DEPARTMENT OF COMPUTER & INFORMATION SYSTEMS ENGINEERING

Course Code: CS-323

Course Title: Artificial Intelligence

Open Ended Lab

TE Batch 2022, Fall Semester 2024

Grading Rubric

Group Members:

Student No.	Name	Roll No.
S1		
S2		
S3		

CRITERIA AND SCALES				**Marks Obtained**		
				S1	**S2**	**S3**
Criterion 1: Has the student appropriately simulated the working of the genetic algorithm?						
0	1	2	-			
The explanation is too basic.	The algorithm is explained well with an example.	The explanation is much more comprehensive.				
Criterion 2: How well is the student's understanding of the genetic algorithm?						
0	1	2	3			
The student has no understanding.	The student has a basic understanding.	The student has a good understanding.	The student has an excellent understanding.			
Criterion 3: How good is the programming implementation?						
0	1	2	3			
The project could not be implemented.	The project has been implemented partially.	The project has been implemented completely but can be improved.	The project has been implemented completely and impressively.			
Criterion 4: How good is the selected application?						
0	1	2	-			
The chosen application is too simple.	The application is fit to be chosen.	The application is different and impressive.				
Criterion 5: How well-written is the report?						
0	1	2	-			
The submitted report is unfit to be graded.	The report is partially acceptable.	The report is complete and concise.				
			Total Marks:			

Kanpsack Problem

The Knapsack problem is an optimization problem that deals with filling up a knapsack with a bunch of items such that the value of the Knapsack is maximized.

The Problem Setup

The goal is to maximize the value of items that can fit into a knapsack with a weight limit (3000 in this case).

Items Definition

Each item is represented by the Thing class, which has:

- name: A string describing the item.
- value: The value of the item (benefit it provides).
- weight: The weight of the item.

The list things defines 10 such items.

The Genetic Algorithm

A genetic algorithm is a heuristic search method inspired by natural evolution. It uses concepts like selection, crossover, and mutation to optimize solutions.

Key Components

- **Genome**: Represents a solution as a binary list, where each position corresponds to whether an item is included (1) or not (0).
  - Example: [1, 0, 1, 0] means the first and third items are included.
- **Fitness Function**: Measures how "good" a solution is.
  - Here, knapsack_fitness() evaluates a genome based on the total value of the selected items, ensuring the total weight does not exceed the limit.

Genetic Operators

- **Population Initialization**: The initial population of solutions is generated using generate_population(), which creates random binary genomes of the given length (len(things)).
- **Selection**: selection_pair() selects two "parent" solutions based on their fitness.
- **Crossover**: single_point_crossover() creates a new solution by combining parts of two parents.
- **Mutation**: mutation() introduces randomness to avoid getting stuck in local optima.

Fitness Evaluation

The knapsack_fitness() function evaluates a genome by:

- Summing the values of selected items (binary 1 in the genome).
- Ensuring the total weight of selected items is within the weight limit (3000).

Steps to Calculate Fitness Value

Inputs:

1. **genome**: A binary list (e.g., [1, 0, 1, 1, 0]), where:
    - 1: The corresponding item is included in the knapsack.
    - 0: The corresponding item is excluded.
2. **things**: A list of items (Thing objects), where each item has:
    - value: The benefit or worth of the item.
    - weight: The item's weight.
3. **weight_limit**: The maximum allowable weight for the knapsack.

Procedure:

1. **Initialize weight and value counters**:

    ```
    weight = 0
    value = 0
    ```

    These will track the cumulative weight and value of selected items.

2. **Iterate through the genome**: For each gene (0 or 1) in the genome:
    - If the gene is 1, include the corresponding item from things:
        - Add the item's weight to the weight counter.
        - Add the item's value to the value counter.
3. **Check the weight constraint**:
    - If at any point the cumulative weight exceeds the weight_limit, the solution is infeasible:

        ```
        if weight > weight_limit:
           return 0
        ```

        In this case, the fitness value is immediately set to 0.

4. **Return the total value**: If the solution is feasible (weight ≤ weight limit), return the accumulated value.

Purpose of the Fitness Limit Parameter

Define a Goal or Optimal Solution:

- The **fitness limit** represents the target fitness value that indicates a solution is "good enough" to solve the problem.
- For example, in the knapsack problem, if you know the maximum achievable value is 740, setting the fitness limit to 740 allows the algorithm to stop early when an optimal or near-optimal solution is found.

Early Termination:

- Without a fitness limit, the algorithm might continue evolving for a fixed number of generations, even if a perfect solution is already found. This wastes computational resources.
- When a solution meets or exceeds the fitness limit, the algorithm can terminate early, saving time and effort.

Control Over Evolution:

- The fitness limit provides a way to stop the algorithm when a solution satisfies the problem's requirements, even if it isn't the absolute best. This is useful for problems where finding an exact optimal solution is computationally expensive.

Ensure Problem-Specific Requirements:

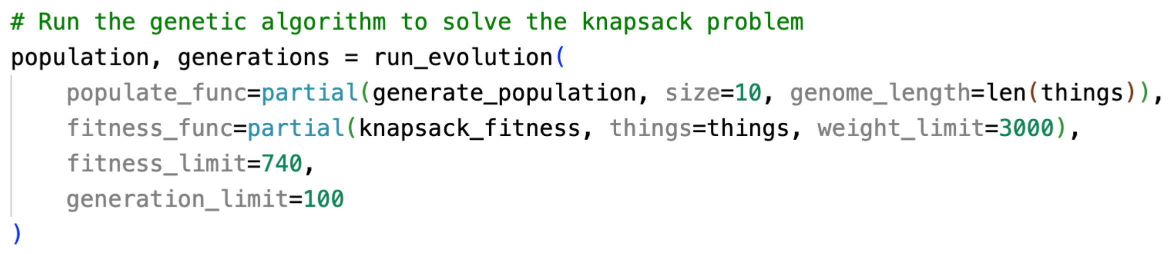
- By setting the fitness limit based on the problem's constraints or desired outcomes, the algorithm is directed toward practical, domain-specific solutions.

The Purpose of partial

In the context of your genetic algorithm, partial is used to simplify function calls by "pre-setting" the parameters that will remain constant throughout the algorithm's execution (such as population size, item list, and weight limit). This way, you don't have to keep passing these parameters repeatedly in each iteration or function call.

Running the Algorithm

The algorithm evolves the population for up to 100 generations or until a solution with a fitness value of at least 740 is found:

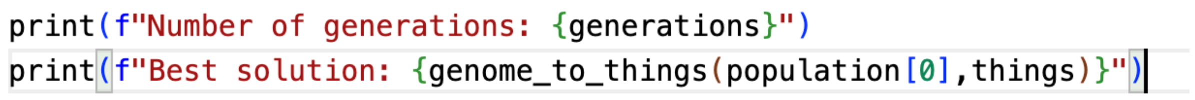


```python
# Run the genetic algorithm to solve the knapsack problem
population, generations = run_evolution(
    populate_func=partial(generate_population, size=10, genome_length=len(things)),
    fitness_func=partial(knapsack_fitness, things=things, weight_limit=3000),
    fitness_limit=740,
    generation_limit=100
)
```

- **Population Size**: 10 genomes.
- **Genome Length**: Matches the number of items (len(things)).
- **Fitness Limit**: Stops if a genome reaches a fitness value of 740.
- **Generation Limit**: Stops after 100 generations.

Extracting Results

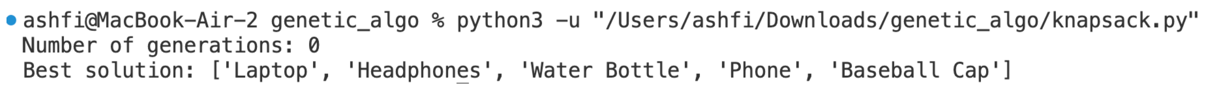
The best solution is stored in population[0], and the genome_to_things() function maps the genome to the corresponding item names:



```python
print(f"Number of generations: {generations}")
print(f"Best solution: {genome_to_things(population[0],things)}")
```

Output

- The number of generations taken to find the solution.
- The list of items (names) corresponding to the best solution:



```
ashfi@MacBook-Air-2 genetic_algo % python3 -u "/Users/ashfi/Downloads/genetic_algo/knapsack.py"
Number of generations: 0
Best solution: ['Laptop', 'Headphones', 'Water Bottle', 'Phone', 'Baseball Cap']
```

Summary

The code systematically uses a genetic algorithm to maximize the total value of items in a knapsack while staying under the weight limit. Each step in the algorithm mimics biological evolution to arrive at an optimal or near-optimal solution.