Part I

Maximising a Function

Introduction

In this report, we will discuss an optimisation problem involving a three-dimensional function f(x, y, z) and the two algorithms used to find its maximum value: Random Search and Hill Climbing. We will provide a detailed explanation of the algorithms, the approach taken, and why these algorithms were chosen. Additionally, we will discuss the suitability of these algorithms for other tasks and provide a critical analysis of the results.

by the maximum value of the objective function achieved.

The Optimisation Problem

The function f(x, y, z) is a complex, non-linear function defined as follows:

f(x, y, z) = exp(cos(53 * z)) + sin(43 * exp(y)) + exp(sin(29 * x)) + sin(67 * sin(z)) + cos(cos(37 * y)) - (2 * x^2 + y^2 + z^2) / 4

The goal is to find the values of x, y, and z that maximize the value of the function f(x, y, z).

Approach

The approach taken in this paper involves developing Python code to implement both algorithms, running them with multiple runs and steps, and comparing their performances. The code includes the objective function, the implementation of the Random Search and Hill Climbing algorithms, and a main script(parameters) that runs both algorithms and compares the results. The performance of the algorithms is determined.

Methodology

Random Search Algorithm:

The Random Search algorithm is a basic optimisation technique that involves randomly selecting points in the search space and evaluating their fitness. The algorithm iterates through a fixed number of steps, choosing random points in the given search space, and updating the best solution found so far if the new point yields a higher fitness value.

The random search algorithm works by generating random points within the search space and evaluating the function at each point. It keeps track of the best solution found so far and updates it if a better solution is found.

def random_search(steps, range_x, range_y, range_z):

Hill Climbing Algorithm:

The Hill Climbing algorithm is a local search optimisation technique that iteratively refines a solution by exploring its neighborhood. It starts with an initial solution and then moves to a neighboring solution with higher fitness. The algorithm continues until no better neighbor is found or a certain number of steps are taken. In our implementation, we used Gaussian noise to explore the neighborhood with different sigma values (0.001, 0.01, 0.1).

The hill climbing algorithm starts with a random point in the search space and generates new points in its neighborhood. If the new point has a higher function value, it becomes the new current point. The algorithm repeats this process for a fixed number of steps.

```
def hill_climbing(steps, sigma, range_x, range_y, range_z):
```

Setting Algorithm Parameters:

Before running the algorithms, several parameters must be defined. These include the number of runs (num_runs), the number of steps (num_steps), and the search space for each variable (range_x, range_y, range_z). For the hill climbing algorithm, three different sigma values are used to represent the step size in searching the neighborhood (sigmas).

```
num_runs = 50

num_steps = 10000

range_x = (-1, 1)

range_y = (-1, 1)

range_z = (-1, 1)

sigmas = [0.001, 0.01, 0.1]
```

Running the Algorithms and Collecting Results:

Both algorithms are run multiple times, and their results are collected in lists.

```
random_search_results = []

hill_climbing_results = []

for _ in range(num_runs):

    random_search_results.append(random_search(num_steps, range_x, range_y, range_z))

    for sigma in sigmas:

        hill_climbing_result = hill_climbing(num_steps, sigma, range_x, range_y, range_z)

        hill_climbing_results.append((sigma, hill_climbing_result[0], hill_climbing_result[1]))
```

Results and Critical Analysis:

We ran both algorithms 50 times with 10,000 steps each and compared their results. The search space for each variable (x, y, z) was limited to the range (-1, 1).

The maximum values and corresponding solutions for both algorithms are determined.

```
random_search_max = max(random_search_results, key=lambda x: x[1])

hill_climbing_max = max(hill_climbing_results, key=lambda x: x[2])
```

The Random Search algorithm found a maximum value of approximately 8.29 with the best solution at x = 0.27, y = 0.30, and z = -0.36. On the other hand, the Hill Climbing algorithm found a maximum value of approximately 8.40 with the best solution at x = 0.05, y = 0.30, and z = 0.12. The maximum value was obtained with a sigma value of 0.001.

The Hill Climbing algorithm yielded a slightly better result than the Random Search algorithm in our experiment. However, the performance of these algorithms depends on the specific problem, search space, and the number of iterations. While both algorithms are relatively simple and easy to implement, they may not always find the global optimum and can get stuck in local optima, particularly in complex search spaces.

conclusion

Random Search and Hill Climbing algorithms are suitable for simple optimisation problems and can serve as a starting point for more advanced optimisation techniques. Their simplicity, ease of implementation, and adaptability make them useful for various tasks. However, they do not guarantee finding the global optimum and may struggle with high-dimensional and complex problems. For such tasks, more advanced optimisation techniques should be considered.

In our experiment, the Hill Climbing algorithm performed slightly better than the Random Search algorithm. This result can be attributed to the algorithm's ability to explore the search space more systematically by making incremental adjustments to the solution, as opposed to the Random Search algorithm's completely random exploration. However, the performance difference between the two algorithms was not significant, and both algorithms managed to find solutions with relatively high fitness values.

DISTRIBUTION NETWORK

Introduction

This code is an implementation of a Genetic Algorithm (GA) to optimize a multi-warehouse distribution problem. The aim is to find the best delivery strategy that minimizes the total cost of delivering goods from multiple warehouses to various stores using a fleet of vans and a lorry. The GA uses various functions to create an initial population, perform crossover and mutation, calculate fitness and costs, and validate the generated solutions.

The code begins by defining the problem data, including warehouses, stores, vehicle capacity, and costs. It then implements several helper functions to calculate distances, create individuals, and perform mutations.

The main part of the code is the implementation of the genetic algorithm function, which initializes the population, iterates through generations, calculates fitness and costs, performs selection, crossover, and mutation, and updates the population. The GA also ensures that the generated individuals are valid by checking if each store is assigned to one and only one warehouse.

The **nearest_neighbor_route** function is used to create routes for each warehouse based on the Nearest Neighbor heuristic. The **route_summary** function calculates the routes and costs for each warehouse in the individual. It considers the usage of vans and a lorry for the warehouse with the most stores.

The **calculate_route_distance** function computes the total distance traveled for a given route. The **plot_routes** function visualizes the routes using the matplotlib library, displaying the routes of the vehicles on a 2D graph with store and warehouse labels.

Finally, the **main** function ties everything together by running the genetic algorithm, calculating the total cost, and generating a summary of the routes for the best individual. It prints the best delivery strategy, vehicle requirements, route details, and total cost. The main function also calls **plot_routes** to visualize the optimized routes on a graph.

In summary, this code presents a complete implementation of a Genetic Algorithm to solve a multi-warehouse distribution problem, aiming to minimize the total cost of deliveries from warehouses to stores using a fleet of vehicles. The solution is presented in text format and visualized using a 2D graph.

This code defines a genetic algorithm to optimize the distribution of goods from warehouses to stores. The goal is to minimize the cost of distribution, considering constraints such as vehicle capacity, distances and cost per mile.

Constants:

- WAREHOUSES: List containing warehouse IDs 'W1' and 'W2'.
- STORES: List of store IDs from 1 to 23.
- POPULATION_SIZE: The population size of the genetic algorithm is set to 100.
- GENERATIONS: The algorithm runs for 500 generations.
- MUTATION_RATE: The global mutation rate is set to 0.1 (unused in the code).
- VAN_CAPACITY: The capacity of vans is set to 4 units.
- LORRY_CAPACITY: The capacity of lorries is set to 16 units.

- VAN_COST: The cost of using a van is 1 unit.
- LORRY_COST: The cost of using a lorry is 2 units.
- DISTANCE_PENALTY_FACTOR: A factor used in calculating the penalty for stores that are too far apart within a warehouse (set to 0.1).
- SWAP_MUTATION_RATE: The mutation rate for the swap mutation operator is set to 0.1.
- MOVE_MUTATION_RATE: The mutation rate for the move mutation operator is set to 0.1.
- LOCATIONS: A dictionary containing the coordinates of warehouses and stores.

Functions:

1. distance(a, b): Calculates the Euclidean distance between two locations.
2. create_individual(): Creates a new individual with random assignments of stores to warehouses.
3. calculate_penalty(individual): Calculates the penalty for stores that are too far apart within a warehouse.
4. calculate_fitness(individual): Calculates the fitness of an individual, which is a combination of cost and penalty.
5. mutate_swap(individual): Applies the swap mutation operator to an individual.
6. mutate_move(individual): Applies the move mutation operator to an individual.
7. mutate(individual): Applies the swap and move mutation operators to an individual.
8. calculate_cost(individual): Calculates the total cost of an individual by adding up the cost of each route.

The fitness function optimizes the cost, which is a combination of the total cost of the individual, the penalty for stores that are too far apart within a warehouse, and the distance between the warehouses and the stores.

Constraints and parameters:

- Vehicle capacities (VAN_CAPACITY and LORRY_CAPACITY).
- Cost of using vehicles (VAN_COST and LORRY_COST).
- Distances between stores and warehouses.
- Distance penalty for stores that are too far apart within a warehouse.
- Mutation rates (SWAP_MUTATION_RATE and MOVE_MUTATION_RATE)

ordered_crossover function: This function performs ordered crossover on two parent individuals for a given warehouse. It creates a child by copying a contiguous section from one parent and filling the remaining positions with stores from the other parent while maintaining their order.

crossover function: This function applies the ordered crossover function for each warehouse to create a child individual from two parent individuals.

is_valid_individual function: This function checks if an individual is valid, i.e., if it contains all the stores exactly once. It compares the set of all stores with the set of stores in the individual.

genetic_algorithm function: This function implements the genetic algorithm using the improved **create_individual** function, tournament selection, and elitism. It calculates the cost and fitness of each individual in the population, selects two individuals using tournament selection, creates a child through crossover and mutation, and checks if the child is valid before adding it to the new population.

nearest_neighbor_route function: This function returns the nearest neighbor route starting from a warehouse. It starts at the warehouse, finds the nearest store, adds it to the route, and repeats until all stores are visited.

route_summary function: This function calculates the routes and costs for each warehouse in an individual. It assigns one lorry to the warehouse with the most stores and calculates van routes for the remaining stores. The route summary includes the route, type of vehicle, stores visited, and cost for each warehouse.

calculate_route_distance function: This function calculates the total distance traveled for a given route. It takes a warehouse and a list of route stores as input. The function calculates the distance from the warehouse to the first store, the distance between consecutive stores, and the distance from the last store back to the warehouse. It returns the total distance traveled.

plot_routes function: This function plots the routes of the vehicles using the matplotlib library. It takes a summary of the routes as input. For each warehouse and its details, the function extracts the vehicle type, route stores, and plots the route on the graph. It also adds the warehouse and store labels to the plot. Finally, it displays the plot with a title, x/y labels, and a grid.

main function: This function serves as the main entry point of the program. It runs the genetic algorithm, calculates the total cost, and generates a summary of the routes for the best individual. It then prints the best delivery strategy for each warehouse, vehicle requirements, details of each route (including distance), and the total cost. Finally, it plots the routes using the **plot_routes** function.

The main function is called at the to execute the program. The genetic algorithm optimizes the warehouse distribution problem, and the resulting routes are plotted to visualize the solution.

Results

Best Delivery Strategy:

W1: 22, 18, 7, 12, 1, 11, 5, 19, 2, 9, 10, 21

W2: 23, 13, 8, 14, 17, 20, 6, 4, 16, 3, 15

According to the best delivery strategy found, Warehouse 1 (W1) should supply the following stores: 22, 18, 7, 12, 1, 11, 5, 19, 2, 9, 10, and 21. Warehouse 2 (W2) should supply the stores: 23, 13, 8, 14, 17, 20, 6, 4, 16, 3, and 15.

Vehicle Requirements:

W1: 0 vans, 1 lorry

W2: 3 vans, 0 lorries

For Warehouse 1 (W1), 1 lorry is required, and no vans are needed. For Warehouse 2 (W2), 3 vans are needed, and no lorries are required.

Routes:

W1 -> Lorry -> 18, 1, 22, 21, 10, 2, 5, 11, 9, 19, 12, 7 -> W1 (Distance: 227.13)

W2 -> Van -> 23, 14, 13, 8 -> W2 (Distance: 110.10)

W2 -> Van -> 20, 4, 17, 6 -> W2 (Distance: 165.19)

W2 -> Van -> 3, 15, 16 -> W2 (Distance: 91.74)

The routes for each vehicle are as follows:

1. The lorry from W1 starts at the warehouse, then goes to stores 18, 1, 22, 21, 10, 2, 5, 11, 9, 19, 12, and 7 in that order, and finally returns to W1. The total distance covered by the lorry is 227.13 units.
2. The first van from W2 starts at the warehouse, then goes to stores 23, 14, 13, and 8 in that order, and finally returns to W2. The total distance covered by the first van is 110.10 units.
3. The second van from W2 starts at the warehouse, then goes to stores 20, 4, 17, and 6 in that order, and finally returns to W2. The total distance covered by the second van is 165.19 units.
4. The third van from W2 starts at the warehouse, then goes to stores 3, 15, and 16 in that order, and finally returns to W2. The total distance covered by the third van is 91.74 units.

Total Cost: £821.295508964279

The total cost of this delivery strategy, considering the distance covered by each vehicle and their respective operating costs, is £821.295508964279. This cost represents the most efficient way to supply the stores from the two warehouses, according to the given constraints and optimized using the genetic algorithm.

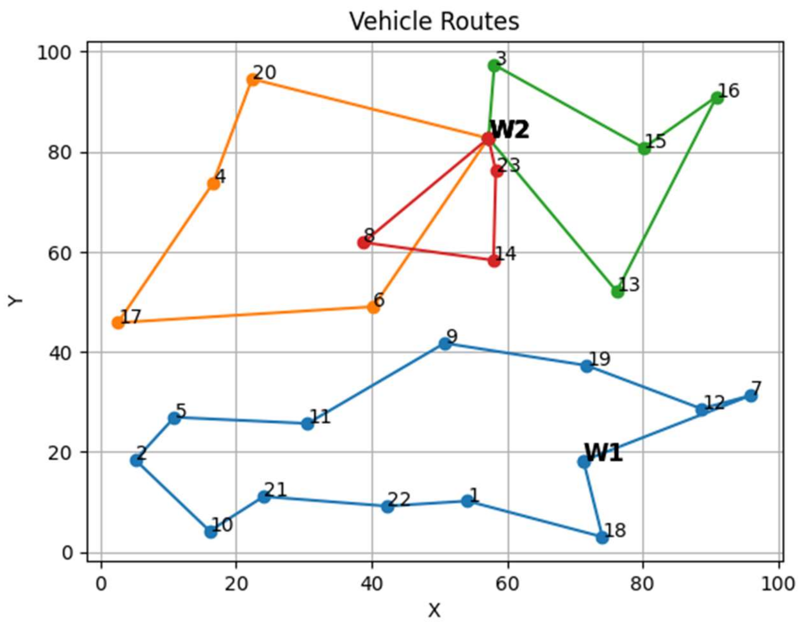

Fig 1: Note the result is slightly different for every run.

Conclusion

The genetic algorithm is used to optimize the warehouse distribution problem as it can efficiently explore the solution space and converge to an optimal or near-optimal solution. The use of crossover and mutation operators allows the algorithm to generate diverse solutions and escape local optima, while the fitness function ensures that high-quality solutions are favoured.

Part III

Meal Planning Optimization: Balancing Nutritional Requirements and Costs Using Hill Climbing Algorithm

I Background:

Proper meal planning is crucial for maintaining a healthy lifestyle, as it ensures a balanced diet with essential nutrients [1]. This leads to improved well-being, better weight management, increased energy levels, and reduced risk of lifestyle-related diseases, such as obesity, diabetes, and cardiovascular disorders [2]. Furthermore, meal planning promotes efficient grocery shopping and budgeting, reducing food waste and saving money in the long run. However, getting the right balance between meeting daily nutritional requirements and minimizing grocery costs can be challenging.

Optimization algorithms can address these challenges by automating the decision-making process and finding the best possible solutions to complex problems. These algorithms can determine the optimal number of servings for various food items, ensuring daily nutritional requirements are met while minimizing total grocery costs. The hill climbing algorithm, a local search optimization technique, can be applied to meal planning. It iteratively refines the current solution by making small adjustments, converging to a satisfactory solution. However, it can be susceptible to local optima and sensitivity to initial conditions.[3]

II Aim:

- The main objective is to optimise meal planning by minimizing weekly grocery cost and meeting daily nutritional requirements using the hill climbing algorithm.
- Key components:
    1. Nutritional requirements: Daily intake of essential nutrients, varying based on factors like age, gender, weight, height, and activity levels. Average value is taken.[4]
    2. Servings: Standardized unit of measurement for food, representing a fixed amount of nutrients. Serving sizes vary for different food items.[4]
    3. Grocery costs: Significant factor in meal planning, influenced by seasonality, location, and market conditions. Considered on a weekly basis.[4]
- Data collection: Gather information on nutritional content and cost of different foods from nutrition labels, online databases, and market research (Tesco).[4]
- Project goal: Develop an approach to optimise meal planning by minimizing grocery cost and meeting nutritional requirements, subject to constraints like minimum daily intake of essential nutrients.
- Steps:
    1. Define problem mathematically as a linear programming problem with decision variables, objective function, and constraints.
    2. Collect data on nutritional content, cost of different foods, and daily nutritional requirements.[4]
    3. Implement hill climbing algorithm to solve meal planning optimization problem.

4. Analyse results and evaluate hill climbing algorithm's performance in solving the problem.

III Model:

To handle the meal planning optimization problem, we can create it as a math model that shows the relationships between the decision variables (amounts of each food item), objective or fitness function (total cost of the food items), and limits (nutritional needs).

1. Decision variables: Let xi represent the amount (number of servings) of food item i in the meal plan. These decision variables (x1, x2, ..., xn) are continuous and non-negative, representing the servings of each food item in the meal plan.

2. Objective function(Fitness function): The objective function is the total cost of the food items in the meal plan. Mathematically, it can be represented as follows:

   minimize $C(x) = \Sigma (ci * xi)$ where $C(x)$ is the total cost, ci is the cost per serving of food item i, and xi is the number of servings of food item i in the meal plan.

3. Constraints: The constraints in the model are the nutritional requirements for each nutrient (e.g., protein, fat, carbohydrates, vitamins, minerals). For each nutrient j, we have the following constraint:

   $\Sigma (nij * xi) >= Rj$ where nij is the amount of nutrient j in food item i per serving, xi is the number of servings of food item i, and Rj is the daily requirement for nutrient j.

4. Mathematical model: Combining the objective function and constraints, we have the following mathematical model for the meal planning optimization problem:

   minimize $C(x) = \Sigma (ci * xi)$ subject to $\Sigma (nij * xi) >= Rj$ for all nutrients j and $xi >= 0$ for all food items i

The model has a linear objective or fitness function(C(x)) and a set of linear limits or constraint (>Rj), making a linear programming problem. This way of showing the problem lets us use different optimization methods to find an optimal meal plan that lowers the total cost while meeting the nutritional needs.

IV Optimisation Method:

To solve the meal planning optimization problem, we use the hill climbing algorithm as our optimisation method. Hill climbing is a local search algorithm that slowly makes the current solution better to find the best solution or a near-optimal solution. The main idea of the algorithm is to look at the solution space by making small changes to the current solution and accepting changes that make the objective function value better (cutting down the total cost in our case) and meet the limits (nutritional needs).

The hill climbing algorithm for our problem is implemented as follows:

1. Initialization: Generate an initial solution with random amounts (number of servings) for each food item. Calculate the total cost of this initial solution.[5]

2. Iteration: Perform a specified number of iterations for this case we used 50,000 iterations to explore the solution space. In each iteration, follow these steps: a. Slightly modify the current solution by adding or subtracting a small step 0.1 servings to a randomly selected food item. b. Calculate the total cost and nutrients of the modified solution. c. Check if the modified solution meets the nutritional requirements. d. If the

modified solution has a lower cost and meets the requirements, update the current solution with the modified solution.[5]

3. Termination: After completing the specified number of iterations, the algorithm returns the final solution and its total cost. This final solution represents the optimal or near-optimal meal plan that minimizes the total cost while meeting the nutritional requirements.[5]

The hill climbing algorithm is well-suited for this problem because it can efficiently explore the solution space and handle the constraints. However, it is essential to note that the algorithm may get stuck in local optima, resulting in suboptimal solutions. To mitigate this issue, we can perform multiple runs of the algorithm with different initial solutions and choose the best solution among them.

V Results:

The results of weekly groceries obtained from our implementation of the hill climbing algorithm are as follows:

- Meets daily nutritional requirements.
- Minimizes cost.
- Provides varying results upon each execution, landing close to the global optima.

Example meal plan (Total cost: £11.70):

> Apple: 2 servings (£1.00), Chicken: 2 servings (£3.00), Rice: 3 servings (£0.60), Broccoli: 2 servings (£1.80), Milk: 2 servings (£1.80), Bread: 2 servings (£0.40), Egg: 3 servings (£0.90), Spinach: 1 serving (£0.50), Yogurt: 1 serving (£0.80), Beans: 3 servings (£0.90),

Nutrients:

> Protein: 143.0, Fat: 44.1, Carbs: 301.1, Vitamin C: 215.7

While the hill climbing algorithm doesn't guarantee the absolute best solution, we can trust that our generated meal plan is sufficiently close to optimal for our needs. The meal plan satisfies daily nutritional requirements while keeping costs low, making it practical and efficient.

Additionally, since dietary variety is recommended, finding the exact global optimal solution may not be critical for our case.

By running the algorithm multiple times with varying starting points and step sizes, we can explore a wider range of possible solutions. This improves our likelihood of discovering an optimal or near-optimal solution. Ultimately, our meal plan is likely to be adequate due to its practicality, efficiency, and the algorithm's exploration strategy.

VI Conclusion:

This project aimed to create an optimised meal plan that satisfies daily nutritional requirements while minimizing the overall cost of groceries. The hill climbing algorithm was employed to explore the solution space and derive an optimal meal plan that meets the specified criteria. The final meal plan generated is cost-effective and nutritionally balanced, demonstrating the algorithm's capability to provide practical guidance for individuals seeking to optimise their grocery shopping and meal planning.

Strengths:

1. Simplicity: Easy-to-understand code and modular structure.
2. Optimization approach: Hill-climbing algorithm for cost minimization.
3. Scalability: Extendable for more food items, nutrients, or requirements.
4. Adaptability: Can be easily modified to incorporate new features.

Weaknesses:

1. Incorporate personalization: Consider food preferences and dietary restrictions.
2. Introduce meal structure: Differentiate between breakfast, lunch, and dinner for a more practical plan.
3. Recipe-based planning: Include recipes and their ingredients for enjoyable and complete meals.
4. Expand nutrient considerations: Add more nutrients, vitamins, and minerals for a balanced and healthy diet.
5. Update cost data: Use accurate, up-to-date pricing information to improve cost optimization.

Despite these limitations, the project successfully demonstrates the potential of using the hill climbing algorithm to optimise meal planning and grocery shopping. By refining the model to incorporate more accurate data and addressing the limitations mentioned, the model's function can be further improved, Additionally, the model can be refined to consider more food items, nutrients, and dietary preferences or restrictions, making it more applicable to real-world meal planning scenarios.

VII References

[-] A. Hoyle, "Optimisation - Lecture Sessions," lectures and practical in Optimisation, University of Stirling, Stirling, United Kingdom, Jan. - Apr. 2023.

[1] "Healthy Meal Planning Tips," National Institute on Aging, National Institutes of Health. [Online]. Available: https://www.nia.nih.gov/health/healthy-meal-planning-tips-older-adults. [Accessed: Apr. 4, 2023].

[2] "Health benefits of eating well," NHS Inform Scotland. [Online]. Available: https://www.nhsinform.scot/healthy-living/food-and-nutrition/eating-well/health-benefits-of-eating-well. [Accessed: Apr. 18, 2023].

[3] S. S. Rao, Optimization Techniques for Solving Complex Problems. 2nd ed. Hoboken, NJ, USA: Wiley, 2009. [Online]. Available: https://www.wiley.com/en-gb/Optimization+Techniques+for+Solving+Complex+Problems-p-9780470293324. [Accessed: Apr. 12, 2023].

[4] U.S. Department of Agriculture, Agricultural Research Service, "FoodData Central," 2021. [Online]. Available: https://fdc.nal.usda.gov/. [Accessed: 13- Apr- 2023].

[5] N. Sharma, "How to Implement the Hill Climbing Algorithm in Python," Towards Data Science, Oct. 26, 2019. [Online]. Available: https://towardsdatascience.com/how-to-implement-the-hill-climbing-algorithm-in-python-1c65c29469de. [Accessed: Apr. 13, 2023].

VIII Appendix: Python Code and Explanation

The Python code implementing the hill climbing algorithm, along with detailed explanations, can be found in a separate file. The code includes functions for generating an initial solution, calculating total cost and nutrients of a solution, checking if the solution meets daily nutritional requirements, and optimizing the meal plan using hill climbing algorithm. A main function is provided for running the optimization, calculating nutrients of optimal solution, and printing results in a user-friendly manner.

The simulated dataset used in code consists nutrient values and costs for various food items, such as apples, chicken, rice, broccoli, milk. Daily nutritional requirements are predefined in code. Hill climbing algorithm is applied to this dataset for finding an optimal meal plan that satisfies daily nutritional requirements while minimizing total cost of groceries.