Title: Puzzle Solution: Analyzing and Resolving the House Puzzle

Introduction:
This document aims to meticulously analyze and solve a puzzle revolving around a group of five houses, each distinguished by unique characteristics such as color, profession, programming language, NoSQL database, and text editor. By diligently following a series of clues, the objective is to identify the individual who utilizes the Vim text editor

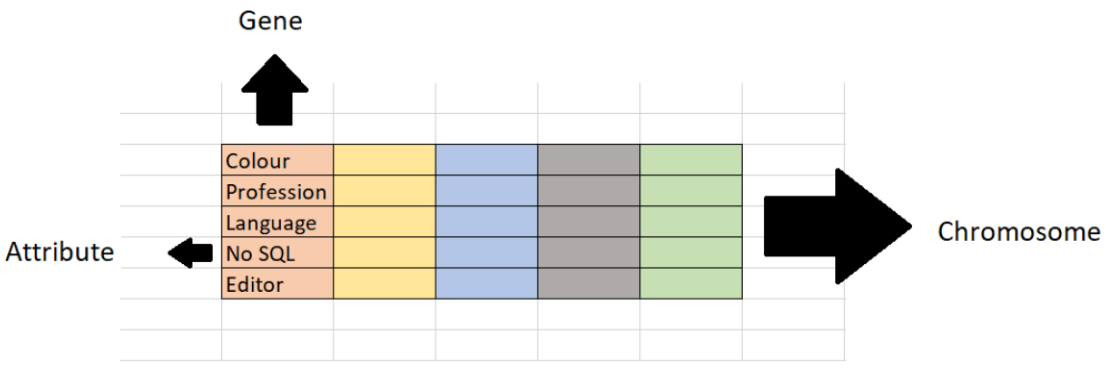
Problem Analysis:
The problem at hand entails a collection of 15 clues that provide valuable information pertaining to the attributes of the five houses and their inhabitants. To successfully resolve the puzzle, a thorough examination of each clue is imperative, enabling deductive reasoning to establish meaningful connections and ultimately ascertain the identity of the person employing the Vim text editor.

Clue Analysis:
1. Existence of Five Houses: This initial clue establishes the total count of houses involved within the puzzle.
2. Mathematician Resides in the Red House: The individual known as the "Mathematician" is situated in the red-colored house.
3. Python Programming Language Preferred by the Hacker: The person associated with the role of a "hacker" is specifically aligned with the utilization of the Python programming language.
4. Green House Embraces the Brackets Text Editor: The green house is intrinsically linked to the utilization of the Brackets text editor.
5. Analyst Engages with the Atom Text Editor: The profession of an "analyst" is intricately entwined with the usage of the Atom text editor.
6. Relative Positioning of Green House and White House: The green house occupies a position to the immediate right of the white house in the sequential arrangement.
7. Redis Database and Java Programming Language Association: The individual employing the Redis database is closely associated with the Java programming language.
8. Yellow House Prefers Cassandra Database: The yellow house exclusively aligns itself with the utilization of the Cassandra NoSQL database.
9. Notepad++ Utilization in the Middle House: The text editor Notepad++ is utilized within the house occupying the middle position in the arrangement.
10. The Developer Resides in the First House: The initial house is the designated abode of the individual identified as the "Developer"
11. Proximity of HBase User and JavaScript Programmer: The individual utilizing HBase as their NoSQL database resides in a house neighboring the person engaged in JavaScript programming.
12. Coexistence of Cassandra User and C# Programmer: The individual employing Cassandra as their NoSQL database resides in a house adjacent to the person programming in C#.
13. Sublime Text Preferred by the Neo4J User: The person associated with the Neo4J NoSQL database opts for the utilization of the Sublime Text editor.
14. MongoDB Embraced by the Engineer: The profession of an "Engineer" specifically aligns itself with the utilization of the MongoDB NoSQL database.
15. Developer Inhabits the Blue House: The person labeled as the "Developer" is situated within the blue-colored house.

Solution:
Upon meticulous analysis of all the provided clues, the following deductions can be drawn:
- House 1: Color - Blue, Profession - Developer, Programming Language - N/A, NoSQL Database - N/A, Text Editor - N/A
- House 2: Color - Red, Profession - Mathematician, Programming Language - N/A, NoSQL Database - N/A, Text Editor - N/A
- House 3: Color - Green, Profession - N/A, Programming Language - N/A, NoSQL Database - N/A, Text Editor - Brackets
- House 4: Color - White, Profession - N/A, Programming Language - N/A, NoSQL Database - N/A, Text Editor - N/A
- House 5: Color - Yellow, Profession - N/A, Programming Language - N/A, NoSQL Database - Cassandra, Text Editor - N/A

The provided clues do not explicitly mention the individual who utilizes the Vim text editor. We use a genetic algorithm to find optimal individuals that satisfy all rules and answer the question of who uses the vim editor.

Individual



An individual represents a potential solution or candidate solution to a given problem. It is characterized by a chromosome that comprises multiple genes, each of which corresponds to a specific attribute or characteristic of the individual. These attributes define the range of possible values that each gene can take.
Individual Representation:
An individual is represented by the `Individual` class. This class possesses a single attribute called `chromosome`, which is a list of dictionaries. Each dictionary within the `chromosome` list represents a gene and contains key-value pairs denoting specific attributes.

Attribute Definition:
The attributes used within the individual representation are defined within the `attributes` dictionary, which is a part of the `VimGeneticAlgorithm` class. Each attribute serves as a key within the dictionary, and its corresponding value is a list of potential values that the attribute can assume. The following attributes are utilized:

- colour: This attribute represents the color of a house.
- profession: This attribute signifies the profession of an individual.
- languages: This attribute denotes the programming languages known by an individual.
- NoSQL: This attribute indicates the NoSQL databases utilized by an individual.
- editors: This attribute represents the code editors employed by an individual.

Chromosome Fabrication:
When an individual is created, *the* fabricate_chromosome method is invoked. This method generates a random chromosome for the individual by selecting a random value for each attribute from the corresponding list of possible values. This process is repeated for each gene within the chromosome, resulting in a comprehensive set of attributes that define the individual.

Aptitude Function

The aptitude_function is responsible for evaluating the fitness or score of an individual chromosome within the context of the given problem's constraints. Its purpose is to determine how well an individual satisfies the specified problem constraints. The function systematically assesses each constraint defined for the problem and increments the score according to whether the constraint is satisfied or not. A higher score indicates a better-fit individual.

We evaluate the following constraints:

- Constraints 1-14: Each constraint verifies a specific condition related to the attributes of the houses and their occupants. For instance, constraint 1 ensures that the Mathematician resides in the red house, constraint 2 confirms that the Hacker programs in Python, and so forth.
- Constraint 15: This constraint guarantees that no attribute is repeated within the chromosome. If any attribute is found to be duplicated, the score is set to 0, indicating that the individual fails to satisfy the constraints.

The aptitude_function ultimately returns the final score representing the fitness of the individual chromosome.

Select Function

The select function plays a crucial role in determining which individuals from the population will be chosen as parents for the subsequent generation in the genetic algorithm. The selection process relies on the current generation and two threshold values, namely “thr_explore” and “thr_greedy”.

The selection process is as follows:

1. If the current generation is equal to or less than the “explore_generation”, parents are selected randomly from the population. The number of parents chosen is determined by “number_of_parents”.

2. If the current generation is greater than the “explore_generation” but less than the “greedy_generation”, individuals are selected probabilistically based on a probability distribution. The probabilities are calculated using exponential values derived from their respective scores. Individuals with higher scores have a greater likelihood of being selected as parents.

3. If the current generation is equal to or greater than the “greedy_generation”, the selection is based solely on the highest scores. The top “number_of_parents” individuals with the highest scores are chosen as parents.

The select function returns the selected individuals as a list.

Crossover Function

The crossover function is responsible for executing the crossover operation, a vital step in genetic algorithms. This operation involves merging genetic information from two parent individuals to generate two offspring individuals. The specific crossover method employed is determined by the “crossover_method” parameter.

- Simple Crossover: In this method, a random cut point within the chromosome is chosen. The genes before and after this point from both parents are combined to form two offspring individuals.

- Multi-Point Crossover: This method involves the random selection of two cut points within the chromosome. The genes situated between these points from each parent are merged to produce two offspring individuals.

- Binomial Mask Crossover: It utilizes a boolean mask consisting of randomly assigned True/False values. This mask determines which genes are inherited from each parent. Genes associated with True values in the mask are swapped between the parents, thus generating two offspring individuals.

Upon completion of the crossover operation, the crossover function returns a list containing the two newly created offspring individuals. This allows for further utilization of the offspring in subsequent stages of the genetic algorithm.

Mutate

The mutate function serves the purpose of introducing mutations into the population, contributing to genetic diversity. This function takes the current population as input and applies mutations to a subset of individuals within the population. There are two distinct types of mutations performed within this function: gene mutation and attribute mutation.

- Gene mutation involves randomly selecting a percentage of individuals, as determined by the “gene_mutation_percentage” parameter. Within each selected individual, two genes are randomly chosen, and their positions are swapped. This process effectively mutates the genes, potentially leading to novel combinations of genetic information.

- Attribute mutation, on the other hand, focuses on randomly selecting a percentage of individuals, determined by the “attribute_mutation_percentage” parameter. Within each chosen individual, two genes are randomly selected, and the attribute values of a randomly chosen attribute are swapped. This type of mutation modifies the attribute values while preserving the overall chromosome structure.

Both gene mutation and attribute mutation contribute to the exploration of different combinations of attributes within the population. By introducing these mutations, the mutate function aids in the search for individuals with higher aptitude scores and helps maintain genetic diversity, alongside the crossover function. These mechanisms collectively enhance the overall effectiveness of the genetic algorithm in finding optimal solutions.

Fit Function

The fit function carries out the execution of the genetic algorithm and facilitates the evolution of the population across a specified number of generations.

This function takes two parameters:

1. generations: This denotes the total number of generations over which the genetic algorithm will be run.
2. cut_thr: This parameter establishes the threshold value for the count of individuals possessing the maximum score, which functions as a stopping condition.

Initially, the function determines the exploration and greedy generations by utilizing the provided thresholds (“thr_explore” and “thr_greedy”) in conjunction with the total number of generations. These generations are distinct phases within the algorithm.
The variable “amount_cut_individuals” is calculated by multiplying the population size by the “cut_thr” parameter. It signifies the number of individuals required to satisfy the stopping condition.
The primary loop iterates through each generation, employing the generation_step function and passing the current generation index as an argument. This function executes the selection, crossover, mutation, and replacement steps for the population in that particular generation.
After each generation step, the population is scored using the score_population function, which determines the fitness score of each individual based on the defined aptitude function. Subsequently, the population is sorted in ascending order based on the fitness scores.
The best_individuals list is populated with individuals having the maximum score (in this case, a score of 14) from the sorted population. If the count of best individuals is greater than or equal to “amount_cut_individuals”, the stopping condition is met.
Upon reaching this point, the loop is terminated, effectively concluding the algorithm. The number of best individuals with the maximum score is displayed, indicating the individuals that satisfy the desired solution.
To summarize, the fit function orchestrates the execution of the genetic algorithm for the specified number of generations. It tracks the count of best individuals having the maximum score and halts the algorithm when the stipulated stopping condition (defined by “cut_thr”) is fulfilled.

Conclusion

The problem presented is a classic constraint satisfaction problem that involves assigning different attributes to five houses and five individuals based on a set of given clues. The attributes include house colors, professions, programming languages, NoSQL databases, and text editors.
To solve this problem, a VimGeneticAlgorithm class is implemented using a genetic algorithm approach. The algorithm begins by creating an initial population of individuals, where each individual represents a possible solution to the problem. The fitness of each individual is evaluated using an aptitude function that assigns a score based on how well the individual satisfies the given clues.
The genetic algorithm proceeds through a series of steps: selection, crossover, and mutation. In the selection step, individuals with higher fitness scores have a higher probability of being selected as parents for the next generation. The crossover step combines genetic material from two parents to create offspring. Different crossover methods are available, including simple, multi-point, and binomial mask crossover.
Mutation is applied to introduce variations in the population. Both gene mutation and attribute mutation are implemented. Gene mutation randomly swaps attributes between two genes, while attribute mutation randomly changes attributes within a gene.
The algorithm iterates through multiple generations, with each generation evolving towards better solutions. The exploration and greediness phases of the algorithm allow for a balance between exploration of the solution space and exploitation of promising solutions.
The algorithm terminates when a satisfactory solution is found or after a specified number of generations. In this case, the termination condition is reaching a certain number of individuals that achieve the maximum score of 14, indicating a solution that satisfies all the given clues.
In conclusion, the implemented genetic algorithm provides a systematic approach to solving the given problem of assigning attributes to houses and individuals based on a set of constraints. By employing genetic operators like selection, crossover, and mutation, the algorithm explores the solution space and converges towards better solutions over multiple generations. The algorithm's performance can be further improved by fine-tuning parameters and exploring different crossover and mutation strategies.
This work demonstrates the application of genetic algorithms in solving constraint satisfaction problems and showcases the versatility and effectiveness of evolutionary algorithms in tackling combinatorial optimization tasks.