Design and Analysis of Algorithms

Cyber Security Department

CYL-2002

Fall 2024

Instructor:

Arslan Aslam

Contents

Algorithm Report: Shortest Path and Influence Chain 3
Introduction 3
Part 1: A* Algorithm for Shortest Path in a Social Network Graph 3
    Pseudocode 3
        1. Create Graph Function 3
        2. Heuristic Function 4
        3. A* Algorithm 4
        4. Main Function 5
    Time Complexity Analysis 5
    Execution of A Algorithm for Shortest Path* 6
Part 2: Longest Influence Chain in a Social Network 6
        1. Parse Graph Function 6
        2. Parse Influence Scores Function 6
        3. Find Longest Path Function (Dynamic Programming) 7
        4. Main Function 7
    Time Complexity Analysis 8
    Execution of Longest Influence Chain Algorithm 8
Conclusion 8

Algorithm Report: Shortest Path and Influence Chain

Introduction

In this project, we explore two key algorithms applied to a social network graph. The first algorithm, implemented in part1.cpp, uses the A* search algorithm to determine the shortest path between two nodes (users) in the graph. This approach leverages a heuristic function to optimize pathfinding by evaluating the number of direct connections of a node.

The second algorithm, implemented in part2.cpp, identifies the longest influence chain within the social network. This is achieved through a dynamic programming approach that finds the longest path where each subsequent user has a higher influence score than the previous one. The project also includes time complexity analysis to evaluate the efficiency of these algorithms.

The following sections provide detailed pseudocode, explanations, and complexity analysis for both algorithms.

Part 1: A* Algorithm for Shortest Path in a Social Network Graph

Pseudocode

1. Create Graph Function

Input: Graph file with edges and weights
Process:
1. Open the file and read each line.
2. Parse the user1, user2, and weight values.
3. Create an undirected graph using an adjacency list.

```
FUNCTION createGraph(filename):
    OPEN file(filename)
    FOR each line in file:
        PARSE user1, user2, weight
        ADD (user2, weight) to graph[user1]
        ADD (user1, weight) to graph[user2]
    END FOR
    CLOSE file
    PRINT "Graph successfully created."
```

2. Heuristic Function

- **Input:** Graph, current node
- **Output:** Number of connections (degree) of the node

```
FUNCTION heuristic(graph, node):
    RETURN SIZE of graph[node]
```

3. A* Algorithm

- **Input:** Graph, start node, goal node
- **Output:** Shortest path and total cost

Process:

- Initialize the priority queue with the start node and its heuristic.
- Track the cost (gCost) and parent nodes for path reconstruction.
- For each node, update the cost if a shorter path is found.
- Reconstruct the path once the goal is reached.

```
FUNCTION aStar(graph, start, goal):
    INITIALIZE priorityQueue with (heuristic(start), start)
    SET gCost[start] = 0
    SET parent[start] = -1

    WHILE priorityQueue is not empty:
        current = priorityQueue.top().node
        REMOVE current from priorityQueue

        IF current == goal:
            RECONSTRUCT and PRINT path using parent
            PRINT gCost[goal]
            RETURN

        FOR each neighbor of current:
            tentativeG = gCost[current] + edgeWeight
            IF tentativeG < gCost[neighbor]:
                gCost[neighbor] = tentativeG
                fCost = tentativeG + heuristic(neighbor)
                ADD (fCost, neighbor) to priorityQueue
                SET parent[neighbor] = current

    PRINT "No path found."
```

4. Main Function

```
FUNCTION main():
    graph = CREATE empty graph
    createGraph("social-network-proj-graph.txt")

    INPUT startNode and goalNode
    aStar(graph, startNode, goalNode)
```

Time Complexity Analysis

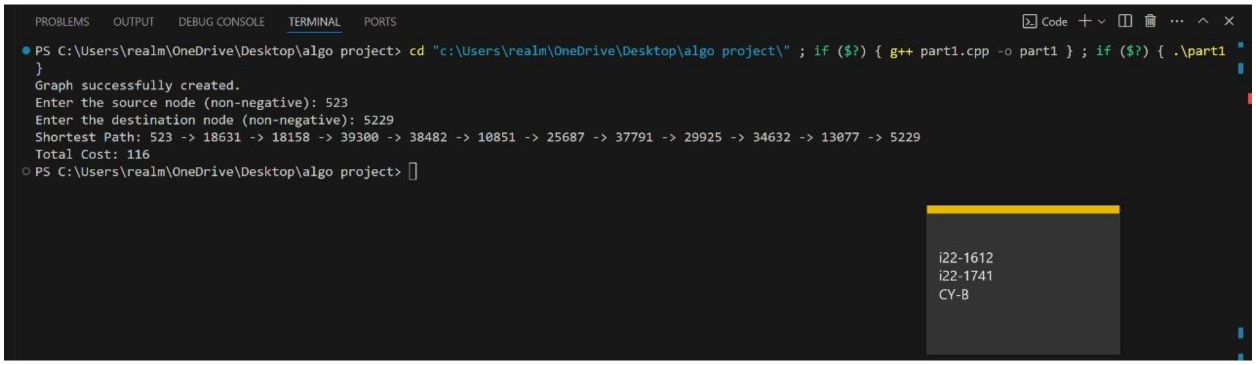
- **Graph Creation:** O(E), where **E** is the number of edges.

    1. In the graph creation step, each edge is read and processed once. For each edge, two operations are performed to add the connection to both user1 and user2 (since the graph is undirected). Therefore, the complexity is linear with respect to the number of edges.

- *A Algorithm:** O((V + E) * log V), where **V** is the number of vertices and **E** is the number of edges.

    1. The A* algorithm uses a priority queue (min-heap) to determine the next node to process. For each node, operations like insertion and removal in the priority queue take **O(log V)** time.
    2. Processing each node and its neighbors leads to a total of **V** nodes and **E** edges being considered. Thus, the overall complexity is **O((V + E) * log V)**.
    3. In the worst case, when every node and edge is processed, this ensures efficient pathfinding even in large graphs.

Execution of A Algorithm for Shortest Path*



```
PS C:\Users\realm\OneDrive\Desktop\algo project> cd "c:\Users\realm\OneDrive\Desktop\algo project\" ; if ($?) { g++ part1.cpp -o part1 } ; if ($?) { .\part1
}
Graph successfully created.
Enter the source node (non-negative): 523
Enter the destination node (non-negative): 5229
Shortest Path: 523 -> 18631 -> 18158 -> 39300 -> 38482 -> 10851 -> 25687 -> 37791 -> 29925 -> 34632 -> 13077 -> 5229
Total Cost: 116
PS C:\Users\realm\OneDrive\Desktop\algo project>
```

i22-1612
i22-1741
CY-B

Part 2: Longest Influence Chain in a Social Network

1. Parse Graph Function

Input: Graph file

```
FUNCTION parseGraph(filename):
    OPEN file(filename)
    FOR each line in file:
        PARSE user1, user2, weight
        ADD user2 to graph[user1]
        ADD user1 to graph[user2]
    CLOSE file
```

2. Parse Influence Scores Function

Input: Influence scores file

```
FUNCTION parseInfluence(filename):
    OPEN file(filename)
    FOR each line in file:
        PARSE user, score
        SET influenceScores[user] = score
    CLOSE file
```

3. Find Longest Path Function (Dynamic Programming)

- **Input:** Node
- **Output:** Longest path starting from the node

```
FUNCTION findLongestPath(node):
    IF node in memoLengths:
        RETURN memoLengths[node], memoPaths[node]

    maxLength = 1
    bestPath = [node]

    FOR each neighbor of node:
        IF influenceScores[neighbor] > influenceScores[node]:
            length, path = findLongestPath(neighbor)
            IF length + 1 > maxLength:
                maxLength = length + 1
                bestPath = [node] + path

    memoLengths[node] = maxLength
    memoPaths[node] = bestPath
    RETURN maxLength, bestPath
```

4. Main Function

```
FUNCTION main():
    parseGraph("social-network-proj-graph.txt")
    parseInfluence("social-network-proj-Influences.txt")

    maxLength = 0
    longestPath = []

    FOR each node in influenceScores:
        length, path = findLongestPath(node)
        IF length > maxLength:
```

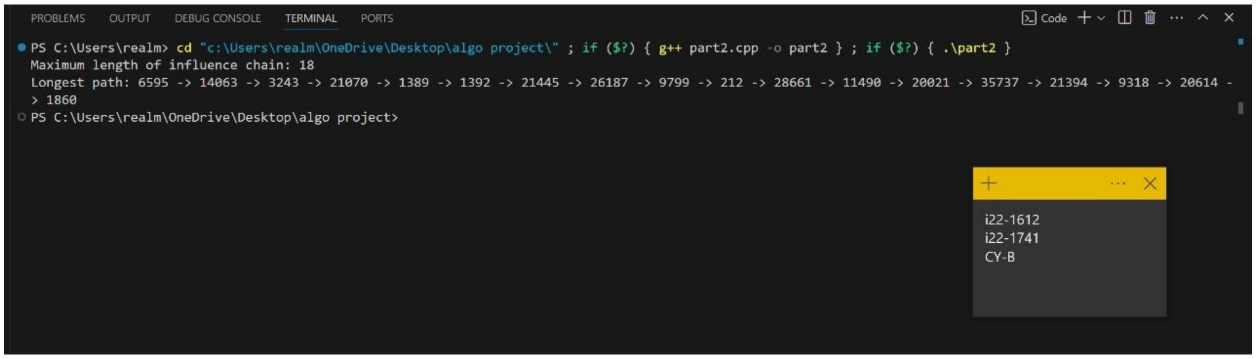
```
        maxLength = length
        longestPath = path

   PRINT maxLength and longestPath
```

Time Complexity Analysis

- **Graph Parsing:** O(E), where **E** is the number of edges.
    1. The graph parsing function reads through each edge once and adds the corresponding connections.
- **Finding Longest Path:** O(V + E), where **V** is the number of vertices and **E** is the number of edges.
    1. The longest path function uses dynamic programming with memorization. Each node is processed once, and each edge is considered once, making the complexity linear with respect to the size of the graph.

Execution of Longest Influence Chain Algorithm



```
PS C:\Users\realm> cd "c:\Users\realm\OneDrive\Desktop\algo project\" ; if ($?) { g++ part2.cpp -o part2 } ; if ($?) { .\part2 }
Maximum length of influence chain: 18
Longest path: 6595 -> 14063 -> 3243 -> 21070 -> 1389 -> 1392 -> 21445 -> 26187 -> 9799 -> 212 -> 28661 -> 11490 -> 20021 -> 35737 -> 21394 -> 9318 -> 20614 -
> 1860
PS C:\Users\realm\OneDrive\Desktop\algo project>
```

i22-1612
i22-1741
CY-B

Conclusion

In this report, we explored two critical algorithms for social network analysis: the A* algorithm for finding the shortest path and a dynamic programming approach for determining the longest influence chain. The A* algorithm leverages heuristics to optimize pathfinding, balancing exploration efficiency with accuracy. Its time complexity of **O((V + E) * log V)** ensures it scales effectively for large networks. On the other hand, the longest influence chain algorithm dynamically explores nodes based on their influence scores, efficiently caching results to avoid redundant calculations. These methods collectively provide powerful tools for understanding user connectivity and influence in social networks, making them valuable for applications like recommendation systems, network optimization, and viral marketing strategies.