**GROUP ASSIGNMENT (05)**

**GROUP MEMBERS:**

|  |  |
| --- | --- |
| NAME | STUDENT ID |
| Henry Amponsah | 11358466 |
| Emma Adobea Osei | 11325735 |
| Enoch Ankrah | 11261979 |
| Francis Musah | 11333369 |
| Lordy Adu Nkansah | 11012698 |
|  |  |
|  |  |

**REPORT**

**Algorithm Implemented: Prim’s Minimum Spanning Tree (MST)**

Prim’s Minimum Spanning Tree (MST) algorithm is a greedy algorithm that finds the minimum spanning tree for a weighted, undirected graph. The algorithm ensures that the resulting tree is of minimum total weight, connecting all vertices without forming any cycles. The application is designed using Object-Oriented Programming (OOP) principles in Java. The key classes include `Graph`, `Edge`, and `PrimMST`. The `Graph` class represents the graph with its vertices and edges, while the `Edge` class represents an edge with a source, destination, and weight. The `PrimMST` class contains the implementation of Prim’s algorithm. The algorithm starts from an arbitrary vertex, uses a priority queue to select the smallest edge, and builds the MST incrementally.

**Algorithm Implemented: Quicksort and Merge Sort**

**Quick Sort Algorithm:**

Quicksort is a fast-sorting algorithm that uses the divide-and-conquer technique to sort items. The method works by picking a 'pivot' element from the array and dividing the other items into two sub-arrays based on whether they are less than or greater than the pivot. The subarrays are then recursively sorted. Quicksort has an average time complexity of O (n log n). Quicksort has a worst-case time complexity of O(n2), This happens when the pivot is always the lowest or biggest member in the array, for example, when it is already sorted.

Quicksort is an in-place sorting algorithm with O (log n) extra space complexity owing to the recursion stack. Quicksort is not stable by default; however, reliable implementations do exist.

**Merge Sort Algorithm:**  
  
Merge Sort is another efficient, stable, comparison-based sorting algorithm that uses the divide-and-conquer paradigm. The algorithm divides the array into two parts, sorts them recursively, and then combines the sorted halves to form the sorted array. Merge Sort’s average case time complexity is O (n log n). However, Merge Sort has a constant O (n log n) time complexity. Merge Sort needs O(n) more space to store the temporary arrays created during the merge operation. Merge Sort is stable, which means it maintains the relative order of equal items. More dependable for larger datasets and those that demand steady sorting. It is ideal for external sorting (sorting enormous datasets that do not fit in memory) because of its constant speed and consistency.

**Algorithm Implemented: Quickhull**

QuickHull is an efficient algorithm for finding the convex hull of a set of points in a 2D plane. The algorithm uses a divide-and-conquer approach like QuickSort. It starts by identifying the extreme points (min and max x-coordinates) and recursively finding the hull points on either side of a dividing line. QuickHull’s average case time complexity is O(n log n). However, in the worst case, the time complexity can be O(n^2). QuickHull requires O(n) space to store the points in the convex hull. QuickHull is suitable for moderate-sized datasets and offers a balance between simplicity and performance.

**Algorithm Implemented: Travelling Salesman Problem**

The Nearest Neighbor Heuristic for the Travelling Salesman Problem (TSP) is a simple and quick method to find an approximate solution to the TSP. The algorithm starts at a random city and repeatedly visits the nearest unvisited city until all cities are visited. The Nearest Neighbor Heuristic’s time complexity is O(n^2). The space complexity is O(n), used to keep track of visited cities. While it does not guarantee the optimal solution, it often finds a tour within a factor of two of the optimal. This heuristic is useful for obtaining a quick, though not necessarily optimal, solution for moderate-sized TSP instances.

**Algorithm Implemented: Dijkstra's Algorithm**

Dijkstra's Algorithm is a well-known algorithm for finding the shortest paths from a source vertex to all other vertices in a weighted graph with non-negative weights. The algorithm uses a priority queue to efficiently select the vertex with the smallest known distance. Dijkstra's Algorithm’s time complexity is O(V^2) when using an adjacency matrix and O((V + E) log V) with a priority queue and adjacency list, where V is the number of vertices and E is the number of edges. The space complexity is O(V) for storing distances and visited vertices. Dijkstra's Algorithm is ideal for finding the shortest paths in dense graphs and networks with non-negative weights.

**Algorithm Implemented: Closest Pair of Points**

The Closest Pair of Points algorithm finds the closest pair of points in a set of points in a 2D plane using a divide-and-conquer approach. The algorithm recursively splits the points into smaller subsets, finds the closest pairs within each subset, and then combines the results. The Closest Pair of Points algorithm has an average and worst-case time complexity of O(n log n). The space complexity is O(n), required to store the points and intermediate results. This algorithm is highly efficient for large datasets and is particularly useful in applications requiring precise distance calculations, such as geographic information systems and computer graphics.

**Algorithm Implemented: Strassen's Matrix**

Strassen's Matrix Multiplication is an advanced algorithm for multiplying two matrices more quickly than the conventional O(n^3) approach. It recursively divides the matrices into smaller submatrices and combines the results using a clever combination of additions and multiplications. Strassen's Matrix Multiplication’s time complexity is O(n^2.81) for square matrices. The space complexity is O(n^2) to store the matrices and intermediate results. Strassen's algorithm is more efficient for large matrices but has higher constants and overhead, making it less practical for small matrices compared to standard matrix multiplication. It is particularly suitable for applications requiring the multiplication of large matrices, such as in scientific computing and large-scale simulations.

**Algorithm Implemented: Huffman Coding**

Huffman Coding is an optimal, lossless, variable-length prefix coding algorithm used for data compression. The algorithm builds a binary tree called the Huffman Tree, where each leaf node represents a character and its frequency. The tree is built by repeatedly merging the two nodes with the lowest frequencies until only one node remains. Huffman Coding’s average and worst-case time complexity for constructing the tree is O(n log n), where n is the number of unique characters. The space complexity is O(n) for storing the nodes and the encoded characters. Huffman Coding is ideal for reducing the size of data for transmission or storage, such as in text compression.

**Algorithm Implemented: Kruskal's Algorithm**

Kruskal's Algorithm is an efficient algorithm for finding the Minimum Spanning Tree (MST) of a connected, undirected graph. The algorithm sorts all the edges in non-decreasing order of their weights and then adds the shortest edge to the MST, ensuring no cycles are formed, until all vertices are included. Kruskal's Algorithm’s time complexity is O(E log E), where E is the number of edges, primarily due to the sorting step. The space complexity is O(V + E) to store the edges and subsets for union-find operations. Kruskal's Algorithm is particularly suitable for sparse graphs and is commonly used in network design and other applications requiring MSTs.