KNAPSACK PROBLEM SOLVER BY GENETIC ALGORITHM

Group Members
Moatasim Qureshi (CS-22117)
M.Yahya Alyan (CS-22141)
M.Safeer Abdullah (CS-22144)
Submitted to: Dr. Hameeza Ahmed

What is Genetic Algorithm?

It is method for solving both constrained and unconstrained optimization problems that are based on natural selection, the process that drives biological evolution.

The genetic algorithm is evolved through many iterations using following key operations:

1. Defining the fitness Function:

   First define the problem in to some kind of mathematical function known as fitness function.

2. Fitness Evaluation:

   The quality of each solution is evaluated based on a fitness function that reflects how well the solution solves the problem.

3. Selection of parents:

   Chromosomes are selected based on their fitness (how good they are at solving the problem). Better solutions have a higher chance of being chosen as parents.

4. Crossover:

   Two parent chromosomes are combined to create offspring, a random point is selected and then the parts of both parents are intermixed creating two off springs.

5. Mutation:

   Random changes are applied to the chromosomes. There is mutation rate at which these changes are applied. At random a bit or two is flipped in a chromosome ultimately leading more optimal solution.

Introduction

the knapsack is a classic combinatorial optimization problem that makes to select some items that have highest value but their weight should not exceed a certain defined knapsack capacity. The selected items should have the maximum possible value without exceeding the knapsack's weight capacity.

In this this Project Genetic Algorithm was applied to solve 0/1 Knapsack Problem. This report outlines the design and implementation of the Genetic Algorithm to find a high-value solution

for the knapsack problem. The problem involves a predefined set of items, each with a specific weight and value, and a knapsack with a limited carrying capacity. The algorithm evaluates solutions based on a fitness function that prioritizes high-value selections while penalizing solutions exceeding the weight constraint. Over successive generations, the population evolves, improving the quality of the solutions through genetic operations.

Genetic Algorithm For solving Knap Sack Problem:

Under this heading we will be telling how the genetic algorithm was applied to the knapsack problem (1/0)-

1) Chromosome representations:

In the Genetic algorithm each solution is represented as a chromosome. In the 0/1 Knapsack Problem. A chromosome is a binary string of the number of items representing length. Each bit represents the presence and absence of that item.

- **1**: the item is selected
- **0**: the item is not selected

2) Fitness Function:

The fitness function is used to evaluate how good a solution (chromosome) is. In this case of the 0/1 Knapsack Problem, the fitness function needs two factors:

1. **Value**: The total value of the selected items.
2. **Weight**: The total weight of the selected items.

The goal of this algorithm is to maximize the total value while ensuring that the weight does not exceed the total knapsack capacity.

- If the total weight of selected items is **less than or equal to** the knapsack's capacity, the fitness is simply the **total value** of the selected items.
- If the total weight exceeds the knapsack capacity, the fitness is **penalized** and returns a value of **0** (indicating that this solution is invalid).

```
def fitness_function(chromosome):
    total_weights = 0
    total_values = 0
    for i in range(len(chromosome)):
        if chromosome[i] == 1:
            total_weights += items[i][0]
            total_values += items[i][1]
    if total_weights <= 15:
        return total_values
    return 0
```

3) Population Initialization:

The **initial population** is a set of randomly generated chromosomes. Each chromosome represents a potential solution to the knapsack problem. The population is typically generated at random, ensuring diversity in the initial solutions. Over time, the algorithm will evolve these solutions through genetic operations to improve their fitness.

```
def initilize_population():
    population = []
    for i in range(population_size):
        chromosome = []
        for j in range(chromosome_length):
            chromosome.append(random.randint(0,1))
        population.append(chromosome)
    return population
```

4) Selection of Parents:

Selection is the process of choosing **parent chromosomes** for the crossover step. The Genetic Algorithm uses the fitness values to determine how likely each chromosome is to be selected.

The **roulette wheel selection** method is often used for this purpose. It works as follows:

- The probability of selection for each chromosome is proportional to its fitness. Higher fitness chromosomes have a higher chance of being selected.
- The selection process works like a roulette wheel, where the "wheel" is divided into segments with sizes proportional to the fitness of each chromosome.

```
def selecting_parents(population, fitness):
    total_fitness = sum(fitness)
    selection_probabiities = []
    for i in fitness:
        selection_probabiities.append(i/total_fitness)

    parent1 = population[np.random.choice(len(population),p=selection_probabiities)]
    parent2 = population[np.random.choice(len(population),p=selection_probabiities)]

    return parent1,parent2
```

5) Crossover:

After selecting the parents, a crossover operation is performed. A random splitting point is chosen within the parent chromosomes, and genetic material from each parent is interchanged from that point onward. This process results in the creation of two offspring, each inheriting a combination of traits from both parents.

For example:

- **Parent 1:** [1, 0, 1, 0, 1, 0]
- **Parent 2:** [0, 1, 0, 1, 1, 1]

The off springs will be, taking crossover point 3

- **Child 1:** [1, 0, 1, 1, 1, 1]
- **Child 2:** [0, 1, 0, 0, 1, 0]

```
def crossover(parent1,parent2):
    crossover_point = random.randint(0,len(parent1))
    if np.random.rand() < crossover_rate:
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent1[crossover_point:] + parent2[:crossover_point]
        return child1,child2
    return parent1,parent2
```

6) Mutation:

Mutation introduces random changes in the chromosome to main diversity of solutions. It ensured that the algorithm does not get stuck in local optima.

For a knapsack problem, mutation involves flipping a random gene of chromosome.

```python
def mutate(chrmosome):
    mutated_chromosome = []
    for i in chrmosome:
        if np.random.rand() < crossover_rate:
            mutated_chromosome.append(1-i)
        else:
            mutated_chromosome.append(i)
    return mutated_chromosome
```

7) Updating Generation

Once the offspring have been created through crossover and mutation, they are evaluated using the fitness function. The population is then updated with the new generation, replacing the old population.

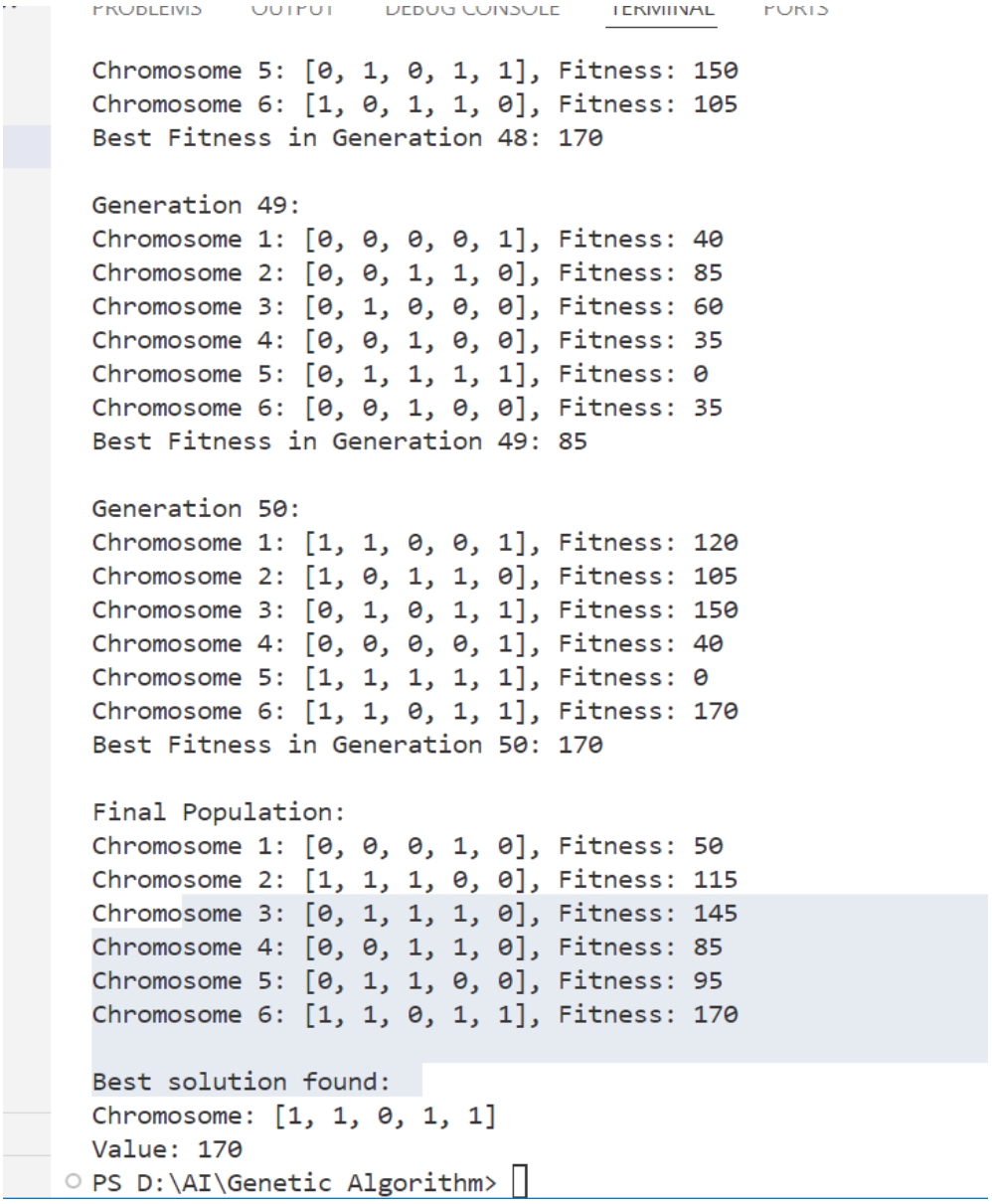
8) Termination:

The algorithm is terminated when the number of generations have been reached or until a satisfactory solution is achieved.

Genetic Algorithm on Python:

```python
def genetic_algo():
    population = initilize_population()
    for generation in range(generations):
        fitness = calculating_fitness(population)
        print(f"\nGeneration {generation + 1}:")
        for i, chromosome in enumerate(population):
            print(f"Chromosome {i + 1}: {chromosome}, Fitness: {fitness[i]}")

        new_population = []
        while len(new_population) < population_size:
            parent1, parent2 = selecting_parents(population, fitness)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)

            new_population.append(child1)
            if len(new_population) < population_size:
                new_population.append(child2)

        population = new_population

        best_fitness = max(fitness)
        print(f"Best Fitness in Generation {generation + 1}: {best_fitness}")

    print("\nFinal Population:")
    for i, chromosome in enumerate(population):
        print(f"Chromosome {i + 1}: {chromosome}, Fitness: {fitness_function(chromosome)}")

    fitness = calculating_fitness(population)
    best_index = fitness.index(max(fitness))
    best_solution = population[best_index]

    print("\nBest solution found:")
    print("Chromosome:", best_solution)
    print("Value:", fitness_function(best_solution))
```

OUTPUT:



Conclusion:

In conclusion, the algorithm successfully identified the optimal solution by selecting the items with the highest value while ensuring the total weight remained within the specified limit.