Subham Adhikari
Coventry University Student ID Number: 14812262
B.Sc. (Hons.) Computing, Softwarica College of IT and E-commerce, Coventry University
ST5008CEM: Programming for Developers
Shrawan Thakur
March 02, 2025

1. a) Ans:

The main goal is to find the minimum number of measurement or attempts to find the critical temperature for the identical materials to change their properties.

Approach

To solve this, we use a dynamic programming (dp) approach. We have to determine the optimal plan to minimize the number of measurements/attempts needed to find the critical temperature. We consider the following given condition:

- If the material changes properties at a given temperature, it cannot be used for the further measurement and we cannot go past that temperature level.
- If the material does not change properties, the material can be reused for further tests.

Let **dp[k][n]** is the maximum number of temperatures that can be checked with given **n** temperature levels and **k** identical sample materials to find the critical temperature **f**.

Algorithm and Steps

- Initialize a 2D array **dp[k][n]** where **k** represents number of sample materials and **n** represents the number of temperature levels to find the critical temperature **f**.
- Enter a loop until **k** materials is iterated. For each material **k**, iterate for each temperature levels **n**. Make sure to start the iteration from 1, since material and levels cannot be 0.
- Create a **DP** table for the easy understanding. The row represents **n** temperature level and column represents **k** sample materials.
- Inside the nested loop, for each combination of **i** (samples) and **j** (temperature levels) check whether **i** > 1 and **j** > 1. If true then enter a loop which tracks a **cj** current row and **pj** previous row. And if **i** = **j** = 1 then, assign **dp[i][j]** = **j** or **dp[i][j]** = 1 respectively.
- For each possible temperature levels **x** (assumed) from 1 to **j**, calculate the cases:
  - If material does not change its property, then use same number of materials and test temperature levels below. Then the number of measurements becomes **value1** = **dp[i][cj]** or **value1** = **dp[i][j-x]**.

- If material changes its property, then use one less material and test temperature levels below. Then the number of measurements becomes **value2** = **dp[i-1][pj]** or **value2** = **dp[i-1] [x-1]**.

- Take the maximum measurements **val** of these two cases to handle the worst-case.
- Find the minimum number of measurements **min** (minimum of maximum measurements **val**) across all possible temperature levels to find the optimal measurement or attempts.
- Populate the table, for each **i** and **j**, set **dp[i][j]** = **min** + 1, where **min** is the minimum worst-case attempts found and extra addition of 1 represents the attempts taken before hand.
- After filling the DP table, the final result is stored in **dp[k][n]**, which gives the minimum measurements required to find the **f** critical temperature with **k** sample materials and **n** temperature levels.

Dynamic Programming (DP) Table

DP	**n** temperature levels										
k sample materials		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	2	3	4	5	6	7	8	9	10
	2	0	1	2	2	3	3	3	4	4	4	4
	3	0	1	2	2	3	3	3	3	4	4	4
	4	0	1	2	2	3	3	3	3	4	4	4
	5	0	1	2	2	3	3	3	3	4	4	4

Code Implementation

```java
import java.util.*;

public class MinMeasurementTemp {

    public static int minMeasurement(int k, int n){  1 usage
        int[][] dp = new int[k+1][n+1];
        // dp[k][n] represents the minimum measurements
        // k -> samples materials
        // n -> temperature levels
        // To find minimum measurement/attempts to find critical temperature for the materials to change their properties.

        for (int i = 1; i <= k ; i++) {
            for (int j = 1; j <= n; j++) {
                if (i == 1) {   // if k (sample material) is 1, the attempts is n
                    dp[i][j] = j;
                }
                else if(j == 1){    // if n (temperature level) is 1, the attempts is 1
                    dp[i][j] = 1;
                }
                else{
                    int min = Integer.MAX_VALUE;
                    for (int cj = j-1, pj = 0; cj >=0 ; cj--, pj++) { // cj -> current row      // pj -> previous row
                        int value1 = dp[i][cj];     // does not change
                        int value2 = dp[i-1][pj];   // changes
                        int val = Math.max(value1, value2);
                        min = Math.min(val, min);
                    }
                    dp[i][j] = min + 1;
                }
            }
        }
        return dp[k][n];
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // k -> samples material and n -> temperature levels
        System.out.print("Enter the no. of material samples: ");
        int k = scan.nextInt();
        System.out.print("Enter the no. of temperature levels: ");
        int n = scan.nextInt();

        System.out.println("The minimum measurement/attempts to find critical temperature is " + minMeasurement(k, n));

    }
}
```

Example-1

Input: k = 2, n = 5

Output (no of attempts): 3

```
Enter the no. of material samples: 2
Enter the no. of temperature levels: 5
The minimum measurement/attempts to find critical temperature is 3
```

Example-2

Input: k = 5, n = 7

Output (no of attempts): 3

```
Enter the no. of material samples: 5
Enter the no. of temperature levels: 7
The minimum measurement/attempts to find critical temperature is 3
```

Example-3

Input: k = 15, n = 20

Output (no of attempts): 5

```
Enter the no. of material samples: 15
Enter the no. of temperature levels: 20
The minimum measurement/attempts to find critical temperature is 5
```

1. b) Ans:

The goal is to find the **k**th lowest combined return where we select one element from each of the two sorted arrays and calculating their product.

There are two sorted arrays: **returns1** and **returns2** given to calculate kth smallest combined return from both arrays.

Approach

To solve this problem, we can use a Binary Search Tree (BST). The main approach is to

- Insert all the possible combinations into the BST.
- Traverse through nodes in sorted order from smallest product to largest.
- The **k**th node visited during the traversal will give the **k**th lowest combined return.

Algorithm and Steps

- Initialize a Binary Search Tree (BST) class **BST** and node class **BSTNode** where **i**, **j** represents indices of **returns1** and **returns2** respectively and **product** represents the product between the indices of **returns1** and **returns2** i.e., **product = returns1[i] * returns2[j]** and **left**, **right** represent child nodes – left and right of BST.
- In **BST** class, define methods **insert()** and **insertRec()** which adds a new node into the BST based on product value. The smaller products go to the left side and larger products go to the right side of the tree.
- Similarly in **BST** class, define another method **kthSmallest(int k)** which performs an in-order traversal by visiting every nodes in increasing order. It uses a counter **count** to track the number of nodes visited. When **count** equals **k**, it records the product of current node.
- In main class, define a method **findKthSmallestProduct()** where it uses nested loops to iterate through all indices **i** from **returns1** and **j** from **returns2**. Insert all possible product combinations into the BST.
- For each pair, compute **product = returns1[i] * returns2[j]** and insert a new **BSTNode(i, j, product)** into the BST. If **product** is smaller, insert into the left subtree and if **product** is larger, insert into the right subtree.

- After building the BST with all possible combinations, an in-order traversal is performed to get the **k**th smallest product return. In-order traversal approach:
  - Traverse the left subtree.
  - Visit the current node and increment a count.
  - If count equals **k**, store the product of node.
  - Traverse the right subtree.
- The **k**th visited node in in-order traversal holds the **k**th smallest product.
- The product at the **k**th lowest product return which is based on sorted order is returned as the output.

Time and Space Complexity Analysis

1. Time Complexity:

- To build BST:
  - There is **m * n** possible products where **m = returns1.length**, **n = returns2.length**.
  - Each insertion into the BST takes **O(log(m * n))** in the worst case.
  - Total BST insertions take **O(m * n log(m * n))**.
- To find the kth smallest product, in-order traversal takes O(k) time to reach the kth node.

Total Time Complexity in the worst case: **O(m * n log(m * n) + k) ≈ O(m * n log(m * n))**.

2. Space Complexity:

- To store BST in the memory, it takes up to **O(m * n)** nodes in all product pairs.
- The recursive stack for in-order traversal requires **O(m * n)** space in worst case.

Code Implementation

```java
// Binary Search Tree (BST) Node Class
class BSTNode {  8 usages
    int i, j;                   // Indices from returns1 and returns2  1 usage
    int product;                // The product returns1[i] * returns2[j]  4 usages
    BSTNode left, right;        // Left and right children  4 usages


    BSTNode(int i, int j, int product) {  1 usage
        this.i = i;
        this.j = j;
        this.product = product;
        this.left = this.right = null;
    }
}
```

```java
// Binary Search Tree Class
class BST {  2 usages
    BSTNode root;  4 usages

    public BST() {  1 usage
        root = null;
    }

    // Insert a new node into the BST
    public void insert(BSTNode node) {  1 usage
        root = insertRec(root, node);
    }

    public BSTNode insertRec(BSTNode root, BSTNode node) {  3 usages
        if (root == null) {
            root = node;
            return root;
        }

        if (node.product < root.product) {
            root.left = insertRec(root.left, node);
        }
        else {
            root.right = insertRec(root.right, node);
        }
        return root;
    }
```

```java
    // In-order traversal to find the k-th smallest product
    public int kthSmallest(int k) {  1 usage
        int[] result = new int[1];
        int[] count = new int[1]; // To keep track of how many nodes we've visited
        inOrderTraversal(root, k, count, result);
        return result[0];
    }

    // Recursive in-order traversal
    private void inOrderTraversal(BSTNode node, int k, int[] count, int[] result) {  3 usages
        if (node == null) return;

        // Traverse the left subtree
        inOrderTraversal(node.left, k, count, result);

        // Visit the current node
        count[0]++;
        if (count[0] == k) {
            result[0] = node.product;
            return;
        }

        // Traverse the right subtree
        inOrderTraversal(node.right, k, count, result);
    }
}
```

```java
public class KthSmallestInvestmentProduct {
    public static int findKthSmallestProduct(int[] returns1, int[] returns2, int k) {  3 usages
        BST bst = new BST();
        // Insert all possible combinations of products into the BST
        for (int i = 0; i < returns1.length; i++) {
            for (int j = 0; j < returns2.length; j++) {
                int product = returns1[i] * returns2[j];
                bst.insert(new BSTNode(i, j, product));
            }
        }
        // Use in-order traversal to find the k-th smallest product
        return bst.kthSmallest(k);
    }

    public static void main(String[] args) {
        // Example 1
        int[] returns1 = {2, 5};
        int[] returns2 = {3, 4};
        int k = 2;
        System.out.println("The 2nd smallest product is: " + findKthSmallestProduct(returns1, returns2, k)); // Output: 8

        // Example 2
        int[] returns1_2 = {-4, -2, 0, 3};
        int[] returns2_2 = {2, 4};
        int k_2 = 6;
        System.out.println("The 6th smallest product is: " + findKthSmallestProduct(returns1_2, returns2_2, k_2)); // Output: 0

        // Example 3
        int[] returns1_3 = {1, 2, 3, 4};
        int[] returns2_3 = {-1, 0, 1, 2, 3};
        int k_3 = 3;
        System.out.println("The 3th smallest product is: " + findKthSmallestProduct(returns1_3, returns2_3, k_3)); // Output: -2
    }
}
```

Example-1

Input: returns1 = {2, 5}, returns2 = {3, 4}, k = 2

```
int[] returns1 = {2, 5};
int[] returns2 = {3, 4};
int k = 2;
System.out.println("The 2nd smallest product is: " + findKthSmallestProduct(returns1, returns2, k));
```

Output: 8

```
The 2nd smallest product is: 8
```

Example-2

Input: returns1 = {-4, -2, 0, 3}. returns2 = {2, 4}, k = 6

```
int[] returns1_2 = {-4, -2, 0, 3};
int[] returns2_2 = {2, 4};
int k_2 = 6;
System.out.println("The 6th smallest product is: " + findKthSmallestProduct(returns1_2, returns2_2, k_2));
```

Output: 0

```
The 6th smallest product is: 0
```

Example-3

Input: returns1= {1000, 200, 3000, 400, 2500}, returns2= {1000, 100, 3500, 2000, 500}, k = 5

```
int[] returns1_4 = {1000, 200, 3000, 400, 2500};
int[] returns2_4 = {1000, 100, 3500, 2000, 500};
int k_4 = 5;
System.out.println("The 5th smallest product is: " + findKthSmallestProduct(returns1_4, returns2_4, k_4));
```

Output: 200000

```
The 5th smallest product is: 200000
```

2. a) Ans:

The goal is to assign the minimum number of rewards to a team of **n** employees where each employee is given at least one reward and the employees with a high-performance rating more rewards than other employees.

There is given an integer array rating: **ratings**

Approach

In this, the two-pass greedy approach is used. We assume an array **ratings** such that the rating of each employee is stored.

- Left-to-Right Pass:

  If the rating of an employee is higher than the previous employee, they receive one more reward than the previous employee.
- Right-to-Left Pass:

  If the rating of an employee is higher than the next employee, they receive at least one more reward than the next employee.

The two-pass greedy method ensures that both left and right conditions are satisfied.

Algorithm and Steps

- Define a method **calculateMinRewards()** which takes a integer array **ratings** as parameter. First the validation of array **ratings** is checked i.e.,
  - If **ratings** null or empty, return 0.
  - If the length of **ratings** is equal to, return 1.
- Inside this method, initialize an integer array **rewards** of length **n** which represent rewards received by employees and number of employees respectively.
- Start a left to right pass iteration which iterates through each employee from **1** to **n-1** and checks whether **ratings[i] > ratings[i - 1]** then sets **rewards[i] = rewards[i - 1] + 1**. This ensures that any employee with a higher rating than the previous one receives one more reward.
- Similarly, Start a right to less pass iteration which iterates through each employee from **n-2** down to **0** and checks whether **ratings[i] > ratings[i + 1]** then sets **rewards[i] =**

Math.max(rewards[i], rewards[i + 1] + 1). This ensures that employee with a higher rating than the next one receives at least one more reward than that neighbor.

- Then, sum up all values in the rewards array to find the total minimum rewards needed. Finally, the output is returned.

Time and Space Complexity Analysis

1. Time Complexity:

- The iteration through array takes **O(n)** in worst case where **n** is the number of employees.

2. Space Complexity:

- The space used by an array of size **n** to store rewards is **O(n)** in worst case.

Code Implementation

```java
import java.util.*;

public class EmployeeRewards {
    public static int calculateMinRewards(int[] ratings) {  3 usages
        // Validation
        if (ratings == null || ratings.length == 0) {
            return 0;
        }
        else if (ratings.length == 1) {
            return 1;
        }

        int n = ratings.length;
        int[] rewards = new int[n];
        // Initialize rewards array with 1 for each employee
        // Fills the value in array. It is efficient than using loops
        Arrays.fill(rewards,  val: 1);

        /*
        Left-to-Right Pass:
        If current employee has higher rating than previous,
        give current employee one more reward than the previous.
        */
        for (int i = 1; i < n; i++) {
            if (ratings[i] > ratings[i - 1]) {
                rewards[i] = rewards[i - 1] + 1;
            }
        }
```

```
        /*
         Right-to-Left Pass:
         If current employee has higher rating than next,
         ensure the current employee gets more rewards than the next one.
         */
        for (int i = n - 2; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1]) {
                // Math.max -> preserve any higher values from Left-to-Right pass
                rewards[i] = Math.max(rewards[i], rewards[i + 1] + 1);
            }
        }

        // Sum up all the rewards to get the total minimum rewards needed.
        int totalRewards = 0;
        for (int reward : rewards) {
            totalRewards += reward;
        }

        return totalRewards;
    }

    public static void main(String[] args) {
        // Example 1
        int[] ratings1 = {1, 0, 2};
        System.out.println("Minimum rewards of ratings1 = " + calculateMinRewards(ratings1));

        // Example 2
        int[] ratings2 = {1, 2, 2};
        System.out.println("Minimum rewards of ratings2 = " + calculateMinRewards(ratings2));

        // Example 3
        int[] ratings3 = {2, 4, 0, 5, 7};
        System.out.println("Minimum rewards of ratings3 = " + calculateMinRewards(ratings3));
    }
}
```

Example-1

Input: ratings = {1, 0, 2}

```
// Example 1
int[] ratings1 = {1, 0, 2};
System.out.println("Minimum rewards of ratings1 = " + calculateMinRewards(ratings1)); // Expected output: 5
```

Output: 5

```
Minimum rewards of ratings1 = 5
```

Example-2

Input: ratings = {1, 2, 2}

```
// Example 2
int[] ratings2 = {1, 2, 2};
System.out.println("Minimum rewards of ratings2 = " + calculateMinRewards(ratings2)); // Expected output: 4
```

Output: 4

```
Minimum rewards of ratings2 = 4
```

Example-2

Input: ratings = {2, 4, 0, 5, 7}

```
// Example 3
int[] ratings3 = {2, 4, 0, 5, 7};
System.out.println("Minimum rewards of ratings3 = " + calculateMinRewards(ratings3)); // Expected output: 9
```

Output: 9

```
Minimum rewards of ratings3 = 9
```

2. b) Ans:

The main goal is to find out the lexicographically close pair of points with the smallest distance where smallest distance is calculated using:

| x_coords [i] - x_coords [j]| + | y_coords [i] - y_coords [j]|

where **x_coords** and **y_coords** represent array of x-coordinates and y-coordinates of the points respectively.

Approach

In this problem, we apply Brute-force because this approach directly computes distances and is optimal for small to moderate points. It iterates through all pairs **(i, j)** with **i < j**, compute their distance and track the pair with the smallest distance. If multiple pairs have the same distance, then it selects the lexicographically smallest one.

Algorithm and Steps

- Declare a method **findClosestPair()** inside initialize variables **minDistance** to a large value **Integer.MAX_VALUE**, **bestI** and **bestJ** to store the indices of the best (closest) pair.
- Start a nested iteration that iterates over all the pairs **(i, j)** with **i < j**. The outer loop runs from **i=0** to **n-1** and inner loop from **j=i+1** to **n-1**.
- For each iteration in the nested loop, the distance is calculated i.e.
  distance = Math.abs(x_coords[i] - x_coords[j]) + Math.abs(y_coords[i] - y_coords[j])
- Similarly, check the conditions in each iteration:
  - If **distance < minDistance**, then update **minDistance=distance** and set **bestI = i**, **bestJ = j**.
  - If **distance == minDistance**, check lexicographical order i.e., if **i<bestI** or **i==bestI** and **j<bestJ**, update **bestI=i** and **bestJ=j**.
- After checking all pairs, return the array pair **[bestI, bestJ]** as the result.

Time and Space Complexity

1. Time Complexity:

- The nested loops to compare every pair through two arrays co-ordinates takes **$O(n^2)$** in worst case where **n** is the length of array co-ordinates.

2. Space Complexity:

- The space used is **O(1)** in worst case.

Code Implementation

```java
import java.util.Arrays;

public class LexicographicalClosestPair {
    // Method to find the lexicographically smallest pair with the minimum distance
    public static int[] findClosestPair(int[] x_coords, int[] y_coords) {  2 usages
        int n = x_coords.length;

        // Initialize minDistance to a large value and bestPair as invalid indices
        int minDistance = Integer.MAX_VALUE;
        int bestI = -1, bestJ = -1;
```

```java
        // Iterate over all pairs (i, j) with i < j
        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                // Calculate distance between points i and j
                int distance = Math.abs(x_coords[i] - x_coords[j]) + Math.abs(y_coords[i] - y_coords[j]);

                // If we find a smaller distance, update minDistance and best pair
                if (distance < minDistance) {
                    minDistance = distance;
                    bestI = i;
                    bestJ = j;
                }
                // If the same distance is found, check lexicographical order:
                // (i, j) is lexicographically smaller if i < bestI or (i == bestI and j < bestJ)
                else if (distance == minDistance) {
                    if (i < bestI || (i == bestI && j < bestJ)) {
                        bestI = i;
                        bestJ = j;
                    }
                }
            }
        }

        return new int[] {bestI, bestJ};
    }
```

```java
    public static void main(String[] args) {
        // Example 1:
        int[] x_coords1 = {1, 2, 3, 2, 4};
        int[] y_coords1 = {2, 3, 1, 2, 3};
        int[] result1 = findClosestPair(x_coords1, y_coords1);
        System.out.println("The closest pair is: " + Arrays.toString(result1));

        // Example 2:
        int[] x_coords2 = {1, 5, 3, 9, 4};
        int[] y_coords2 = {8, 3, 1, 6, 5};
        int[] result2 = findClosestPair(x_coords2, y_coords2);
        System.out.println("The closest pair is: " + Arrays.toString(result2));

    }
}
```

Example-1

Input: x_coords = {1, 2, 3, 2, 4}, y_coords= {2, 3, 1, 2, 3}

```java
// Example 1:
int[] x_coords1 = {1, 2, 3, 2, 4};
int[] y_coords1 = {2, 3, 1, 2, 3};
int[] result1 = findClosestPair(x_coords1, y_coords1);
System.out.println("The closest pair is: " + Arrays.toString(result1)); // Expected output: [0, 3]
```

Output: result = [1, 3]

```
The closest pair is: [0, 3]
```

Example-2

Input: x_coords = {1, 5, 3, 9, 4}, y_coords= {8, 3, 1, 6, 5}

```java
// Example 2:
int[] x_coords2 = {1, 5, 3, 9, 4};
int[] y_coords2 = {8, 3, 1, 6, 5};
int[] result2 = findClosestPair(x_coords2, y_coords2);
System.out.println("The closest pair is: " + Arrays.toString(result2)); // Expected output: [1, 4]
```

Output: result = [1, 4]

```
The closest pair is: [1, 4]
```

3. a) Ans:

The goal is to find the minimum total cost to connect **n** devices in the network where each device can either install a communication module at a given cost which is specified by the **modules** array or be connected using a direct connection which is specified in **connections** where **connections[j] = [device1j, device2j, costj]** which represents the cost to connect devices **device1j and device2j**.

Approach

For this problem, we assume a Minimum Spanning Tree (MST) problem on an augmented graph.

- Add a special virtual node (node 0) which represents module installation.
- For each device **i**, add an edge from the virtual node to device **i** with weight **modules[i]**.
- Add the given bidirectional connections as edges between devices.

We use Kruskal's algorithm to build an MST that connects all nodes (the virtual node and all devices) at the minimum total cost and Disjoint Set Union (DSU) structure the algorithm.

Algorithm and Steps

- Define classes **Edge** represents an edge in the graph with endpoints u, v and a connection cost and **DSU** to perform efficient union-find operations in Kruskal's algorithm.
- Declare a method in main class **minCostToConnect()** where the minimum total cost is calculated to connect all devices using an MST approach.
- Start a loop such that for each device **i** from 1 to **n**, add an edge between the virtual node (node 0) and device **i**. The cost of this edge is the module installation cost given in **modules[i-1]**.
- Similarly, for each connection provided (an array like [u, v, cost]), add an edge between device **u** and device **v** with the given cost. The connections should be bidirectional.
- Sort all edges in ascending order of cost to ensure we pick the lowest cost edges first.
- Initialize a **DSU** to manage which devices are connected.
- Start an iteration through sorted edges where for each **DSU** edge, check whether the two nodes are in different sets. If they are not connected i.e., adding the edge does not create a

cycle, then add the cost of edge to the total cost and merge the two sets. Stop when **n** edges are added where edges connect **n+1** nodes.

- Return the **totalCost** as minimum total cost required all the devices.

Time and Space Complexity

1. Time Complexity:

- The nested loops to compare every pair through two arrays co-ordinates takes **$O(n^2)$** in worst case where **n** is the length of array co-ordinates.
- To build graph, it took **O(v + e)** where **v=n+1** and **e** is the total number of edges (the virtual node and the provided connections).
- To sort edges, it took **O(e log e)**.
- The Kruskal's Algorithm took **O(e log e)**.

Overall time complexity is **O(e log e)** in worst where **e** is the number of edges.

2. Space Complexity:

- The space used to store graph which is edges list is **O(e)**.
- The space used by DSU is **O(v)**.

Overall space complexity is **O(v + e)** in worst case where v = n+1 and e is the number of edges.

Code Implementation

```java
import java.util.*;

// Edge class represents an edge in the graph with endpoints u and v, and a connection cost
class Edge implements Comparable<Edge> {  6 usages
    int u, v, cost;  2 usages
    Edge(int u, int v, int cost) {  2 usages
        this.u = u;
        this.v = v;
        this.cost = cost;
    }
    @Override
    public int compareTo(Edge other) {
        return Integer.compare(this.cost, other.cost);
    }
}
```

```java
// DSU (Disjoint Set Union) class for efficient union-find operations in Kruskal's algorithm
class DSU {  2 usages
    int[] parent;  7 usages
    DSU(int n) {  1 usage
        parent = new int[n + 1]; // +1 for virtual node 0
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {  3 usages
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // Path compression
        }
        return parent[x];
    }
    boolean union(int x, int y) {  1 usage
        int px = find(x), py = find(y);
        if (px == py) return false;
        parent[py] = px;
        return true;
    }
}
```

```java
public class MinConnectionCost {
    public static int minCostToConnect(int n, int[] modules, int[][] connections) {  1 usage
        List<Edge> edges = new ArrayList<>();

        // Add edges from the virtual node (0) to each device (1-indexed) with cost equal to module installation cost
        for (int i = 0; i < n; i++) {
            edges.add(new Edge( u: 0,  v: i + 1, modules[i]));
        }

        // Add all bidirectional connections (device indices are assumed 1-indexed)
        for (int[] conn : connections) {
            int u = conn[0], v = conn[1], cost = conn[2];
            edges.add(new Edge(u, v, cost));
        }

        // Sort edges by cost (Kruskal's algorithm)
        Collections.sort(edges);

        DSU dsu = new DSU(n);
        int totalCost = 0, edgesAdded = 0;
```

Example-1

Input: n = 3, modules = {1, 2, 3}, connections = {{1, 2, 1}, {2, 3, 1}}

```
// Example-1
int n1 = 3;
int[] modules1 = {1, 2, 2};
int[][] connections1 = {{1, 2, 1}, {2, 3, 1}};
int result1 = minCostToConnect(n1, modules1, connections1);
System.out.println("The minimum total cost to connect all devices: " + result1); // Expected Output: 3
```

Output: 3

```
The minimum total cost to connect all devices: 3
```

Example-2

Input: n = 4, modules = {2, 3, 1, 4}, connections = {{1, 2, 1}, {1, 3, 3}, {2, 4, 2}, {3, 4, 4}}

```
// Example-2
int n2 = 4;
int[] modules2 = {2, 3, 1, 4};
int[][] connections2 = {{1, 2, 1}, {1, 3, 3}, {2, 4, 2}, {3, 4, 4}};
int result2 = minCostToConnect(n2, modules2, connections2);
System.out.println("The minimum total cost to connect all devices: " + result2); // Expected Output: 6
```

Output: 6

Example-3

Input: n = 1, modules = {5}, connections = {}

```
// Example-3
int n3 = 1;
int[] modules3 = {5};
int[][] connections3 = {};  // No connections
int result3 = minCostToConnect(n3, modules3, connections3);
System.out.println("The minimum total cost to connect all devices: " + result3); //Expected Output: 5
```

Output: 5

```
The minimum total cost to connect all devices: 5
```

3. b) Ans:

The goal is to create a Tetris Game using queue **pieceQueue** to store the sequence of the falling blocks and stack **boardStack** to represent the current state of the game board like a matrix.

Approach

The solution uses simple data structures to model Tetris. A queue is used to store the falling tetris block pieces **pieceQueue** so that the game can easily display a preview of the next piece. A stack (implemented as a list of rows and columns) **boardStack** is the game board, where each row is an array of fixed length. The game loop is driven by a timer: on each tick, the current piece moves down; if it can't move down any longer (due to collision with the board or other blocks), it is locked into position down at bottom, full rows are cleared (which raises the score and level), and a new piece is spawned. Mouse (on-screen image buttons) and keyboard (arrow keys and "**R**" for restart) controls are included to move, rotate, and drop pieces. A game-over animation displaying a pulsating, fading "**GAME OVER**" message and a prompt to restart the game is triggered when a piece cannot be spawned because the top row is full.

Algorithm and Steps

1. Initialization:

- The board **boardStack** is initialized as an empty stack of rows, each row an integer arrays the size of the width of the board (composed of 10 cells), where there are 20 rows making a board. The value for each cell in the board is initialized to 0, meaning they are all empty.
- A queue **pieceQueue** is initialized to hold the pieces that are falling. Some (let's say three) random tetris block pieces are initialized and enqueued. The top of the queue is dequeued to form the currently falling tetris block.
- The main game panel (showing the board), a side panel for score, level, and next piece preview and buttons (with onboard graphics) for left, right, rotate and drop actions are initialized.

- Then the keyboard key bindings are established for keyboard key presses (**LEFT**, **RIGHT**, **UP**, **DOWN**, **R**) and mouse buttons that perform the same control methods.

2. Game Loop:

- Try to move the current piece one cell down. If the move is feasible (no collision), move to that position. If not (collided), freeze the piece by copying its matrix onto the board.
- Check for each row from bottom to top: if any row is complete (i.e., contains no zeros), remove that row from the board and add a new blank row on top. Add in the score by increasing (lines cleared × **100** × current level) and increase the level based on the new score.
- Take the next tetris block from the queue into the current piece. Enqueue a new random piece. Repaint the preview panel such that the next block is always displayed.
- Should the new piece already contact the board (the top row is filled), the start of the game over sequence will then be triggered.

3. Game Over & Reset

- The main game timer stops when the game ends, and an animation timer begins running with a pulsating effect on "**GAME OVER**" text and a transition from transparency (alpha) to scale. The game board has its semi-transparent overlay announcing "**GAME OVER**", with "Press **R** to restart."
- Will generate an instant reset of the game with the score, level, and the piece queue reinitialized. Press "**R**" to clear the board and begin the main game timer again.

4. User Input Handling

- Left and right arrow key move the piece left and right, respectively, up arrow key rotates the piece, down arrow key drops the piece to the bottom instantly. Pressing R will restart the game as soon as it ends.
- On-screen buttons (implemented as custom image buttons) in the side panel also call the same control methods.

Time and Space Complexity

1. Time Complexity:

- Each collision detection, motion, and locking operations are on a small tetris block matrix (of constant size) and the board has fixed dimensions (20 x 10). Therefore each update tick runs in **O(1)** time.
- Checking through the board for complete rows is **O(BOARD_HEIGHT × BOARD_WIDTH)**, habitually constant because **BOARD_HEIGHT and BOARD_WIDTH** are fixed.

Overall: Each tick of the game loop is **O(1)** time, which means the total time spent by the update in-between ticks is **O(1)**. In n ticks, this gives **O(n)**, which is likely to work in practice given the fixed board size.

2. Space Complexity:

- The board uses a fixed amount of space - 20 rows x 10 columns, giving **O(1)** space.
- The queue stores a constant number of tetris block pieces - for example, 3-giving **O(1)**.
- Other game state variables (score, level, timers, etc.) use constant space.

Overall: In terms of board and game state sizes, the space complexity is **O(1)**.

Code Implementation

```java
package tetrisgame;

import javax.swing.*;
import java.awt.*;
import java.util.*;


public class Tetris extends JFrame {
    // Game Constants
    public static final int BOARD_WIDTH = 10;
    public static final int BOARD_HEIGHT = 20;
    public static final int CELL_SIZE = 35;

    // Piece queue
    private Queue<TetrisBlock> pieceQueue = new LinkedList<>();

    private TetrisMainPanel mainPanel;

    public Tetris() {
        initUI();
    }

    private void initUI() {
        setTitle("Tetris Game");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setResizable(false);

        mainPanel = new TetrisMainPanel(BOARD_WIDTH, BOARD_HEIGHT, CELL_SIZE, pieceQueue);
        add(mainPanel);

        pack();
        setLocationRelativeTo(null);
    }

    public static void main(String[] args) {
        EventQueue.invokeLater(() -> new Tetris().setVisible(true));
    }
}
```

```java
class TetrisMainPanel extends JPanel {
    private TetrisGameArea gameArea;
    private TetrisSidePanel sidePanel;

    public TetrisMainPanel(int gridColumns, int gridRows, int cellSize, Queue<TetrisBlock> pieceQueue) {
        setBackground(TetrisColors.COLORS[0]);
        setLayout(new BorderLayout(15, 15));
        setBorder(BorderFactory.createEmptyBorder(15, 15, 15, 15));

        gameArea = new TetrisGameArea(gridColumns, gridRows, cellSize, pieceQueue);
        add(gameArea, BorderLayout.CENTER);

        sidePanel = new TetrisSidePanel(pieceQueue, gridRows, cellSize, gameArea);
        add(sidePanel, BorderLayout.EAST);

        // Let the game area update the side panel (score, level, preview)
        gameArea.setSidePanel(sidePanel);
    }

    public TetrisGameArea getGameArea() {
        return gameArea;
    }
}
```

```java
package tetrisgame;

import javax.swing.*;
import javax.swing.Timer;
import java.awt.*;
import java.awt.event.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.util.*;

public class TetrisGameArea extends JPanel {
    private int gridColumns;
    private int gridRows;
    private int cellSize;

    // Board as a stack (top row at index 0)
    private Stack<int[]> boardStack;
    private Queue<TetrisBlock> pieceQueue;
    private TetrisBlock currentPiece;
    private int score = 0;
    private int level = 1;
    private boolean gameOver = false;

    private Timer gameTimer;

    // Game Over animation parameters
    private Timer gameOverAnimationTimer;
    private float gameOverAlpha = 1.0f;
    private boolean alphaIncreasing = false;
    private double gameOverScale = 1.0;
    private boolean scaleIncreasing = true;

    // Reference to side panel and preview panel
    private TetrisSidePanel sidePanel;
    private TetrisPreview previewPanel;

    public TetrisGameArea(int gridColumns, int gridRows, int cellSize, Queue<TetrisBlock> pieceQueue) {
        this.gridColumns = gridColumns;
        this.gridRows = gridRows;
        this.cellSize = cellSize;
        this.pieceQueue = pieceQueue;
        setBackground(TetrisColors.COLORS[0]);
        setPreferredSize(new Dimension(cellSize * gridColumns, cellSize * gridRows));

        initGame();
        setupKeyboardControls();
        startGame();
    }
```

```
50      public void setSidePanel(TetrisSidePanel sp) {
51          this.sidePanel = sp;
52          this.previewPanel = sp.getPreviewPanel();
53      }
54
55      private void initGame() {
56          boardStack = new Stack<>();
57          for (int i = 0; i < gridRows; i++) {
58              int[] row = new int[gridColumns];
59              Arrays.fill(row, 0);
60              boardStack.add(0, row);
61          }
62          pieceQueue.clear();
63          // Enqueue three tetrominoes
64          for (int i = 0; i < 3; i++) {
65              pieceQueue.add(TetrisBlock.random(gridColumns));
66          }
67          currentPiece = pieceQueue.poll();
68      }
69
70      private void startGame() {
71          gameTimer = new Timer(getFallSpeed(), e -> gameUpdate());
72          gameTimer.start();
73      }
74
75
76      private int getFallSpeed() {
77          return Math.max(50, 1000 - (level * 75));
78      }
```

```
80      private void setupKeyboardControls() {
81          InputMap im = getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW);
82          ActionMap am = getActionMap();
83
84          im.put(KeyStroke.getKeyStroke("LEFT"), "left");
85          im.put(KeyStroke.getKeyStroke("RIGHT"), "right");
86          im.put(KeyStroke.getKeyStroke("UP"), "rotate");
87          im.put(KeyStroke.getKeyStroke("DOWN"), "drop");
88          im.put(KeyStroke.getKeyStroke("R"), "restart");
89
90          am.put("left", new AbstractAction() {
91              public void actionPerformed(ActionEvent e) { processControl("left"); }
92          });
93          am.put("right", new AbstractAction() {
94              public void actionPerformed(ActionEvent e) { processControl("right"); }
95          });
96          am.put("rotate", new AbstractAction() {
97              public void actionPerformed(ActionEvent e) { processControl("rotate"); }
98          });
99          am.put("drop", new AbstractAction() {
100             public void actionPerformed(ActionEvent e) { processControl("drop"); }
101         });
102         am.put("restart", new AbstractAction() {
103             public void actionPerformed(ActionEvent e) {
104                 if (gameOver) resetGame();
105             }
106         });
107     }
108
109     // Unified control processing (used by both keyboard and button actions)
110     public void processControl(String cmd) {
111         if (gameOver) return;
112         switch (cmd) {
113             case "left":
114                 tryMove(currentPiece, -1, 0);
115                 break;
116             case "right":
117                 tryMove(currentPiece, 1, 0);
118                 break;
119             case "rotate":
120                 rotatePiece();
121                 break;
122             case "drop":
123                 while (tryMove(currentPiece, 0, 1)) { }
124                 break;
125         }
126         repaint();
127     }
```

```
    private void rotatePiece() {
        int rows = currentPiece.matrix.length;
        int cols = currentPiece.matrix[0].length;
        int[][] rotated = new int[cols][rows];
        for (int y = 0; y < rows; y++) {
            for (int x = 0; x < cols; x++) {
                rotated[x][rows - 1 - y] = currentPiece.matrix[y][x];
            }
        }
        TetrisBlock testPiece = new TetrisBlock(rotated, currentPiece.colorIndex);
        testPiece.x = currentPiece.x;
        testPiece.y = currentPiece.y;
        if (!collision(testPiece, 0, 0)) {
            currentPiece.matrix = rotated;
        }
    }

    // ----------------------------
    //    Main Game Loop
    // ----------------------------
    private void gameUpdate() {
        if (!tryMove(currentPiece, 0, 1)) {
            lockPiece();
            clearCompletedRows();
            spawnPiece();
        }
        repaint();
        previewPanel.repaint();
//        if (previewPanel != null) previewPanel.repaint();
    }

    private void lockPiece() {
        for (int y = 0; y < currentPiece.matrix.length; y++) {
            for (int x = 0; x < currentPiece.matrix[y].length; x++) {
                if (currentPiece.matrix[y][x] != 0) {
                    int boardX = currentPiece.x + x;
                    int boardY = currentPiece.y + y;
                    if (boardY >= 0 && boardY < gridRows && boardX >= 0 && boardX < gridColumns) {
                        boardStack.get(boardY)[boardX] = currentPiece.colorIndex;
                    }
                }
            }
        }
    }
```

```
174     private void clearCompletedRows() {
175         int linesCleared = 0;
176         for (int i = gridRows - 1; i >= 0; i--) {
177             boolean full = true;
178             for (int cell : boardStack.get(i)) {
179                 if (cell == 0) { full = false; break; }
180             }
181             if (full) {
182                 boardStack.remove(i);
183                 int[] emptyRow = new int[gridColumns];
184                 Arrays.fill(emptyRow, 0);
185                 boardStack.add(0, emptyRow);
186                 linesCleared++;
187                 i++; // recheck same index after shift
188             }
189         }
190         if (linesCleared > 0) {
191             score += linesCleared * 100 * level;
192             level = 1 + score / 1000;
193             gameTimer.setDelay(getFallSpeed());
194             if (sidePanel != null) {
195                 sidePanel.updateStatus(score, level);
196             }
197         }
198     }
199 
200     private void spawnPiece() {
201         currentPiece = pieceQueue.poll();
202         pieceQueue.add(TetrisBlock.random(gridColumns));
203         if (previewPanel != null) previewPanel.repaint();
204         if (collision(currentPiece, 0, 0) || isTopRowFilled()) {
205             triggerGameOver();
206         }
207     }
208 
209 //    private boolean isRowFull(int[] row) {
210 //        for (int cell : row) {
211 //            if (cell == 0) return false;
212 //        }
213 //        return true;
214 //    }
215 
216     private boolean isTopRowFilled() {
217         int[] topRow = boardStack.get(0);
218         for (int cell : topRow) {
219             if (cell != 0) return true;
220         }
221         return false;
222     }
```

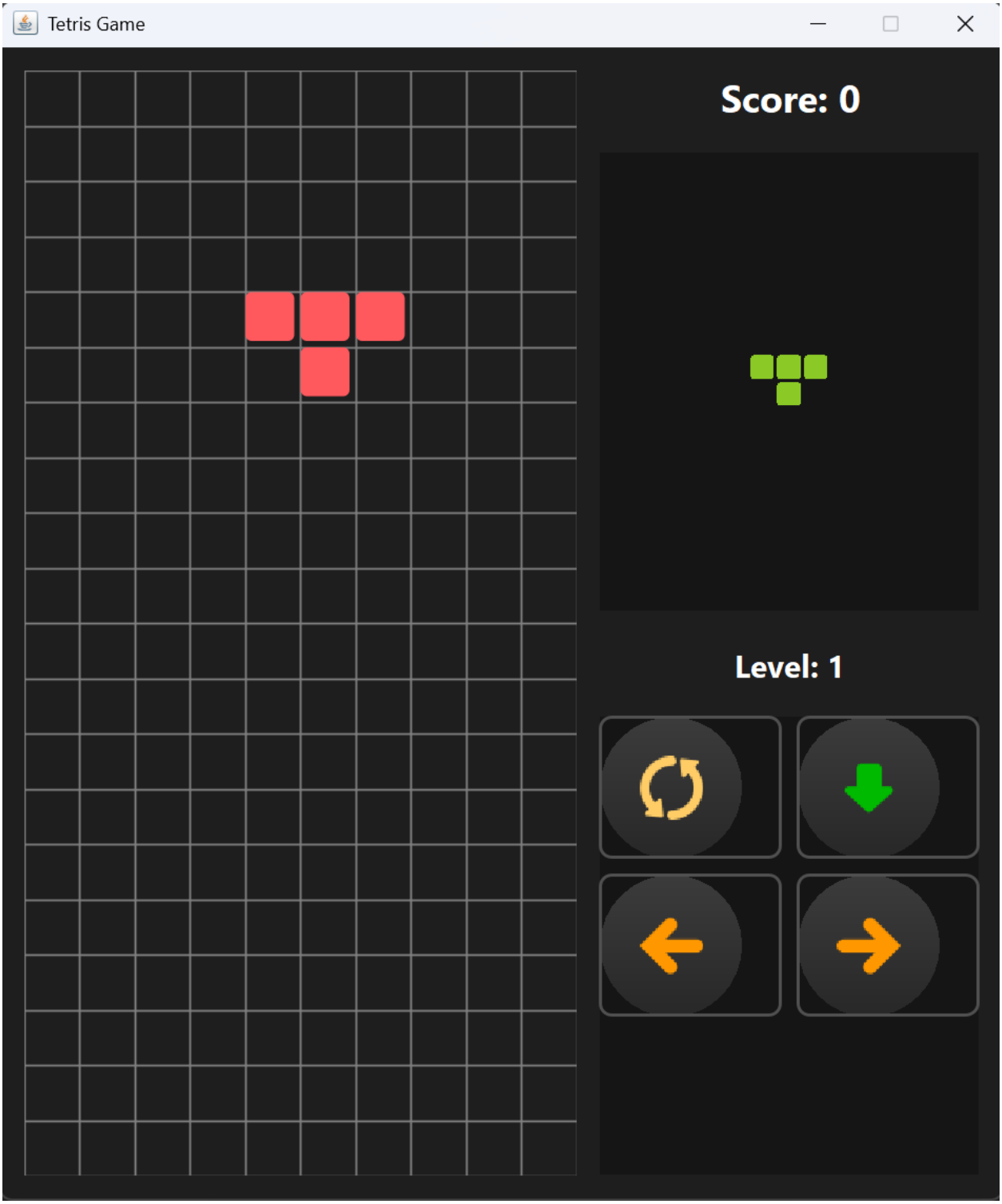
```
225         // ----------------------------
226         //    Collision & Movement
227         // ----------------------------
228         private boolean tryMove(TetrisBlock piece, int dx, int dy) {
229             if (!collision(piece, dx, dy)) {
230                 piece.move(dx, dy);
231                 return true;
232             }
233             return false;
234         }
235 
236         private boolean collision(TetrisBlock piece, int dx, int dy) {
237             for (int y = 0; y < piece.matrix.length; y++) {
238                 for (int x = 0; x < piece.matrix[y].length; x++) {
239                     if (piece.matrix[y][x] != 0) {
240                         int newX = piece.x + x + dx;
241                         int newY = piece.y + y + dy;
242                         if (newX < 0 || newX >= gridColumns || newY >= gridRows) return true;
243                         if (newY >= 0 && boardStack.get(newY)[newX] != 0) return true;
244                     }
245                 }
246             }
247             return false;
248         }
249 
250         // ----------------------------
251         //   Game Over & Animation
252         // ----------------------------
253         private void triggerGameOver() {
254             gameTimer.stop();
255             gameOver = true;
256             startGameOverAnimation();
257         }
258 
259         private void startGameOverAnimation() {
260             gameOverAlpha = 1.0f;
261             alphaIncreasing = false;
262             gameOverScale = 1.0;
263             scaleIncreasing = true;
264             gameOverAnimationTimer = new Timer(50, e -> updateGameOverAnimation());
265             gameOverAnimationTimer.start();
266         }
```

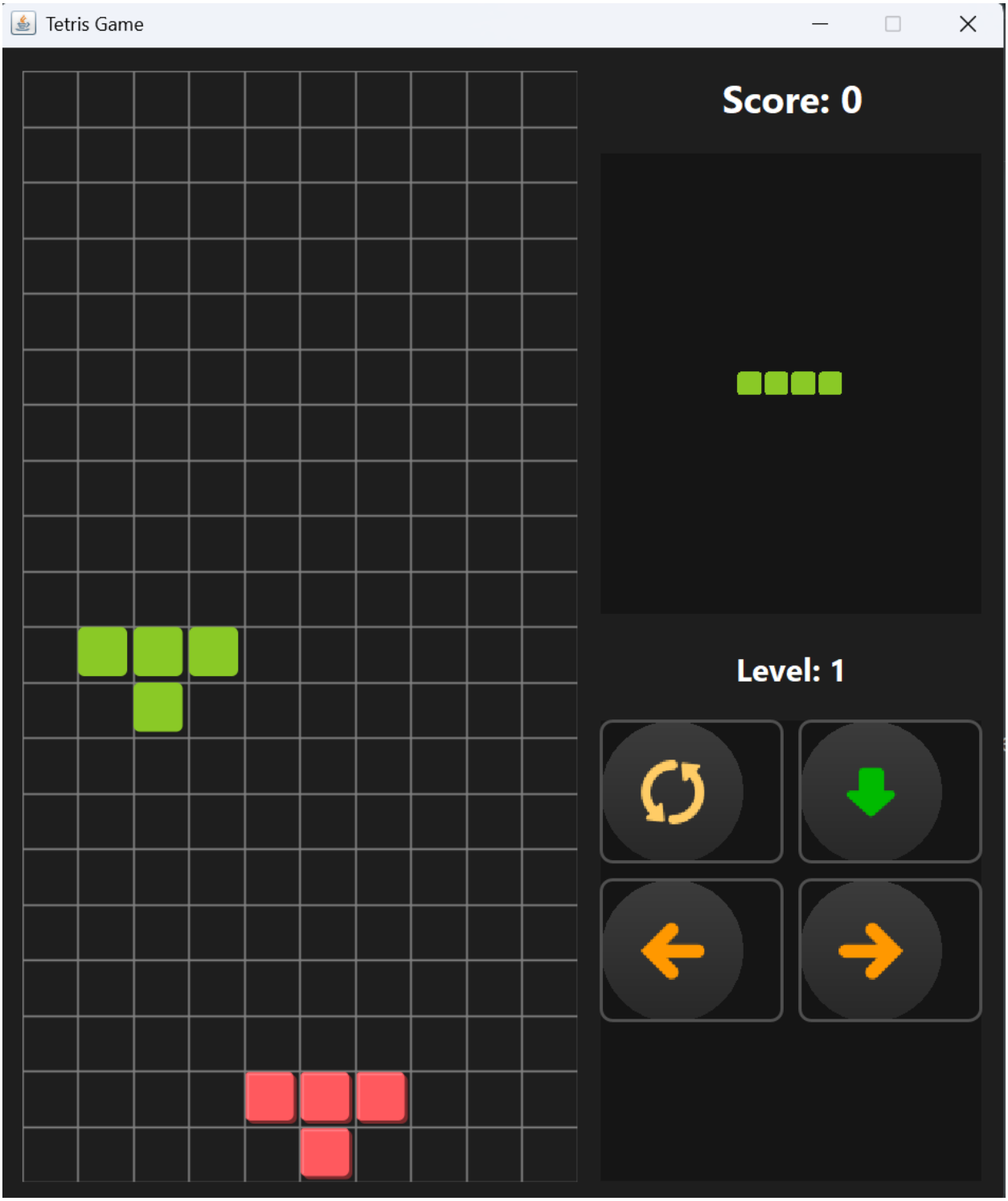
```
268     private void updateGameOverAnimation() {
269         // Oscillate alpha between 0.5 and 1.0
270         if (alphaIncreasing) {
271             gameOverAlpha += 0.05f;
272             if (gameOverAlpha >= 1.0f) {
273                 gameOverAlpha = 1.0f;
274                 alphaIncreasing = false;
275             }
276         } else {
277             gameOverAlpha -= 0.05f;
278             if (gameOverAlpha <= 0.5f) {
279                 gameOverAlpha = 0.5f;
280                 alphaIncreasing = true;
281             }
282         }
283         // Oscillate scale between 1.0 and 1.2
284         if (scaleIncreasing) {
285             gameOverScale += 0.01;
286             if (gameOverScale >= 1.2) {
287                 gameOverScale = 1.2;
288                 scaleIncreasing = false;
289             }
290         } else {
291             gameOverScale -= 0.01;
292             if (gameOverScale <= 1.0) {
293                 gameOverScale = 1.0;
294                 scaleIncreasing = true;
295             }
296         }
297         this.repaint();
298     }
299 
300     private void resetGame() {
301         gameOver = false;
302         score = 0;
303         level = 1;
304         initGame();
305         gameTimer.setDelay(getFallSpeed());
306         gameTimer.start();
307         if (sidePanel != null) sidePanel.updateResetStatus(score, level, this);
308     }
309 
310     @Override
312     protected void paintComponent(Graphics g) {
313         super.paintComponent(g);
314         Graphics2D g2 = (Graphics2D) g;
315         g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
316         drawBoard(g2);
317         drawCurrentPiece(g2);
318         if (gameOver) drawGameOver(g2);
    }
```

```
320     private void drawBoard(Graphics2D g2) {
321         for (int y = 0; y < boardStack.size(); y++) {
322             int[] row = boardStack.get(y);
323             for (int x = 0; x < gridColumns; x++) {
324                 drawCell(g2, x, y, row[x]);
325                 g2.setColor(Color.gray);
326                 g2.drawRect(x * cellSize, y * cellSize, cellSize, cellSize);
327             }
328         }
329     }
330 
331     private void drawCurrentPiece(Graphics2D g2) {
332         currentPiece.draw(g2, cellSize);
333     }
334 
335     private void drawCell(Graphics2D g2, int x, int y, int colorIdx) {
336         if (colorIdx == 0) return;
337         int xPos = x * cellSize;
338         int yPos = y * cellSize;
339         Color color = TetrisColors.COLORS[colorIdx];
340         // Draw cell shadow.
341         g2.setColor(color.darker().darker());
342         g2.fillRoundRect(xPos + 2, yPos + 2, cellSize - 4, cellSize - 4, 8, 8);
343         // Draw cell body.
344         g2.setColor(color);
345         g2.fillRoundRect(xPos, yPos, cellSize - 4, cellSize - 4, 8, 8);
346         // Draw highlight.
347         g2.setColor(color.brighter());
348         g2.drawLine(xPos + 2, yPos + 2, xPos + cellSize - 6, yPos + 2);
349         g2.drawLine(xPos + 2, yPos + 2, xPos + 2, yPos + cellSize - 6);
350     }
```

```
352         private void drawGameOver(Graphics2D g2) {
353             // Draw semi-transparent overlay.
354             g2.setColor(new Color(0, 0, 0, 200));
355             g2.fillRect(0, 0, getWidth(), getHeight());
356             String text = "GAME OVER";
357             Font baseFont = new Font("Monospaced", Font.BOLD, 75);
358             AffineTransform old = g2.getTransform();
359             AffineTransform at = new AffineTransform();
360             at.translate(getWidth() / 1.3, getHeight() / 1.5);
361             at.scale(gameOverScale, gameOverScale);
362             g2.setTransform(at);
363             g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, gameOverAlpha));
364             FontRenderContext frc = g2.getFontRenderContext();
365             TextLayout layout = new TextLayout(text, baseFont, frc);
366             Shape textShape = layout.getOutline(null);
367             Rectangle2D textBounds = textShape.getBounds2D();
368             double textWidth = textBounds.getWidth();
369             double textHeight = textBounds.getHeight();
370             AffineTransform centerText = AffineTransform.getTranslateInstance(-textWidth / 2.0, textHeight / 2.0);
371             textShape = centerText.createTransformedShape(textShape);
372             float topY = (float) textBounds.getY();
373             float bottomY = (float) (textBounds.getY() + textBounds.getHeight());
374             GradientPaint textGradient = new GradientPaint(
375                 0, topY,
376                 new Color(255, 165, 0),
377                 0, bottomY,
378                 new Color(160, 50, 200)
379             );
380             g2.setPaint(textGradient);
381             g2.fill(textShape);
382             g2.setStroke(new BasicStroke(2.0f));
383             g2.setColor(Color.black);
384             g2.draw(textShape);
385             g2.setTransform(old);
386             g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 1.0f));
387             String restartText = "Press R to restart";
388             g2.setFont(new Font("Monospaced", Font.PLAIN, 18));
389             FontMetrics fm = g2.getFontMetrics();
390             int textW = fm.stringWidth(restartText);
391             int x = (getWidth() - textW) / 2;
392             int y = (int) (getHeight() / 2.0 + 80);
393             g2.setColor(Color.WHITE);
394             g2.drawString(restartText, x, y);
395         }
396     }
```

Output




Tetris Game
Score: 0
Level: 1

4. a) Ans:

The main goal is to return the result table ordered by count of hashtag which is in descending order. An input data (in table) is passed with **userId**, **tweetId**, **tweetDate**, **tweetText** where **userId** and **tweetId** are primary keys which uniquely identify the user and tweet respectively.

Approach

To solve this, we use file handling to persist auto-incrementing primary keys i.e., userId and tweetId so that every new tweet gets a unique ID even when the program restarts. Each tweet is represented by a **Tweet** object that extracts hashtags from its text. The program then prints the tweets in a table and computes the top 3 trending hashtags for February 2024 by counting every hashtag occurrence. When two hashtags have the same count, they are sorted alphabetically in ascending order.

Algorithm and Steps

- Initialization a class **PrimaryKeyGenerator** which reads from a file ("ids.txt") that contains two lines in the format "userId: <value>" and "tweetId: <value>". If the file exists, it initializes the counters from it. Otherwise, it starts with 0.
- Start a loop where each time a new **Tweet** is created, the generator returns the next available userId and tweetId, increments the counters and updates the file.
- Initialize a class **Tweet**. When a **Tweet** object is created, it uses the **PrimaryKeyGenerator** to get unique IDs.
- Declare a **printInputTable()** method which formats and prints each tweet along with its details (IDs, text, hashtags, and date) in an table.
- Declare a method **findTopTrendingHashtags()** which performs:
  - It filters the tweets to only those dated in February 2024.
  - It then counts every occurrence of each hashtag with duplicates ones using a HashMap.
  - The hashtags are sorted by count in descending order. if two hashtags have the same count, they are sorted alphabetically in ascending order.
  - Hashtags with maximum count are returned.

- Finally, the top 3 trending hashtags and their counts are printed in another output table.

Time and Space Complexity

1. Time Complexity:

- For **PrimaryKeyGenerator**, reading and writing the file is constant time (**O(1)**) since the file is very small.
- For **Tweet**, extracting hashtags from a tweet takes O(m), where m is the length of the tweet.
- For **findTopTrendingHashtags**:
  - Filtering and counting hashtags over n tweets takes **O(n * m)** time in the worst case.
  - Sorting the unique hashtags takes **O(k log k)**, where **k** is the number of unique hashtags.

Overall time complexity is **O(n * m + k log k)** in worst case.

2. Space Complexity:

- The tweet objects and their extracted hashtags require **O(n * m)** space.
- The counting HashMap uses **O(k)** space.

Overall space complexity is **O(n * m + k)** in worst case.

Code Implementation

```java
import java.time.LocalDate;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.io.*;

// This class manages auto-increment primary keys by reading from/writing to a text file.
class PrimaryKeyGenerator {  2 usages
    private static final String FILE_NAME = "ids.txt";  2 usages
    private static int currentUserId;  6 usages
    private static int currentTweetId;  6 usages
```

```java
    // Static initializer: load keys from file if exists; otherwise, set default values.
    static {
        File file = new File(FILE_NAME);
        if (file.exists()) {
            try (Scanner scanner = new Scanner(file)) {
                if (scanner.hasNextLine()) {
                    // Expecting a line like "userId: 135"
                    String userLine = scanner.nextLine().trim();
                    currentUserId = Integer.parseInt(userLine.split( regex: ":")[1].trim());
                }
                if (scanner.hasNextLine()) {
                    // Expecting a line like "tweetId: 13"
                    String tweetLine = scanner.nextLine().trim();
                    currentTweetId = Integer.parseInt(tweetLine.split( regex: ":")[1].trim());
                }
            }
            catch (Exception e) {
                System.err.println("Error reading primary keys. Using default values.");
                currentUserId = 0;
                currentTweetId = 0;
            }
        }
        else {
            currentUserId = 0;
            currentTweetId = 0;
        }
    }
```

```
41      // Updates the file with the current keys in the format:
42      // userId: <currentUserId>
43      // tweetId: <currentTweetId>
44      private static void updateFile() {  2 usages
45          try (FileWriter writer = new FileWriter(FILE_NAME,  append: false)) {
46              writer.write( str: "userId: " + currentUserId + "\n");
47              writer.write( str: "tweetId: " + currentTweetId);
48          } catch (IOException e) {
49              System.err.println("Error updating primary keys file: " + e.getMessage());
50          }
51      }
52
53      public static synchronized int getNextUserId() {  1 usage
54          int id = currentUserId;
55          currentUserId++;
56          updateFile();
57          return id;
58      }
59
60      public static synchronized int getNextTweetId() {  1 usage
61          int id = currentTweetId;
62          currentTweetId++;
63          updateFile();
64          return id;
65      }
66  }
```

```java
class Tweet {  12 usages
    private final int userId;  2 usages
    private final int tweetId;  2 usages
    private final LocalDate tweetDate;  2 usages
    private final String tweet;  2 usages

    // Store ALL extracted hashtags (including duplicates if they appear).
    private final List<String> hashtags;  3 usages

    // Constructor uses PrimaryKeyGenerator to assign unique IDs.
    public Tweet(LocalDate tweetDate, String tweet) {  7 usages
        this.userId = PrimaryKeyGenerator.getNextUserId();
        this.tweetId = PrimaryKeyGenerator.getNextTweetId();
        this.tweetDate = tweetDate;
        this.tweet = tweet;

        // Extract hashtags from the tweet text.
        this.hashtags = new ArrayList<>();
        Pattern pattern = Pattern.compile( regex: "#\\w+");
        Matcher matcher = pattern.matcher(tweet);
        while (matcher.find()) {
            this.hashtags.add(matcher.group());
        }
    }
```

```
93      public int getUserId() {  1 usage
94          return userId;
95      }
96      public int getTweetId() {  1 usage
97          return tweetId;
98      }
99      public LocalDate getTweetDate() {  2 usages
100         return tweetDate;
101     }
102     public String getTweet() {  1 usage
103         return tweet;
104     }
105     public List<String> getHashtags() {  2 usages
106         return hashtags;
107     }
108 }
109
110 public class TopTrendingHashtags {
```

```java
public class TopTrendingHashtags {
    /**
     * Prints the input tweets in an ASCII table with columns:
     * user_id, tweet_id, tweet, hashtags, tweet_date.
     */
    private static void printInputTable(List<Tweet> tweets) {  1 usage
        int userIdWidth = 7;     // "user_id"
        int tweetIdWidth = 8;    // "tweet_id"
        int tweetWidth = 80;     // "tweet"
        int tagsWidth = 50;      // "hashtags"
        int dateWidth = 10;      // "tweet_date"

        String headerLine = "+"
                + repeat( c: '-',  times: userIdWidth + 2) + "+"
                + repeat( c: '-',  times: tweetIdWidth + 2) + "+"
                + repeat( c: '-',  times: tweetWidth + 2) + "+"
                + repeat( c: '-',  times: tagsWidth + 2) + "+"
                + repeat( c: '-',  times: dateWidth + 2) + "+";

        System.out.println(headerLine);
        System.out.printf("| %-" + userIdWidth + "s | %-" + tweetIdWidth + "s | %-"
                        + tweetWidth + "s | %-" + tagsWidth + "s | %-" + dateWidth + "s |\n",
                "user_id", "tweet_id", "tweet", "hashtags", "tweet_date");
        System.out.println(headerLine);
```

```java
        for (Tweet t : tweets) {
            String hashtagsStr = String.join( delimiter: ",", t.getHashtags());
            System.out.printf("| %-" + userIdWidth + "d | %-" + tweetIdWidth + "d | %-" +
                            tweetWidth + "s | %-" + tagsWidth + "s | %-" + dateWidth + "s |\n",
                    t.getUserId(),
                    t.getTweetId(),
                    t.getTweet(),
                    hashtagsStr,
                    t.getTweetDate().toString());
        }
        System.out.println(headerLine);
    }
```

```java
    /**
     * Finds the top 3 trending hashtags for February 2024.
     * This counts EVERY occurrence of a hashtag (duplicates included).
     * When counts tie, hashtags are sorted in ascending alphabetical order.
     * Returns a list of [hashtag, count] pairs.
     */
    public List<List<String>> findTopTrendingHashtags(List<Tweet> tweets) {  1 usage
        Map<String, Integer> hashtagCounts = new HashMap<>();

        for (Tweet tweet : tweets) {
            LocalDate date = tweet.getTweetDate();
            if (date.getYear() == 2024 && date.getMonthValue() == 2) {
                for (String hashtag : tweet.getHashtags()) {
                    hashtagCounts.put(hashtag, hashtagCounts.getOrDefault(hashtag,  defaultValue: 0) + 1);
                }
            }
        }

        List<Map.Entry<String, Integer>> sortedList = new ArrayList<>(hashtagCounts.entrySet());
        sortedList.sort((a, b) -> {
            int countCmp = b.getValue().compareTo(a.getValue());
            if (countCmp == 0) {
                return a.getKey().compareTo(b.getKey());
            }
            return countCmp;
        });
```

```java
        List<List<String>> result = new ArrayList<>();
        int limit = 3;
        for (Map.Entry<String, Integer> entry : sortedList) {
            if (limit-- == 0) break;
            result.add(Arrays.asList(entry.getKey(), String.valueOf(entry.getValue())));
        }
        return result;
    }

    /**
     * Prints the top hashtags in an ASCII table with columns: Hashtag, Count.
     */
    private static void printHashtagsTable(List<List<String>> topHashtags) {  1 usage
        String line = "+----------------------+-------+";
        System.out.println(line);
        System.out.printf("| %-20s | %-5s |\n", "Hashtag", "Count");
        System.out.println(line);
        for (List<String> row : topHashtags) {
            System.out.printf("| %-20s | %-5s |\n", row.get(0), row.get(1));
        }
        System.out.println(line);
    }
```

```
199 @    private static String repeat(char c, int times) {  5 usages
200          StringBuilder sb = new StringBuilder(times);
201          for (int i = 0; i < times; i++) {
202              sb.append(c);
203          }
204          return sb.toString();
205      }
206
207 ▶    public static void main(String[] args) {
208          // Create tweets using auto-generated userId and tweetId from file.
209          List<Tweet> tweets = Arrays.asList(
210              new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 1),
211                      tweet: "Enjoying a great start to the day. #HappyDay #MorningVibes #HappyDay"),
212              new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 3),
213                      tweet: "Another #HappyDay with good vibes! #FeelGood"),
214              new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 4),
215                      tweet: "Productivity peaks! #WorkLife #ProductiveDay"),
216              new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 5),
217                      tweet: "Exploring new tech frontiers. #TechLife #Innovation"),
218              new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 5),
219                      tweet: "Gratitude for today's moments. #HappyDay #Thankful"),
220              new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 7),
221                      tweet: "Innovation drives us. #TechLife #FutureTech"),
222              new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 8),
223                      tweet: "Connecting with nature's serenity. #Nature #Peaceful")
224          );
```

```
226          // 1) Print the input table with extracted hashtags.
227          System.out.println("Input Tweets Table:\n");
228          printInputTable(tweets);
229
230          // 2) Compute top trending hashtags in February 2024.
231          TopTrendingHashtags solver = new TopTrendingHashtags();
232          List<List<String>> topHashtags = solver.findTopTrendingHashtags(tweets);
233
234          // 3) Print the output table of top hashtags.
235          System.out.println("\nTop 3 Trending Hashtags (February 2024):\n");
236          printHashtagsTable(topHashtags);
237      }
238
239  }
```

Example-1

Input:

```
// Create tweets using auto-generated userId and tweetId from file.
List<Tweet> tweets = Arrays.asList(
        new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 1),
                tweet: "Enjoying a great start to the day. #HappyDay #MorningVibes #HappyDay"),
        new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 3),
                tweet: "Another #HappyDay with good vibes! #FeelGood"),
        new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 4),
                tweet: "Productivity peaks! #WorkLife #ProductiveDay"),
        new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 5),
                tweet: "Exploring new tech frontiers. #TechLife #Innovation"),
        new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 5),
                tweet: "Gratitude for today's moments. #HappyDay #Thankful"),
        new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 7),
                tweet: "Innovation drives us. #TechLife #FutureTech"),
        new Tweet(LocalDate.of( year: 2024,  month: 2,  dayOfMonth: 8),
                tweet: "Connecting with nature's serenity. #Nature #Peaceful")
);
```

user_id	tweet_id	tweet	hashtags	tweet_date
0	0	Enjoying a great start to the day. #HappyDay #MorningVibes #HappyDay	#HappyDay,#MorningVibes,#HappyDay	2024-02-01
1	1	Another #HappyDay with good vibes! #FeelGood	#HappyDay,#FeelGood	2024-02-03
2	2	Productivity peaks! #WorkLife #ProductiveDay	#WorkLife,#ProductiveDay	2024-02-04
3	3	Exploring new tech frontiers. #TechLife #Innovation	#TechLife,#Innovation	2024-02-05
4	4	Gratitude for today's moments. #HappyDay #Thankful	#HappyDay,#Thankful	2024-02-05
5	5	Innovation drives us. #TechLife #FutureTech	#TechLife,#FutureTech	2024-02-07
6	6	Connecting with nature's serenity. #Nature #Peaceful	#Nature,#Peaceful	2024-02-08

Output:

```
Top 3 Trending Hashtags (February 2024):

+----------------------+-------+
| Hashtag              | Count |
+----------------------+-------+
| #HappyDay            | 4     |
| #TechLife            | 2     |
| #FeelGood            | 1     |
+----------------------+-------+
```

4. b) Ans:

The goal is to find the minimum number of roads i.e., **roads** needed to traverse to collect all packages i.e., **packages**.

Approach

We are given an undirected graph which represents a city of with **n** nodes where each node either has a package (**packages[i] =1**) or not (**packages[i]=0**)

We can start at any node and:

- Collect packages from all nodes within distance 2 (i.e., the current node, its neighbors, and neighbors-of-neighbors).
- Move to an adjacent node (costing 1 traversal of a road).
- Our main goal is to collect all packages and return to the same start node with minimal total road traversals.

To solve the problem correctly and with best optimal way, we can use a BFS approach. Each state is defined as: (**currentNode**, **bitmaskOfCollectedPackages**) where **bitmaskOfCollectedPackages** is a bitmask representing which package-nodes have been collected so far. From each state, we can:

- Collect packages in the radius-2 neighborhood of **currentNode** (updating the bitmask).
- Move to any neighbor of **currentNode**, incurring **+1** to the road count.

We search until all packages are collected (the bitmask indicates we have them all) and we have returned to the starting node. Among all ways to do this, we want the minimum road traversals.

Algorithm and Steps

- We need to identify which nodes have packages where **packages[i] == 1**).
- Build an adjacency list from roads for the undirected graph.

- For each node, precompute a function **collectMask(node)** that returns a bitmask of which package-nodes could be collected if we "stand" on that node, i.e., we can collect the packages from that node within a distance of less than or equal to 2.
- A state is (**node**, **collectedMask**, **startNode**). However, we can store startNode once per BFS run (or we can do a BFS from each possible start node).
- **collectedMask** is an integer bitmask of length p (where p is the number of package-nodes). If the j-th package-node is collected, bit j is 1.
- For each node start in [0 to n−1], do a BFS in the state-space. The initial state is (**start**, **collectMask(start)**) with cost 0. We keep a queue of states and a distance map **dist[state] = cost**.
- From a state (**u**, **mask**) with cost **costSoFar**:
  - Already collect packages from u's radius-2 neighborhood: newMask = mask OR collectMask(u).
  - If newMask is the mask with all packages collected and u==start, we have a valid route back to start. Record costSoFar as a candidate answer.
  - Otherwise, for each neighbor **v** of **u**, we can move to (**v**, **newMask**) with **cost = costSoFar + 1**. If we have not visited (**v**, **newMask**) or found a cheaper cost for it, we push it into the BFS queue.
- Among all BFS runs (one per possible start node) or among all solutions found in a single BFS that allows starting at each node, take the minimum cost that collects all packages and returns to the start.

Time and Space Complexity

1. Time Complexity:

- Precompute coverage for each node: **O(n * n1)** where **n1** is neighbors up to distance 2.
- BFS in state-space: up to **O(n×2p)** states, each with up to d neighbors, so **O(n*2p * d)**.

2. Space Complexity:

- We store an adjacency list for the graph: **O(n + e)** where **e** is edges.
- We store the BFS dist map for up to **$n \times 2^p$** states.
- The **collectMask** array is size **n**.
- Overall **$O(n \times 2^p)$** in the worst case for BFS state and distances, plus adjacency overhead.

Code Implementation

```java
import java.util.*;

public class CollectPackages {
    static int n;                   // number of nodes  8 usages
    static List<Integer>[] adj;     // adjacency list  6 usages
    static int[] packages;          // 0 or 1 per node  1 usage
    static int fullMask;            // bitmask of all packages  2 usages
    static int[] collectMask;       // collectMask[node] = bitmask of packages collectible from node  4 usages

    public static int minRoadsToCollectAll(int[] packages, int[][] roads) {  3 usages
        n = packages.length;
        CollectPackages.packages = packages;

        // Build adjacency
        adj = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            adj[i] = new ArrayList<>();
        }
        for (int[] e : roads) {
            adj[e[0]].add(e[1]);
            adj[e[1]].add(e[0]);
        }

        // Identify which nodes have packages and set up bitmasks
        List<Integer> packageNodes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if (packages[i] == 1) packageNodes.add(i);
        }
        int p = packageNodes.size();   // number of package-nodes
        if (p == 0) return 0;          // no packages, cost is 0
        fullMask = (1 << p) - 1;
```

```
        // Precompute collectMask for each node (which packages can be collected from distance ≤ 2)
        collectMask = new int[n];
        for (int i = 0; i < n; i++) {
            // BFS up to distance 2 from node i
            int mask = 0;
            Queue<int[]> q = new LinkedList<>();
            boolean[] visited = new boolean[n];
            q.offer(new int[]{i, 0}); // (node, dist)
            visited[i] = true;
            while (!q.isEmpty()) {
                int[] cur = q.poll();
                int nd = cur[0], dist = cur[1];
                // If nd is a package-node, set the bit
                if (packages[nd] == 1) {
                    int idx = packageNodes.indexOf(nd);
                    if (idx != -1) {
                        mask |= (1 << idx);
                    }
                }
                if (dist < 2) {
                    for (int nxt : adj[nd]) {
                        if (!visited[nxt]) {
                            visited[nxt] = true;
                            q.offer(new int[]{nxt, dist+1});
                        }
                    }
                }
            }
            collectMask[i] = mask;
        }
```

```
        // Precompute collectMask for each node (which packages can be collected from distance ≤ 2)
        collectMask = new int[n];
        for (int i = 0; i < n; i++) {
            // BFS up to distance 2 from node i
            int mask = 0;
            Queue<int[]> q = new LinkedList<>();
            boolean[] visited = new boolean[n];
            q.offer(new int[]{i, 0}); // (node, dist)
            visited[i] = true;
            while (!q.isEmpty()) {
                int[] cur = q.poll();
                int nd = cur[0], dist = cur[1];
                // If nd is a package-node, set the bit
                if (packages[nd] == 1) {
                    int idx = packageNodes.indexOf(nd);
                    if (idx != -1) {
                        mask |= (1 << idx);
                    }
                }
                if (dist < 2) {
                    for (int nxt : adj[nd]) {
                        if (!visited[nxt]) {
                            visited[nxt] = true;
                            q.offer(new int[]{nxt, dist+1});
                        }
                    }
                }
            }
            collectMask[i] = mask;
        }
```

```
            // Move to neighbors
            for (int nxt : adj[node]) {
                int nxtMask = newMask; // we also collect from next node
                // but let's unify it by collecting only from standing positions
                // so we do not forcibly collect from edges
                long nxtState = encode(nxt, nxtMask);
                if (!dist.containsKey(nxtState)) {
                    dist.put(nxtState, costSoFar + 1);
                    q.offer(new long[]{nxt, nxtMask});
                }
            }
        }
        return Integer.MAX_VALUE; // cannot collect all returning to start
    }

    private static long encode(int node, int mask) {  3 usages
        // Combine node + mask into a single long
        // node in lower bits, mask in upper bits or vice versa
        // but for safety just combine carefully
        return (((long) mask) << 20) | (node & 0xFFFFF);
    }

    // For testing
    public static void main(String[] args) {
        // Example - 1
        int[] packages1 = {1,0,0,0,0,1};
        int[][] roads1 = {{0,1},{1,2},{2,3},{3,4},{4,5}};
        int result1 = minRoadsToCollectAll(packages1, roads1);
        System.out.println("The minimum number of roads needed to traverse: " + result1); // Expect 2

        // Example - 2
        int[] packages2 = {0,0,0,1,1,0,0,1};
        int[][] roads2 = {{0,1},{0,2},{1,3},{1,4},{2,5},{5,6},{5,7}};
        int result2 = minRoadsToCollectAll(packages2, roads2);
        System.out.println("The minimum number of roads needed to traverse: " + result2); // Expect 2


        // Example - 3
        int[] packages3 = {1,0,0,1,0,0};
        int[][] roads3 = {{0,1},{1,2},{2,3},{3,4},{4,5}};
        int result3 = minRoadsToCollectAll(packages3, roads3);
        System.out.println("The minimum number of roads needed to traverse: " + result3); // Expect 0
    }
}
```

Example-1

Input: packages1 = {1,0,0,0,0,1}, roads1 = {{0,1},{1,2},{2,3},{3,4},{4,5}}

```
// Example - 1
int[] packages1 = {1,0,0,0,0,1};
int[][] roads1 = {{0,1},{1,2},{2,3},{3,4},{4,5}};
int result1 = minRoadsToCollectAll(packages1, roads1);
System.out.println("The minimum number of roads needed to traverse: " + result1); // Expect 2
```

Output: 2

```
The minimum number of roads needed to traverse: 2
```

Example-2

Input: packages2 = {0,0,0,1,1,0,0,1}, roads2 = {{0,1},{0,2},{1,3},{1,4},{2,5},{5,6},{5,7}}

```
// Example - 2
int[] packages2 = {0,0,0,1,1,0,0,1};
int[][] roads2 = {{0,1},{0,2},{1,3},{1,4},{2,5},{5,6},{5,7}};
int result2 = minRoadsToCollectAll(packages2, roads2);
System.out.println("The minimum number of roads needed to traverse: " + result2); // Expect 2
```

Output: 2

```
The minimum number of roads needed to traverse: 2
```

Example-3

Input: packages3 = {1,0,0,1,0,0}, roads3 = {{0,1},{1,2},{2,3},{3,4},{4,5}}

```
// Example - 3
int[] packages3 = {1,0,0,1,0,0};
int[][] roads3 = {{0,1},{1,2},{2,3},{3,4},{4,5}};
int result3 = minRoadsToCollectAll(packages3, roads3);
System.out.println("The minimum number of roads needed to traverse: " + result3); // Expect 0
```

Output:

```
The minimum number of roads needed to traverse: 0
```

5. Ans:

Approach

We design a GUI-based network optimizer that aids administrators in visually designing and optimizing network topologies. The network is modeled as an undirected graph where nodes represent servers or clients, and edges represent possible network connections. Each edge is described with two significant attributes: cost and bandwidth. The requirements are twofold:

- Network Optimization: Find a subnetwork that minimizes total cost while reducing the likelihood of information transmission latency. This modified Kruskal algorithm takes each weighted graph edge with respect to a newly introduced combined metric:
  weight=α * cost * β * (LATENCY_FACTOR/bandwidth)
  In this function description, □ and **β** represent user- specified weights.

- Shortest Path Calculation: Users select any two nodes and utilize Dijkstra's algorithm to discover the shortest (lowest latency) path from the first to the second, with each edge weight calculated by:
  LATENCY=LATENCY_FACTOR/ bandwidth.

The application supports interactive node and edge creation (editing and removal), dynamic visualization of the network, real-time display of total cost, and average latency, and implements a dual-functionality GUI with both optimization and pathfinding features.

Algorithms

- Graph Construction and GUI Setup:
  - First, the nodes are created with unique IDs and positions.
  - Edges connect nodes and include attributes: cost and bandwidth.
  - The GraphPanel is created which renders nodes and edges (with gradient fills and labels).

- The ControlPanel is created which offers controls for network optimization (via input fields for α and β), path finding (using combo boxes for start and end nodes), and node management (e.g., removing nodes).
- The ToolPanel is created which allows the user to switch between selection, node creation, and edge creation modes.
- Users can add, edit (double-click to change cost/bandwidth), or remove nodes/edges via mouse and context menus.
- Dragging a node updates its position, and combo boxes are refreshed to reflect current nodes.

- Network Optimization (modified Kruskal):
  - For each edge:

    weight = α×cost+β*(LATENCY_FACTOR / bandwidth)
  - Sort all edges in order of the calculated weights, in ascending order.
  - Using Union-Find (Kruskal's algorithm), select the edges one at a time, such that cycles are avoided until connections are made between all nodes.
  - Include the chosen edges among active ones to be accentuated in the GUI.
- Shortest Path Calculation (Dijkstra):
  - The combo boxes are at the disagreement, allowing selecting a start node and an end node.
  - Each edge's weight is computed as **LATENCY=LATENCY_FACTOR/ bandw**idth.
  - Run Dijkstra’s algorithm on a starting node and backtrack from the end node once the shortest path is found.
  - Visually highlight the edges along the shortest path.
- Real-Time Evaluation and Metrics:
  - In ControlPanel, metrics like total costs incurred while traversing the network and average latencies are continuously updated.
  - The re-rendering of the graph takes place whenever a change is applied to the network, thus allowing users to interactively update the topology and observe the outcome immediately.

Time and Space Complexity

1. Network Optimization (Kruskal's Algorithm):

- Time Complexity:
  - The first step is to sort all edges in **O(e log e)**, where **e** is the number of edges.
  - The time to perform union-find operations will be about **O(e α (v))** where α is the inverse Ackermann function and v is the number of nodes/vertices.
  - Finally, the optimization continues in **O(e log e)**.
- Space Complexity:
  - Storing the list of edges and nodes takes **O(v + e)**.
  - The union-find data structure uses **O(v)** space.

2. Shortest Path (Dijkstra Algorithm):

- Time Complexity:
  - Using a priority queue, Dijkstra's algorithm runs in **O(e log v)**.
- Space Complexity:
  - Requires **O(v)** space for the distance and predecessor maps.

Code Implementation

```java
package com.mycompany.networkoptimizer;

import javax.swing.*;
import javax.swing.border.*;
import com.formdev.flatlaf.FlatDarkLaf;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import java.util.List;
import java.util.concurrent.atomic.AtomicReference;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

/**
 * A refined Network Optimizer application with:
 * - Separate combo boxes for Start and End nodes in path finding
 * - Automatic selection of different nodes if available
 * - A 'Remove Node' feature in ControlPanel
 * - Double-click editing of edge cost/bandwidth
 */
public class NetworkOptimizer extends JFrame {

    // ----------------------------
    // Fields & Constants
    // ----------------------------
    private static final double LATENCY_FACTOR = 1000.0;

    // Data Model
    private final List<Node> nodes = new ArrayList<>();
    private final List<Edge> edges = new ArrayList<>();

    private Node selectedNode;
    private final AtomicReference<Edge> tempEdge = new AtomicReference<>();

    // UI Components
    private GraphPanel graphPanel;
    private ControlPanel controlPanel;
    private ToolPanel toolPanel;
    private JLabel statusBar; // for user messages

    // ----------------------------
    // Constructor & Setup
    // ----------------------------
    public NetworkOptimizer() {
        configureLookAndFeel();
        initComponents();
        configureWindow();
    }
```

```
51      private void configureLookAndFeel() {
52          try {
53              UIManager.setLookAndFeel(new FlatDarkLaf());
54              UIManager.put("Button.arc", 8);
55              UIManager.put("Component.arc", 8);
56          } catch (Exception e) {
57              e.printStackTrace();
58          }
59      }
60
61      private void initComponents() {
62          // Create the main UI panels
63          graphPanel = new GraphPanel(nodes, edges, tempEdge);
64          controlPanel = new ControlPanel(
65              this::handleOptimize,
66              this::handlePathFind,
67              this::deleteNode, // removeNodeHandler
68              nodes
69          );
70          toolPanel = new ToolPanel(this::handleToolChange);
71
72          // Status Bar
73          statusBar = new JLabel("Ready");
74          statusBar.setBorder(new EmptyBorder(5, 10, 5, 10));
75          statusBar.setForeground(Color.WHITE);
76          statusBar.setBackground(new Color(45, 45, 50));
77          statusBar.setOpaque(true);
78
79          // Add mouse interactions for graph
80          setupGraphInteractions();
81
82          // Layout
83          JPanel mainPanel = new JPanel(new BorderLayout());
84          mainPanel.add(toolPanel, BorderLayout.NORTH);
85          mainPanel.add(graphPanel, BorderLayout.CENTER);
86          mainPanel.add(controlPanel, BorderLayout.EAST);
87          mainPanel.add(statusBar, BorderLayout.SOUTH);
88
89          add(mainPanel);
90      }
```

```
92      private void configureWindow() {
93          setPreferredSize(new Dimension(1280, 720));
94          pack();
95          setLocationRelativeTo(null);
96          setDefaultCloseOperation(EXIT_ON_CLOSE);
97          setVisible(true);
98      }
99
100     // --- Interaction Handlers ---
101     private void handleToolChange(Tool tool) {
102         graphPanel.clearPreview();
103         graphPanel.repaint();
104         updateStatus("Tool changed to: " + tool.toString());
105     }
106
107     // ----------------------------
108     // Graph Interaction Setup
109     // ----------------------------
110     private void setupGraphInteractions() {
111         MouseAdapter mouseAdapter = new MouseAdapter() {
112
113             @Override
114             public void mousePressed(MouseEvent e) {
115                 handleMousePress(e);
116             }
117
118             @Override
119             public void mouseReleased(MouseEvent e) {
120                 handleMouseRelease(e);
121             }
122
123             @Override
124             public void mouseDragged(MouseEvent e) {
125                 handleMouseDrag(e);
126             }
127
128             @Override
129             public void mouseMoved(MouseEvent e) {
130                 if (toolPanel.getCurrentTool() == Tool.EDGE_CREATION && tempEdge.get() != null) {
131                     graphPanel.setPreviewPoint(e.getPoint());
132                 }
133             }
134
135             @Override
136             public void mouseClicked(MouseEvent e) {
137                 // Check for double-click on edge label (to edit cost/bandwidth)
138                 if (e.getClickCount() == 2) {
139                     handleEdgeDoubleClick(e);
140                 }
141             }
142         };
```

```java
    // ----------------------------
    // Mouse Handlers
    // ----------------------------
    private void handleMousePress(MouseEvent e) {
        Point p = e.getPoint();
        if (SwingUtilities.isRightMouseButton(e)) {
            // Show context menu
            showContextMenu(p);
            return;
        }

        // Check if user clicked on an existing node
        Optional<Node> clickedNode = nodes.stream()
            .filter(n -> n.getBounds().contains(p))
            .findFirst();

        if (clickedNode.isPresent()) {
            selectedNode = clickedNode.get();
            if (toolPanel.getCurrentTool() == Tool.EDGE_CREATION) {
                // Begin edge creation
                tempEdge.set(new Edge(selectedNode, null, 0, 0));
                graphPanel.setPreviewPoint(p);
                updateStatus("Edge creation: select target node.");
            }
        }
        // Otherwise, if in Add Node mode, create a new node
        else if (toolPanel.getCurrentTool() == Tool.NODE_CREATION) {
            Node newNode = new Node("Node " + (nodes.size() + 1), p.x, p.y);
            nodes.add(newNode);
            controlPanel.refreshCombos(nodes);
            graphPanel.repaint();
            updateStatus("New node created: " + newNode.getId());
        }
    }

    private void handleMouseRelease(MouseEvent e) {
        if (tempEdge.get() != null && selectedNode != null) {
            // If user released over a different node, finalize edge creation
            nodes.stream()
                .filter(n -> n != selectedNode && n.getBounds().contains(e.getPoint()))
                .findFirst()
                .ifPresent(target -> {
                    Edge newEdge = promptForEdgeProperties(selectedNode, target);
                    if (newEdge != null) {
                        edges.add(newEdge);
                        controlPanel.refreshCombos(nodes);
                        updateStatus("New edge created between " +
                            selectedNode.getId() + " and " + target.getId());
                    }
                });
            tempEdge.set(null);
            graphPanel.clearPreview();
        }
        selectedNode = null;
        graphPanel.repaint();
    }
```

```
205     private void handleMouseDrag(MouseEvent e) {
206         if (selectedNode != null && toolPanel.getCurrentTool() == Tool.SELECTION) {
207             // Drag the selected node
208             selectedNode.setPosition(e.getX(), e.getY());
209             graphPanel.repaint();
210             updateStatus("Moving node: " + selectedNode.getId());
211         }
212         else if (toolPanel.getCurrentTool() == Tool.EDGE_CREATION && tempEdge.get() != null) {
213             // Update preview line
214             graphPanel.setPreviewPoint(e.getPoint());
215             graphPanel.repaint();
216         }
217     }
218
219     // Double-click on the midpoint of an edge label to edit cost/bandwidth
220     private void handleEdgeDoubleClick(MouseEvent e) {
221         Point p = e.getPoint();
222         for (Edge edge : edges) {
223             int midX = (edge.source.getX() + edge.target.getX()) / 2;
224             int midY = (edge.source.getY() + edge.target.getY()) / 2;
225             if (p.distance(midX, midY) < 20) {
226                 // user double-clicked near the edge label
227                 editEdgeProperties(edge);
228                 break;
229             }
230         }
231     }
232
233     // ----------------------------
234     // Context Menu for Right-Click
235     // ----------------------------
236     private void showContextMenu(Point p) {
237         JPopupMenu contextMenu = new JPopupMenu();
238
239         // Node deletion
240         nodes.stream()
241             .filter(n -> n.getBounds().contains(p))
242             .findFirst()
243             .ifPresent(node -> {
244                 JMenuItem deleteNode = new JMenuItem("Delete Node");
245                 deleteNode.addActionListener(e -> {
246                     this.deleteNode(node);
247                 });
248                 contextMenu.add(deleteNode);
249             });
250
251         // Edge deletion
252         edges.stream()
253             .filter(edge -> edge.containsPoint(p))
254             .findFirst()
255             .ifPresent(edge -> {
256                 JMenuItem deleteEdge = new JMenuItem("Delete Edge");
257                 deleteEdge.addActionListener(e -> {
258                     this.deleteEdge(edge);
259                 });
260                 contextMenu.add(deleteEdge);
261             });
262
263         contextMenu.show(graphPanel, p.x, p.y);
264     }
```

```
266         // ----------------------------
267         // Editing & Deletion
268         // ----------------------------
269         private Edge promptForEdgeProperties(Node source, Node target) {
270             JPanel panel = new JPanel(new GridLayout(0, 1));
271             JTextField costField = new JTextField("10");
272             JTextField bandwidthField = new JTextField("100");
273 
274             panel.add(new JLabel("Cost:"));
275             panel.add(costField);
276             panel.add(new JLabel("Bandwidth (Mbps):"));
277             panel.add(bandwidthField);
278 
279             int result = JOptionPane.showConfirmDialog(
280                 this, panel, "Edge Properties",
281                 JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE
282             );
283 
284             if (result == JOptionPane.OK_OPTION) {
285                 try {
286                     double cost = Double.parseDouble(costField.getText());
287                     double bandwidth = Double.parseDouble(bandwidthField.getText());
288                     if (cost <= 0 || bandwidth <= 0) {
289                         JOptionPane.showMessageDialog(this,
290                             "Cost and Bandwidth must be positive numbers.",
291                             "Input Error", JOptionPane.ERROR_MESSAGE);
292                         return null;
293                     }
294                     return new Edge(source, target, cost, bandwidth);
295                 }
296                 catch (NumberFormatException ex) {
297                     JOptionPane.showMessageDialog(this, "Invalid numeric values.",
298                         "Error", JOptionPane.ERROR_MESSAGE);
299                 }
300             }
301             return null;
302         }
303 
304         // Double-click editing
305         private void editEdgeProperties(Edge edge) {
306             JPanel panel = new JPanel(new GridLayout(0, 1));
307             JTextField costField = new JTextField(String.valueOf(edge.getCost()));
308             JTextField bandwidthField = new JTextField(String.valueOf(edge.getBandwidth()));
309 
310             panel.add(new JLabel("Edit Cost:"));
311             panel.add(costField);
312             panel.add(new JLabel("Edit Bandwidth (Mbps):"));
313             panel.add(bandwidthField);
314 
315             int result = JOptionPane.showConfirmDialog(
316                 this, panel, "Edit Edge Properties",
317                 JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE
318             );
```

```
320             if (result == JOptionPane.OK_OPTION) {
321                 try {
322                     double cost = Double.parseDouble(costField.getText());
323                     double bandwidth = Double.parseDouble(bandwidthField.getText());
324                     if (cost <= 0 || bandwidth <= 0) {
325                         JOptionPane.showMessageDialog(this,
326                             "Cost and Bandwidth must be positive numbers.",
327                             "Input Error", JOptionPane.ERROR_MESSAGE);
328                         return;
329                     }
330                     edge.setCost(cost);
331                     edge.setBandwidth(bandwidth);
332                     graphPanel.repaint();
333                     updateMetrics();
334                     updateStatus("Edge updated between " + edge.source.getId() + " and " + edge.target.getId());
335                 }
336                 catch (NumberFormatException ex) {
337                     JOptionPane.showMessageDialog(this, "Invalid numeric values.",
338                         "Error", JOptionPane.ERROR_MESSAGE);
339                 }
340             }
341         }
342 
343         // Deletion from context menu or the ControlPanel
344         private void deleteNode(Node node) {
345             nodes.remove(node);
346             edges.removeIf(e -> e.connects(node));
347             controlPanel.refreshCombos(nodes);
348             graphPanel.repaint();
349             updateStatus("Deleted node: " + node.getId());
350         }
351 
352         private void deleteEdge(Edge edge) {
353             edges.remove(edge);
354             graphPanel.repaint();
355             updateStatus("Deleted an edge.");
356         }
```

```
358         // ----------------------------
359         // Optimization & Path Finding
360         // ----------------------------
361         private void handleOptimize(double alpha, double beta) {
362             if (!isGraphConnected()) {
363                 JOptionPane.showMessageDialog(this, "Graph is not connected!",
364                     "Optimization Error", JOptionPane.ERROR_MESSAGE);
365                 updateStatus("Optimization failed: Graph is not connected.");
366                 return;
367             }
368 
369             new SwingWorker<List<Edge>, Void>() {
370                 @Override
371                 protected List<Edge> doInBackground() {
372                     return new KruskalOptimizer().findOptimalNetwork(nodes, edges, alpha, beta);
373                 }
374 
375                 @Override
376                 protected void done() {
377                     try {
378                         List<Edge> optimalEdges = get();
379                         edges.forEach(e -> e.setActive(false));
380                         optimalEdges.forEach(e -> e.setActive(true));
381                         updateMetrics();
382                         graphPanel.repaint();
383                         updateStatus("Optimization complete. Optimal network highlighted.");
384                     }
385                     catch (Exception ex) {
386                         ex.printStackTrace();
387                         updateStatus("Error during optimization.");
388                     }
389                 }
390             }.execute();
391         }
392 
393         private void handlePathFind(Node start, Node end) {
394             new SwingWorker<List<Node>, Void>() {
395                 @Override
396                 protected List<Node> doInBackground() {
397                     return new DijkstraPathFinder().findShortestPath(edges, start, end);
398                 }
```

```
            @Override
            protected void done() {
                try {
                    List<Node> path = get();
                    if (path.isEmpty()) {
                        JOptionPane.showMessageDialog(NetworkOptimizer.this,
                            "No path exists between selected nodes",
                            "Path Finding",
                            JOptionPane.INFORMATION_MESSAGE);
                        updateStatus("Path finding: No path exists.");
                    }
                    else {
                        highlightPath(path);
                        updateStatus("Path found between " + start.getId() + " and " + end.getId());
                    }
                }
                catch (Exception ex) {
                    ex.printStackTrace();
                    updateStatus("Error during path finding.");
                }
            }
        }.execute();
    }

    // ------------------------------
    // Helper Methods
    // ------------------------------
    private void updateMetrics() {
        double totalCost = edges.stream().mapToDouble(Edge::getCost).sum();
        double avgLatency = edges.stream()
            .mapToDouble(e -> (e.getBandwidth() > 0
                ? LATENCY_FACTOR / e.getBandwidth()
                : Double.POSITIVE_INFINITY))
            .average()
            .orElse(0);
        controlPanel.updateMetrics(totalCost, avgLatency);
    }

    private void highlightPath(List<Node> path) {
        edges.forEach(e -> e.setHighlighted(false));
        for (int i = 0; i < path.size() - 1; i++) {
            Node a = path.get(i);
            Node b = path.get(i + 1);
            edges.stream()
                .filter(e -> e.connects(a, b))
                .forEach(e -> e.setHighlighted(true));
        }
        graphPanel.repaint();
    }
```

```
450     private boolean isGraphConnected() {
451         if (nodes.isEmpty()) {
452             return false;
453         }
454         Set<Node> visited = new HashSet<>();
455         Queue<Node> queue = new LinkedList<>();
456         queue.add(nodes.get(0));
457 
458         while (!queue.isEmpty()) {
459             Node current = queue.poll();
460             if (visited.add(current)) {
461                 edges.stream()
462                         .filter(e -> e.connects(current))
463                         .map(e -> e.otherEnd(current))
464                         .forEach(queue::add);
465             }
466         }
467         return visited.size() == nodes.size();
468     }
469 
470     private void updateStatus(String message) {
471         statusBar.setText(message);
472     }
473 
474     // ----------------------------
475     // Main Method
476     // ----------------------------
477     public static void main(String[] args) {
478         SwingUtilities.invokeLater(NetworkOptimizer::new);
479     }
480 
481     // ------------------------------------
482     // Inner Classes: Node, Edge, Optimizers
483     // ------------------------------------
484     static class Node {
485         private final String id;
486         private int x, y;
487         private static final int SIZE = 40;
488 
489         public Node(String id, int x, int y) {
490             this.id = id;
491             this.x = x;
492             this.y = y;
493         }
```

```
495         public Rectangle getBounds() {
496             return new Rectangle(x - SIZE / 2, y - SIZE / 2, SIZE, SIZE);
497         }
498
499         public void setPosition(int x, int y) {
500             this.x = x;
501             this.y = y;
502         }
503
504         public String getId() {
505             return id;
506         }
507
508         public int getX() {
509             return x;
510         }
511
512         public int getY() {
513             return y;
514         }
515     }
516
517     static class Edge {
518         private final Node source, target;
519         private double cost;
520         private double bandwidth;
521         private boolean active;
522         private boolean highlighted;
523
524         public Edge(Node source, Node target, double cost, double bandwidth) {
525             this.source = source;
526             this.target = target;
527             this.cost = cost;
528             this.bandwidth = bandwidth;
529         }
530
531         public boolean containsPoint(Point p) {
532             Line2D line = new Line2D.Float(
533                 source.getX(), source.getY(),
534                 target.getX(), target.getY());
535             return line.ptSegDist(p) < 5;
536         }
```

```
538         public boolean connects(Node node) {
539             return source == node || target == node;
540         }
541
542         public boolean connects(Node a, Node b) {
543             return (source == a && target == b)
544                 || (source == b && target == a);
545         }
546
547         public Node otherEnd(Node node) {
548             return node == source ? target : source;
549         }
550
551         // Getters / Setters
552         public double getCost() {
553             return cost;
554         }
555         public void setCost(double cost) {
556             this.cost = cost;
557         }
558         public double getBandwidth() {
559             return bandwidth;
560         }
561         public void setBandwidth(double bandwidth) {
562             this.bandwidth = bandwidth;
563         }
564
565         public void setActive(boolean active) {
566             this.active = active;
567         }
568         public void setHighlighted(boolean highlighted) {
569             this.highlighted = highlighted;
570         }
571         public boolean isActive() { return active; }
572         public boolean isHighlighted() { return highlighted; }
573     }
```

```java
    static class KruskalOptimizer {
        public List<Edge> findOptimalNetwork(List<Node> nodes, List<Edge> edges,
                                             double alpha, double beta) {
            List<Edge> sortedEdges = new ArrayList<>(edges);
            sortedEdges.sort(Comparator.comparingDouble(e ->
                alpha * e.getCost() + beta * (LATENCY_FACTOR / e.getBandwidth())));

            UnionFind uf = new UnionFind(nodes);
            List<Edge> mst = new ArrayList<>();

            for (Edge edge : sortedEdges) {
                Node a = edge.source;
                Node b = edge.target;
                if (uf.find(a) != uf.find(b)) {
                    uf.union(a, b);
                    mst.add(edge);
                    if (mst.size() == nodes.size() - 1) {
                        break;
                    }
                }
            }
            return mst;
        }
    }

    static class DijkstraPathFinder {
        public List<Node> findShortestPath(List<Edge> edges, Node start, Node end) {
            Map<Node, Double> dist = new HashMap<>();
            Map<Node, Node> prev = new HashMap<>();
            PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingDouble(dist::get));

            dist.put(start, 0.0);
            pq.add(start);

            while (!pq.isEmpty()) {
                Node u = pq.poll();
                for (Edge e : getAdjacentEdges(edges, u)) {
                    Node v = e.otherEnd(u);
                    double weight = (e.getBandwidth() > 0)
                        ? LATENCY_FACTOR / e.getBandwidth()
                        : Double.POSITIVE_INFINITY;
                    double alt = dist.getOrDefault(u, Double.MAX_VALUE) + weight;
                    if (alt < dist.getOrDefault(v, Double.MAX_VALUE)) {
                        dist.put(v, alt);
                        prev.put(v, u);
                        pq.remove(v); // ensure updated priority
                        pq.add(v);
                    }
                }
            }
            return buildPath(prev, end);
        }
```

```
628         private List<Edge> getAdjacentEdges(List<Edge> edges, Node node) {
629             List<Edge> adjacent = new ArrayList<>();
630             for (Edge e : edges) {
631                 if (e.connects(node)) {
632                     adjacent.add(e);
633                 }
634             }
635             return adjacent;
636         }
637 
638         private List<Node> buildPath(Map<Node, Node> prev, Node end) {
639             LinkedList<Node> path = new LinkedList<>();
640             for (Node at = end; at != null; at = prev.get(at)) {
641                 path.addFirst(at);
642             }
643             // If path has only 1 node, there's no actual route
644             return path.size() > 1 ? path : Collections.emptyList();
645         }
646     }
647 
648     static class UnionFind {
649         private final Map<Node, Node> parent = new HashMap<>();
650 
651         public UnionFind(List<Node> nodes) {
652             for (Node n : nodes) {
653                 parent.put(n, n);
654             }
655         }
656 
657         public Node find(Node n) {
658             if (parent.get(n) != n) {
659                 parent.put(n, find(parent.get(n)));
660             }
661             return parent.get(n);
662         }
663 
664         public void union(Node a, Node b) {
665             Node rootA = find(a);
666             Node rootB = find(b);
667             if (rootA != rootB) {
668                 parent.put(rootB, rootA);
669             }
670         }
671     }
```

```
673         // ------------------------------
674         // UI Classes
675         // ------------------------------
676         static class GraphPanel extends JPanel {
677             private static final Color NODE_GRADIENT_START = new Color(100, 200, 255);
678             private static final Color NODE_GRADIENT_END   = new Color(0, 100, 200);
679             private static final Color ACTIVE_EDGE         = new Color(50, 200, 100);
680             private static final Color HIGHLIGHT_EDGE      = new Color(255, 100, 100);
681 
682             private static final Stroke DASHED_STROKE = new BasicStroke(
683                 2, BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL,
684                 0, new float[]{5}, 0);
685 
686             private final List<Node> nodes;
687             private final List<Edge> edges;
688             private final AtomicReference<Edge> tempEdge;
689 
690             private Point previewPoint = null; // for edge preview
691 
692             public GraphPanel(List<Node> nodes, List<Edge> edges, AtomicReference<Edge> tempEdge) {
693                 this.nodes = nodes;
694                 this.edges = edges;
695                 this.tempEdge = tempEdge;
696                 setBackground(new Color(30, 30, 35));
697             }
698 
699             public void setPreviewPoint(Point p) {
700                 this.previewPoint = p;
701                 repaint();
702             }
703 
704             public void clearPreview() {
705                 this.previewPoint = null;
706                 repaint();
707             }
708 
709             @Override
                protected void paintComponent(Graphics g) {
711                 super.paintComponent(g);
712                 Graphics2D g2 = (Graphics2D) g;
713                 g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
714 
715                 paintEdges(g2);
716                 if (previewPoint != null) {
717                     paintPreviewEdge(g2);
718                 }
719                 paintNodes(g2);
720             }
```

```
722         private void paintEdges(Graphics2D g2) {
723             for (Edge e : edges) {
724                 Color color = e.isHighlighted() ? HIGHLIGHT_EDGE
725                         : e.isActive() ? ACTIVE_EDGE : new Color(150, 150, 150);
726                 paintEdge(g2, e, color, e.isHighlighted() ? 4 : e.isActive() ? 3 : 2);
727             }
728         }
729 
730         private void paintEdge(Graphics2D g2, Edge e, Color color, int width) {
731             g2.setColor(color);
732             g2.setStroke(new BasicStroke(width, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND));
733             g2.drawLine(e.source.getX(), e.source.getY(), e.target.getX(), e.target.getY());
734 
735             // Label: cost + latency
736             double latency = (e.getBandwidth() > 0) ? LATENCY_FACTOR / e.getBandwidth() : 0;
737             String label = String.format("$%.1f | %.1fms", e.getCost(), latency);
738 
739             FontMetrics fm = g2.getFontMetrics();
740             Point2D mid = new Point2D.Float(
741                 (e.source.getX() + e.target.getX()) / 2f,
742                 (e.source.getY() + e.target.getY()) / 2f
743             );
744 
745             // Label background
746             g2.setColor(new Color(255, 255, 255, 200));
747             g2.fillRoundRect(
748                 (int) mid.getX() - fm.stringWidth(label) / 2 - 3,
749                 (int) mid.getY() - fm.getHeight() / 2 - 3,
750                 fm.stringWidth(label) + 6,
751                 fm.getHeight() + 6,
752                 8, 8
753             );
754             // Label text
755             g2.setColor(Color.BLACK);
756             g2.drawString(label,
757                 (int) mid.getX() - fm.stringWidth(label) / 2,
758                 (int) mid.getY() + fm.getAscent() / 2);
759         }
760 
761         private void paintPreviewEdge(Graphics2D g2) {
762             if (tempEdge.get() != null && previewPoint != null) {
763                 Node source = tempEdge.get().source;
764                 g2.setColor(Color.YELLOW);
765                 g2.setStroke(DASHED_STROKE);
766                 g2.drawLine(source.getX(), source.getY(), previewPoint.x, previewPoint.y);
767             }
768         }
```

```
770         private void paintNodes(Graphics2D g2) {
771             for (Node n : nodes) {
772                 // Shadow
773                 g2.setColor(new Color(0, 0, 0, 50));
774                 g2.fillOval(n.getX() - 22, n.getY() - 22, 44, 44);
775
776                 // Gradient fill
777                 GradientPaint gradient = new GradientPaint(
778                     n.getX() - 20, n.getY() - 20, NODE_GRADIENT_START,
779                     n.getX() + 20, n.getY() + 20, NODE_GRADIENT_END
780                 );
781                 g2.setPaint(gradient);
782                 g2.fillOval(n.getX() - 20, n.getY() - 20, 40, 40);
783
784                 // Border
785                 g2.setColor(new Color(255, 255, 255, 100));
786                 g2.setStroke(new BasicStroke(2));
787                 g2.drawOval(n.getX() - 20, n.getY() - 20, 40, 40);
788
789                 // Label
790                 g2.setColor(Color.WHITE);
791                 g2.setFont(new Font("Segoe UI", Font.BOLD, 14));
792                 FontMetrics fm = g2.getFontMetrics();
793                 g2.drawString(n.getId(),
794                     n.getX() - fm.stringWidth(n.getId()) / 2,
795                     n.getY() + fm.getAscent() / 4);
796             }
797         }
798     }
799
800     /**
801      * The ControlPanel includes:
802      * - Network Optimization (alpha, beta)
803      * - Path Finding (Start, End combos)
804      * - Node Management (Remove Node)
805      * - Real-time metrics
806      */
807     static class ControlPanel extends JPanel {
808
809         private final JLabel costLabel    = new JLabel("Total Cost: $0.00");
810         private final JLabel latencyLabel = new JLabel("Avg Latency: 0ms");
811
812         private final JComboBox<Node> startCombo = new JComboBox<>();
813         private final JComboBox<Node> endCombo   = new JComboBox<>();
814         private final JComboBox<Node> removeCombo = new JComboBox<>();
815
```

```
816             public ControlPanel(
817                 BiConsumer<Double, Double> optimizeHandler,
818                 BiConsumer<Node, Node> pathHandler,
819                 Consumer<Node> removeNodeHandler,
820                 List<Node> nodes
821             ) {
822                 setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
823                 setBorder(new EmptyBorder(15, 15, 15, 15));
824                 setBackground(new Color(45, 45, 50));
825
826                 // --- Optimization Controls ---
827                 JPanel optimizePanel = createSectionPanel("Network Optimization");
828                 JTextField alphaField = new JTextField("1.0");
829                 JTextField betaField  = new JTextField("1.0");
830
831                 JButton optimizeBtn = createButton("Optimize", e -> {
832                     try {
833                         double alpha = Double.parseDouble(alphaField.getText());
834                         double beta  = Double.parseDouble(betaField.getText());
835                         optimizeHandler.accept(alpha, beta);
836                     } catch (NumberFormatException ex) {
837                         showError("Invalid weights");
838                     }
839                 });
840                 optimizeBtn.setToolTipText("Optimize the network based on the given weights.");
841
842                 optimizePanel.add(createLabelField("Cost Weight (α):", alphaField));
843                 optimizePanel.add(Box.createVerticalStrut(10));
844                 optimizePanel.add(createLabelField("Latency Weight (β):", betaField));
845                 optimizePanel.add(Box.createVerticalStrut(15));
846                 optimizePanel.add(optimizeBtn);
847
848                 // --- Path Finding Controls ---
849                 JPanel pathPanel = createSectionPanel("Path Finding");
850                 startCombo.setRenderer(new NodeRenderer());
851                 endCombo.setRenderer(new NodeRenderer());
852
853                 JButton pathBtn = createButton("Find Path", e -> {
854                     Node start = (Node) startCombo.getSelectedItem();
855                     Node end   = (Node) endCombo.getSelectedItem();
856                     if (start != null && end != null && start != end) {
857                         pathHandler.accept(start, end);
858                     }
859                 });
860                 pathBtn.setToolTipText("Find the shortest path between selected nodes.");
861
862                 pathPanel.add(createLabelCombo("Start:", startCombo));
863                 pathPanel.add(Box.createVerticalStrut(5));
864                 pathPanel.add(createLabelCombo("End:", endCombo));
865                 pathPanel.add(Box.createVerticalStrut(15));
866                 pathPanel.add(pathBtn);
867
868                 // --- Node Management (Remove) ---
869                 JPanel nodeMgmtPanel = createSectionPanel("Node Management");
870                 removeCombo.setRenderer(new NodeRenderer());
```

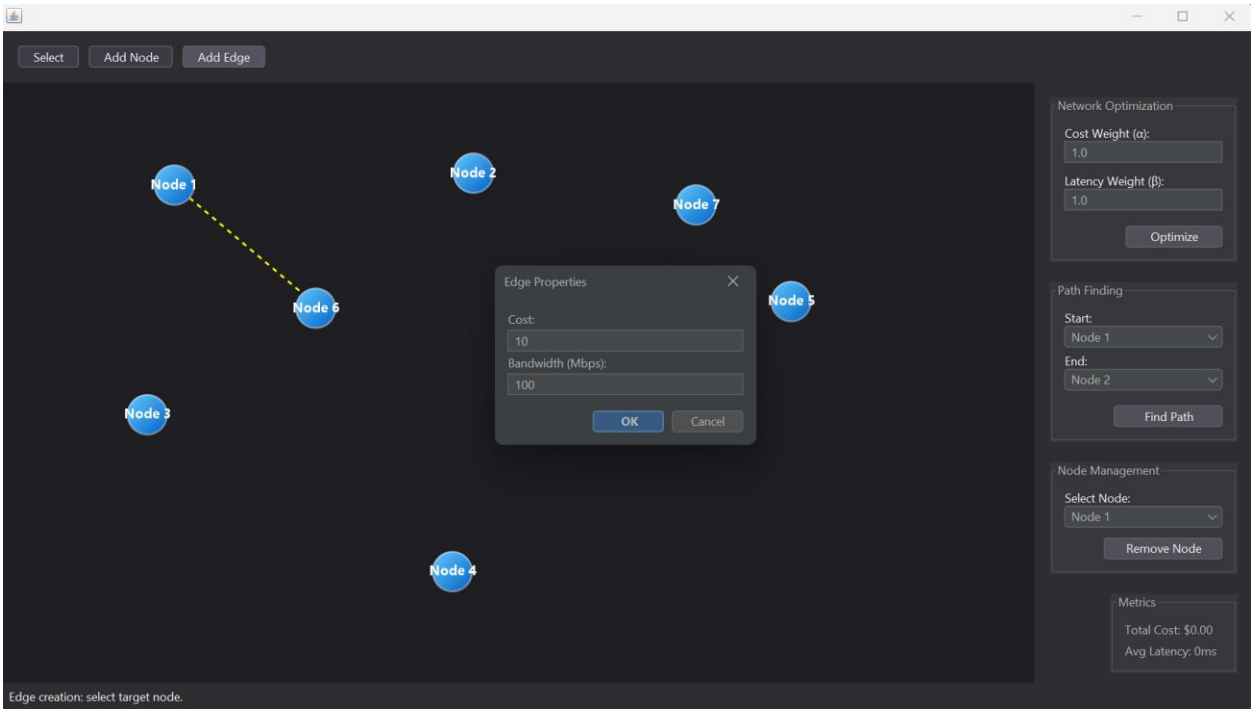
```
872             JButton removeBtn = createButton("Remove Node", e -> {
873                 Node toRemove = (Node) removeCombo.getSelectedItem();
874                 if (toRemove != null) {
875                     removeNodeHandler.accept(toRemove);
876                 }
877             });
878             removeBtn.setToolTipText("Remove the selected node and all connected edges.");
879 
880             nodeMgmtPanel.add(createLabelCombo("Select Node:", removeCombo));
881             nodeMgmtPanel.add(Box.createVerticalStrut(10));
882             nodeMgmtPanel.add(removeBtn);
883 
884             // --- Metrics Display ---
885             JPanel metricsPanel = createSectionPanel("Metrics");
886             metricsPanel.add(costLabel);
887             metricsPanel.add(Box.createVerticalStrut(5));
888             metricsPanel.add(latencyLabel);
889 
890             // Layout
891             add(optimizePanel);
892             add(Box.createVerticalStrut(20));
893             add(pathPanel);
894             add(Box.createVerticalStrut(20));
895             add(nodeMgmtPanel);
896             add(Box.createVerticalStrut(20));
897             add(metricsPanel);
898             add(Box.createVerticalGlue());
899 
900             // Populate combos
901             refreshCombos(nodes);
902         }
903 
904         /**
905          * Refresh all combo boxes (start, end, remove) with the latest list of nodes.
906          * By default, if at least two nodes exist, we select different nodes for start & end.
907          */
908         public void refreshCombos(List<Node> nodes) {
909             // Separate models so the user can choose different combos
910             DefaultComboBoxModel<Node> modelStart  = new DefaultComboBoxModel<>();
911             DefaultComboBoxModel<Node> modelEnd    = new DefaultComboBoxModel<>();
912             DefaultComboBoxModel<Node> modelRemove = new DefaultComboBoxModel<>();
913 
914             for (Node n : nodes) {
915                 modelStart.addElement(n);
916                 modelEnd.addElement(n);
917                 modelRemove.addElement(n);
918             }
919             startCombo.setModel(modelStart);
920             endCombo.setModel(modelEnd);
921             removeCombo.setModel(modelRemove);
922 
923             // If at least 2 nodes, select different items by default
924             if (modelStart.getSize() > 1) {
925                 startCombo.setSelectedIndex(0);
926                 endCombo.setSelectedIndex(1);
927             }
```

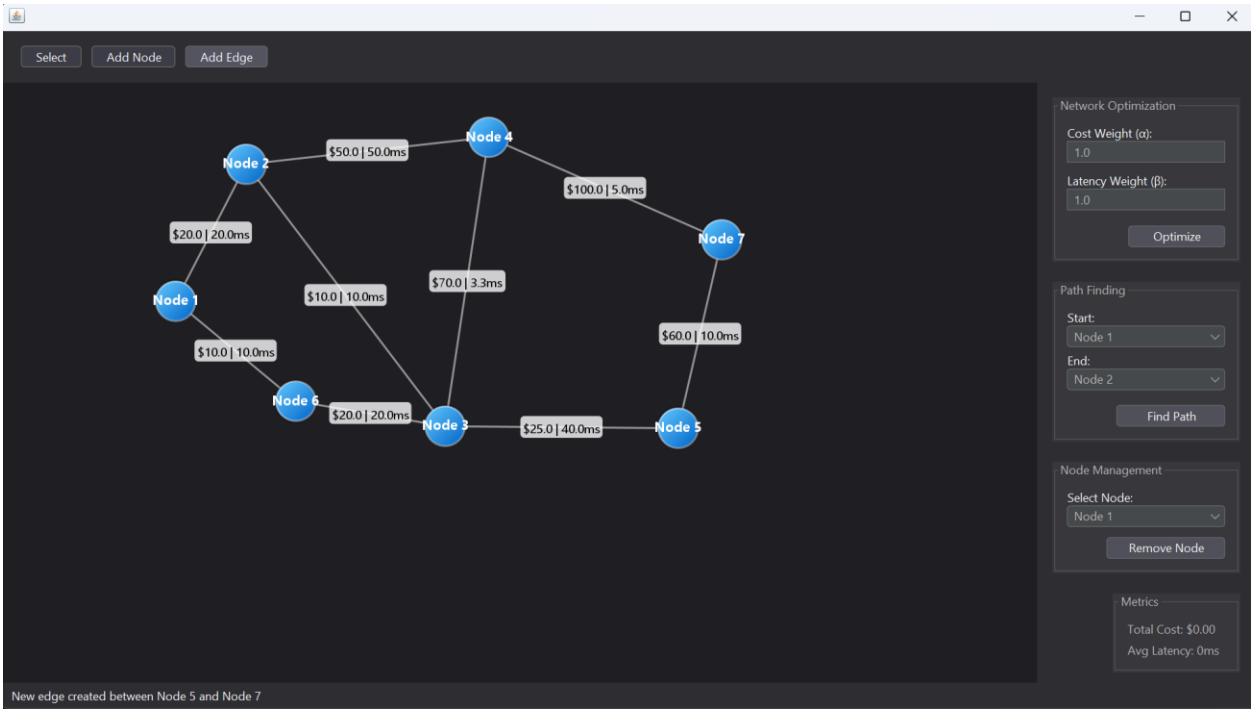
```
934         public void updateMetrics(double cost, double latency) {
935             costLabel.setText(String.format("Total Cost: $%.2f", cost));
936             latencyLabel.setText(String.format("Avg Latency: %.2fms", latency));
937         }
938 
939         // ----------------------------
940         // Helper UI Methods
941         // ----------------------------
942         private JPanel createSectionPanel(String title) {
943             JPanel panel = new JPanel();
944             panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
945             panel.setBorder(new CompoundBorder(
946                 new TitledBorder(title),
947                 new EmptyBorder(10, 10, 10, 10)
948             ));
949             panel.setBackground(new Color(55, 55, 60));
950             return panel;
951         }
952 
953         private JButton createButton(String text, ActionListener listener) {
954             JButton btn = new JButton(text);
955             btn.addActionListener(listener);
956             btn.setAlignmentX(Component.LEFT_ALIGNMENT);
957             btn.setMaximumSize(new Dimension(200, 30));
958             btn.setBackground(new Color(80, 80, 90));
959             btn.setForeground(Color.WHITE);
960             return btn;
961         }
962 
963         private JPanel createLabelField(String labelText, JComponent comp) {
964             JPanel panel = new JPanel(new BorderLayout());
965             JLabel label = new JLabel(labelText);
966             label.setForeground(Color.WHITE);
967             panel.add(label, BorderLayout.NORTH);
968             panel.add(comp, BorderLayout.CENTER);
969             panel.setOpaque(false);
970             return panel;
971         }
972 
973         private JPanel createLabelCombo(String labelText, JComboBox<Node> combo) {
974             JPanel panel = new JPanel(new BorderLayout());
975             JLabel label = new JLabel(labelText);
976             label.setForeground(Color.WHITE);
977             panel.add(label, BorderLayout.NORTH);
978             panel.add(combo, BorderLayout.CENTER);
979             panel.setOpaque(false);
980             return panel;
981         }
982 
983         private void showError(String message) {
984             JOptionPane.showMessageDialog(this, message, "Input Error", JOptionPane.ERROR_MESSAGE);
985         }
986     }
```

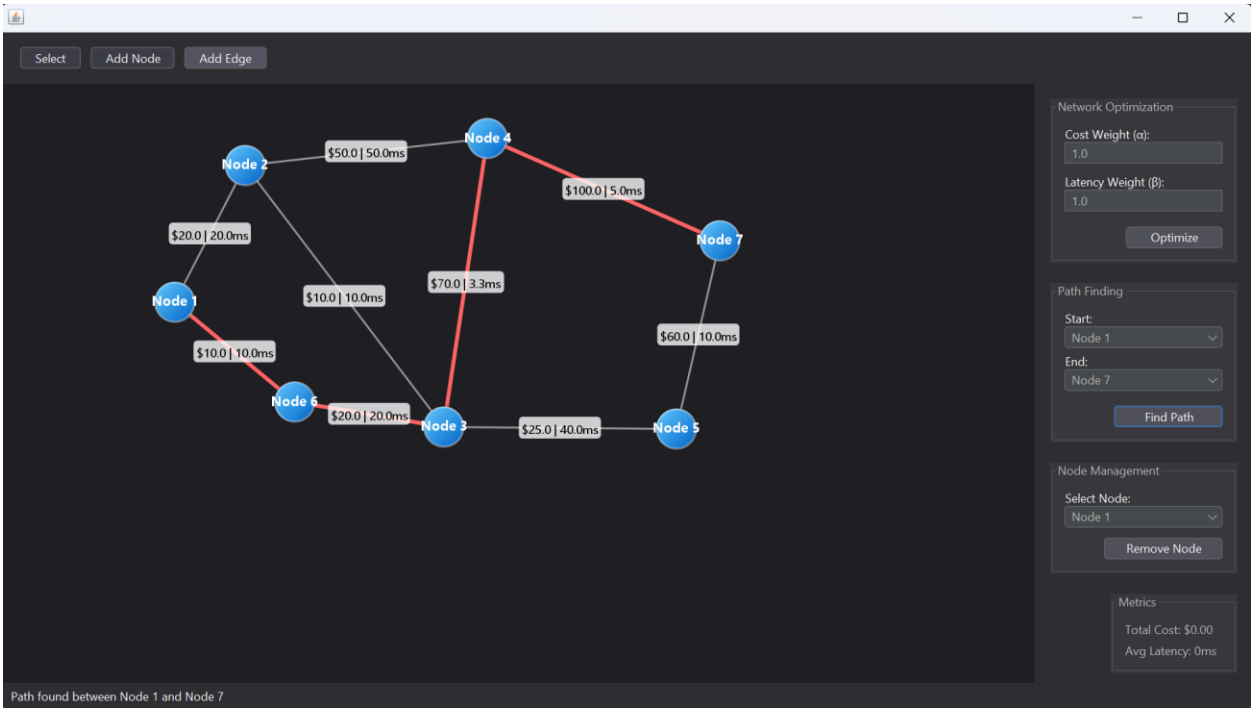
```
988         static class ToolPanel extends JPanel {
989             private final ButtonGroup group = new ButtonGroup();
990             private Tool currentTool = Tool.SELECTION;
991
992             public ToolPanel(Consumer<Tool> toolConsumer) {
993                 setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));
994                 setBorder(new EmptyBorder(5, 10, 5, 10));
995                 setBackground(new Color(45, 45, 50));
996
997                 Arrays.stream(Tool.values()).forEach(tool -> {
998                     JToggleButton btn = new JToggleButton(tool.toString());
999                     btn.setFocusPainted(false);
1000                    btn.setBackground(new Color(60, 60, 70));
1001                    btn.setForeground(Color.WHITE);
1002                    btn.addActionListener(e -> {
1003                        currentTool = tool;
1004                        toolConsumer.accept(tool);
1005                    });
1006                    btn.setToolTipText("Tool: " + tool.toString());
1007                    group.add(btn);
1008                    add(btn);
1009                });
1010
1011                // Select the first tool by default (SELECTION)
1012                if (group.getElements().hasMoreElements()) {
1013                    group.getElements().nextElement().setSelected(true);
1014                }
1015            }
1016
1017            public Tool getCurrentTool() {
1018                return currentTool;
1019            }
1020        }
1021
1022        enum Tool {
1023            SELECTION("Select"),
1024            NODE_CREATION("Add Node"),
1025            EDGE_CREATION("Add Edge");
1026
1027            private final String label;
1028            Tool(String label) {
1029                this.label = label;
1030            }
1031            @Override
1033            public String toString() {
1034                return label;
1035            }
        }
```

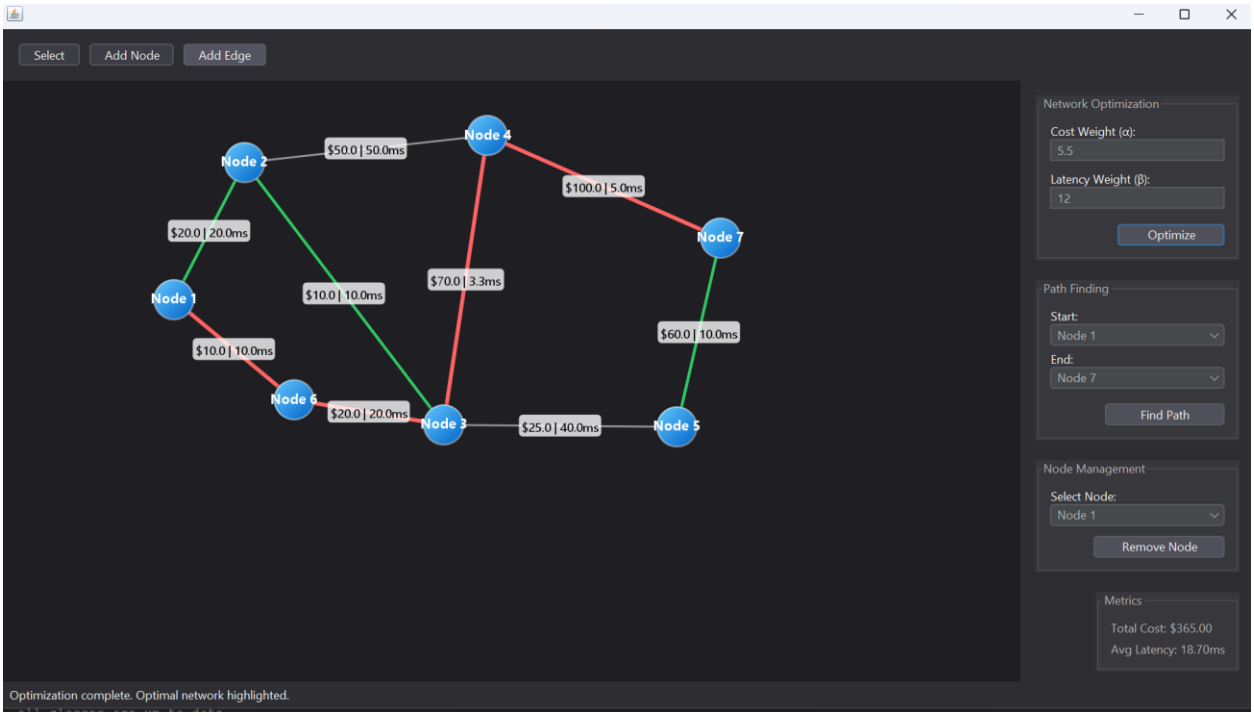
```
    static class NodeRenderer extends DefaultListCellRenderer {
        @Override
        public Component getListCellRendererComponent(JList<?> list, Object value,
                                                      int index, boolean isSelected, boolean cellHasFocus) {
            super.getListCellRendererComponent(list, value, index, isSelected, cellHasFocus);
            if (value instanceof Node) {
                setText(((Node) value).getId());
            }
            return this;
        }
    }
}
```

Output


Select
Add Node
Add Edge
Node 1
Node 2
Node 7
Node 6
Node 5
Node 3
Node 4
Edge Properties
Cost:
10
Bandwidth (Mbps):
100
OK
Cancel
Network Optimization
Cost Weight (α):
1.0
Latency Weight (β):
1.0
Optimize
Path Finding
Start:
Node 1
End:
Node 2
Find Path
Node Management
Select Node:
Node 1
Remove Node
Metrics
Total Cost: $0.00
Avg Latency: 0ms
Edge creation: select target node.


Select
Add Node
Add Edge
Node 4
Node 2
$50.0 | 50.0ms
$100.0 | 5.0ms
Node 7
$20.0 | 20.0ms
$70.0 | 3.3ms
Node 1
$10.0 | 10.0ms
$60.0 | 10.0ms
$10.0 | 10.0ms
Node 6
$20.0 | 20.0ms
Node 3
$25.0 | 40.0ms
Node 5
Network Optimization
Cost Weight (α):
1.0
Latency Weight (β):
1.0
Optimize
Path Finding
Start:
Node 1
End:
Node 2
Find Path
Node Management
Select Node:
Node 1
Remove Node
Metrics
Total Cost: $0.00
Avg Latency: 0ms
New edge created between Node 5 and Node 7


Select
Add Node
Add Edge
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7
$50.0 | 50.0ms
$100.0 | 5.0ms
$20.0 | 20.0ms
$10.0 | 10.0ms
$70.0 | 3.3ms
$60.0 | 10.0ms
$10.0 | 10.0ms
$20.0 | 20.0ms
$25.0 | 40.0ms
Network Optimization
Cost Weight (α):
1.0
Latency Weight (β):
1.0
Optimize
Path Finding
Start:
Node 1
End:
Node 7
Find Path
Node Management
Select Node:
Node 1
Remove Node
Metrics
Total Cost: $0.00
Avg Latency: 0ms
Path found between Node 1 and Node 7


Select
Add Node
Add Edge
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7
$50.0 | 50.0ms
$100.0 | 5.0ms
$20.0 | 20.0ms
$10.0 | 10.0ms
$70.0 | 3.3ms
$60.0 | 10.0ms
$10.0 | 10.0ms
$20.0 | 20.0ms
$25.0 | 40.0ms
Network Optimization
Cost Weight (α):
5.5
Latency Weight (β):
12
Optimize
Path Finding
Start:
Node 1
End:
Node 7
Find Path
Node Management
Select Node:
Node 1
Remove Node
Metrics
Total Cost: $365.00
Avg Latency: 18.70ms
Optimization complete. Optimal network highlighted.

6. a) Ans:

The goal is to sequence "0102030405..." up to number **n** using three threads **ZeroThread**, **EvenThread** and **OddThread**. A class **NumberPrinter** is given with three methods: **printZero()**, **printEven()** and **printOdd()** to print zeros, even numbers and odd numbers respectively in a correct order.

Approach

To solve this problem, we use a shared state with synchronization through **Lock** and **Condition**. The state transitions control which thread is allowed to execute next.

- **ZeroThread**: It prints "0" and then signals **OddThread** if the next number is odd or signals **EvenThread** if even.
- **OddThread** and **EvenThread**: Wait for the turn, print the corresponding number, increment the counter and signal ZeroThread.

Algorithm and Steps

- Initialize variables **current = 1** and **state = 0**. Also initialize a class **NumberPrinter** and a controller with the maximum number **n** and a **NumberPrinter** instance as parameters.
- For **ZeroThread**, start a loop which iterates while **current <= n**.
  - If **state != State.ZERO**, it waits on the condition.
  - After the condition is met, **printer.printZero()** is called with prints “0”.
  - Check if **current** is odd, it sets state to **State.ODD**; otherwise, it sets it to **State.EVEN**. Then, **condition.signalAll()** is called which signals all waiting threads so that the correct thread can proceed.
- For **OddThread**, start a loop which iterates while **current <= n**.
  - If **state != State.ZERO** and **current** is odd, it waits on the condition.
  - After the condition is met, **printer.printOdd(current)** is called with prints “0”.
  - The **current** is incremented i.e., **current++**
  - Set the state to **ZERO** i.e., **state = State.ZERO** and signals all threads so that the **ZeroThread** can print the next zero.
- Similarly, just like for **EvenThread**, same process is repeated.

- The threads keep on repeating until **current** exceeds **n**.
- Finally, any remaining threads are signaled to ensure proper termination and the output is printed.

Time and Space Complexity

1. Time Complexity:

- The loops through **n** takes **O(n).**
- Each thread operation (print and state update) is done in **O(1)**.

Overall time complexity is **O(n)** in worst where **n** is the **nth** of the number.

2. Space Complexity:

- A fixed number of variables are such as lock, condition, state, and counter which takes **O(1)**.

Overall space complexity is **O(1)** in worst case.

Code Implementation

```java
import java.util.concurrent.locks.*;

class NumberPrinter {  4 usages
    // Print Zero
    public void printZero() {  1 usage
        System.out.print("0");
    }
    // Print Odd
    public void printOdd(int num) {  1 usage
        System.out.print(num);
    }
    // Print Odd
    public void printEven(int num) {  1 usage
        System.out.print(num);
    }
}
```

```java
public class ThreadController {
    private final int n;                    // Maximum number to print  5 usages
    private final NumberPrinter printer;    // Object to print numbers  4 usages
    private int current = 1;                // Next number to print  11 usages
    // Enum to represent which thread's turn it is
    private enum State { ZERO, ODD, EVEN }  10 usages
    private State state = State.ZERO;       // Start with ZeroThread  7 usages
    private final Lock lock = new ReentrantLock();  7 usages
    private final Condition condition = lock.newCondition();  7 usages

    // Constructor to initialize the controller with n and a NumberPrinter
    public ThreadController(int n, NumberPrinter printer) {  1 usage
        this.n = n;
        this.printer = printer;
    }

    // Start the three threads for printing zero, odd, and even numbers
    public void startThreads() {  1 usage
        new Thread(this::zeroThread,  name: "ZeroThread").start();
        new Thread(this::oddThread,  name: "OddThread").start();
        new Thread(this::evenThread,  name: "EvenThread").start();
    }
```

```java
// ZeroThread: Prints "0" and then signals either OddThread or EvenThread based on the next number
private void zeroThread() {  1 usage
    lock.lock();
    try {
        while (current <= n) {
            // Wait until it's zero's turn
            while (state != State.ZERO) {
                condition.await();
            }
            printer.printZero();
            if (current > n) break;
            // Set next state: ODD if current is odd, EVEN if even
            state = (current % 2 == 1) ? State.ODD : State.EVEN;
            condition.signalAll();
        }
        // Wake up waiting threads to allow termination
        state = State.ODD;
        condition.signalAll();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    } finally {
        lock.unlock();
    }
}
```

```java
// OddThread: Waits for its turn, prints the odd number, increments current, then signals ZeroThread
private void oddThread() {  1 usage
    lock.lock();
    try {
        while (current <= n) {
            // Wait until it's odd's turn and the current number is odd
            while (state != State.ODD || current % 2 == 0) {
                condition.await();
            }
            printer.printOdd(current);
            current++;
            state = State.ZERO;
            condition.signalAll();
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    } finally {
        lock.unlock();
    }
}
```

```
    // EvenThread: Waits for its turn, prints the even number, increments current, then signals ZeroThread
    private void evenThread() {  1 usage
        lock.lock();
        try {
            while (current <= n) {
                // Wait until it's even's turn and the current number is even
                while (state != State.EVEN || current % 2 == 1) {
                    condition.await();
                }
                printer.printEven(current);
                current++;
                state = State.ZERO;
                condition.signalAll();
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock();
        }
    }
```

```
    public static void main(String[] args) {
        NumberPrinter printer = new NumberPrinter();

        // Example-1
        int n1 = 5;
        ThreadController controller1 = new ThreadController(n1, printer);
        System.out.print("The number is ");
        controller1.startThreads(); // Expected output: 01020304050

        /*
        // Example-2
        int n2 = 10;
        ThreadController controller2 = new ThreadController(n2, printer);
        System.out.print("The number is ");
        controller2.startThreads(); // Expected output: 0102030405060708090100 11
         */
    }
}
```

Example-1

Input: n = 5

```
// Example-1
int n1 = 5;
ThreadController controller1 = new ThreadController(n1, printer);
System.out.print("The number is ");
controller1.startThreads(); // Expected output: 01020304050
```

Output: 01020304050

```
The number is 01020304050
```

Example-2

Input: n = 10

```
// Example-2
int n2 = 10;
ThreadController controller2 = new ThreadController(n2, printer);
System.out.print("The number is ");
controller2.startThreads(); // Expected output: 0102030405060708090100 11
```

Output: 0102030405060708090100 11

```
The number is 0102030405060708090100 11
```

6. b) Ans:

The main goal is to create a web crawler application that can crawl multiple web pages concurrently using multithreading to improve performance.

Approach

In this solution, we build a multithreaded web crawler that improves performance by processing multiple URLs concurrently.

The design comprises:

- Thread-Safe Data Structures:
  - A **BlockingQueue** (in particular, a **LinkedBlockingQueue**) that contains URLs yet to be crawled.
  - A **ConcurrentHashMap-based** set to keep tabs on URLs which have been visited (guaranteeing that each URL is processed once).
  - A **ConcurrentMap** for holding crawled page content.
- Thread Pool (ExecutorService):
  - A fixed thread pool (while its limit is configurable) submits and executes crawling tasks concurrently.
- Task Design:
  - Each task peeks from the queue, attempts to download a specific page using Jsoup with retry logic, harvests hyperlinks off that page within a stream-based filter excluding unwanted ones, and adds new URLs to the queue.

- Crawl Delay Enforcement:
  - The crawler shall initiate a delay per domain in being polite not to overload any given site.

- Error Handling:
  - Each task retries unsuccessfully fetched URLs several times and handles exceptions so that a single failure doesn't disrupt the entire process.

- Result Printing:
  - Pages retrieved are printed after the thread pool is done (with title and little text).

Algorithm and Steps

- Initialization:
  - Set up a URL Queue: Seed URLs are put inside a thread-safe blocking queue.
  - Visited Set and Data Map: Create a concurrent set to track visited URLs and a map to store the HTML content of each crawled URL.
  - Thread Pool: Create a fixed thread pool (using ExecutorService) with a predefined number of threads.

- The Crawling Task (executed by each thread):
  - Polls a URL from the queue repeatedly within 500 ms waiting time.
  - If the URL has not been visited, mark it as a visited one and then proceed it:
    - Crawl Delay: Introduce crawl delay for all domains to obey crawl policies.
    - Fetch Web Page: Try to download the page using Jsoup. If the request fails, try a fixed number of attempts.
    - Store Result: Save the content of the page in the crawled data map.
    - Get More Active Links: Examine the page for links that satisfy certain criteria and place all unseen URLs in the URL queue.

- Termination:
  - Once tasks are submitted to all threads, shut down the executor and wait for termination, with timeout.
  - When finished (or timed out), print the results after parsing the saved HTML for each URL.

- Additional Handling:
  - This **isCrawlAllowed ()** method is stubbed just to check the robots.txt of the site; in this case, it allows any crawls.

- The concurrent visited set ensures that the same URL is never processed twice.

Time and Space Complexity

1. Time Complexity:

- Each URL is processed by doing a fixed number of network operations (using as many as **MAX_RETRIES**), parsing the HTML and extracting the links. While this will depend on many factors, since we are measuring for the actual work in processing a URL, it's considered **O(1)** in computational complexity.
- If **n** URLs are finally processed, we have **O(n)** total work done in processing all the URLs, because each URL is processed exactly once. However, the running time may be reduced greatly through concurrency (if there are t threads, then the total work for coming up with the link list may approximately be **O(n/t)**, not accounting for overhead).

2. Space Complexity:

- Worst case, near all discovered URLs come from settings will give O(**n**) space, in which **n** is the number of unique URLs visited.
- Storing the content of each page will also take up a space of O(**n**).
- Indeed, the delays in domain access and temporary objects are also O(**n**) space.

Thus, space complexity is O(**n**).

Code Implementation

```java
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.select.Elements;
import java.util.*;
import java.util.concurrent.*;
import java.io.IOException;
import java.util.stream.Collectors;

public class MultithreadedWebCrawler {
    // Configuration constants
    private static final String USER_AGENT = "Mozilla/5.0 (compatible; MyWebCrawler/1.0)";  1 usage
    private static final int REQUEST_TIMEOUT_MS = 10_000;  1 usage
    private static final int MAX_RETRIES = 3;  3 usages
    private static final int CRAWL_DELAY_MS = 1000;  2 usages

    // Thread-safe data structures
    private final BlockingQueue<String> urlQueue = new LinkedBlockingQueue<>();  3 usages
    private final Set<String> visitedUrls = ConcurrentHashMap.newKeySet();  5 usages
    private final ConcurrentMap<String, String> crawledData = new ConcurrentHashMap<>();  2 usages
    private final ConcurrentMap<String, Long> domainDelays = new ConcurrentHashMap<>();  2 usages
    private final ExecutorService executor;  4 usages
    private final int numThreads;  2 usages

    public MultithreadedWebCrawler(int numThreads) {  1 usage
        this.numThreads = numThreads;
        this.executor = Executors.newFixedThreadPool(numThreads);
    }

    public void startCrawling(List<String> seedUrls) {  1 usage
        seedUrls.stream()
                .filter(url -> !visitedUrls.contains(url))
                .forEach(urlQueue::add);
```

```
        for (int i = 0; i < numThreads; i++) {
            executor.submit(this::crawlTask);
        }

        executor.shutdown();
        try {
            if (!executor.awaitTermination( timeout: 2, TimeUnit.MINUTES)) {
                System.err.println("Crawling timeout reached");
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void crawlTask() {  1 usage
        try {
            while (!Thread.currentThread().isInterrupted()) {
                String url = urlQueue.poll( timeout: 500, TimeUnit.MILLISECONDS);
                if (url == null) continue;

                if (visitedUrls.add(url)) {
                    processUrl(url);
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
```

```java
    private void processUrl(String url) {  1 usage
        try {
            if (!isCrawlAllowed(url)) {
                return;
            }

            String content = fetchWebPageWithRetry(url);
            if (content == null) return;

            crawledData.put(url, content);
            System.out.println("\nSuccessfully crawled: " + url);

            List<String> links = extractAndFilterLinks(url, content);
            links.stream()
                    .filter(link -> !visitedUrls.contains(link))
                    .forEach(link -> urlQueue.offer(link));

        } catch (Exception e) {
            System.err.println("Error processing " + url + ": " + e.getMessage());
        }
    }

@   private String fetchWebPageWithRetry(String url) {  1 usage
        for (int attempt = 0; attempt < MAX_RETRIES; attempt++) {
            try {
                enforceCrawlDelay(url);

                Document doc = Jsoup.connect(url)
                        .userAgent(USER_AGENT)
                        .timeout(REQUEST_TIMEOUT_MS)
                        .get();

                return doc.outerHtml();
            } catch (IOException e) {
                if (attempt == MAX_RETRIES - 1) {
                    System.err.println("Failed to fetch " + url + " after " + MAX_RETRIES + " attempts");
                }
            }
        }
        return null;
```

```java
    private void enforceCrawlDelay(String url) {  1 usage
        try {
            String domain = new java.net.URL(url).getHost();
            Long lastAccess = domainDelays.get(domain);
            if (lastAccess != null) {
                long elapsed = System.currentTimeMillis() - lastAccess;
                if (elapsed < CRAWL_DELAY_MS) {
                    Thread.sleep( millis: CRAWL_DELAY_MS - elapsed);
                }
            }
            domainDelays.put(domain, System.currentTimeMillis());
        } catch (Exception e) {
            System.err.println("Error enforcing crawl delay: " + e.getMessage());
        }
    }

    private List<String> extractAndFilterLinks(String url, String html) {  1 usage
        try {
            Document doc = Jsoup.parse(html, url);
            Elements links = doc.select( cssQuery: "a[href]");

            return links.stream()  Stream<Element>
                    .map(link -> link.absUrl( attributeKey: "href"))  Stream<String>
                    .filter(this::isValidLink)
                    .collect(Collectors.toList());
        } catch (Exception e) {
            System.err.println("Error parsing links from " + url);
            return Collections.emptyList();
        }
    }

@   private boolean isValidLink(String url) {  1 usage
        return url.startsWith("http") &&
                !url.contains("mailto:") &&
                !url.contains("javascript:") &&
                !url.endsWith(".pdf") &&
                !url.endsWith(".zip");
    }
```

```
144         private boolean isCrawlAllowed(String url) {  1 usage
145             try {
146                 java.net.URL target = new java.net.URL(url);
147                 String robotsUrl = target.getProtocol() + "://" + target.getHost() + "/robots.txt";
148
149                 if (!visitedUrls.contains(robotsUrl)) {
150                     visitedUrls.add(robotsUrl);
151                     // Implement robots.txt parsing here using crawler-commons
152                     // For simplicity, we'll allow all crawls in this example
153                 }
154                 return true;
155             } catch (Exception e) {
156                 System.err.println("Error checking robots.txt for " + url);
157                 return false;
158             }
159         }
160
161         public void printResults() {  1 usage
162             System.out.println("\n\nCrawling Results:");
163             System.out.println("==================");
164
165             crawledData.forEach((url, content) -> {
166                 Document doc = Jsoup.parse(content);
167                 String title = doc.title();
168                 String textSnippet = doc.body() != null ?
169                         doc.body().text().replaceAll( regex: "\\s+",  replacement: " ").substring(0, 150) + "..." :
170                         "[No text content]";
171
172                 System.out.println("URL: " + url);
173                 System.out.println("Title: " + title);
174                 System.out.println("Content: " + textSnippet);
175                 System.out.println("----------------------------");
176             });
177         }
```

```
    public static void main(String[] args) {
        List<String> seedUrls = List.of(
//                  "https://en.wikipedia.org/wiki/Web_crawler",
                "https://solo-leveling.fandom.com/wiki/Solo_Leveling_Wiki"
        );

        MultithreadedWebCrawler crawler = new MultithreadedWebCrawler( numThreads: 4);
        crawler.startCrawling(seedUrls);
        crawler.printResults();
    }
}
```

Output

```
Successfully crawled: https://solo-leveling.fandom.com/wiki/Solo_Leveling_Wiki

Successfully crawled: https://www.fandom.com/

Successfully crawled: https://solo-leveling.fandom.com

Successfully crawled: https://createnewwiki-143.fandom.com/Special:CreateNewWiki

Successfully crawled: https://solo-leveling.fandom.com/wiki/Solo_Leveling_Wiki#

Successfully crawled: https://solo-leveling.fandom.com/f
```

```
Successfully crawled: https://solo-leveling.fandom.com/wiki/Collect_Demon_Souls!_(1)

Successfully crawled: https://solo-leveling.fandom.com/wiki/Quests

Successfully crawled: https://solo-leveling.fandom.com/wiki/Courage_of_the_Weak

Successfully crawled: https://solo-leveling.fandom.com/wiki/Collect_Demon_Souls!_(2)

Successfully crawled: https://solo-leveling.fandom.com/wiki/Job_Change_Quest

Successfully crawled: https://solo-leveling.fandom.com/wiki/Kill_the_Enemies
```

```
Successfully crawled: https://solo-leveling.fandom.com/wiki/Ahjin_Guild

Successfully crawled: https://solo-leveling.fandom.com/wiki/Knights_Guild

Successfully crawled: https://solo-leveling.fandom.com/wiki/Fiend_Guild

Successfully crawled: https://solo-leveling.fandom.com/wiki/Category:Races

Successfully crawled: https://solo-leveling.fandom.com/wiki/Adam_White

Successfully crawled: https://solo-leveling.fandom.com/wiki/Category:Human

Successfully crawled: https://solo-leveling.fandom.com/wiki/Draw_Sword_Guild
```

```
Successfully crawled: https://solo-leveling.fandom.com/wiki/Kaisellin

Successfully crawled: https://solo-leveling.fandom.com/wiki/Ice_Bears

Successfully crawled: https://solo-leveling.fandom.com/wiki/Kamish

Successfully crawled: https://solo-leveling.fandom.com/wiki/Shadows

Successfully crawled: https://solo-leveling.fandom.com/wiki/Beru

Successfully crawled: https://solo-leveling.fandom.com/wiki/Igris

Successfully crawled: https://solo-leveling.fandom.com/wiki/Greed
```

Appendix

GitHub Link: - https://github.com/SubhamAdhikari20/Programming-for-Developers-Coursework