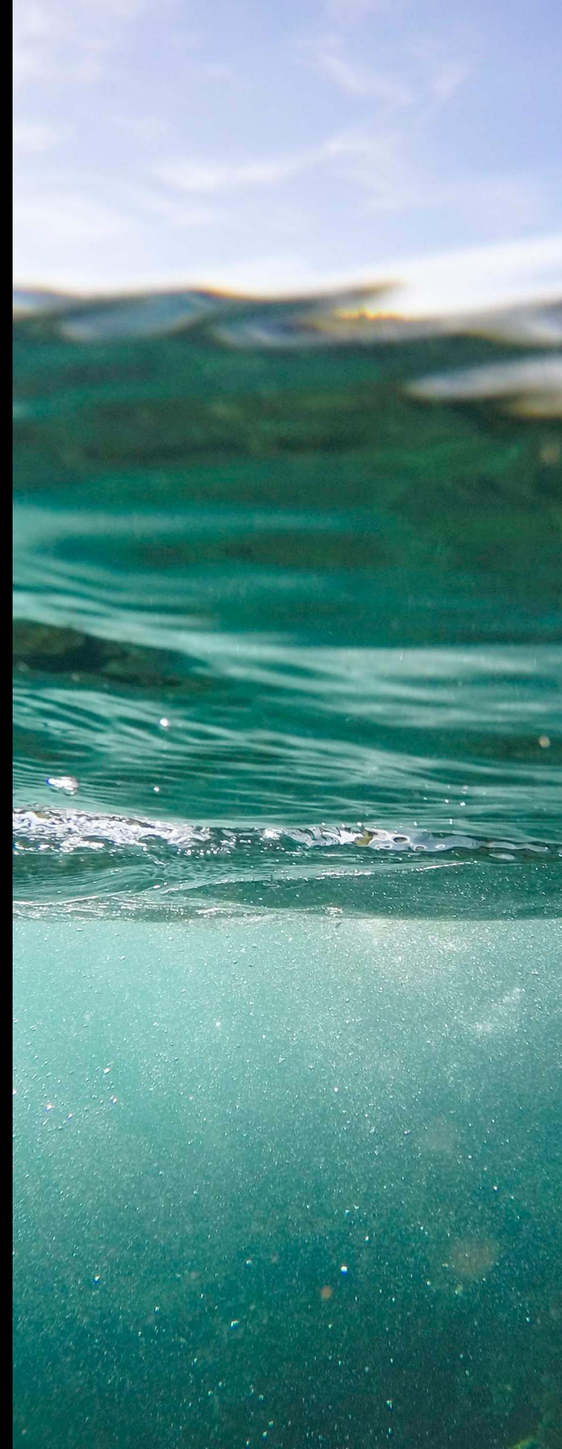
WATER CONSERVATION:

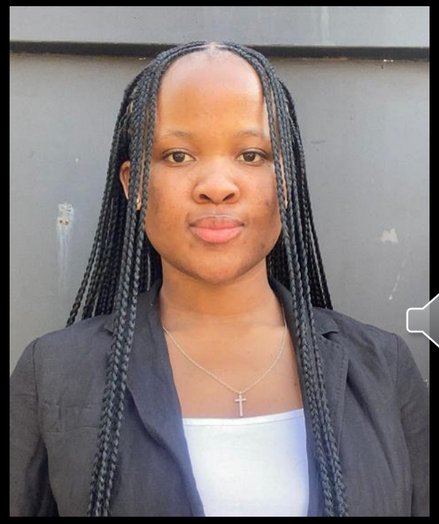
PRESERVING LIFE ON EARTH

The Importance of Water Conservation



Group Members

George Mugale



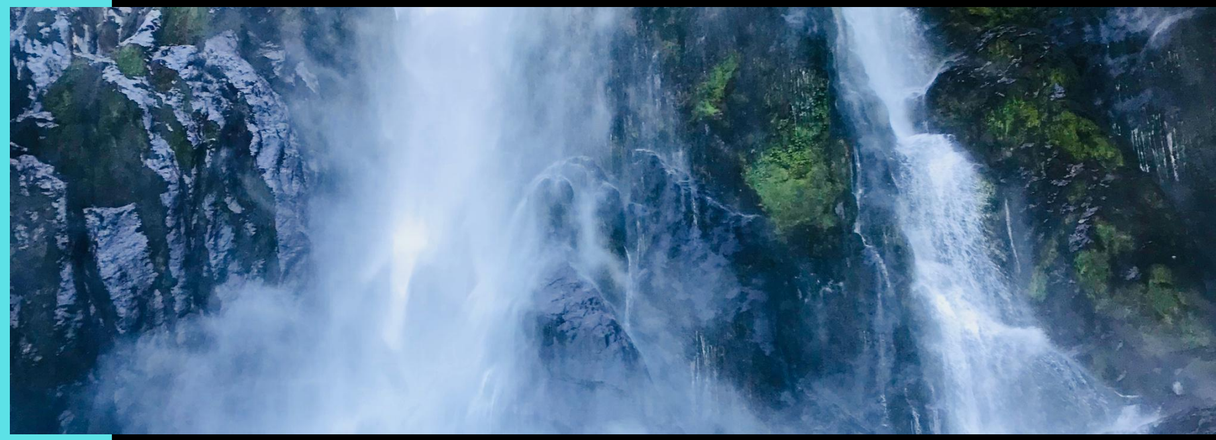
Primrose Makweng

Ntokozo Dlamini

Sinqobile Myeza

The Importance of Water

Water is critical for all life on Earth. It supports ecosystems, agriculture, and human survival. Scarcity and pollution harm global health and economy.



Current Challenges

Global water crisis worsens with population growth and industry.

Climate change, pollution, and unsustainable use are major causes.

Impacts include water quality decline and biodiversity loss.

Impact on society

Clean Water is essential for drinking cooking and sanitation.Lack of access to clean water can lead to waterbone disease such as cholera

Overuse and pollution of water bodies can damage ecosystems, affecting wildlife and biodiversity

Solutions?

Automated Water Quality Detection Application (Java Desktop)

Reason for this approach

01 Accuracy – Ensures reliable classification of water quality.

02 Efficiency – Automates detection, reducing manual effort and human error.

03 Trend Insights – Identifies contamination patterns for water management.

Graph construction

- Pixel to Vertex Conversion – Transform image pixels into graph vertices.
- Vertex Attributes – Store location and colour for each vertex.
- Edge Weighting – Calculate colour differences to determine edge strengths.

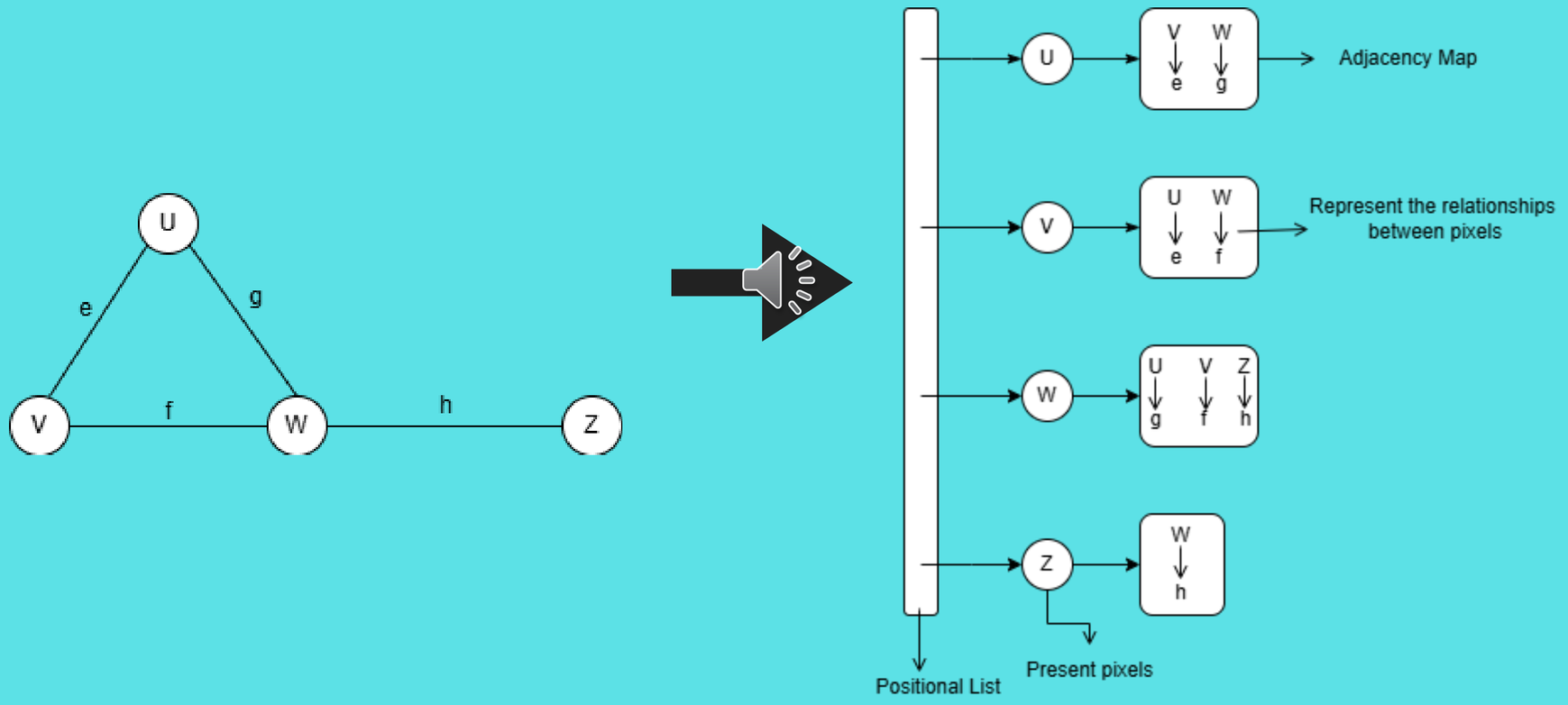
Graph construction

- Edge Pruning – Remove weak edges iteratively to simplify the graph.
- Adaptive Super pixels – Dynamically adjust region size:
- Large images → Use larger super pixels relative to graph size.
- Small images → Treat each pixel as its own region (size = 1).

```
    final int width = image.getWidth();
    final int height = image.getHeight();
    final int gridWidth = width / pixelRegionSize + 1;
    final int gridHeight = height / pixelRegionSize + 1;

    // create a grid that will temporarily store pixels next to each other
    @SuppressWarnings("unchecked")
    Vertex<Point>[][] vertexGrid = new Vertex[gridHeight][gridWidth];

    // Only two directions (left and up) to prevent duplicate edges
    final int[][] neighborDirections = { { -1, 0 }, { 0, -1 } };

    for (int y = 0; y < height; y += pixelRegionSize) {
        for (int x = 0; x < width; x += pixelRegionSize) {
            // Calculate relative grid coordinates
            final int gridX = x / pixelRegionSize;
            final int gridY = y / pixelRegionSize;

            // Compute average color for the current block
            Color averageColor = calculateBlockAverageColor(x, y);
            // creates a point representation of the points in the region
            Point regionPoint = createRegionPoint(x, y, averageColor);

            // if the point does not already exist
            if (pixelVertices.get(regionPoint) == null) {
                // Create and store vertex
                Vertex<Point> vertex = graph.insertVertex(regionPoint);
                // add the vertex to the adjacency map
                vertexGrid[gridY][gridX] = vertex;
                pixelVertices.put(regionPoint, vertex);

                // Connect to neighbors with smart edge pruning
                connectToNeighbors(vertexGrid, vertex, gridX, gridY, averageColor, neighborDirections);
            }
        }
    }
```

Graph

Adjacency Map Structure



Graph

Our graph ADT includes:

- an undirected graph,
- a Vertex (InnerVertex<V>) and an Edge (InnerEgde<E>) class,
- maitaining a positional list of all vertices and edges,
- each vertex(pixel) has a map of adjacent vertices.

Adjacency Map Structure?

- Stores pixel and their neighbours.
- Efficient for MST construction.
- Easy to retrieve pixel information.

Prim's Minimum Spanning Tree

Find the minimum-cost paths that link all graph nodes without cycles.

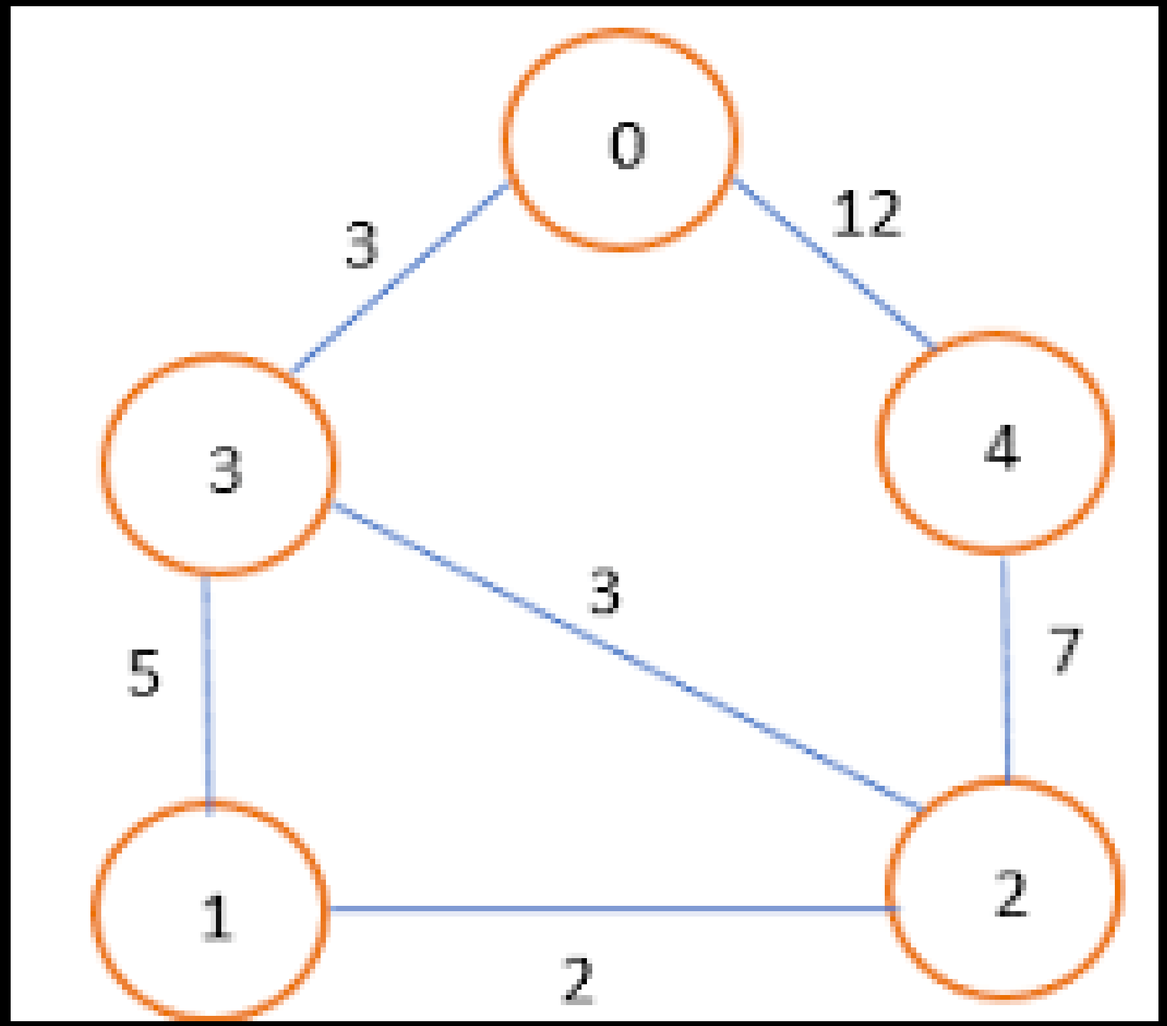
01 Total weight of connected edges

02 Number of edges in the path

03 The average weight of edges in the path

04 The variance of edge weights

Prim's Minimum Spanning Tree



Graph Edit Distance

Calculates the cost of edits to convert graph A to graph B

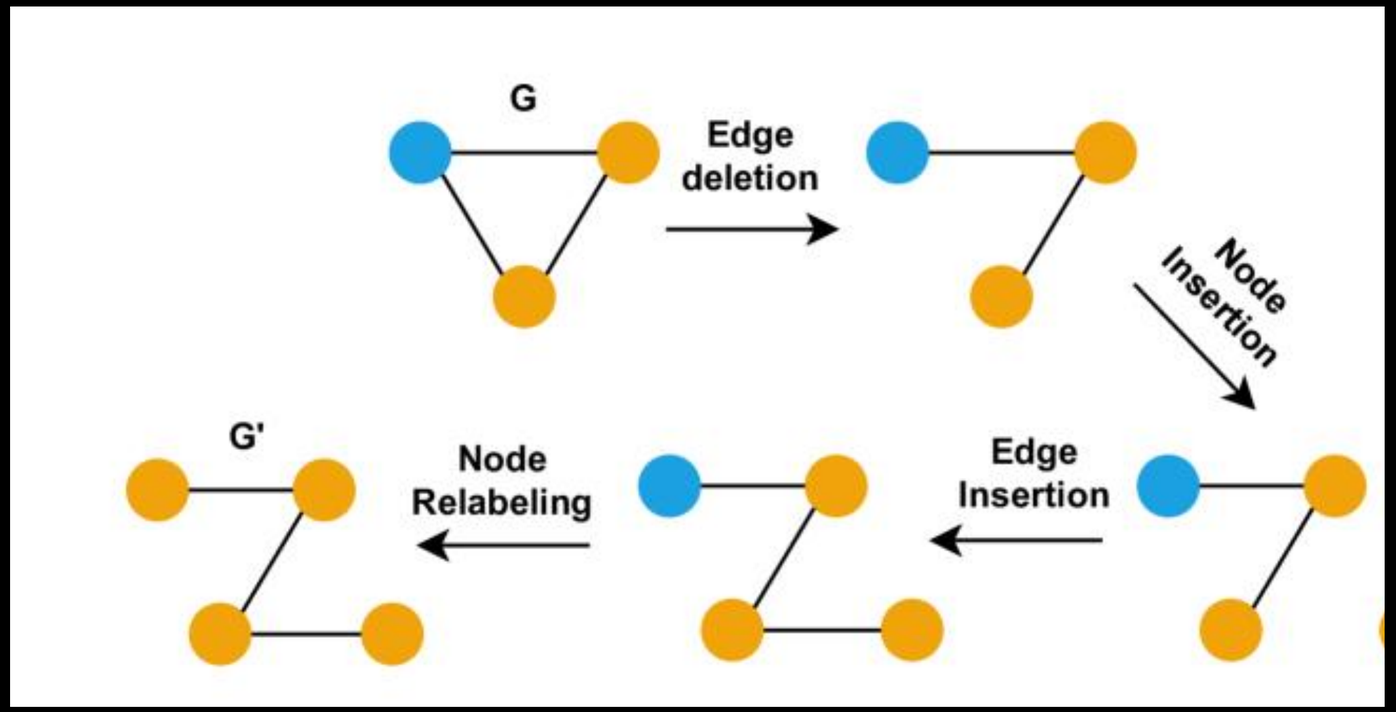
01 Extract graph from new image

02 Compare it to a known clean/ dirty water graphs

03 Simulate convertion to see how similar A and B are

04 Smart cost functions determine how expensive convertion operation would be

Graph Edit Distance



```java
     *
     * @param graphA the graph being queried
     * @param graphB the graph being compared with
     * @return the value of the edit distance
     */
    public float calculateGraphEditDistance(ImageGraph graphA, ImageGraph graphB) {
        float editDistance = 0.0f;

        // Vertex matching, matching vertices from graphA to graphB to compute costs
        Iterator<Vertex<Point>> graphAVertices = graphA.getGraph().vertices().iterator();
        // sort graph b vertices and edges to compare
        Iterator<Vertex<Point>> graphBVertices = graphB.getGraph().vertices().iterator();

        // Iterate through both to calculate vertex substitution cost
        while (graphAVertices.hasNext() && graphBVertices.hasNext()) {
            Point graphAPoint = graphAVertices.next().getElement();
            Point graphBPoint = graphBVertices.next().getElement();

            editDistance += vertexSubstitutionCost(graphAPoint, graphBPoint);
        }

        editDistance += VERTEX_INSERTION_DELETION_COST
                * Math.abs(graphA.getVerticies().size() - graphB.getVerticies().size());

        // 2. Edge matching, matching edges from graphA to graphB to compute costs
        Iterator<Edge<Float>> graphAEdges = graphA.getGraph().edges().iterator();

        Iterator<Edge<Float>> graphBEdges = graphB.getGraph().edges().iterator();

        while (graphAEdges.hasNext() && graphBEdges.hasNext()) {
            Float graphAWeight = graphAEdges.next().getElement();
            Float graphBWeight = graphBEdges.next().getElement();

            editDistance += edgeSubstitutionCost(graphAWeight, graphBWeight);
        }

        editDistance += EDGE_INSERTION_DELETION_COST
                * Math.abs(graphA.getGraph().numEdges() - graphB.getGraph().numEdges());

        return (editDistance / MAXIMUM_POSSIBLE_GED) * 1000;
    }
```

k-Nearest Neighbor

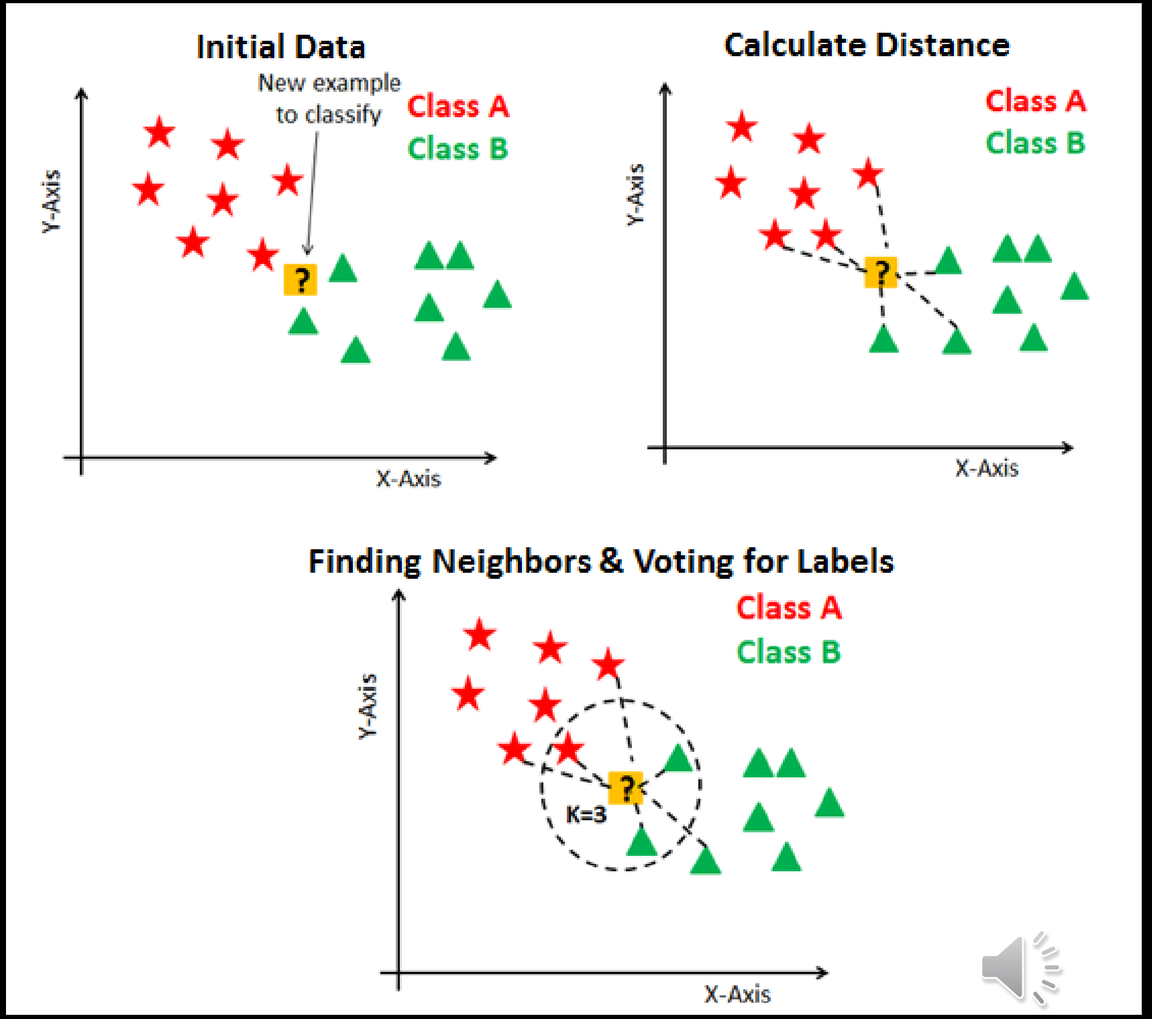
01 Compute distances between the new graph and other reference graphs

02 Get the k closest reference graphs with regards to the distance calulated

03 Determine the most common label amongs the k reference graphs

04 The mst common label is the the “winner”

k-Nearest Neighbor



```
// sort the list in ascending order (from biggest to smallest)
Collections.sort(matchList);

/*
 * get only the first K pairs in the list (check that K is not more than the
 * size of the classification list)
 */
matchList = matchList.subList(0, Math.min(K, matchList.size()));

/* create a map that will store the label and the frequency of occurrences
and the weight of occurrences the list*/
Map<MATCH_TYPE, Double> weightedVotes = new AdjacencyMap<>();

// add the labels and the votes
for (KClass<MATCH_TYPE, Double> kClass : matchList) {
    Double currentVote = weightedVotes.get(kClass.label);

    double weight = 1.0 / (kClass.distance + Double.MIN_VALUE);

    // if it is a new entry
    if (currentVote == null) {
        weightedVotes.put(kClass.label, weight);
    } else {
        weightedVotes.put(kClass.label, currentVote + weight);
    }
}

MATCH_TYPE bestMatch = MATCH_TYPE.BLACK;
double highestVote = Float.MIN_VALUE;
/* Find the label with the most occurrences and least distance (highest vote) */
Iterator<MATCH_TYPE> iterator = weightedVotes.keySet().iterator();
// iterate through all keys and find the one with the highest vote
while (iterator.hasNext()) {
    MATCH_TYPE currentMatch = iterator.next();
    double currentVote = weightedVotes.get(currentMatch);
    if (currentVote > highestVote) {
        highestVote = currentVote;
        bestMatch = currentMatch;
    }
}
return bestMatch;
```

Reference Data Set

Prepare and train data

- Gather images
- Label and categorize manually images
- Calculate and store MST features
- Calculate the mean and standard deviation of the feature vector
- Normalize MST features vector

Graph-Based Tasks

Similarity

- Similarity measures help the system assess how closely a new water sample matches previously identified contaminated or safe samples, aiding in accurate contamination detection.

Classification

- Classification allows the system to automatically label water samples into predefined categories enabling quick decision-making without manual analysis.

Classification

- Get new graphs MST features
- Determine MST Feature Euclidean distances
- Get K closest feature distances
- Classify under the most common category within the K set of graphs

```
75
76          // add the labels and the votes
77          for (KClass<CATEGORY_TYPE, Distance> kClass : classificationList) {
78              Integer currentFrequency = frequencyMap.get(kClass.label);
79              // if it is a new entry
80              if (currentFrequency == null) {
81                  frequencyMap.put(kClass.label, 1);
82              }
83          }
84
85          CATEGORY_TYPE bestMatch = CATEGORY_TYPE.ONLY_WATER_TOP_VIEW;
86          int highestFrequency = Integer.MIN_VALUE;
87
88          Iterator<CATEGORY_TYPE> iterator = frequencyMap.keySet().iterator();
89          // iterate through all keys and find check each vote
90          while (iterator.hasNext()) {
91              CATEGORY_TYPE currentMatch = iterator.next();
92              int currentFrequency = frequencyMap.get(currentMatch);
93              if (currentFrequency > highestFrequency) {
94                  highestFrequency = currentFrequency;
95                  bestMatch = currentMatch;
96              }
97          }
98
99          return bestMatch;
100     }
101
```

```java
public static CATEGORY_TYPE classify(ImageGraph inputGraph, ImageIterator referenceGraphs, int K) {

    if (referenceGraphs == null || !referenceGraphs.hasNext() || K <= 0) {
        throw new IllegalArgumentException("Invalid input parameters");
    }

    List<KClass<CATEGORY_TYPE, Distance>> classificationList = new ArrayList<>();
    Prims_MST<Point> MST = new Prims_MST<>();
    MSTFeatures inputGraphFeatures = MST.CalcMST(inputGraph.getGraph());

    while (referenceGraphs.hasNextMSTFeature()) {
        MSTFeatures currentFeature = referenceGraphs.nextFeature();

        if (currentFeature != null) {
            classificationList.addLast(new KClass<CATEGORY_TYPE, Distance>(currentFeature.category_TYPE,
                    MSTFeatures.calculateDistance(inputGraphFeatures, currentFeature)));
        }
    }

    referenceGraphs = null;
    System.gc();

    Collections.sort(classificationList);

    classificationList = classificationList.subList(0, Math.min(K, classificationList.size()));

    // create a map that will store the label and the frequency of occurrences
    Map<CATEGORY_TYPE, Integer> frequencyMap = new AdjacencyMap<>();

    // add the labels and the votes
    for (KClass<CATEGORY_TYPE, Distance> kClass : classificationList) {
        Integer currentFrequency = frequencyMap.get(kClass.label);
        // if it is a new entry
        if (currentFrequency == null) {
            frequencyMap.put(kClass.label, 1);
        }
    }

    CATEGORY_TYPE bestMatch = CATEGORY_TYPE.ONLY_WATER_TOP_VIEW;
    int highestFrequency = Integer.MIN_VALUE;

    Iterator<CATEGORY_TYPE> iterator = frequencyMap.keySet().iterator();
    // iterate through all keys and find check each vote
    while (iterator.hasNext()) {
        CATEGORY_TYPE currentMatch = iterator.next();
        int currentFrequency = frequencyMap.get(currentMatch);
        if (currentFrequency > highestFrequency) {
            highestFrequency = currentFrequency;
```

```java

public class kNearestNeighbor {

    public static class KClass<A, B extends Comparable<B>> implements Comparable<KClass<A, B>> {
        public A label;
        public B distance;

        public KClass(A label, B distance) {
            this.label = label;
            this.distance = distance;
        }

        @Override
        public String toString() {
            // TODO Auto-generated method stub
            return String.format("KClass{label: %s distance: %s}", label, distance);
        }

        @Override
        public int compareTo(KClass<A, B> o) {
            return this.distance.compareTo(o.distance);
        }
    }
```

Similarity Detection

- Find most similar graphs under the classified data set
- MST quick reject to further filter unlikely matches
- GED as a similarity metric between graphs
- Get K most similar graphs
- Image is mostly similar with the most common type

Our ADTs

- Graph
- Map
- Array List
- Linked Positional List
- Heap Priority Queue

Social Impact

Critical for biodiversity: rivers, lakes, wetlands.

Community involvement vital for water quality.

Local efforts show collective action works.

Some novelty

01 Functional Interfaces and Lambda Expressions

02 Asynchronous Programming

03 Observer and Proxy design patterns

Limitations

Data set collection and training

Similarity relies on classification

Possible Improvments

Finding node mappings

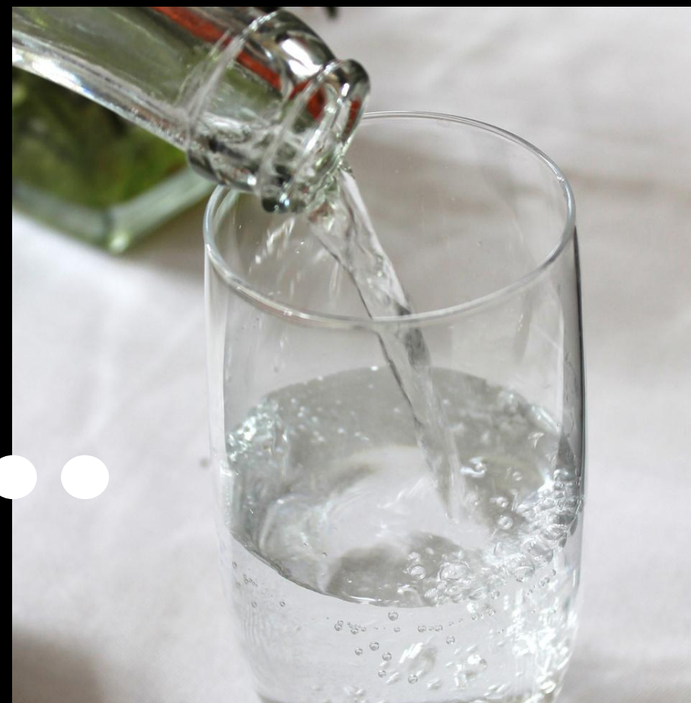
Distributed database and more reference graphs

Industry and Agriculture Impact

Early Contamination Detection

Faster Response Time

Cost-Effective Monitoring



```
File Name: Screenshot 2025-05-10 011626undrink.png
Classify: ONLY_WATER_SIDE_VIEW Similar To: RED
MST Features:
Total Weight: -0.71284837
Average Weight: -0.017945576
Variance: -0.22445753
Edge Count: -1
Degree Map: null
File Name: Screenshot 2025-05-17 162854.png
```

```
File Name: Screenshot 2025-05-17 162854.png
Classify: ONLY_WATER_SIDE_VIEW Similar To: GREEN
MST Features:
Total Weight: -0.7028841
Average Weight: -0.4428747
Variance: -0.5961988
Edge Count: -1
Degree Map: null
```

Results

Water Conservation at Home

01 Tips: fix leaks, use efficient appliances, save water in baths.

02 Every drop saved counts.

Call to Action

- Take daily steps for water conservation.
- Support policies for sustainable water management.
- Join local and global initiatives.