Introduction

The code provided is a simulation of a rabbit in a board game. The rabbit has to collect carrots while avoiding obstacles and traps. The simulation uses a genetic algorithm to optimize the rabbit's moves in order to collect the maximum number of carrots while avoiding traps and consuming the minimum amount of energy.

Experimentation Explained

The code consists of three main parts. The first part generates the board, initializes the rabbit's position, direction, moves, and energy, and displays them. The second part is the genetic algorithm that uses the fitness function to evaluate the fitness of each individual and selects the best individuals for reproduction. The third part executes the best individual's genes (actions) to simulate the rabbit's moves and display the final state of the board, rabbit, and carrots.

Algorithm

1. generate_board(): This function generates a random board of size ROWS x COLS with the given cell types and probabilities using random.choices() function. The cell types are defined as constants: EMPTY = 0, CARROT = 1, OBSTACLE = 2, and TRAP = 3. The function returns the generated board.

2. display_board(board): This function displays the board in a 2D way using the symbols defined for each cell type in a dictionary: symbols = {EMPTY: ".", CARROT: "C", OBSTACLE: "O", TRAP: "T"}. The function takes a board as an argument and prints the board in a 2D way.

3. generate_rabbit(): This function generates a random initial position, direction, and returns them as x, y, and direction.

4. display_rabbit(x, y, direction, moves, energy): This function displays the rabbit's position, direction, moves, and energy. It uses a dictionary to map actions to direction symbols: directions = {UP: "↑", DOWN: "↓", LEFT: "←", RIGHT: "→"}.

5. fitness(individual, board, x, y, direction, moves, energy): This function evaluates the fitness of an individual (a list of genes) by simulating the rabbit's moves on the board and updating the rabbit's position, direction, energy, and score based on the cell types and actions in each position. The function returns the score as the fitness value.

6. genetic_algorithm(board, x, y, direction, moves, energy, population_size=100, num_generations=50, mutation_rate=0.1): This function uses a genetic algorithm to optimize the rabbit's moves to collect the maximum number of carrots while avoiding traps and consuming the minimum amount of energy. It takes the board, rabbit's position, direction, moves, and energy as arguments and optional parameters for population size, number of generations, and mutation rate. It initializes the population with random individuals, evaluates the fitness of each individual using the fitness function, selects the best individuals for reproduction, generates offspring by crossover and mutation, replaces the old population with the new population (elites + offspring), and repeats this process for a given number of generations. Finally, it selects the best individual as the solution and simulates the rabbit's moves using the individual's genes (actions).

Analysis

The simulation uses a genetic algorithm to optimize the rabbit's moves based on a fitness function that evaluates the score of the rabbit's moves. The fitness function simulates the rabbit's moves on the board by updating the rabbit's position, direction, energy, and score based on the cell types and actions in each position. The genetic algorithm initializes the population with random individuals, evaluates the fitness of each individual, selects the best individuals for reproduction, generates offspring by crossover and mutation, and replaces the old population with the new population (elites + offspring). The algorithm repeats this process for a fixed number of generations.

The performance of the algorithm depends heavily on the size of the board, number of moves, and population size. A larger board with more obstacles and traps may make it harder for the algorithm to find the optimal solution. Similarly, a smaller population size may limit the diversity of the population and prevent the algorithm from finding better solutions.

The algorithm may also get stuck in local optima, where it is unable to find a better solution because it is trapped in a suboptimal area of the search space. This can be overcome by using additional techniques, such as increasing the mutation rate, introducing new genes randomly, or using multiple runs with different random seeds.

Despite these limitations, the genetic algorithm is a powerful tool for solving optimization problems and can be used to find the optimal solution to a wide range of problems. It has been successfully applied in various fields, such as engineering design, finance, and bioinformatics.

In conclusion, the analysis shows that the genetic algorithm is an effective method for optimizing the rabbit's moves in the carrot-collecting problem. However, the performance of the algorithm may depend on the size of the board, number of moves, and population size, and may require additional techniques to overcome local optima.

Output

Defined Functions Output

```
    # Display the rabbit's position, direction, moves, and energy
    def display_rabbit(x, y, direction, moves, energy):
        directions = {UP: "↑", DOWN: "↓", LEFT: "←", RIGHT: "→"}
        print(f"Rabbit's position: ({x}, {y})")
        print(f"Rabbit's direction: {directions[direction]}")
        print(f"Rabbit's moves left: {moves}")
        print(f"Rabbit's energy: {energy}")
        print()
[6] ✓ 0.0s

    # Test the code by generating and displaying a board and a rabbit
    board = generate_board()
    display_board(board)
    x, y, direction = generate_rabbit()
    display_rabbit(x, y, direction, MOVES, ENERGY)
[7] ✓ 0.0s
...
    . O . . C
    . . . . .
    . . . T C
    . . . . .
    . O C O C

    Rabbit's position: (2, 0)
    Rabbit's direction: ←
    Rabbit's moves left: 20
    Rabbit's energy: 10
```

Final Output

```
    board = generate_board()
    display_board(board)
    x, y, direction = generate_rabbit()
    display_rabbit(x, y, direction, MOVES, ENERGY)
    genetic_algorithm(board, x, y, direction, MOVES, ENERGY)

 ✓ 0.1s
Output exceeds the size limit. Open the full output data in a text editor
C C . T .
. . C . .
. O T C T
O . T . .
T . . . C

Rabbit's position: (4, 0)
Rabbit's direction: ↑
Rabbit's moves left: 20
Rabbit's energy: 10

Generation 1: best fitness score = 9
Generation 2: best fitness score = 9
Generation 3: best fitness score = 9
Generation 4: best fitness score = 9
Generation 5: best fitness score = 9
Generation 6: best fitness score = 9
Generation 7: best fitness score = 9
Generation 8: best fitness score = 9
Generation 9: best fitness score = 9
Generation 10: best fitness score = 9
Generation 11: best fitness score = 9
Generation 12: best fitness score = 9
Generation 13: best fitness score = 9
Generation 14: best fitness score = 9
...
Rabbit's direction: 3
Rabbit's moves left: 15
Rabbit's energy: 7
Number of carrots collected: 7
```

Conclusion

In conclusion, the report documents the implementation and analysis of a genetic algorithm to find the optimal path for a rabbit to collect carrots while avoiding obstacles and traps on a randomly generated board. The algorithm uses a fitness function to evaluate the fitness of each individual in the population and a genetic algorithm to generate a population of random individuals and iteratively select the best individuals for reproduction, generate offspring by crossover and mutation, and replace the old population with the new population.

The algorithm's performance depends heavily on the size of the board, number of moves, and population size. The algorithm may also get stuck in local optima and may require additional techniques to escape these optima. However, the genetic algorithm is a powerful tool for solving optimization problems and can be used to find the optimal solution to a wide range of problems.

Overall, the implementation and analysis of the algorithm provide insights into the effectiveness and limitations of genetic algorithms in solving optimization problems.