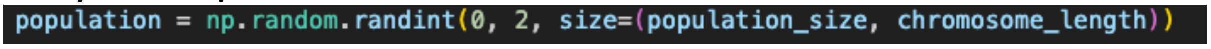
CT421 Assignment 1 - Bin Packing Problem

Eoghan O’Brien ID:20451106

<u>Representation of Solutions:</u>

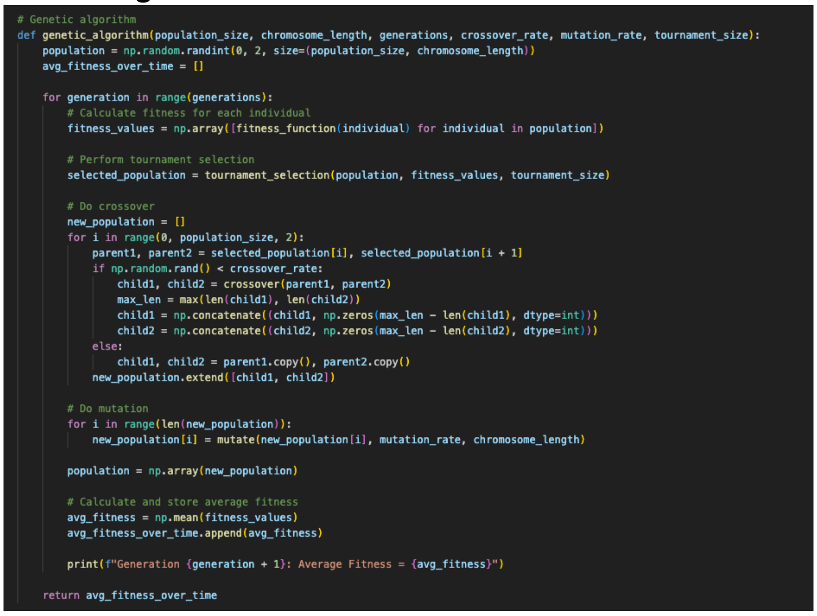
In the genetic algorithm for this Bin Packing problem, solutions are represented using binary arrays. Each array represents a possible packing layout in which an item’s presence in a bin is shown by a binary value (1 for being present, 0 for absent). While maintain the combinatorial aspect of the problem, this binary representation makes it easier to manipulate solutions efficiently during crossover and mutation processes.

Binary matrix representation line of code:



```python
population = np.random.randint(0, 2, size=(population_size, chromosome_length))
```

Genetic algorithm:

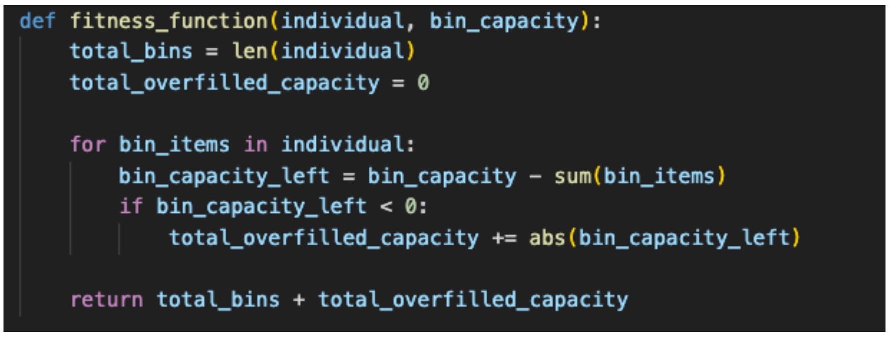


```python
# Genetic algorithm
def genetic_algorithm(population_size, chromosome_length, generations, crossover_rate, mutation_rate, tournament_size):
    population = np.random.randint(0, 2, size=(population_size, chromosome_length))
    avg_fitness_over_time = []

    for generation in range(generations):
        # Calculate fitness for each individual
        fitness_values = np.array([fitness_function(individual) for individual in population])

        # Perform tournament selection
        selected_population = tournament_selection(population, fitness_values, tournament_size)

        # Do crossover
        new_population = []
        for i in range(0, population_size, 2):
            parent1, parent2 = selected_population[i], selected_population[i + 1]
            if np.random.rand() < crossover_rate:
                child1, child2 = crossover(parent1, parent2)
                max_len = max(len(child1), len(child2))
                child1 = np.concatenate((child1, np.zeros(max_len - len(child1), dtype=int)))
                child2 = np.concatenate((child2, np.zeros(max_len - len(child2), dtype=int)))
            else:
                child1, child2 = parent1.copy(), parent2.copy()
            new_population.extend([child1, child2])

        # Do mutation
        for i in range(len(new_population)):
            new_population[i] = mutate(new_population[i], mutation_rate, chromosome_length)

        population = np.array(new_population)

        # Calculate and store average fitness
        avg_fitness = np.mean(fitness_values)
        avg_fitness_over_time.append(avg_fitness)

        print(f"Generation {generation + 1}: Average Fitness = {avg_fitness}")

    return avg_fitness_over_time
```

<u>Fitness functions and Operators:</u>

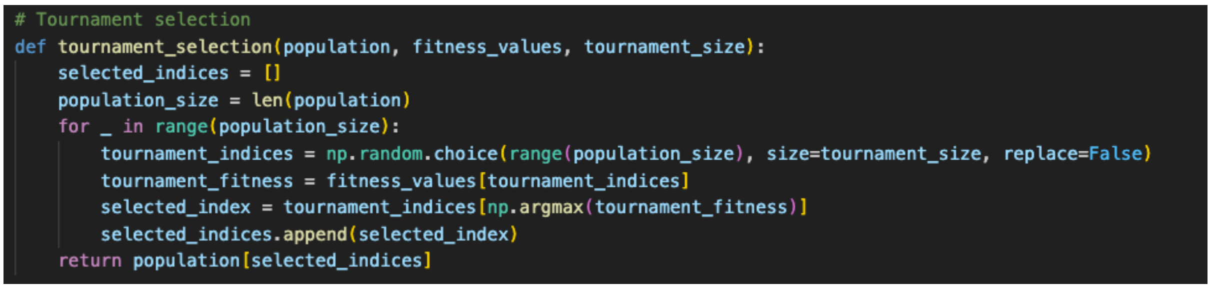
Fitness function:

An important factor when determining the quality of the solutions produced by the genetic algorithm is the fitness function. With an extra penalty for bins that are overfilled, the fitness function is defined as the total numbers of bins used. This method encourages an effective packing technique by rewarding the algorithm to minimise the amount of overfilled capacity as well as the overall numbers of bins used.



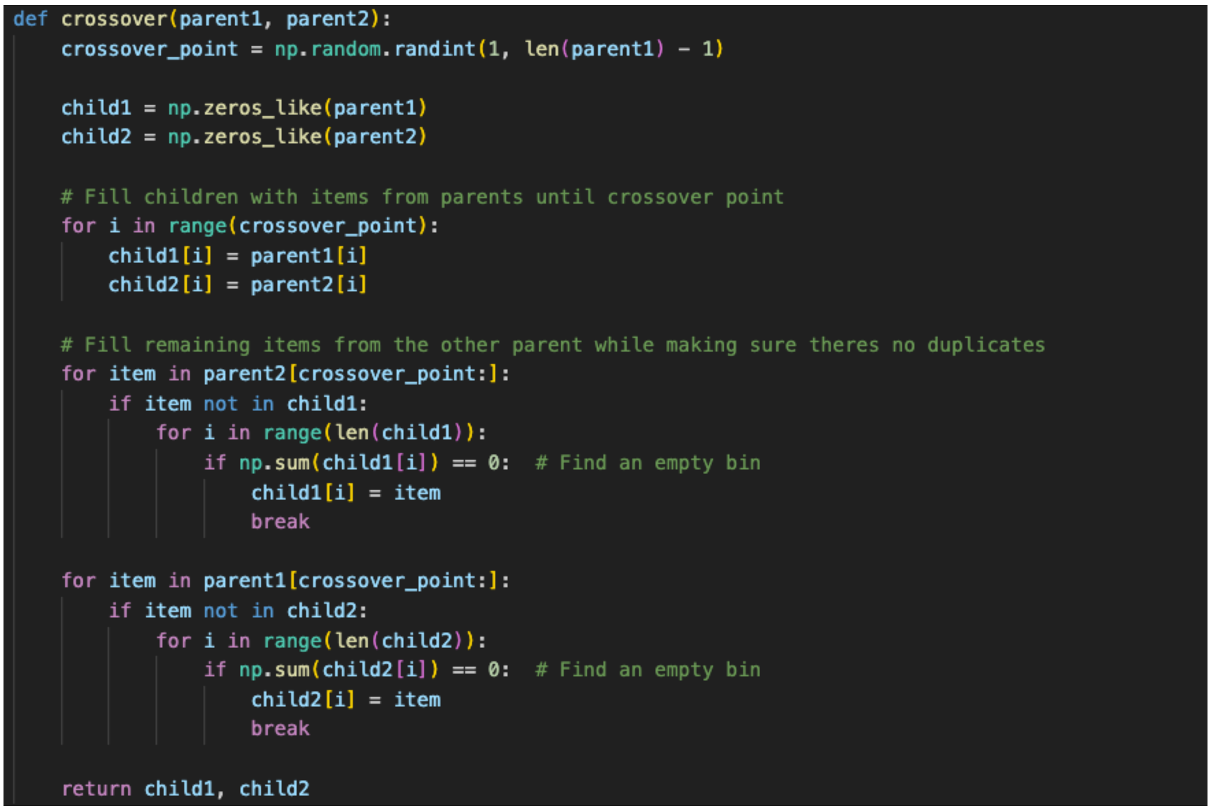
```python
def fitness_function(individual, bin_capacity):
    total_bins = len(individual)
    total_overfilled_capacity = 0

    for bin_items in individual:
        bin_capacity_left = bin_capacity - sum(bin_items)
        if bin_capacity_left < 0:
            total_overfilled_capacity += abs(bin_capacity_left)

    return total_bins + total_overfilled_capacity
```

Selection:
Tournament selection is used in the selection method, in which individuals are chosen at random and put against one another in tournaments. Then, the top performers are selected to be crossover parents. Early convergence is avoided and variety within the population is preserved due to this technique.



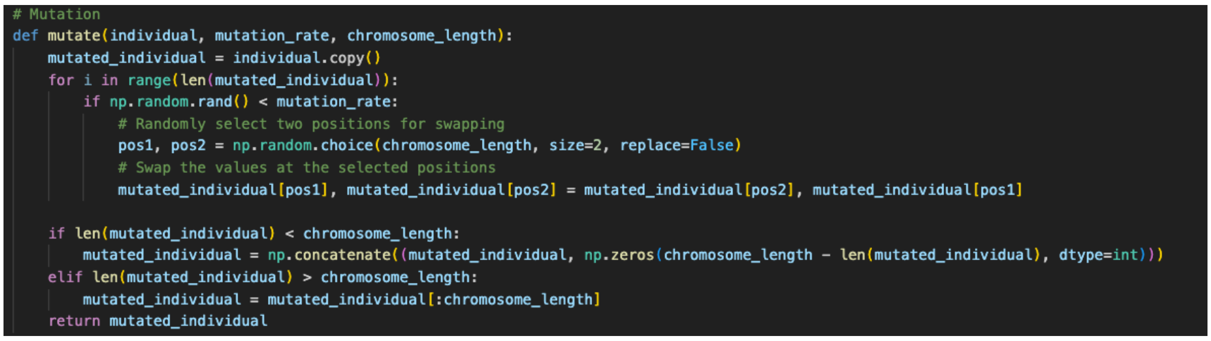
```python
# Tournament selection
def tournament_selection(population, fitness_values, tournament_size):
    selected_indices = []
    population_size = len(population)
    for _ in range(population_size):
        tournament_indices = np.random.choice(range(population_size), size=tournament_size, replace=False)
        tournament_fitness = fitness_values[tournament_indices]
        selected_index = tournament_indices[np.argmax(tournament_fitness)]
        selected_indices.append(selected_index)
    return population[selected_indices]
```

Crossover:
To recombine genetic material from parent solutions, one-point crossover is used. This method encourages exploration of the solution space and makes sure that offspring follow the one item per bin limitation. Through the random crossover point generation process, it produces a range of viable solutions that have beneficial characteristics from both parents.



```python
def crossover(parent1, parent2):
    crossover_point = np.random.randint(1, len(parent1) - 1)

    child1 = np.zeros_like(parent1)
    child2 = np.zeros_like(parent2)

    # Fill children with items from parents until crossover point
    for i in range(crossover_point):
        child1[i] = parent1[i]
        child2[i] = parent2[i]

    # Fill remaining items from the other parent while making sure theres no duplicates
    for item in parent2[crossover_point:]:
        if item not in child1:
            for i in range(len(child1)):
                if np.sum(child1[i]) == 0:  # Find an empty bin
                    child1[i] = item
                    break

    for item in parent1[crossover_point:]:
        if item not in child2:
            for i in range(len(child2)):
                if np.sum(child2[i]) == 0:  # Find an empty bin
                    child2[i] = item
                    break

    return child1, child2
```

Mutation:
In order to maintain genetic variation and avoid stagnation, a mutation operator is used that switches two items within a chromosome at random. Through the introduction of small random adjustments to each person's genetic composition, this approach makes it easier to explore new solution areas.



```python
# Mutation
def mutate(individual, mutation_rate, chromosome_length):
    mutated_individual = individual.copy()
    for i in range(len(mutated_individual)):
        if np.random.rand() < mutation_rate:
            # Randomly select two positions for swapping
            pos1, pos2 = np.random.choice(chromosome_length, size=2, replace=False)
            # Swap the values at the selected positions
            mutated_individual[pos1], mutated_individual[pos2] = mutated_individual[pos2], mutated_individual[pos1]

    if len(mutated_individual) < chromosome_length:
        mutated_individual = np.concatenate((mutated_individual, np.zeros(chromosome_length - len(mutated_individual), dtype=int)))
    elif len(mutated_individual) > chromosome_length:
        mutated_individual = mutated_individual[:chromosome_length]
    return mutated_individual
```

Insights into Problem landscape
The Bin Packing problem provides a difficult combinatorial optimisation task with real-world applications in the allocation of resources and logistics. This problem landscape is defined by a large search space with multiple local optima. It takes careful handling of constraints like the one item per bin rule and a balance between exploration and exploitation to find the most optimal solution. Exploring different mutation rates and crossover techniques become important to effectively explore this complex problem space.