**Introduction**

This project implements various search algorithms to traverse and search for files within a directory structure. It supports multiple search techniques, including **Breadth-First Search (BFS), Depth-First Search (DFS), Iterative Deepening DFS (IDDFS), Hill Climbing, Genetic Algorithm, and Min-Max Search with Alpha-Beta Pruning**. The goal is to efficiently locate a target file by applying different strategies.

**1. Importing Required Libraries**

The script utilizes the following Python libraries:

* **os**: Handles file and directory traversal.
* **random**: Used for genetic algorithm selection and mutation.
* **matplotlib**.pyplot: Visualizes the search process.
* **collections**.deque: Implements queues for BFS and IDDFS.
* **math**: Provides mathematical functions.
* **networkx**: Used for graph visualization of the file tree.

**2. FileNode Class**

This class represents a node in the directory tree.

* **Attributes:**
  + name: Name of the file or folder.
  + path: Full path to the file or folder.
  + is\_dir: Boolean flag indicating if the node is a directory.
  + children: List of child nodes (subdirectories or files).

**3. Building the File Tree**

**Function: build\_file\_tree(directory)**

This function constructs a tree representation of the directory structure using a queue (BFS traversal).

* **Steps:**
  1. Initialize the root node with the given directory.
  2. Traverse through the directory structure using **os.scandir().**
  3. Add subdirectories and files as children to the parent node.
  4. Skip directories with **permission** errors.

**4. Search Algorithms**

**4.1 Breadth-First Search (BFS)**

**Function: bfs\_search(root, target)**

* Uses a queue to explore each level of directories before moving deeper.
* Visualizes the search process using matplotlib.
* Returns the path of the target file if found, else returns None.

**4.2 Depth-First Search (DFS)**

**Function: dfs\_search(root, target)**

* Uses a stack to explore the deepest files first.
* Ensures correct DFS order by reversing the children before adding them to the stack.
* Returns the path of the target file if found, else returns None.

**4.3 Iterative Deepening Depth-First Search (IDDFS)**

**Function: iddfs\_search(root, target, max\_depth)**

* Performs multiple rounds of DFS up to increasing depth limits.
* Uses **dls()** as a helper function for depth-limited search.
* Returns the path of the target file if found, else returns None.

**4.4 Hill Climbing Search**

**Function: hill\_climbing(root, target)**

* A **heuristic-based search algorithm** that always selects the best immediate option.
* Sorts child nodes alphabetically to determine the next search path.
* *May get stuck at local maxima (i.e., a dead-end directory).*

**4.5 Genetic Algorithm Search**

**Function: genetic\_algorithm(root, target, generations=5)**

* Simulates natural selection to find the target file.
* Uses selection, crossover, and mutation techniques:
  + Selects top matching nodes.
  + Combines features of two parents for crossover.
  + Introduces randomness to prevent stagnation.
* Returns the path of the target file if found, else returns None.

**4.6 Min-Max Search with Alpha-Beta Pruning**

**Function: minmax\_search(node, target, depth, maximizing\_player, alpha, beta)**

* A decision-making algorithm using game-theory principles.
* Evaluates nodes at a fixed depth to determine the best path.
* Uses pruning to optimize performance by avoiding unnecessary calculations.

**5. Tree Visualization**

**Function: visualize\_tree(root)**

* Uses **networkx** to draw the directory structure as a graph.
* Each node represents a file or folder.
* Edges represent parent-child relationships.

**6. Printing the File Tree Structure**

**Function: print\_file\_tree(node, indent=0)**

* Recursively prints the directory structure in a human-readable format.
* Uses icons:
  + 📂 for directories.
  + 📄 for files.

**7. Main Function Execution**

**Function: main()**

* Calls build\_file\_tree() to construct the directory tree.
* Prints the file structure using print\_file\_tree().
* Prompts the user for a file search query.
* Executes all search algorithms sequentially and prints their results.
* Visualizes the directory structure and search process.

**8. Execution**

* The script runs when executed directly:

*if \_\_name\_\_ == "\_\_main\_\_":*

*main()*

**9. Edge Cases Considered**

* **Non-existent directories:** The script checks if the given directory exists before attempting traversal.
* **Permission errors:** Catches exceptions when accessing restricted directories.
* **Empty directories:** Ensures the script doesn't break if a directory has no files.
* **Cyclic symbolic links:** Avoids infinite loops caused by symbolic link cycles.

**10. Conclusion**

This project explores various search algorithms for file retrieval. Each method has its own strengths and weaknesses:

* **BFS:** Best for level-wise exploration.
* **DFS:** Efficient but may go too deep before finding a result.
* **IDDFS:** Balances between BFS and DFS.
* **Hill Climbing:** Fast but may get stuck at local optima.
* **Genetic Algorithm:** Introduces randomness but may not always find the target.
* **Min-Max Search:** Uses decision-making principles to optimize file search.

This script serves as a valuable tool for understanding different search techniques and their applications in file system traversal.

Please refer to this github page, if you need exact method to start using this code.

[*https://github.com/Zeemi-Bhai/FileSearch\_using\_AI\_concepts*](https://github.com/Zeemi-Bhai/FileSearch_using_AI_concepts)