PROJECT REPORT

Graph Algorithms Analysis & Optimization for Bitcoin Transaction Network

Muhammad Shahab Alam , Syed Sabih Nasir

Introduction:

The ever-growing complexity of real-world networks, such as social interactions or financial transactions, has led to an increased demand for robust and efficient graph algorithms. In this project, we aim to analyze and optimize various graph algorithms applied to the Bitcoin transaction network, using the dataset available at Stanford Bitcoin Alpha Dataset. Our focus is on solving fundamental graph problems, including finding shortest paths, minimum spanning trees, cycle detection, and sorting, with the ultimate goal of gaining insights into the structure and dynamics of the Bitcoin transaction network.

Objectives:

1. **Normalization of Dataset and Graph Creation**
   - Normalizing the dataset to avoid computational errors when passed to the relevant algorithms.
   - Transformation of the dataset into a suitable graph representation.

2. **Single Source Shortest Path & Algorithm Analysis**
   - Using Dijkstra and Bellman Ford algorithms to find the shortest paths to all the nodes from a single source.
   - Analyzing and comparing the efficiency of these algorithms.

3. **Minimum Spanning Tree & Algorithm Analysis**
   - Finding the minimum spanning trees of the graph using Prims and Kruskal's algorithm.
   - Performing a time complexity analysis on the two algorithms and comparing the algorithms in accordance to the dataset.
4. **Cycle Detection**
   - Creating an algorithm for detecting cycles in a graph.

- Cons of Cycles in the graph and their effect on overall cost during computation.

5. **Sorting the dataset with respect to time**
    - Implementation of Merge, Quick and Heap sort on time (4th column of the dataset).
    - Time values are in EPOCH format so converting them to human readable format
    - Storing those new values in a Separate file.
    - Complexity Analysis for Sorting Algorithms.

Expected Outcomes:

- Comparative study of the efficiency and suitability of various graph algorithms
- Improved understanding of complexities of algorithms and their implications in the real world networks.

1. Normalization of Dataset and Graph Creation

The set used in this project denotes a weighted signed network for the people that use Bitcoin and the weights refer to how much a user can be trusted.As the bitcoin users are anonymous therefore there is a need to maintain a track of the user's reputation to avoid transactions with fraudulent users. The weights range from [-10-10], -10 as in total distrust and 10 being total trust.

Data format : `Source, Target, Rating, Time`
Where:

- Source: NODE ID of Source
- Target: NODE ID of Target
- Rating: Source's rating for the target
- Time: Time of rating

Nodes : 3783
Edges : 24186
Range of Rating: -10 to +10

Normalizing the Dataset:
The dataset contains negative values and those negative values can adversely affect when we calculate the single source shortest path using dijkstra's algorithm. As it has a limitation when dealing with graphs that contain negative edge weights. It is due to the algorithm's assumption that the sum of weights along a path is an increasing function. So to deal with the problem the ratings values have been scaled by a factor of 10 ie. 0 representing the total distrust and 20 being the total trust. So the information related to the normalized dataset is below.

Nodes : 3783
Edges : 24186
Range of Rating: 0 to 20.

Graph Creation:
So based on those new normalized values, the graph is created. All this is happening in the same code so basically there is a `Graph` class which has these methods(`addEdge`, `createGraphfromCSV`). So the code reads the graph data from the "`input.csv`" file, normalizes the ratings and writes the normalized data into "`normalized_output.csv`". Then it creates a graph using the graph class with edges represented by the adjacency matrix using the normalized data.

2. Single Source Shortest Path & Algorithm Analysis

Dijsktra's Algorithm

Introduction

This algorithm efficiently finds the shortest paths between a designated source node and all other nodes in a graph, where the term "shortest" refers to the minimum sum of edge weights along the path.

Pseudocode

```
Input: Graph represented by an adjacency matrix, source node

1. Initialize variables:
   - numNodes = number of nodes in the graph
   - distances = array of distances from the source to each node
(initialized to infinity)
   - visited = array indicating whether each node has been
visited (initialized to false)

2. Set the distance from the source to itself to be 0.
   distances[source] = 0

3. Repeat (numNodes) times:
   a. Find the unvisited node with the minimum distance.
      minNode = unvisited node with the smallest distance

   b. Mark the selected node as visited.
```

```
        visited[minNode] = true
```

c. Update the distances to neighboring unvisited nodes.

```
        for each unvisited neighbor of minNode:
            newDistance = distances[minNode] + weight of the edge
between minNode and neighbor
            distances[neighbor] = min(distances[neighbor],
newDistance)
```

Output: Shortest distances from the source node to all other nodes in the graph.

Advantages	Limitations
• Dijkstra's algorithm has a relatively straightforward implementation, making it accessible for programmers and engineers. The basic structure involves priority queue data structures or arrays for efficient selection of the next node.	• One major limitation is its reliance on non-negative edge weights. Dijkstra's algorithm may produce incorrect results when applied to graphs containing negative weights or in scenarios where negative weights are meaningful.

Time Complexity

V: Number of nodes
E: Number of edges

Initialization the distances and visited arrays takes O(V) time and the main loop, in its each iteration finding the unvisited node with minimum distance takes O(V) time and updating distances to neighboring unvisited nodes takes O(E) time in total.So the overall time complexity of the main loop is O(V^2+E).

So the total time complexity = Initialization + Main Loop = O(V) + O(V^2+E)
= O(V^2+E)

Space Complexity

It has been implemented in the form of an adjacency matrix therefore its space complexity = O(V^2).

Bellman Ford Algorithm

Introduction

Bellman-Ford algorithm is a dynamic programming based algorithm made to find the shortest paths from a single source node to all other nodes in a graph. What gives Bellman Ford an edge over Dijkstra is its ability to deal with negative edges too.

Pseudocode

Input: Graph represented by an adjacency matrix, source node

```
1. Initialize distances array:
   - Set all distances to infinity, except the distance to the source node, which is set to 0.

2. Repeat (numNodes - 1) times:
   a. For each edge (u, v) with weight w:
      - Relax the edge: If distances[u] + w < distances[v], update distances[v] to distances[u] + w.

3. Check for negative cycles:
   a. For each edge (u, v) with weight w:
      - If distances[u] + w < distances[v], print "Graph contains negative cycle. Bellman-Ford not applicable." and exit.

4. Output the shortest paths:
   - Print or store the shortest distances from the source node to all other nodes in the distances array.
```

Advantages	Limitations
● Bellman-Ford can handle graphs with edges that have negative weights, which makes it more applicable and practical in a broader range of scenarios.	● Time Complexity of Bellman Ford is higher compared to Dijkstra's algorithm.

Time Complexity

V: Number of nodes
E: Number of edges

Initialization the distances array takes O(V) time and the main loop takes has an overall time complexity of O((V-1)*E). There is also an additional loop to check for negative cycles which has a time complexity of O(E). **So basically the total time complexity will be = O(Initialization)+O(Main Loop)+O(Negative Cycle)= O((V-1)*E).**

Dijkstra	Bellman Ford
● Efficient for dense graph with positive weights ● Does not work with negative weights	● Can handle negative weights too ● Efficient for sparse graphs, especially when there are negative weights

3. Minimum Spanning Tree & Algorithm Analysis

Prim's Algorithm

Introduction

Prim's algorithm is a greedy algorithm designed for finding the minimum spanning tree in a connected, undirected graph. The algorithm efficiently constructs a tree that spans all the vertices of the graph while minimizing the sum of edge weights.

Pseudocode

Input: Connected, undirected graph represented by an adjacency matrix

1. Select any starting vertex as the initial node.

2. Initialize an empty set to store the vertices included in the minimum spanning tree (MST).
   - Create a priority queue or min-heap to store candidate edges with their weights.

3. Mark the starting vertex as visited and add its adjacent edges to the priority queue.

4. Repeat until all vertices are included in the MST:
   a. Extract the minimum-weight edge (u, v) from the priority queue.
   b. If both vertices u and v are not in the MST set:
      - Add v to the MST set.

- Add edge (u, v) to the MST.
- Mark v has visited.
- Add all edges from v to the priority queue.

5. Output the minimum spanning tree.

Advantages	Limitations
● Prim's algorithm is efficient, especially as it always selects the edge with the minimum weight at each step, by this local optimal choice results in an overall optimal solution.	● Designed specifically for undirected graphs, cannot be applied to directed graphs.

Time Complexity

V: Number of Nodes
E: Number of Edges

(Initialization) Selecting a starting vertex and initializing the minimum spanning tree set has O(V) time complexity and extracting the minimum weight edge from priority queue and updating the priorities is in O((V+E)log E). The main loop repeats until all vertices are included in the MST and in each iteration of the loop extracting minimum weight edge and updating priorities takes O(log V).
Total T.C=O(Initialization)+O(Priority Updation)+O(Main Loop). So the overall T.C is O((V+E)log V).

Kruskal’s Algorithm

Introduction

Kruskal's algorithm is a greedy algorithm used for finding the minimum spanning tree (MST) of a connected, undirected graph. The algorithm incrementally builds the MST by selecting the smallest edges from the graph while avoiding the formation of cycles.

Pseudocode

Input: Connected, undirected graph represented by a list of edges with weights

1. Sort all the edges in non-decreasing order of their weights.

2. Initialize an empty set to represent the disjoint sets (initially, each vertex is its own set).

3. Initialize an empty list to store the edges of the minimum spanning tree (MST).

4. Repeat until the MST has (V - 1) edges (where V is the number of vertices):
   a. Extract the next smallest edge from the sorted list of edges.
   b. Check if adding this edge creates a cycle in the MST:
      - If the two vertices of the edge belong to different sets, add the edge to the MST.
      - Otherwise, discard the edge.

5. Output the minimum spanning tree represented by the list of edges in the MST.

Advantages	Limitations
● This algorithm performs well with sparse graphs as it processes individually, not relying on the density of the graph.	● Does not work for directed graphs, it is only designed to work on undirected graphs.

Time Complexity

In majority of the cases sorting of edges is the dominant term when deducing the overall time complexity. Sorting edges is the most time consuming step which has a time complexity of O(E log E). So the overall time complexity is O(E log E).

Prim	Kruskal
● Well suited for sparse graphs ● Efficient when graph is fully connected ● Is designed in a way that naturally avoids creating cycles.	● Well suited for sparse graphs ● More efficient when the graph has many vertices but relatively few edges ● It can work on disconnected components and connect them as it selects edges

4. Cycles

In graph theory, a cycle is a closed path in a graph, where a path is a sequence of edges that connect a sequence of vertices, and the last vertex is the same as the first one.

Effects of Cycles on Computational Cost

Algorithm Complexity: The presence of cycles can impact the time complexity for certain algorithms For example, algorithms that operate on graphs with cycles may have different time complexities compared to algorithms designed for acyclic graphs.

Infinite Loops: In programming, cycles in code structures (e.g., loops) can lead to infinite loops if not properly handled. Infinite loops can significantly affect the runtime of a program, and they may lead to resource exhaustion and unresponsiveness.

Memory Usage: In certain algorithms or data structures, cycles can lead to increased memory usage. For instance, cyclic references in data structures may prevent memory from being properly deallocated.

Pseudocode for Detecting Cycle

```
function hasCycle():
    for each node in graph:
        if not visited[node] and detectCycleFromNode(node):
            return true  // Cycle detected

    return false  // No cycle detected

function detectCycleFromNode(node):
    visited[node] = true
    recursionStack[node] = true

    for each neighbor in neighbors of node:
        if not visited[neighbor]:
            if detectCycleFromNode(neighbor):
                return true  // Cycle detected
        else if recursionStack[neighbor]:
            return true  // Cycle detected

    recursionStack[node] = false
    return false  // No cycle detected
```

5. Sorting the dataset with respect to time

In this part of the project we sort the fourth column of the dataset (Time) using three sorting algorithms :

- Quick Sort
- Merge Sort
- Heap Sort

The time values in the fourth column are in epoch format. It is a format which represents the number of seconds since 1 Jan, 1970.Epoch time is commonly used in computing systems to facilitate date and time calculations.The use of epoch time simplifies time-related operations, as it represents time as a single integer, making it easy to perform arithmetic calculations.

Quick Sort

Introduction

Quick sort is a sorting algorithm based on the divide and conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Pseudocode

```
function quickSort(array):
    if length of array <= 1:
        return array  // Base case: Already sorted

    pivot = randomly_choose_pivot(array)

    left, right = partition(array, pivot)

    return concatenate(quickSort(left), pivot, quickSort(right))

function partition(array, pivot):
    left = [], right = []

    for element in array:
        if element < pivot:
            append element to left
        else:
            append element to right
  return left, right
```

Advantages	Limitations
• Known for its speed and efficiency, especially in average and best case scenarios.	• Its worst case time complexity is O(N^2), this occurs when the chosen pivot consistently results in poor balanced partitions. Therefore affecting performance.

Merge Sort

Introduction

The idea behind merge sort is to recursively divide an array into smaller segments, sort each segment individually and then merge the segments back together to obtain the final sorted array.

Pseudocode

```
function mergeSort(array):
    if length of array <= 1:
        return array  // Already sorted

    middle = length of array / 2
    left = mergeSort(first half of array)
    right = mergeSort(second half of array)

    return merge(left, right)

function merge(left, right):
    result = []
    while left is not empty and right is not empty:
        if first element of left <= first element of right:
            append first element of left to result
            remove first element from left
        else:
            append first element of right to result
            remove first element from right

    // Append remaining elements, if any
    append remaining elements of left to result
    append remaining elements of right to result
    return result
```

Advantages	Limitations
● Merge Sort is a stable sorting algorithm, meaning it maintains the relative order of equal elements in the sorted output as they were in the input.	● It needs additional memory proportional to the size of the input array for the temporary storage of elements during the merging process. This can be a disadvantage in scenarios where memory usage is a critical factor.

Heap Sort

Introduction

Heap sort is a comparison based sorting technique based on binary heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for remaining elements.

Pseudocode

First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until the size of the heap is greater than 1.

- Build a heap from the given input array.
- Repeat the following steps until the heap contains only one element:
    - Swap the root element of the heap (which is the largest element) with the last element of the heap.
    - Remove the last element of the heap (which is now in the correct position).
    - Heapify the remaining elements of the heap.
- The sorted array is obtained by reversing the order of the elements in the input array.

Advantage	Limitations
● Heap Sort is an in-place sorting algorithm, meaning it doesn't require additional memory for temporary storage during the sorting process.	● Heap Sort is not a stable sorting algorithm. It does not necessarily maintain the relative order of equal elements in the sorted output as they were in the input. If stability is a requirement, other sorting algorithms like Merge Sort may be preferred.

Conclusion

In this project, our primary objectives were: normalizing the dataset and creating a suitable graph representation, followed by the application and analysis of various graph algorithms. The normalization of the dataset was crucial to prevent computational errors when employed in subsequent algorithms..

We began by applying Dijkstra and Bellman Ford algorithms to find single-source shortest paths, subsequently doing an in-depth analysis and comparison of their efficiencies. This exploration provided valuable insights into the performance characteristics of these algorithms.

Moving on to Minimum Spanning Trees, we implemented Prims and Kruskal's algorithms to identify the most efficient connections within the graph. Time complexity analysis was conducted, offering a comprehensive understanding of the computational costs associated with each algorithm, thereby facilitating optimal algorithm selection based on dataset characteristics.

Our project also involved the development of an algorithm for cycle detection within the graph. Understanding the cons of cycles and their impact on overall computation costs provided valuable considerations for algorithmic design and graph structure.

Lastly, we applied sorting algorithms—Merge, Quick, and Heap sort—to organize the dataset based on time values in EPOCH format. Converting these values into a human-readable format and storing them separately allowed for improved interpretability. A thorough complexity analysis of the sorting algorithms was conducted, offering insights into their efficiency and applicability.

In conclusion, this project addressed key aspects of data normalization, graph representation, and the application of fundamental algorithms. The analyses conducted on single-source shortest paths, minimum spanning trees, cycle detection, and sorting algorithms provided a

comprehensive exploration of algorithmic efficiency. The findings of this project contribute to a deeper understanding of algorithmic choices and their implications in diverse computational scenarios.