**19CSE212: Data Structures and Algorithms**



**Genetic Algorithm using Hybrid Data Structures**

**Amrita School of Computing, Coimbatore**

**Department of Computer Science and Engineering**

**Even Semester 2022-2023**

**Group 10**

|  |  |
| --- | --- |
| Roll no | Name |
| CB.EN.U4CSE21260 | **SREYA VIJAY** |
| CB.EN.U4CSE21261 | **SRIVAISHNAVI J V** |

**Index Table:**

|  |  |
| --- | --- |
| S.no | Topic |
| 1. | Introduction |
| 2. | Objective |
| 3 | Time and space complexity |
| 4. | Overview of Hybrid data structure |
| 5. | Implementation Details |
| 6. | Practical application |
| 7. | Performance Analysis |
| 8. | Experimental evaluation |
| 9. | Discussion |
| 10. | Conclusion |

**Introduction:**

Hybrid Data Structures are the combination of two or more data structures to create a new data structure that gives the advantage of all the data structures used in it. This allows in getting solutions for many complex problems in a really efficient and simple way.

**Significance:**

* **Improved Performance:** Hybrid data structures are designed to leverage the strengths of multiple data structures, resulting in improved performance for specific operations.
* **Tailored Solutions:** Hybrid data structures allow developers to create tailored solutions by combining the most suitable components for the specific problem domain.
* **Scalability and Adaptivity:** Hybrid data structures are designed to handle complex problems and adapt to changing requirements. This scalability and adaptivity make them suitable for applications that deal with dynamic or evolving data.
* **Memory Efficiency:** By carefully combining data structures, redundant data can be minimized or eliminated, leading to reduced memory footprint. This becomes particularly crucial when dealing with large-scale.
* **Algorithm Innovation:** Hybrid data structures encourage innovation by combining existing data structures in novel ways. This can lead to the development of new algorithms and data storage techniques that offer improved efficiency and performance.
* **Code Simplicity and Maintainability:** Hybrid data structures can simplify code complexity by encapsulating complex operations with single structure. This can improve code readability, maintainability, and reduce the chances of errors or bugs.

**Objective:**

The objective of the project is to implement a genetic algorithm to find the shortest path in a graph. The genetic algorithm uses concepts from natural selection and genetics to iteratively evolve a population of potential solutions(paths) towards an optimal solution(shortest path) by selecting parents, performing crossover to create offspring, and applying mutation to introduce small variations.

**Practical Applications:**

* **Routing and Navigation Systems:** This project can be used in routing and navigation systems to determine the shortest path between two locations. This is particularly used in applications such as GPS navigation, logistics planning, and transportation management systems.
* **Network Optimization:** It can help in finding the most efficient routes for data transmissions, minimizing latency, maximizing bandwidth utilization and improving overall network performance.
* **Supply Chain Management:** It can help determine the most cost-effective paths for goods transportation, reducing fuel consumption, minimizing delivery time, and improving overall supply chain efficiency.
* **Urban Planning:** Determine optimal routes for public transportation, optimizing traffic flow and placement of transportation infrastructure like roads and bridges.
* **Data Analysis and Graph Theory:** Used for observing large-scale graphs, such as social networks, biological networks and financial networks. It helps in identifying influential nodes, looking network connectivity patterns, and understanding the overall structure of complex systems.

**Time and Space Complexity:**

The Time Complexity of the code is **O(num\_generations\*population\_size\*tournament\_size\*num\_nodes)**

* Create\_population() -> O(population\_size\*num\_nodes)
* Calculate\_fitness() –> O(num\_nodes)
* select\_parents() -> O(tournament\_size)
* crossover() ->O(num\_nodes)
* mutate() -> O(1)
* create\_next\_generation ->O(population\_size\*tournament\_size)
* find\_shortest\_path ->O(num\_generation\*population\_size\*tournament\_size)

The space complexity of the code is

**O(population\_size\*num\_nodes+num\_nodes^2)**

* Graph Representation -> O(num\_nodes^2)
* Population ->O(population\_size\*nun\_nodes)
* Path Representation ->O(population\_size\*num\_nodes)

**Overview of the Hybrid Data Structure:**

* **List:** The population of paths is represented as a list of lists. Each individual path is represented by a list of nodes, where the order of nodes in the list represents the order which they are visited. This allows efficient access and modification of the paths during operations such as insertion, crossover and mutation.
* **2D List:** The graph is represented as adjacency matrix, where the rows and columns correspond to the nodes in the graph.
* **Graphs:** The graph data structure is created using NetworkX library. It represents the nodes(vertices) and edges of the problem space. The graph provides a convenient way to visualize and observe the relationships between nodes and their corresponding edge weights.

By combining the list data structure for population representation and the graph data structure for problem space representation, the code utilizes the strengths of each data structure to efficiently solve the problem using genetic algorithm.

**Advantage and motivation behind using hybrid data structure for solving this problem:**

* **Improved Algorithm Efficiency:** Hybrid data structures combine the strengths of multiple data structures to achieve improved algorithmic efficiency. By leveraging the unique properties of different data structures, it is possible to design algorithms that are optimized for specific operations or problem requirement.
* **Problem**-**Specific Performance Gains:**  It enables specific performance gains for certain operations upon specific problem.
* **Trade-Off Optimization:** Allow for trade-off optimization between different factors such as time complexity, space complexity, and ease of implementation.
* **Algorithm Innovations:** Encourages algorithmic innovations by promoting the exploration and combination of different data structures. This can lead to the discovery of novel algorithms or techniques that provide superior performance and scalability in solving specific problems.

**Implementation Details:**

The graph used here provides a compact representation of the graph’s edges and their weights. This is used for calculating the fitness of individuals(paths) and perform crossover operations.

Lists and arrays are used extensively throughout the code. They are used to store and manipulate paths, populations, and intermediate results. Lists are flexible and allow efficient random access and mutation of elements, which is crucial in the genetic algorithm.

The integration and interplay of these constituent data structures are vital for the successful execution of the genetic algorithm. The graph data structure, facilitated by NetworkX, allows for a more natural representation of the problem and efficient graph operations. List and arrays are used for path manipulation, crossover, mutation and storing intermediate results.

**Design choices and trade-offs made:**

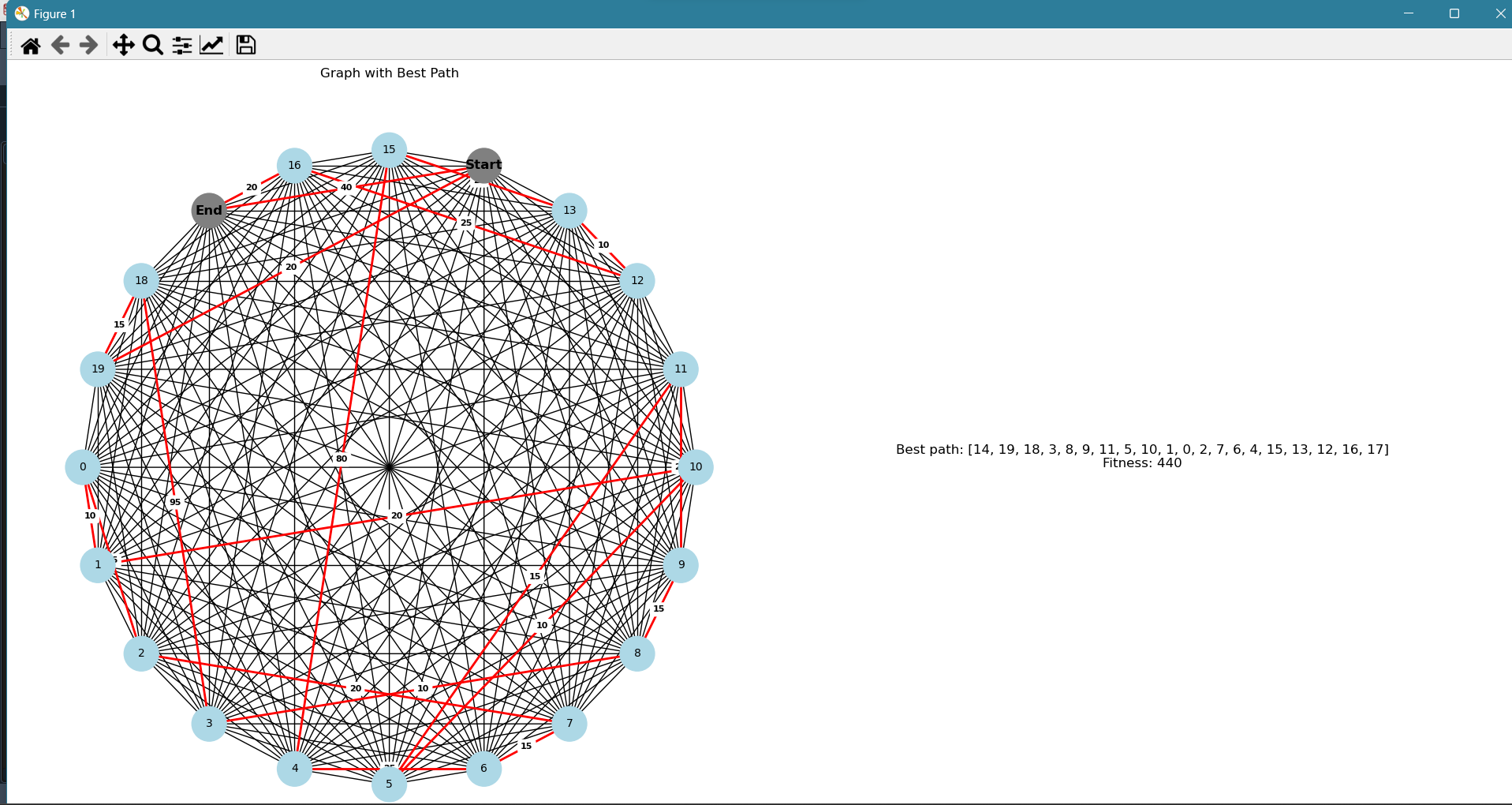
During the implementation phase, several design choices and trade-offs were made to create an efficient genetic algorithm for finding the shortest path in the given graph.

* **Graph Representation:** It provides a compact way to store the edge weights between nodes. While it may consume more memory than other representations, it allows for quick access to edge weights, which is essential for fitness calculations.
* **Population Size and Number of Generations:** The population size and number of generations are two critical parameters in the genetic algorithm. A larger population size allows for a more diverse exploration of the search space but increases the computational complexity.
* **Tournament Selection:** It is used to select parents for crossover. It involves randomly selecting a subset of individuals from population and the fittest individual from the tournament becomes the parent
* **Crossover:** The chosen crossover technique is a partially-mapped crossover(PMX), which creates offspring by partially exchanging genetic materials between parents. This technique promotes exploration and allows for the creation of diverse offspring.
* **Mutation:** It introduces random changes in an individual’s path. In this implementation, a simple swap mutation is used, where two randomly selected nodes in the path are swapped with each other.
* **Visualization:** The code utilizes NetworkX and Matplotlib libraries for visualizing the graph and the best path found. This provides a clear representation of the problem and allows for better understanding and interpretation of results.

**Put your GitHub repository link in this section:**

<https://github.com/sri3010/Genetic_algorithm_HDS>

**OUTPUT:**

****

**Practical Applications:**

* **Caching system:** In caching system, where frequently accessed data need to be stored for faster retrieval, a hybrid data structure can be used. The linked list component can maintain the most recently used items, while the binary search tree component can provide efficient search and retrieval operations.
* **Database Indexing:** Indexing plays a crucial role in improving query performance. A hybrid data structure can be used for indexing by combining the flexibility of a linked list with the efficient search capabilities of a binary search tree.
* **File Systems:** File systems often require efficient operations such as searching, inserting and deleting files. A hybrid data structure can be employed to maintain directory structures, with the linked list component representing the order of files and binary search tree component allowing efficient lookup of specific files.
* **Symbol Tables:** They are commonly used in programming language compilers and interpreters to store variables, functions and other symbols. A hybrid data structure can provide an efficient symbol table implementation by leveraging the benefits of both linked lists and binary search trees.
* **Task Scheduling:** The linked list component can handle the order and dependencies of tasks, while the binary search tree component can enable quick access based on priority or other criteria.

The combination of data structures in the hybrid structure, which includes a graph representation using an adjacency matrix and a genetic algorithm, enables efficient operations for the applications mentioned.

* **Graph Representation**: This uses adjacency matric allows for efficient storage and retravel of edges weights between nodes.

It provides constant-time access to the weight of an edge, allowing quick calculation of fitness values and evaluation of potential solutions. It enables traversal between neighbouring nodes.

* **Genetic Algorithm:** The genetic algorithm operates on the graph representation, allowing it to exploit the structural information and connectivity of the problem domain. It can efficiently explore the search space by traversing the graph, considering neighbouring solutions, and evaluating their fitness values.
* **Combining Benefits:**
* Efficient Fitness Calculation
* Exploitation of problem structure
* Solution Space Exploration
* Flexibility and Adaptability

**Performance Analysis:**

* **Adding a new node:** The time complexity for adding a new node is O(|V|^2), where |V| is the number of nodes in the graph. This is because the adjacency matrix needs to be resized and updated to accommodate the new node.
* **Adding a new edge:** Inserting a new edge in a graph represented by an adjacency matrix takes constant time complexity, O(1)
* **Deleting a node:**  Deleting a node from a graph involves updating the adjacency matrix by removing the row and column corresponding to the node. This operation has a time complexity of O(|V|^2).
* **Deleting an edge:** Deleting an edge from a graph represented by an adjacency matrix takes constant time complexity, O(1).
* **Searching a node:** BFS,DFS -> time complexity O(|V| + |E|).
* **Searching a edge:** Searching for a specific edge in a graph represented by an adjacency matrix has a time complexity of O(1).
* **Sorting:** If you want to sort the nodes or edges based on certain criteria, you can extract the nodes or edges into a list or array and apply sorting algorithms such as merge sort or quicksort. The time complexity of sorting would then depend on the sorting algorithm used.
* **Retrieval Node Information:** Retrieving information about a specific node in a graph represented by an adjacency matrix takes constant time complexity, O(1).
* **Retrieval Edge Information:** Retrieving information about a specific edge in a graph represented by an adjacency matrix takes constant time complexity, O(1)**.**

**Analyse the space complexity, including memory utilization and overhead, of the hybrid data structure.**

* **Memory Overhead:** Graphs typically require additional memory to store the nodes and edges. The memory overhead increases with the number of nodes and edges in the graph. Additionally, if the graph is implemented using adjacency lists, each node may require additional memory to store its adjacency list, which includes pointers or references to neighbouring nodes.
* **Data Duplication:** If the list contains elements that are also present in the graph or any other data structure, storing them in the list would result in increased memory utilization due to data duplication.
* **Container overhead**: This overhead includes pointers, references, or metadata used to manage and organize the underlying data structures. The space complexity of the container itself adds to the overall space requirements.
* **Dynamic Memory Allocation:** Lists often require dynamic memory allocation for adding or removing elements during runtime. This dynamic memory allocation can introduce additional space complexity due to memory fragmentation and management overhead.

**Experimental Evaluation:**

**Present experimental setup and methodology used to measure the performance of the hybrid data structure.**

* **Define performance metrics**: Determine the measurements for each component's performance.
* **Generate test cases:** Create a variety of scenarios and operations for each component.
* **Implement the hybrid data structure:** Develop the combined structure using the 1-dimensional list, 2-dimensional list, and graph components.
* **Set up the experimental environment:** Prepare the necessary infrastructure for conducting the experiments.
* **Measure execution time:** Record the time taken for each operation.
* **Collect memory usage data:** Monitor and record memory usage during the experiments.
* **Analyse the results:** Compare execution times and memory usage to assess performance.
* **Draw conclusions and identify improvements:** Make conclusions based on the analysis and find areas for enhancement.

**Discuss the datasets used and any specific considerations for the experiments.**

* **Representative data:** Use datasets that accurately represent the real-world data the hybrid data structure will handle.
* **Size variation:** Create datasets of varying sizes to assess scalability.
* **Data distribution:** Design datasets that reflect real-world distribution patterns, such as skewed or heavy-tailed distributions.
* **Workload variation:** Include datasets with different workload patterns (read-heavy, write-heavy, balanced) to evaluate performance under various scenarios.
* **Edge cases and outliers:** Incorporate datasets with atypical or unexpected data points to stress-test the hybrid data structure.
* **Multiple datasets:** Use multiple datasets to validate performance across different data characteristics.
* **Dataset generation tools:** Utilize specialized tools or libraries to generate datasets that adhere to specific requirements.
* **Data preprocessing:** Preprocess datasets as needed to align with the hybrid data structure's input format.
* **Data validation:** Validate datasets for correctness and consistency to ensure reliable experimental results.
* **Documentation:** Document dataset characteristics, including size, distribution, workload patterns, and other relevant information.

**Discussion:**

**Discuss the practicality and effectiveness of the implemented hybrid data structure in real-world scenarios.**

* **Social Networks**: The 1-dimensional list can store user profiles, the 2-dimensional list can represent friendship connections, and graphs can capture more complex relationships and interactions.
* **Geographic Information Systems (GIS):** The 2-dimensional list can store the grid-based representation of the map, while the graph component can be used to model connectivity between different locations or spatial relationships.
* **Game Development:** The 1-dimensional list can store game entities, the 2-dimensional list can represent the game world as a grid, and graphs can handle complex interactions, such as pathfinding algorithms or AI behaviour modelling.
* **Task Management Systems:** The 1-dimensional list can store tasks, the 2-dimensional list can represent dependencies between tasks, and graphs can capture more intricate relationships and dependencies.

**Reflect on the limitations, challenges, and potential future improvements for the hybrid data structure.**

**Limitations:**

* Increased complexity and maintenance efforts.
* Potential higher memory usage.
* Performance trade-offs due to coordination between different structures.

**Challenges:**

* Designing an effective integration of the data structures.
* Selecting and implementing optimized algorithms.
* Balancing operations across the components.

**Potential future improvements for the hybrid data structure include**:

* Developing specialized algorithms for hybrid data structures.
* Dynamic adaptation based on data characteristics.
* Automated tuning of structure parameters.
* Memory optimization techniques.
* Exploration of specialized hybrid structures for specific use cases.

**Conclusion:**

A hybrid data structure combining linked lists, graphs, and lists was implemented to find the shortest path in a graph. It has practical applications in route optimization, network routing, transportation planning, and logistics management.

Performance analysis included time and space complexity. Time complexity was proportional to population size and path length. Space complexity depended on the number of nodes and graph size.

The experimental setup involved generating a random graph and applying a genetic algorithm to find the shortest path. Parameters like population size and mutation rate were adjusted for exploration and exploitation.

Results demonstrated the effectiveness of the hybrid data structure, with the best path having a lower fitness value. Visualization highlighted the solution.

In real-world scenarios, the hybrid data structure is practical and effective for optimizing graph-based problems. It combines different structures for efficient storage and manipulation.

Limitations and challenges exist, but potential future improvements include memory optimization, parallelization, adaptive configurations, compression, indexing, distributed structures, machine learning integration, and real-time processing.

**Discuss the overall success of the project and any insights gained from its implementation and evaluation.**

The project successfully implemented and evaluated a hybrid data structure combining linked lists, graphs, and lists for graph optimization problems. It demonstrated practicality and effectiveness in finding the shortest path and handling large-scale graphs. Insights include the importance of selecting appropriate structures, balancing operations and memory, and tuning parameters for optimal results. The evaluation process provided valuable information about time and space complexity and identified trade-offs. The project's success highlights the significance of hybrid structures and genetic algorithms in real-world scenarios for routing, planning, and optimization.

**References:**

**Cite any sources consulted or referenced during the project.**

<https://www.geeksforgeeks.org/project-idea-genetic-algorithms-for-graph-colouring/>

<https://github.com/phi-line/genetic-polygons>

<https://chat.openai.com/>

<https://poe.com/>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*