Bfs

Certainly! The code you've provided implements the Breadth-First Search (BFS) algorithm using Python. Below is the algorithm in pseudocode:

```
1. Initialize an empty list called `visited` to keep track of visited nodes.
2. Initialize an empty list called `queue` to act as the BFS queue.
3. Define the `bfs` function that takes `visited`, `graph`, and a starting `node` as parameters.
4. Append the starting node to `visited` and enqueue it into the `queue`.
5. While the `queue` is not empty:
   - Dequeue a node `m` from the `queue`.
   - Print `m` (this represents visiting the node).
   - For each neighbor `neighbour` of `m` in the graph:
     - If `neighbour` is not in `visited`, add it to `visited` and enqueue it into the `queue`.
6. End of the `bfs` function.
```

The code follows this algorithm by using a graph representation (dictionary) and implementing the BFS algorithm to traverse the graph. Here's a breakdown of the code:

- `graph`: Represents the adjacency list representation of the graph where each node is a key, and its neighbors are stored as values in a list.
- `visited`: Keeps track of visited nodes during traversal to avoid revisiting nodes.
- `queue`: Acts as a FIFO queue to process nodes in a BFS manner.

The `bfs` function performs BFS traversal starting from a specified node. It visits each node and its neighbors, ensuring that each level of nodes is explored before moving to the next level.

Here's a step-by-step execution of the code with the given graph:

1. Start BFS from node '7'.

2. Visit '7' and enqueue '7'.

3. Dequeue '7', print '7', and enqueue its neighbors '8'.

4. Dequeue '8', print '8', and enqueue its neighbor (which is empty).

5. BFS finishes since there are no more nodes in the queue.

The printed output will be: "following is the Breadth First Search : 7 8"

DFS

The code you provided implements the Iterative Deepening Depth-First Search (IDDFS) algorithm. Here's a simplified version of the algorithm:

1. **Initialization**:

   - Initialize a global variable `result` to store the traversal path.

   - Define two dictionaries `dict_hn` and `dict_gn` to represent the heuristic values and the graph.

   - Define two functions: `DLS` (Depth-Limited Search) and `IDDFS` (Iterative Deepening Depth-First Search).

2. **Depth-Limited Search (DLS)**:

   - Takes parameters: `city` (current city), `visitedstack` (list of visited cities), `startlimit` (starting depth), `endlimit` (maximum depth).

   - Appends the current city to `result` and `visitedstack`.

   - Checks if the current city is the goal city, returns 1 if found.

   - Checks if the depth limit has been reached, returns 0 if limit reached.

   - Recursively calls DLS for each neighbor of the current city not in `visitedstack`.

   - Returns 1 if the goal city is found in any of the recursive calls, else returns 0.

3. **Iterative Deepening Depth-First Search (IDDFS)**:

   - Takes parameters: `city` (starting city), `visitedstack` (list of visited cities), `endlimit` (maximum depth limit).

- Iterates through depth limits from 0 to `endlimit`.

- Calls `DLS` with increasing depth limits until the goal city is found or depth limit is reached.

- Prints "Found" if the goal city is found, else "Not Found".

- Prints the traversal path in `result`.

4. **Execution**:

- Initializes `start` and `goal` cities, and calls `IDDFS` with appropriate parameters.

- Prints the final traversal path from `start` to `goal`.

The algorithm uses Depth-First Search (DFS) with a depth limit (`DLS`) and gradually increases the depth limit until the goal city is found (IDDFS). It is an efficient strategy for traversing large graphs with limited memory usage.

The goal of the algorithm is to find a path from the start city to the goal city in the given graph, using a depth-limited depth-first search strategy.

A*

The code you provided implements the A* search algorithm using a priority queue to find the shortest path from a start city to a goal city in a graph. Here's a simplified version of the algorithm:

1. **Initialization**:

- Define two dictionaries: `dict_hn` (heuristic values) and `dict_gn` (graph representation).

- Define the `get_fn` function to calculate the evaluation function value (`fn`) for a given path.

- Initialize `start` and `goal` cities.

- Define the `expand` function to expand nodes in the search using a priority queue.

- Define the `main` function to set up the initial priority queue and start the search.

2. **Evaluation Function (`get_fn`)**:

   - Takes a string representing a path (`citystr`) and calculates `fn` using heuristic (`hn`) and path cost (`gn`).

   - Splits `citystr` to extract cities and calculates `gn` as the sum of edge costs in the path.

   - Calculates `hn` using the heuristic value from `dict_hn`.

   - Returns the sum of `hn` and `gn` as the evaluation function value.

3. **Node Expansion (`expand`)**:

   - Takes a priority queue (`cityq`) containing nodes to be expanded.

   - Dequeues nodes from the priority queue and checks if the goal city is reached.

   - If the goal city is reached, sets the result path (`result`) and stops the search.

   - Otherwise, expands the current node by adding its neighbors to the priority queue with updated evaluation function values.

4. **Main Algorithm (`main`)**:

   - Creates a priority queue (`cityq`) with the initial node (start city) and its evaluation function value.

   - Calls the `expand` function to perform the A* search using the priority queue.

   - Prints the result path (`result`) if a path from the start city to the goal city is found.

The A* search algorithm combines the path cost (`gn`) with the heuristic estimate (`hn`) to guide the search towards the goal efficiently. It uses a priority queue to prioritize nodes with lower evaluation function values, ensuring that the shortest path is found first.

Best FS

The code you provided implements the Breadth-First Search (BFS) algorithm to find the shortest path between a starting city and a destination city in a graph. Here's a simplified version of the algorithm:

1. **Initialization**:

   - Initialize dictionaries `visited`, `distance`, and `parent` to keep track of visited nodes, distances from the starting node, and parent nodes for each city, respectively.

   - Initialize a queue using `Queue` from Python's built-in `queue` module.

- Initialize starting city, destination city, and set initial values for `visited`, `distance`, and `parent`.

2. **BFS Algorithm**:

   - Mark the starting city as visited, set its distance as 0, and enqueue it.
   - While the queue is not empty:
     - Dequeue a city `u`.
     - Explore its adjacent cities (`v`).
     - If an adjacent city `v` is not visited:
       - Mark it as visited.
       - Set its parent as `u`.
       - Update its distance as distance of `u` + 1.
       - Enqueue `v`.
   - After BFS completes, reconstruct the path from the destination city back to the starting city using the `parent` dictionary.

3. **Execution**:

   - Call the `bfs` function with the starting city ('Arad') and the destination city ('Bucharest').
   - The function prints the shortest path from 'Arad' to 'Bucharest'.

The BFS algorithm efficiently explores all reachable nodes from the starting city in a breadth-first manner, ensuring that the shortest path is found first. It uses a queue data structure to manage the traversal order and keeps track of visited nodes to avoid revisiting them.

Genetic Algorithm

Sure, here's a short summary of the Genetic Algorithm implemented in your code:

1. **Initialization**:

   - Set up parameters such as the number of genes, chromosomes, mating pool size, population size, generation count, lower bound (lb), and upper bound (ub) for random gene values.

  - Initialize the population randomly within the specified bounds.

2. **Main Loop**:

  - For each generation, calculate the fitness of each individual in the population based on some fitness function (in this case, the sum of squares of genes).

  - Select parents from the population based on fitness (using a matting pool selection approach where the fittest individuals have a higher chance of being selected).

  - Create offspring through crossover and mutation operations. Offspring inherit genes from parents and undergo random mutation in some genes.

  - Replace part of the population with the new parents and offspring.

  - Repeat the loop for the specified number of generations.

3. **Fitness Evaluation**:

  - After the specified number of generations, calculate the fitness of the final population.

  - Determine the fittest individual (chromosome) and its fitness value.

  - Print the best individual and its fitness.

4. **Key Components**:

  - **Initialization**: Randomly initialize the population.

  - **Fitness Calculation**: Calculate fitness based on a predefined function.

  - **Parent Selection**: Select parents based on fitness.

  - **Crossover**: Create offspring through crossover at a random point.

  - **Mutation**: Implement random mutation in offspring.

  - **Population Update**: Update the population with new parents and offspring.

5. **Algorithm Flow**:

  - Initialize population.

  - For each generation:

    - Calculate fitness.

    - Select parents.

    - Create offspring through crossover and mutation.

    - Update the population with parents and offspring.

  - Calculate final fitness and determine the best individual.

Overall, the Genetic Algorithm aims to evolve a population of individuals (chromosomes) over generations, selecting the fittest ones for reproduction (crossover and mutation) to potentially improve the overall fitness of the population.

Perceptron

This code implements a simple Perceptron classifier from scratch and compares its performance with Scikit-learn's Perceptron on the Iris dataset. Here's a short summary of the algorithm:

1. **Initialization**:

  - Define a Perceptron class with methods for activation, fitting (training), and prediction.

  - Initialize the learning rate and number of epochs in the Perceptron class constructor.

2. **Activation Function**:

  - Use the Heaviside step function as the activation function.

3. **Training (fitting)**:

  - Initialize weights and bias.

  - Iterate through epochs and training instances.

  - Compute the weighted sum of inputs plus bias (z) and apply the activation function.

  - Update weights and bias based on the error between predicted and actual outputs.

4. **Prediction**:

  - Use the trained weights and bias to predict output for new data.

5. **Evaluation**:

- Load the Iris dataset and prepare it for binary classification (class 0 vs. other classes).
  - Split the dataset into training and testing sets.
  - Train the custom Perceptron and Scikit-learn's Perceptron.
  - Make predictions on the test set and calculate accuracy scores.
  - Print the accuracy scores and classification reports.

6. **Key Components**:
   - **Perceptron Class**: Includes methods for activation, fitting, and prediction.
   - **Heaviside Activation**: Used as the activation function.
   - **Training Loop**: Updates weights and bias through epochs.
   - **Prediction**: Uses trained parameters to predict outputs.
   - **Evaluation Metrics**: Calculates accuracy and prints classification reports.

7. **Algorithm Flow**:
   - Initialize Perceptron with learning rate and epochs.
   - Load and prepare dataset.
   - Train custom Perceptron and Scikit-learn's Perceptron.
   - Make predictions and evaluate accuracy.
   - Print results and comparison.

Overall, the code demonstrates the implementation and usage of a basic Perceptron classifier for binary classification tasks and compares its performance with the Perceptron class provided by Scikit-learn.

Tipping easy

Sure, here's a short algorithmic overview of the Fuzzy Logic System for Tipping implemented in the code:

1. **Initialization**:
   - Import necessary libraries: `numpy`, `skfuzzy`, and `ctrl` from `skfuzzy`.

   - Define Antecedent and Consequent objects (`quality`, `service`, `tip`) to represent input and output variables with their respective universe and membership functions.

2. **Membership Functions**:

   - Use `automf` to automatically generate membership functions for `quality` and `service`.

   - Create custom membership functions for `tip` using `fuzz.trimf`.

3. **Rule Definition**:

   - Define fuzzy rules using `ctrl.Rule` to map input variables to output variable memberships.

   - Example rules: "If quality is poor OR service is poor, then tip is low."

4. **Control System Setup**:

   - Define a control system (`tipping_ctrl`) comprising the defined rules.

5. **Control System Simulation**:

   - Create a control system simulation object (`tipping`) based on the control system.

   - Set input values for `quality` and `service`.

   - Compute the output (`tip`) using `tipping.compute()`.

6. **Output Interpretation**:

   - Print the computed tip value (`tipping.output['tip']`).

   - Visualize the output membership function using `tip.view(sim=tipping)`.

7. **Algorithm Flow**:

   - Initialize universe variables (`quality`, `service`, `tip`) with membership functions.

   - Define fuzzy rules mapping input variables to output variable memberships.

   - Create a control system and simulation based on the rules.

   - Set input values and compute the output tip.

   - Print the tip value and visualize the output membership function.

This code snippet demonstrates the use of fuzzy logic to determine tipping amounts based on input variables such as service quality and food quality.

Tiping hard

Here's a short algorithmic overview of the Fuzzy Logic System visualization implemented in the code:

1. **Universe Variables and Membership Functions**:

   - Define universe variables for quality, service, and tip with appropriate ranges.

   - Generate fuzzy membership functions (e.g., `trimf`) for each variable.

2. **Visualization of Membership Functions**:

   - Create subplots to visualize membership functions for quality, service, and tip.

   - Plot the membership functions for different linguistic terms (e.g., low, medium, high) on each subplot.

3. **Fuzzy Logic Activation**:

   - Use `fuzz.interp_membership` to compute the degree of membership for input values.

   - Apply fuzzy rules (e.g., Rule 1: bad food OR service) and determine the degree of activation for each rule.

4. **Visualize Fuzzy Logic Activation**:

   - Create subplots to visualize the activation of output membership functions based on fuzzy rules.

   - Fill between curves to show the degree of activation for each linguistic term (e.g., low, medium, high) in the output.

5. **Aggregation and Defuzzification**:

   - Aggregate the activated output membership functions using `np.fmax`.

   - Defuzzify the aggregated result to obtain a crisp output value (tip amount).

6. **Visualize Aggregated Result**:

   - Plot the aggregated membership functions and the defuzzified result (centroid) on a subplot.

The code demonstrates the step-by-step process of fuzzy logic visualization, including defining membership functions, activating rules, aggregating results, and obtaining a crisp output value for tip amount.

Naïve bayes

Here's an algorithmic overview of the Gaussian Naive Bayes classifier implemented in your code:

1. **Data Preprocessing**:

   - Load data using pandas from a CSV file.

   - Encode class labels into numeric values using `encode_class` function.

   - Convert attribute values to float.

2. **Data Splitting**:

   - Split the data into training and testing sets using the `splitting` function.

3. **Model Training**:

   - Calculate mean and standard deviation for each attribute of each class using `MeanAndStdDevForClass` function.

4. **Model Testing**:

   - For each test instance, calculate the probabilities for each class using Gaussian probability and `calculateClassProbabilities` function.

   - Predict the class label for each test instance using `predict` function.

   - Get predictions for the entire test set using `getPredictions` function.

5. **Performance Evaluation**:

   - Calculate the accuracy rate of the model using the `accuracy_rate` function by comparing predicted labels with actual labels in the test set.

6. **Output**:

   - Print the total number of examples, training examples, and test examples.

   - Print the accuracy of the model on the test set.

The code demonstrates the complete workflow of training and testing a Gaussian Naive Bayes classifier on diabetes data, including data preprocessing, model training, testing, and performance evaluation.