Relatório do Trabalho de Técnicas Avançadas de Projetos e Grafos

Técnicas de Programação Avançada — Ifes — Campus Serra

Alunos: David Vilaça, Harã Heique, Larissa Motta.
Prof. Jefferson O. Andrade

2019/2

Sumário

Lista de Códigos Fonte

Lista de Figuras

1 Introdução

Técnicas Avançadas de Projeto foca na utilização de algoritmos menos sofisticados que os amplamente mais aplicados (dividir e conquistar, randomização e afins). São eles: a programação dinâmica e algoritmos gulosos.
Já na teoria dos grafos é um ramo da matemática que estuda as relações entre os objetos, onde para isso são empregados as estruturas comumente chamada de grafos, G(V,A), no qual V é um conjunto de objetos denominados vértices e A é um conjunto de pares não ordenados do vértice, denominada de arestas.
Neste trabalho é proposto a resolução de um conjunto de problemas envolvendo programação dinâmica, algoritmos gulosos, e algoritmos em grafos vistas na disciplina, onde todos os problemas propostos estão disponíveis no site UVa Online Judge.
Os problemas foram resolvidos usando a linguagem de programação *Java* (versão JDK 11.0.3+) nos Ambientes Integrados de Desenvolvimento (IDE) *Netbeans*, *Apache Netbeans* e no editor de texto *Visual Studio Code*. As resoluções estão dispostos na seção 2 deste documento.
Todos os códigos e imagens presentes nesse trabalho estão disponíveis no GitHub.

2 Problemas

2.1 Programação Dinâmica

2.1.1 UVa 00108 – Maximum Sum

A solução deste problema consiste em somar as sub matrizes da matriz de entrada armazenando os resultados das somas em uma matriz de resultado. Após somar todos os valores da sub matriz é verificado se o resultado até então é menor do que a soma corrente, e caso seja, é necessário propagar a nova informação como sendo a maior. A cada interação de soma a variável *"result"* sempre armazena a maior soma das sub matrizes até então. No final do programa ela permanecerá com a maior soma sendo o resultado correto, ou seja, a cada interação obtemos o melhor resultado presente no array denominado *sum*, que faz o caching das somas, recuperando assim o que está disponível no momento, como é mostrado na linha 40 do Código Fonte 1, caracterizando-o como um algoritmo dinâmico.

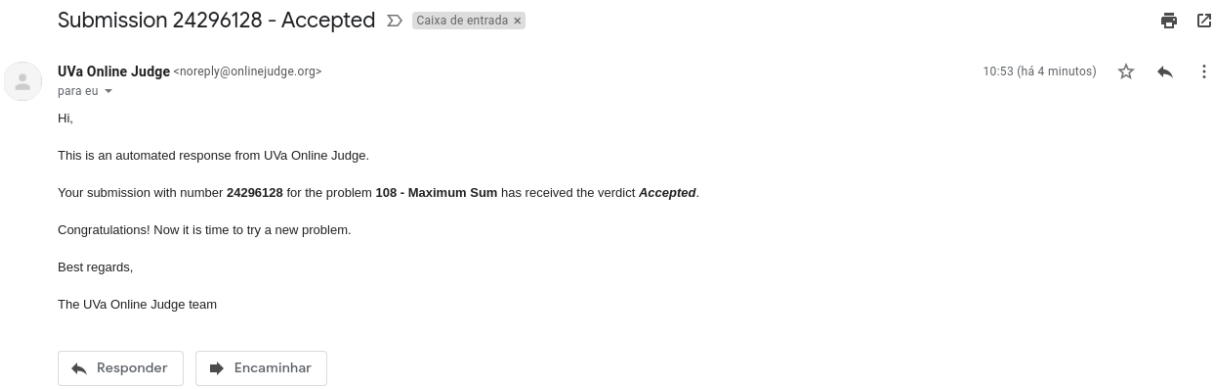

Figura 1: Email de aceitação da solução *UVa 00108–Maximum Sum*.

```java
import java.util.Scanner;

public class Main {

  private static final Scanner scanner = new Scanner(System.in);

  public static void main(String[] args) {
    int n, sum[][], square[][], result;
    while (scanner.hasNext()) {
      n = scanner.nextInt();
      sum = new int[n][n]; // array to cache sum
      square = new int[n][n]; // square matrix
      result = 0; // test case result
      for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
          square[i][j] = scanner.nextInt(); // fill matrix
          if (i == 0 && j == 0) // on begin start result with first element of
          ↪ matrix
            result = square[i][j];

          for (int subI = i; subI > -1; subI--) {
            for (int subJ = j; subJ > -1; subJ--) {

              sum[subI][subJ] = 0; // start sum of sub matrix

              // SUM ALL SUB MATRIX
              if (subI + 1 <= i && subJ + 1 <= j) {
                sum[subI][subJ] += sum[subI + 1][subJ + 1];
              }

              for (int z = subI; z <= i; z++)
                sum[subI][subJ] += square[z][subJ];

              for (int y = subJ; y <= j; y++)
                sum[subI][subJ] += square[subI][y];

              sum[subI][subJ] -= square[subI][subJ];

              // if result is smaller then exchange
              if (sum[subI][subJ] > result)
                result = sum[subI][subJ];
            }
          }
        }
      }
      System.out.println(result);
    }
  }
}
```

Código Fonte 1: Solução para o problema *UVa 00108 – Maximum Sum* (pt 1).

2.1.2 UVa 10684 – The Jackpot

A solução desse problema consiste numa variável para armazenar o valor máximo calculado sempre. A cada entrada de dados de aposta de uma sequência de tamanho N é calculado o maior valor entre a soma da entrada atual mais o valor máximo anterior desse cálculo, e o resultado é testado com o valor máximo atual.
O algoritmo atual é dinâmico porque a cada interação calcula-se o melhor resultado e o armazena numa variável (linha 18 do Código Fonte 2) para não precisar ser recalculado nas interações seguintes, resultando em ganho de performance.

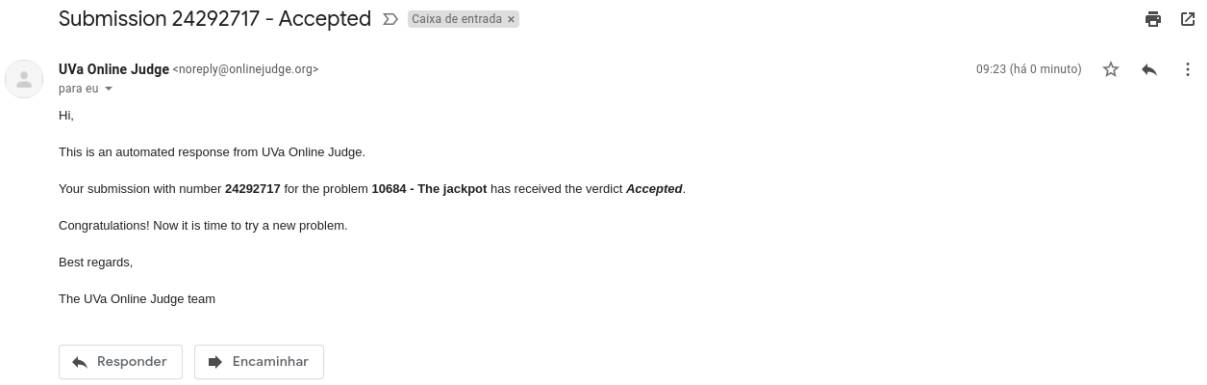

Figura 2: Email de aceitação da solução *UVa 10684 – The Jackpot*.

```java
import java.util.Scanner;

public class Main {

  private static final Scanner scanner = new Scanner(System.in);

  public static void main(String[] args) {
    int n = scanner.nextInt();
    int max, currentValue, aux;

    // length of the sequence
    while (n != 0) {
      max = 0; // to cache the max value
      currentValue = 0;
      for (int i = 0; i < n; i++) { // for each bet of sequence
        aux = scanner.nextInt();
        currentValue = getMax(currentValue + aux, aux); // max value
        max = getMax(max, currentValue); // max value
      }

      output(max);

      n = scanner.nextInt();
    }

  }

  /**
   * get the max value between int value a and int value b
   *
   * @param a
   * @param b
   * @return
   */
  private static int getMax(int a, int b) {
    if (a >= b)
      return a;
    else
      return b;
  }
```

Código Fonte 2: Solução para o problema *UVa 10684 – The Jackpot* (pt. 1).

```
  /**
   * output print
   *
   * @param val
   */
  private static void output(int val) {
    if (val > 0)
      System.out.printf("The maximum winning streak is %d.\n", val);
    else
      System.out.println("Losing streak.");
  }

}
```

Código Fonte 3: Solução para o problema *UVa 10684 – The Jackpot* (pt. 2).

2.2 Algoritmos Gulosos

2.2.1 UVa 11100 – The Trip, 2007

A solução deste problema se inicia com a criação de um array de tamanho N para armazenar as mochilas e logo após, são ordenadas para então conseguir caminhar por elas das menores para as maiores. É realizada a comparação em pares de mochilas verificando se possuem dimensões iguais para acumular um contador. Se as mochilas forem de dimensões diferentes obtém-se o valor máximo entre o acumulador (das mochilas anteriores) e a dimensão da mochila atual e reinicia-se contador para a próxima iteração, onde o processo é repetido. Ao final dessa iteração teremos a quantidade máxima de sacolas que minimizam o número total de peças de bagagem. Com esse número em mãos, a saída se resume em percorrer o array ordenado, novamente, para imprimir cada dimensão de mochila que cabe dentro da sacola atual.
Esse problema se caracteriza como greedy ou "guloso"porque a cada iteração que agimos no array a melhor decisão é tomada, como por exemplo no caso de obter o valor máximo (linha 31 do Código Fonte 6), não precisando assim, calcular todas as possibilidades para ao final escolhermos a melhor.

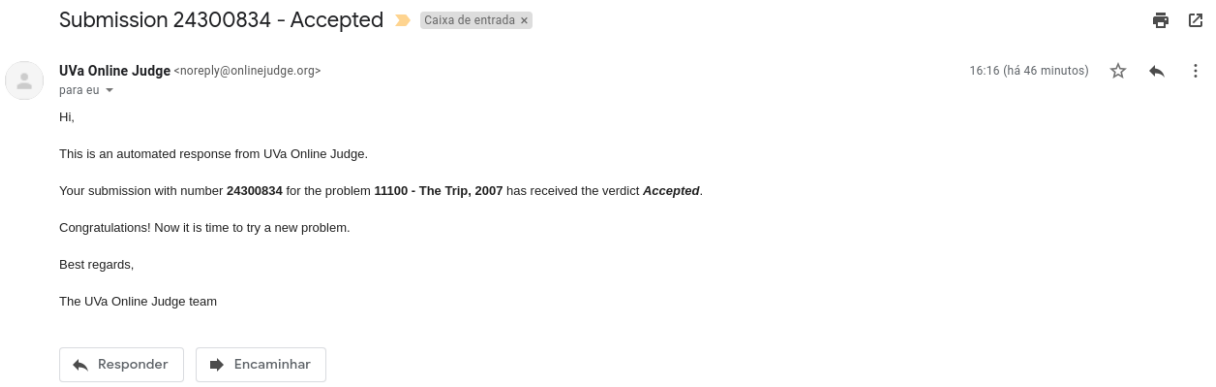

Figura 3: Email de aceitação da solução *UVa 11100 – The Trip, 2007*.

```java
import java.util.Scanner;
import java.util.Arrays;

public class Main {

  private static final Scanner scanner = new Scanner(System.in);

  public static void main(String[] args) {
    int n = scanner.nextInt(), atualBag, amount, outputBagsMax, lastBag;
    Integer[] bags;
    while (n > 0) {
      bags = new Integer[n];
      outputBagsMax = 0;
      amount = 0;
      lastBag = -1;

      // get input and fill bags array
      for (int i = 0; i < n; i++) {
        bags[i] = scanner.nextInt();
      }

      // sort array to scroll in ascending order
      Arrays.sort(bags);

      for (int i = 0; i < n; i++) { // for each bag from smallest to largest
        atualBag = bags[i];

        if (lastBag == atualBag) {
          amount++;
        } else {
          outputBagsMax = Math.max(amount, outputBagsMax);
          amount = 1;
        }

        lastBag = atualBag;
      }

      outputBagsMax = Math.max(amount, outputBagsMax);

      output(outputBagsMax, n, bags);

      n = scanner.nextInt();
    }
  }
```

Código Fonte 4: Solução para o problema *UVa 11100 – The Trip, 2007* (pt. 1).

```
  private static void output(int outputBagsMax, int n, Integer[] bags) {
    System.out.printf("%d\n", outputBagsMax);
    for (int i = 0; i < outputBagsMax; i++) {
      System.out.printf("%d", bags[i]);
      for (int j = i + outputBagsMax; j < n; j += outputBagsMax) {
        System.out.printf(" %d", bags[j]);
      }
      System.out.printf("\n");
    }
    System.out.printf("\n");
  }
}
```

Código Fonte 5: Solução para o problema *UVa 11100 – The Trip, 2007* (pt. 2).

2.2.2 UVa 12405 – Scarecrow

Como se pode notar Código Fonte 6 a solução consiste em caminhar nos caracteres de entrada e a cada caracter '.' encontrado, que significa um local de cultivo, adiciona um espantalho e caminha 3 spots para frente, ou seja, quando achar um spot de cultivo precisamos de um espantalho e este cobre 3 spots, como é mostrado entre as linhas 31 e 33. Já quando achado o caracter '#', que significa uma região infértil, não precisamos de espantalho, como é mostrado nas linhas 34 e 35.
O algoritmo explicado anteriormente é considerado guloso, pois toma a decisão de adicionar um espantalho a cada momento com base no conjunto de informações coletadas na iteração corrente, sendo a alternativa mais promissora naquele momento atual e por fim imprimindo a quantidade de áreas prontas para cultivo caracterizadas pelo espantalho que devem ser colocados nessas áreas.

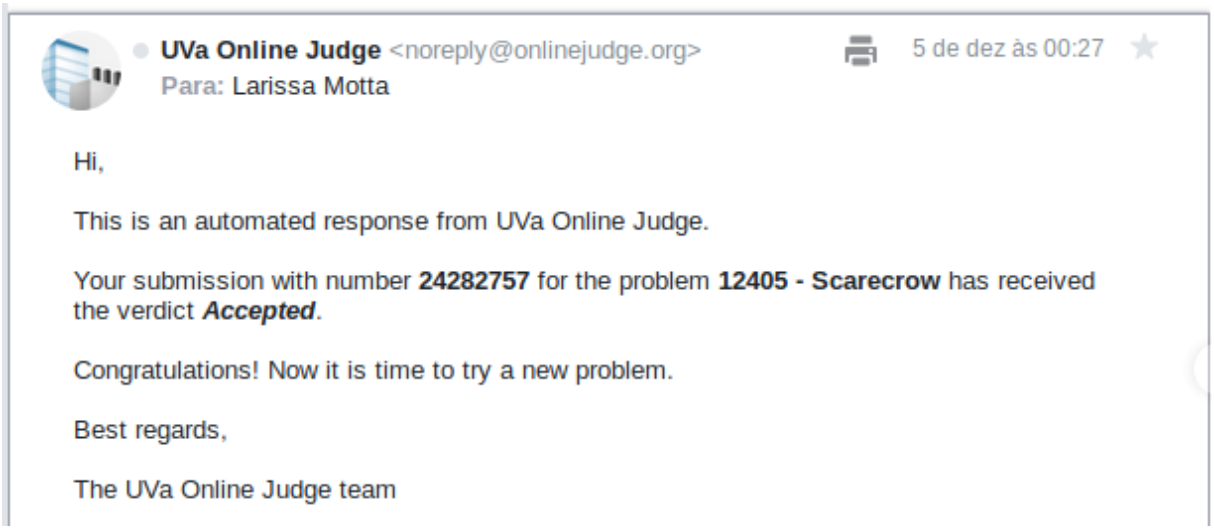

Figura 4: Email de aceitação da solução *UVa 12405 – Scarecrow*.

```java
import java.util.Scanner;

public class Main {

  private static final Scanner scanner = new Scanner(System.in);

  public static void main(String[] args) {
    // get number of test cases
    int t = scanner.nextInt();
    // aux variables
    int n, index, answer;
    char c;
    String line;

    // for each case
    for (int numLines = 0; numLines < t; numLines++) {
      // acc answer value of case
      answer = 0;
      // get number of characters
      n = scanner.nextInt();
      // discard \n
      scanner.nextLine();
      // get characters line of case
      line = scanner.nextLine();

      // for each character of line
      index = 0;
      while (index < n) {
        // current character
        c = line.charAt(index);
        if (c == '.') { // if character is '.' then is crop-growing spot
          answer++;
          index += 3;
        } else { // if character is '#' then is infertile region
          index++;
        }
      }
      // output answer of actual case
      System.out.printf("Case %d: %d\n", numLines + 1, answer);
    }
  }
}
```

Código Fonte 6: Solução para o problema *UVa 12405 – Scarecrow* (pt. 1).

2.3 Algoritmos em Grafos

2.3.1 UVa 00280 – Vertex

No Código Fonte 7 está presente a função principal que é responsável pelo workflow definido pelo problema. Na linha 12 é lido a quantidade de vértices/nós do caso corrente para o grafo, onde este é instanciado na linha 21 pela classe *Graph* que representa uma estrutura *grafo*, passando como argumento um boleano que define se ele será bidirecional, ou seja, uma grafo direcionado ou não direcionado, onde neste é direcionado, logo contendo arestas unidirecionais.
Seguindo a função principal na linha 22 é chamado a função que é responsável por instanciar todos os vértices/nós do grafo e realizar a ligação das arestas entre eles, chamando respectivamente os métodos *addVertex* e *addEdge* da classe *Graph*.
Por fim é feita a análise do grafo atual identificando os grafos inacessíveis definido no problema proposto. Basicamente a estratégia utilizada foi a busca em profundidade performada pelo método *depthFirstSearch* da classe *Graph*, que retorna uma estrutura $Set< T >$ compostos por todos os vértices/nós que o vértice passado como argumento consegue ter acesso. Para determinar quais os vértices que não tem acesso, basicamente é pego a diferença entre todos os vértices do grafo atual com os vértices acessíveis retornados pelo método chamado, como é mostrado na função *identifyInacessibleVertices* do Código Fonte 8 entre as linhas 38 e 41. Por fim é mostrado na linha 42 a função *output*, que imprimi respectivamente o número de vértices inacessíveis pelo vértice analisado assim como quais são esses vértices.

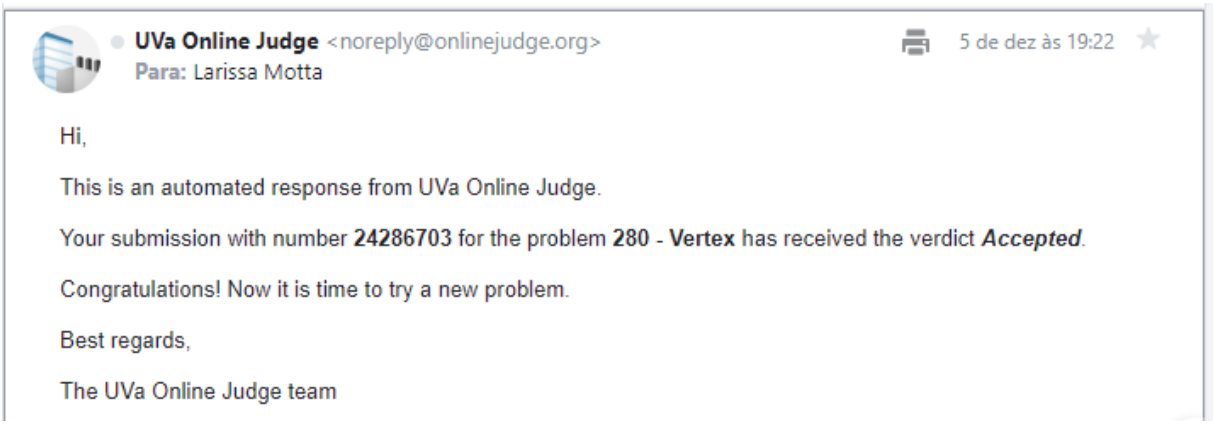

Figura 5: Email de aceitação da solução *UVa 00280 – Vertex*.

```java
import java.util.HashMap;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.Scanner;
import java.util.Set;

public class Main {
    private static final Scanner SC = new Scanner(System.in);

    public static void main(String[] args) {
        while (true) {
            int nVertices = SC.nextInt();
            ignoreLines(1);

            // Stops when find the 0 number of vertices
            if (nVertices == 0) {
                break;
            }

            // Creates a non bidirection graph and populates it
            Graph<Integer> graph = new Graph(false);
            populateGraph(graph, nVertices);

            // Do the analisys of the graph populated printing all the
            ↪  inacessible vertices
            identifyInacessibleVertices(graph);
        }
    }

    private static void output(Set<Integer> elements) {
        StringBuilder builder = new StringBuilder();

        // Getting the count of inacessible vertices
        builder.append(elements.size());

        // Getting the inacessible vertices
        for (int el : elements) {
            builder.append(" ").append(el);
        }

        builder.append("\n");
        System.out.print(builder);
    }

    // Ignore the number of lines in terminal
    private static void ignoreLines(int numberOfLines) {
        for (int i = 0; i < numberOfLines; i++) {
            SC.nextLine();
        }
    }
```

Código Fonte 7: Solução para o problema *UVa 00280 – Vertex* (pt. 1).

```java
// Define the graphs nodes and edges using the generic class Graph<T>
    private static void populateGraph(Graph<Integer> graph, int qntVertices) {
        // First of all create the nodes/vertices before connect with the
        ↪ edges
        for (int i = 1; i <= qntVertices; i++) {
            graph.addVertex(i);
        }
        // Connect the edges
        while (true) {
            String[] groupVertices = SC.nextLine().split(" ");
            // Stops reading when find a zero line
            if (groupVertices.length <= 0 || groupVertices[0].charAt(0) == '0')
            ↪ {
                return;
            }
            // Takes the initial vertex
            int initialVertex = Integer.parseInt(groupVertices[0]);

            // Connects the vertices with the initial vertex
            for (int i = 1; i < groupVertices.length; i++) {
                int destVertex = Integer.parseInt(groupVertices[i]);

                if (destVertex == 0) {
                    break;
                }
                graph.addEdge(initialVertex, destVertex);
            }
        }
    }

    // Check and print the vertices inacessible from the one that is in
    ↪ analisys
    private static void identifyInacessibleVertices(Graph<Integer> graph) {
        String[] groupVerticesToCheck = SC.nextLine().split(" ");
        int nVerticesToCheck = Integer.parseInt(groupVerticesToCheck[0]);

        // Checking the vertices by the depth first search
        for (int i = 1; i <= nVerticesToCheck; i++) {
            int vertex = Integer.parseInt(groupVerticesToCheck[i]);
            // Get all the acessible vertices, but doesn't count the initial
            ↪ one
            Set<Integer> acessibleVertices = graph.depthFirstSearch(vertex,
            ↪ false);
            Set<Integer> inacessibleVertices = graph.getAllVertices();
            // Get the inacessible vertices by the difference between the two
            ↪ collections
            inacessibleVertices.removeAll(acessibleVertices);
            output(inacessibleVertices);
        }
    }
```

Código Fonte 8: Solução para o problema *UVa 00280 – Vertex* (pt. 2).

```java
// Graph implemented by an adjancency list way
class Graph<T> {
    private final HashMap<T, LinkedList<T>> edges;
    private final boolean bidirectional;

    public Graph(boolean hasBidirection) {
        this.edges = new HashMap();
        this.bidirectional = hasBidirection;
    }

    public void addVertex(T s) {
        edges.put(s, new LinkedList());
    }

    public void addEdge(T source, T destination) {
        if (!edges.containsKey(source)) {
            addVertex(source);
        }

        if (!edges.containsKey(destination)) {
            addVertex(destination);
        }

        edges.get(source).add(destination);

        if (this.bidirectional == true) {
            edges.get(destination).add(source);
        }
    }

    public int numberOfVertices() {
        return this.edges.size();
    }

    public int numberOfEdges() {
        int count = 0;

        for (T v : edges.keySet()) {
            count += edges.get(v).size();
        }

        if (this.bidirectional == true) {
            return count / 2;
        }

        return count;
    }
```

Código Fonte 9: Solução para o problema *UVa 00280 – Vertex* (pt. 3).

```
//
    public boolean hasVertex(T s) {
        return this.edges.containsKey(s);
    }

    public boolean hasEdge(T s, T d) {
        return this.edges.get(s).contains(d);
    }

    public Set<T> getAllVertices() {
        return new LinkedHashSet(this.edges.keySet());
    }

    public Set<T> depthFirstSearch(T vertex, boolean countInitialVertex) {
        Set<T> dfsVisitedElements = new LinkedHashSet();

        // Calls the function to get all visited vertices without counting the
        ↪  first one
        if (this.hasVertex(vertex)) {
            DFS_execution(vertex, dfsVisitedElements, countInitialVertex);
        }

        return dfsVisitedElements;
    }

    private void DFS_execution(T visitedVertex, Set<T> dfsVisitedVertices,
    ↪  boolean addVisitedVertex) {
        if (visitedVertex == null) {
            return;
        }

        // Adds into the set as a visited vertex
        if (addVisitedVertex) { dfsVisitedVertices.add(visitedVertex); }

        LinkedList<T> adjacencyVertices = this.edges.get(visitedVertex);
        for (T nextVisitedVertex : adjacencyVertices) {

            // If the vertex is already in the Set is because is not necessary
            ↪  to call again
            if (dfsVisitedVertices.contains(nextVisitedVertex)) {
                continue;
            }

            DFS_execution(nextVisitedVertex, dfsVisitedVertices, true);
        }
    }
```

Código Fonte 10: Solução para o problema *UVa 00280 – Vertex* (pt. 4).

```
    // Returns the adjancency list of each vertex
    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();

        for (T v : edges.keySet()) {
            builder.append(v.toString()).append("-> ");

            for (T w : edges.get(v)) {
                builder.append(w.toString()).append("-> ");
            }

            builder.append("null\n");
        }

        return builder.toString();
    }
}
```

Código Fonte 11: Solução para o problema *UVa 00280 – Vertex* (pt. 5).

2.3.2 UVa 00459 – Graph Connectivity

O problema consiste em encontrar a quantidade de subgrafos conexos em um grafo criado pela entrada fornecida. No código Código Fonte 12 está presente a função principal, onde na linha 13 é lido o número de casos do problema, em que para cada caso é pego o maior valor de um letra no alfabeto, presente na linha 17, instanciado um grafo não direcionado, que é populado e conectado com todos os vértices definido pela leitura de entrada as arestas de conexão, presentes nas linhas 20 e 21, e por fim a partir da chamada do método do objeto instanciado da classe *Graph* denominado *numberOfSubGraphsConnected*, na linha 24, é retornado o número de subgrafos presente no grafo atual.
O Código Fonte 17 demonstra a implementação do método responsável por contar o número de subgrafos. A estratégia utilizada é a utilização do método da classe que faz a busca em profundidade do grafo a partir de um vértice dado, onde é retornado uma estrutura *Set* com todos os vértices visitados, incluindo o passado como argumento. Este *Set* é adicionado em uma lista de sets chamada de *subGraphs*, que como propriamente o nome diz, armazena todos os subgrafos encontrados pela busca em profundidade. Para que não seja retornado dois ou mais sets iguais existe anteriormente, entre as linhas 8 e 19, uma verificação se o vértice atual a ser analisado já está presente em algum subgrafo encontrado. Caso esteja não é necessário realizar novamente a busca em profundidade, pois gerará o mesmo resultado, assim passando para o próximo vértice.

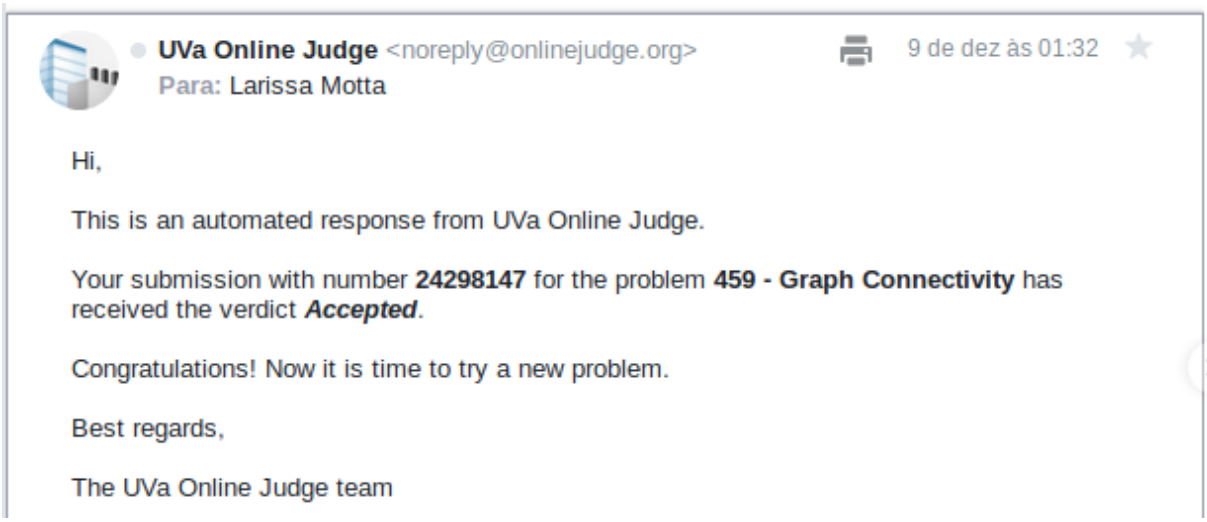

Figura 6: Email de aceitação da solução *UVa 00459 – Graph Connectivity*.

```java
import java.util.HashMap;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.Set;

public class Main {

    private static final Scanner SC = new Scanner(System.in);

    public static void main(String[] args) {
        int nCases = SC.nextInt();
        ignoreLines(2);

        for (int i = 0; i < nCases; i++) {
            char largestNode = SC.nextLine().charAt(0);

            // Populates and connect the graph vertices
            Graph<Character> graph = new Graph(false);
            populateGraph(graph, largestNode);

            // Count the number of subgraphs from the graph method and print
            ↪ it
            int numberOfSubGraphs = graph.numberOfSubGraphsConnected();
            System.out.println(numberOfSubGraphs);

            if (i < nCases - 1) {
                System.out.println();
            }
        }
    }

    // Ignore the number of lines in terminal
    private static void ignoreLines(int numberOfLines) {
        for (int i = 0; i < numberOfLines; i++) {
            SC.nextLine();
        }
    }
```

Código Fonte 12: Solução para o problema *UVa 00459 – Graph Connectivity* (pt. 1).

```java
// Define the graphs nodes and edges using the generic class Graph<T>
    private static void populateGraph(Graph<Character> graph, char
    ↪ largestLetter) {

        /* First of all create the nodes/vertices before connect with the
        ↪  edges
           getting from the first letter (A) to the end letter (largest letter
↪  in input)
        */
        graph.addVertex(generateAlphabet('A', largestLetter));

        // Connect the edges
        while (SC.hasNextLine()) {
            String edgesConnection = SC.nextLine();

            if (edgesConnection.isEmpty()) {
                break;
            }

            // Connect them
            graph.addEdge(edgesConnection.charAt(0),
            ↪  edgesConnection.charAt(1));
        }
    }

    private static List<Character> generateAlphabet(char start, char end) {
        List<Character> lstChars = new LinkedList();

        for (char c = start; c <= end; c++) {
            lstChars.add(c);
        }

        return lstChars;
    }
```

Código Fonte 13: Solução para o problema *UVa 00459 – Graph Connectivity* (pt. 2).

```java
// Graph implemented by an adjancency list where T is a node/vertex
class Graph<T> {

    private final HashMap<T, LinkedList<T>> edges;
    private final boolean isDirectional;

    public Graph(boolean isDirecional) {
        this.edges = new HashMap();
        this.isDirectional = isDirecional;
    }

    public void addVertex(T v) {
        edges.put(v, new LinkedList());
    }

    public void addVertex(List<T> vs) {
        for (T v : vs) {
            edges.put(v, new LinkedList());
        }
    }

    public void addEdge(T source, T destination) {
        if (this.hasEdge(source, destination)) {
            return;
        }

        if (!edges.containsKey(source)) {
            addVertex(source);
        }

        if (!edges.containsKey(destination)) {
            addVertex(destination);
        }

        edges.get(source).add(destination);

        if (!this.isDirectional == true) {
            edges.get(destination).add(source);
        }
    }
```

Código Fonte 14: Solução para o problema *UVa 00459 – Graph Connectivity* (pt. 3).

```
//
    public int numberOfVertices() {
        return this.edges.size();
    }

    public int numberOfEdges() {
        int count = 0;

        for (T v : edges.keySet()) {
            count += edges.get(v).size();
        }

        if (!this.isDirectional == true) {
            return count / 2;
        }

        return count;
    }

    public boolean hasVertex(T s) {
        return this.edges.containsKey(s);
    }

    public boolean hasEdge(T s, T d) {
        return this.edges.get(s).contains(d);
    }

    public Set<T> getAllVertices() {
        return new LinkedHashSet(this.edges.keySet());
    }

    public Set<T> depthFirstSearch(T vertex, boolean countInitialVertex) {
        Set<T> dfsVisitedElements = new LinkedHashSet();

        // Calls the function to get all visited vertices without counting the
        ↪  first one
        if (this.hasVertex(vertex)) {
            DFS_execution(vertex, dfsVisitedElements, countInitialVertex);
        }

        return dfsVisitedElements;
    }
```

Código Fonte 15: Solução para o problema *UVa 00459 – Graph Connectivity* (pt. 4).

```java
public int numberOfSubGraphsConnected() {
        List<Set<T>> subGraphs = new LinkedList();

        for (T vertex : this.getAllVertices()) {

            // Check if the vertex is in some subgraph
            boolean exists = false;
            for (Set<T> subgraph : subGraphs) {
                if (subgraph.contains(vertex)) {
                    exists = true;
                    break;
                }
            }

            // If exists pass to the next vertex to check
            if (exists) {
                continue;
            }

            // Uses the DFS to captures the subgraphs from the current graph
            Set<T> visitedVertices = this.depthFirstSearch(vertex, true);
            subGraphs.add(visitedVertices);
        }

        return subGraphs.size();
    }

    // Returns the adjancency list of each vertex
    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();

        edges.keySet().stream().map((v) -> {
            builder.append(v.toString()).append("-> ");
            return v;
        }).map((v) -> {
            edges.get(v).forEach((w) -> {
                builder.append(w.toString()).append("-> ");
            });
            return v;
        }).forEachOrdered((_item) -> {
            builder.append("null\n");
        });

        return builder.toString();
    }
```

Código Fonte 16: Solução para o problema *UVa 00459 – Graph Connectivity* (pt. 5).

```java
//
    public int numberOfSubGraphsConnected() {
        List<Set<T>> subGraphs = new LinkedList();

        for (T vertex : this.getAllVertices()) {

            // Check if the vertex is in some subgraph
            boolean exists = false;
            for (Set<T> subgraph : subGraphs) {
                if (subgraph.contains(vertex)) {
                    exists = true;
                    break;
                }
            }

            // If exists pass to the next vertex to check
            if (exists) {
                continue;
            }

            // Uses the DFS to captures the subgraphs from the current graph
            Set<T> visitedVertices = this.depthFirstSearch(vertex, true);
            subGraphs.add(visitedVertices);
        }

        return subGraphs.size();
    }

    // Returns the adjancency list of each vertex
    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();

        edges.keySet().stream().map((v) -> {
            builder.append(v.toString()).append("-> ");
            return v;
        }).map((v) -> {
            edges.get(v).forEach((w) -> {
                builder.append(w.toString()).append("-> ");
            });
            return v;
        }).forEachOrdered((_item) -> {
            builder.append("null\n");
        });

        return builder.toString();
    }
```

Código Fonte 17: Solução para o problema *UVa 00459 – Graph Connectivity* (pt. 6).

```
//
    private void DFS_execution(T visitedVertex, Set<T> dfsVisitedVertices,
    ↪  boolean addVisitedVertex) {
        if (visitedVertex == null) {
            return;
        }

        // Adds into the set as a visited vertex
        if (addVisitedVertex) { dfsVisitedVertices.add(visitedVertex); }

        LinkedList<T> adjacencyVertices = this.edges.get(visitedVertex);
        for (T nextVisitedVertex : adjacencyVertices) {

            // If the vertex is already in the Set is because is not necessary
            ↪  to call again
            if (dfsVisitedVertices.contains(nextVisitedVertex)) {
                continue;
            }

            DFS_execution(nextVisitedVertex, dfsVisitedVertices, true);
        }
    }
}
```

Código Fonte 18: Solução para o problema *UVa 00459 – Graph Connectivity* (pt. 7).

2.3.3 UVa 00872 – Ordering

O algoritmo se inicia com a leitura do número de casos do problema, como é mostrado na linha 17 do Código Fonte 19. Após isso para cada interação dos casos é instanciado um grafo não direcionado e nele populado os vértices referentes as variáveis de leitura do problema, como é mostrado nas linhas 22 e 23 da função principal. Dentro da função chamada *readVariablesAndConstraints* além de popular o grafo, como dito anteriormente, é também feita a leitura das constraints, que são basicamente regras que as permutações devem seguir para que sejam válidas.
Continuando na função principal, na linha 26 é chamado o método do grafo instanciado chamado de *permutations*. A partir do Código Fonte 15 é possível visualizar a implementação deste método, onde é responsável por retornar todas as permutações possíveis filtradas segundo as regras lidas de entrada. Na linha 30 é chamado o método *doPermutation* que realiza todas as permutações possíveis e os retorna em uma estrutura de lista de array de lista, onde cada um representa uma permutação. Já na linha 33 é chamado o método, também privado, que filtra todas as permutações a partir das regras passada como argumento do método. Por fim é retornado as permutações validadas e chamado o método *output*, que basicamente imprime as informações da maneira que o problema deseja.
A estratégia utilizada para realizar as permutações está presente no Código Fonte 23, onde primeiramente é checado os casos bases que são em caso de a lista de permutação (*lstPermute*) não possuir nenhum ou um elemento. Após isso, através de um loop, é recuperado da lista de permutação o primeiro elemento da permutação resultante e chamado recursivamente em um loop interno a função *doPermutation*, passando como parâmetro a lista restante, com exceção do primeiro elemento de permutação, como pode ser visto entre as linhas 17 e 27. Logo a ideia é extrair um por um todos os elementos, e sempre colocá-los na primeira posição da lista de combinação (*lstCombination*) e por fim retornar a lista restante até que seja realizada todas as permutações possíveis.

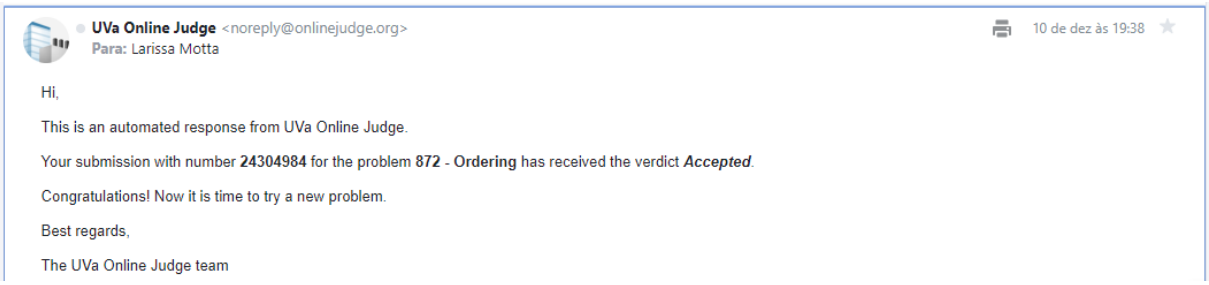

Figura 7: Email de aceitação da solução *UVa 00872 – Ordering*.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.Set;
import java.util.stream.Collectors;

public class Main {

    private static final Scanner SC = new Scanner(System.in);
    private static final String REGEX = " ";

    public static void main(String[] args) throws Exception {
        int nCases = SC.nextInt();
        ignoreLines(2);

        for (int i = 0; i < nCases; i++) {
            // Populates the graph and reads the variables and constraints
            Graph<String> graph = new Graph(false);
            String[] constraints = readVariablesAndConstraints(graph);

            // Get all the permutations from the graph
            List<ArrayList<String>> permutations =
            ↪  graph.permutations(constraints);
            output(permutations);

            if (i < nCases - 1) {
                System.out.println();
            }
        }
    }

    // Ignore the number of lines in terminal
    private static void ignoreLines(int numberOfLines) {
        for (int i = 0; i < numberOfLines; i++) {
            SC.nextLine();
        }
    }
```

Código Fonte 19: Solução para o problema *UVa 00872 – Ordering* (pt. 1).

```java
// Reads the variable and constraints of the given input and returns the
↪  constraints
    private static String[] readVariablesAndConstraints(Graph<String> graph)
    ↪  throws Exception {
        String[] variables = SC.nextLine().split(REGEX);
        String[] constraints = SC.nextLine().split(REGEX);

        // Reads the empty line
        if (SC.hasNextLine()) {
            SC.nextLine();
        }

        // Validating the rules in the given problem description
        if (variables.length < 2 || variables.length > 20) {
            throw new Exception("The numbers of variables must be greater than
            ↪  or equal to 2 and less than or equal to 20.");
        }
        if (constraints.length < 1 || constraints.length > 50) {
            throw new Exception("The numbers of constraints must be greater
            ↪  than or equal to 1 and less than or equal to 50.");
        }

        // Populates the graph with the variables as vertices
        graph.addVertex(variables);

        return constraints;
    }

    private static void output(List<ArrayList<String>> permutations) {
        if (permutations.isEmpty()) {
            System.out.println("NO");
        }
        List<String> lstStrPerm = new ArrayList();

        // Converting the List of string to a List of strings
        for (List<String> perm : permutations) {
            lstStrPerm.add(perm.stream()
                               .map(String::valueOf)
                               .collect(Collectors.joining(REGEX)));
        }

        // After this sort the list of permutations and then print them out
        Collections.sort(lstStrPerm);
        for (int i = 0; i < lstStrPerm.size(); i++) {
            System.out.println(lstStrPerm.get(i));
        }
    }
}
```

Código Fonte 20: Solução para o problema *UVa 00872 – Ordering* (pt. 2).

```java
// Graph implemented by an adjancency list where T is a node/vertex
class Graph<T> {

    private final HashMap<T, LinkedList<T>> edges;
    private final boolean isDirectional;

    public Graph(boolean isDirecional) {
        this.edges = new HashMap();
        this.isDirectional = isDirecional;
    }

    public void addVertex(T v) {
        this.edges.put(v, new LinkedList());
    }

    public void addVertex(T[] vs) {
        for (T v : vs) {
            this.edges.put(v, new LinkedList());
        }
    }

    public void addEdge(T source, T destination) {
        if (this.hasEdge(source, destination)) {
            return;
        }

        if (!this.edges.containsKey(source)) {
            this.addVertex(source);
        }

        if (!this.edges.containsKey(destination)) {
            this.addVertex(destination);
        }

        this.edges.get(source).add(destination);

        if (!this.isDirectional == true) {
            this.edges.get(destination).add(source);
        }
    }

    public int numberOfVertices() {
        return this.edges.size();
    }
```

Código Fonte 21: Solução para o problema *UVa 00872 – Ordering* (pt. 3).

```
//
    public int numberOfEdges() {
        int count = 0;

        for (T v : this.edges.keySet()) {
            count += this.edges.get(v).size();
        }

        if (!this.isDirectional == true) {
            return count / 2;
        }

        return count;
    }

    public boolean hasVertex(T s) {
        return this.edges.containsKey(s);
    }

    public boolean hasEdge(T s, T d) {
        return this.edges.get(s).contains(d);
    }

    public Set<T> getAllVertices() {
        return new LinkedHashSet(this.edges.keySet());
    }

    public List<ArrayList<T>> permutations(String constraints[]) throws
    ↪ Exception {
        ArrayList<T> vertices = new ArrayList(this.edges.keySet());
        List<ArrayList<T>> result = doPermutation(vertices);

        // Filter the permutations by the constraint given
        this.filterBasedOnConstraints(result, constraints);
        if (result.size() > 300) {
            throw new Exception("The numbers of orderings consistent with the
            ↪  constraints must be less than or equal to 300.");
        }

        return result;
    }
```

Código Fonte 22: Solução para o problema *UVa 00872 – Ordering* (pt. 4).

```
//
    private List<ArrayList<T>> doPermutation(ArrayList<T> lstPermute) {
        List<ArrayList<T>> structurePermute = new ArrayList();

        // Check the bases of the collection (zero or one element)
        if (lstPermute.isEmpty()) {
            return structurePermute;
        }

        if (lstPermute.size() == 1) {
            structurePermute.add(lstPermute);
            return structurePermute;
        }

        // Check for more than one element
        for (int i = 0; i < lstPermute.size(); i++) {
            T firstElementPermute = lstPermute.get(i);

            // Retrieve all the list except the first element of permutation
            ArrayList<T> lstRetrieve = (ArrayList<T>) lstPermute.clone();
            lstRetrieve.remove(i);

            // Generating all permutations from the retrieve list
            for (ArrayList<T> lstCombination : this.doPermutation(lstRetrieve))
            ↪ {
                // firstElementPermute is always the first element of the
                ↪ current permutation
                lstCombination.add(0, firstElementPermute);
                structurePermute.add(lstCombination);
            }
        }

        return structurePermute;
    }
```

Código Fonte 23: Solução para o problema *UVa 00872 – Ordering* (pt. 5).

```
//
    private void filterBasedOnConstraints(List<ArrayList<T>> permutations,
    ↪  String[] constraints) {
        if (constraints.length == 0) {
            return;
        }

        /* For each constraint check if filter the permutations wich is not
           following the rules of the constraint
         */
        List<ArrayList<T>> removePermutations = new ArrayList();
        for (String constraint : constraints) {
            String var1 = String.valueOf(constraint.charAt(0));
            String var2 = String.valueOf(constraint.charAt(2));

            for (ArrayList<T> lstPermut : permutations) {

                // Get the index of the variables inside the list of
                ↪  permutations
                int indexVar1 = lstPermut.indexOf(var1);
                int indexVar2 = lstPermut.indexOf(var2);

                if (indexVar1 == -1 || indexVar2 == -1) {
                    continue;
                }

                /* Always var1 < var2, so if index of var1 > index of var2
                ↪  adds
                   adds inside the removePermutations list
                 */
                if (indexVar1 > indexVar2) {
                    removePermutations.add(lstPermut);
                }
            }
        }

        // Remove all failed permutations according to the constraints given
        permutations.removeAll(removePermutations);
    }
```

Código Fonte 24: Solução para o problema *UVa 00872 – Ordering* (pt. 6).

```
    // Returns the adjancency list of each vertex
    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();

        edges.keySet().stream().map((v) -> {
            builder.append(v.toString()).append("-> ");
            return v;
        }).map((v) -> {
            edges.get(v).forEach((w) -> {
                builder.append(w.toString()).append("-> ");
            });
            return v;
        }).forEachOrdered((_item) -> {
            builder.append("null\n");
        });

        return builder.toString();
    }
}
```

Código Fonte 25: Solução para o problema *UVa 00872 – Ordering* (pt. 7).

2.3.4 UVa 10034 – Freckles

O Código Fonte 26 apresenta a função principal do algoritmo, onde na linha 18 faz a leitura do número de casos do problema. Logo depois na linha 22 é lido o número de *freckles* (sardas) do caso corrente. Na linha 26 é instanciado um grafo não direcionado e tipado pela classe *Freckle*. Basicamente esta classe *Freckle* representa as coordenadas em que a sarda está presente nas costas do indivíduo do problema.
No mesmo código fonte na linha 31 é feita a chamada da função *identifyAndConnectFreckles*, onde são instanciados os n sardas do problema, assim como sua posições, e inserido dentro do grafo de forma a conectar todos os freckles criados, ou seja, criando um grafo de freckles completo. Perceba que no Código Fonte 27 que o peso calculado de cada ligação dos vértices do grafo é feita pelo cálculo da distância euclidiana entre dois freckles distintos, como é mostrado na linha 22.
Após isso, voltando a função principal, é chamado o método da classe *Graph* mais importante para resolução do problema chamado de *minimumCostConnectAllVertices* presente na linha 37. Ele é responsável por determinar o menor custo total para conectar todos os vértices do grafo, ou seja, conectar todos os freckles. A estratégia utilizada foi a seguindo a ideia do algoritmo de *dijkstra*, que soluciona o problema do caminho mais curto para grafos dirigidos ou não dirigidos com arestas de pesos positivos, como é o caso do problema. Nos Código Fonte 32 e Código Fonte 33 estão presentes a implementação do método. Note que no primeiro loop *for* é atualizado os pesos de todos os vértices adjacentes ao vértice atual que está sendo analisado. Após a atualização é executado outro loop *for*, onde este é responsável por encontrar o vértice com o menor caminho em relação ao vértice atual baseando pelo peso da aresta. Após isso é definido o vértice que será analisado novamente pelo algoritmo e marcado o vértice anterior que estava sendo analisado através da estrutura *HashSet* chamada *closed*, onde o vértice é adicionado não sendo analisado novamente. A cada loop que é encontrado o menor caminho entre os vértices e é somado o valor do peso da aresta, o qual este é retornado no fim do método.

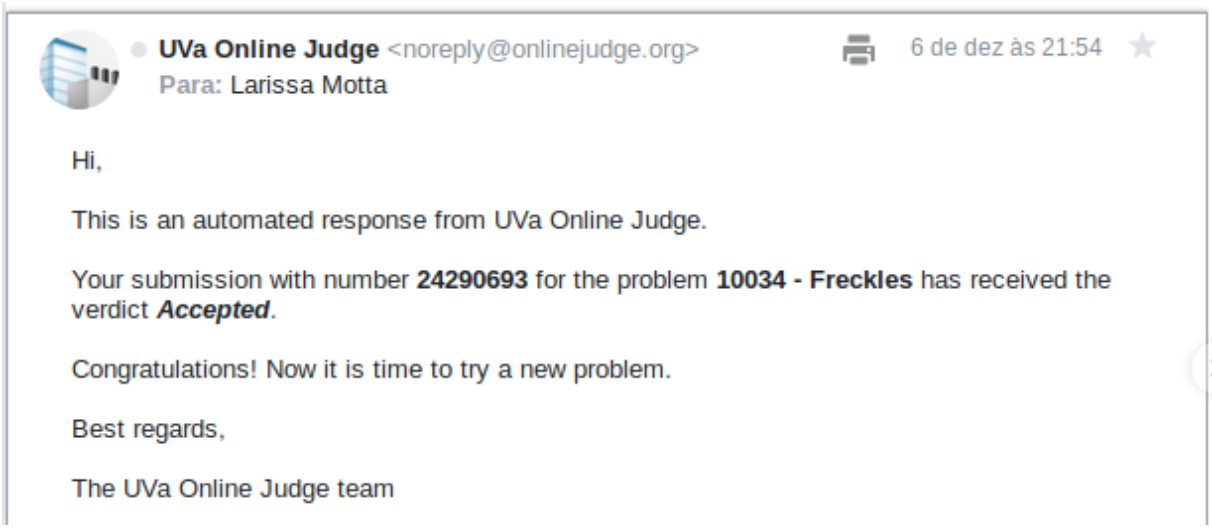

Figura 8: Email de aceitação da solução *UVa 10034 – Freckles*.

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Objects;
import java.util.Scanner;
import java.util.Set;
import java.util.stream.Collectors;

public class Main {

    private static final Scanner SC = new Scanner(System.in);
    private static final String REGEX = " ";

    public static void main(String args[]) {
        int nCases = SC.nextInt();
        //ignoreLines(2); // /n input and the blank line

        for (int i = 1; i <= nCases; i++) {
            int nFreckles = SC.nextInt();
            ignoreLines(1);

            // Creates a non direction graph of freckles
            Graph<Freckle> frecklesGraph = new Graph(false);

            /* Connect the edges of all the vertices connecting all the
            ↪  freckles
               and calculating the weight by the euclidian distance
            */
            List<Freckle> freckles = identifyAndConnectFreckles(frecklesGraph,
            ↪  nFreckles);
            //ignoreLines(1); // Ignore the next blank line

            // Finds the minimum cost of ink to connect all the lines
            if (!freckles.isEmpty()) {
                Freckle initialFreckle = freckles.get(0);
                double cost =
                ↪  frecklesGraph.minimumCostConnectAllVertices(initialFreckle);

                System.out.printf("%.2f\n", cost);
                if (i < nCases) { System.out.println(); }
            }
        }
    }
```

Código Fonte 26: Solução para o problema *UVa 10034 – Freckles* (pt. 1).

```java
/* Get all freckles from the current case and creates it populating the graph
       and then connect to all other freckles inside the graph
    */
    private static List<Freckle> identifyAndConnectFreckles(Graph<Freckle>
    ↪  graph, int qntFreckles) {
        List<Freckle> frecklesAux = new ArrayList();

        for (int i = 1; i <= qntFreckles; i++) {
            String[] coordinates = SC.nextLine().split(REGEX);
            double x = Double.parseDouble(coordinates[0]);
            double y = Double.parseDouble(coordinates[1]);

            // Creates a freckle where the i is his identifier then adds into
            ↪  the graph
            Freckle currentFreckle = new Freckle(i, x, y);
            graph.addVertex(currentFreckle);

            // Adds into freckles list to make the connections among the other
            ↪  freckles
            frecklesAux.add(currentFreckle);

            // Connects the current freckle to all the others already create
            ↪  it
            for (int k = frecklesAux.size() - 2; k >= 0; k--) {
                Freckle destFreckle = frecklesAux.get(k);
                double weight = currentFreckle.euclidianDistance(destFreckle);
                graph.addEdge(currentFreckle, destFreckle, weight);
            }
        }

        return frecklesAux;
    }

    // Ignore the number of lines in terminal
    private static void ignoreLines(int numberOfLines) {
        for (int i = 0; i < numberOfLines; i++) {
            SC.nextLine();
        }
    }
}
```

Código Fonte 27: Solução para o problema *UVa 10034 – Freckles* (pt. 2).

```java
// Simple representation of the freckles coordinates
class Freckle {

    private int id;
    public double x;
    public double y;

    public Freckle(int identifier, double x, double y) {
        this.id = identifier;
        this.x = x;
        this.y = y;
    }

    public int checkId() {
        return this.id;
    }

    public double euclidianDistance(Freckle f) {
        double deltaX = f.x - this.x;
        double deltaY = f.y - this.y;

        return Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));
    }

    @Override
    public int hashCode() {
        int hash = 3;
        hash = 83 * hash + this.id;

        return hash;
    }

    @Override
    public boolean equals(Object o) {
        if (o == null || this == null || this.getClass() != o.getClass()) {
            return false;
        }

        Freckle f = (Freckle) o;

        return this.id == f.id;
    }

    @Override
    public String toString() {
        return "" + this.id;
    }
}
```

Código Fonte 28: Solução para o problema *UVa 10034 – Freckles* (pt. 3).

```java
// Graph implemented by an adjancency list way
class Graph<T> {

    private final static class Edge<T> {
        private T srcVertex;
        private T destVertex;
        private double weight;

        public Edge(T src, T dest) {
            this.srcVertex = src;
            this.destVertex = dest;
            this.weight = 0d;
        }
        public Edge(T src, T dest, double weight) {
            this.srcVertex = src;
            this.destVertex = dest;
            this.weight = weight;
        }

        public void changeWeight(double weight) {
            this.weight = weight;
        }

        public void changeConnection(T src, T dest) {
            if (src != null) {
                this.srcVertex = src;
            }
            if (dest != null) {
                this.destVertex = dest;
            }
        }

        @Override
        public int hashCode() {
            int hash = 7;
            hash = 53 * hash + Objects.hashCode(this.srcVertex);
            hash = 53 * hash + Objects.hashCode(this.destVertex);

            return hash;
        }
        @Override
        public boolean equals(Object o) {
            if (o == null || this == null || this.getClass() != o.getClass()) {
                return false;
            }
            Edge<T> e = (Edge) o;
            return this.srcVertex.equals(e.srcVertex) &&
            ↪  this.destVertex.equals(e.destVertex);
        }
    }
```

Código Fonte 29: Solução para o problema *UVa 10034 – Freckles* (pt. 4).

```
//
    private final HashMap<T, LinkedList<Edge<T>>> edges;
    private final boolean isDirectional;

    public Graph(boolean isDirectional) {
        this.edges = new HashMap();
        this.isDirectional = isDirectional;
    }

    public void addVertex(T v) {
        edges.put(v, new LinkedList());
    }

    public void addEdge(T source, T destination, double weight) {
        if (!edges.containsKey(source)) {
            addVertex(source);
        }

        if (!edges.containsKey(destination)) {
            addVertex(destination);
        }

        Edge<T> srcEdge = new Edge(source, destination, weight);
        this.edges.get(source).add(srcEdge);

        if (!this.isDirectional) {
            Edge<T> destEdge = new Edge(destination, source, weight);
            this.edges.get(destination).add(destEdge);
        }
    }

    public int numberOfVertices() {
        return this.edges.size();
    }

    public int numberOfEdges() {
        int count = 0;

        for (T v : edges.keySet()) {
            count += edges.get(v).size();
        }

        if (!this.isDirectional) {
            return count / 2;
        }

        return count;
    }
```

Código Fonte 30: Solução para o problema *UVa 10034 – Freckles* (pt. 5).

```
//
    public boolean hasVertex(T v) {
        return this.edges.containsKey(v);
    }

    public boolean hasEdge(T s, T d) {
        Edge<T> edge = new Edge(s, d);

        return this.edges.get(s).contains(edge);
    }

    public Set<T> getAllVertices() {
        return new LinkedHashSet(this.edges.keySet());
    }

    public Set<T> depthFirstSearch(T vertex, boolean countInitialVertex) {
        Set<T> dfsVisitedElements = new LinkedHashSet();

        // Calls the function to get all visited vertices without counting the
        ↪  first one
        if (this.hasVertex(vertex)) {
            DFS_execution(vertex, dfsVisitedElements, countInitialVertex);
        }

        return dfsVisitedElements;
    }

    // Returns the adjancency list of each vertex
    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();

        for (T v : this.edges.keySet()) {
            builder.append(v.toString()).append("-> ");

            for (T w : this.getAdjacencyVertices(v)) {
                builder.append(w.toString()).append("-> ");
            }

            builder.append("null\n");
        }

        return builder.toString();
    }
```

Código Fonte 31: Solução para o problema *UVa 10034 – Freckles* (pt. 6).

```java
// Returns the minimum sum of weight edge to connect all vertices
    public double minimumCostConnectAllVertices(T InitialVertex) {
        // Uses the dikjstra approach to take the result

        int qntVertices = this.numberOfVertices();
        Set<T> vertices = this.edges.keySet();

        /* Distance of all vertices where T is the vertex and initialize
           all of them as infinite
        */
        HashMap<T, Double> distanceMap = new HashMap();

        for (T v : vertices) {
            distanceMap.put(v, Double.POSITIVE_INFINITY);
        }
        distanceMap.put(InitialVertex, 0d);

        // A Set of object that has been selected as a short path based on edge
        ↪ weight
        HashSet<T> closed = new HashSet();

        double totalSum = 0d;
        T analysingVertex = InitialVertex;

        while (!closed.contains(analysingVertex)) {
            // Adds into closed wich means there will not be analysed anymore
            totalSum += distanceMap.get(analysingVertex);
            closed.add(analysingVertex);

            if (closed.size() == qntVertices) {
                break;
            }

            // Update the weight of the vertex adjacent
            for (Edge<T> edgeAdjacent : this.edges.get(analysingVertex)) {
                T adjacentVertex = edgeAdjacent.destVertex;

                // Is not necessary to update distance if the vertex is
                ↪ visited
                if (closed.contains(adjacentVertex)) {
                    continue;
                }

                double adjcentVertexWeight = distanceMap.get(adjacentVertex);

                if (adjcentVertexWeight > edgeAdjacent.weight) {
                    distanceMap.put(adjacentVertex, edgeAdjacent.weight);
                }
            }
```

Código Fonte 32: Solução para o problema *UVa 10034 – Freckles* (pt. 7).

```
            // Take the next vertex to analyze based on his weight edge
            double minWeight = Double.POSITIVE_INFINITY;
            for (T adjacentVertex: distanceMap.keySet()) {

                // Is not necessary to update distance if the vertex is visited
                if (closed.contains(adjacentVertex)) {
                    continue;
                }

                double adjcentVertexWeight = distanceMap.get(adjacentVertex);

                if (minWeight > adjcentVertexWeight) {
                    minWeight = adjcentVertexWeight;
                    analysingVertex = adjacentVertex;
                }
            }
        }
        //double totalSum = distanceMap.values().stream().mapToDouble(i ->
        ↪  i).sum();

        return totalSum;
    }
```

Código Fonte 33: Solução para o problema *UVa 10034 – Freckles* (pt. 8).

```
//
    private void DFS_execution(T visitedVertex, Set<T> dfsVisitedVertices,
    ↪  boolean addVisitedVertex) {
        if (visitedVertex == null) {
            return;
        }

        // Adds into the set as a visited vertex
        if (addVisitedVertex) {
            dfsVisitedVertices.add(visitedVertex);
        }

        List<T> adjacencyVertices = this.getAdjacencyVertices(visitedVertex);

        for (T nextVisitedVertex : adjacencyVertices) {
            // If the vertex is already in the Set is because is not necessary
            ↪  to call again
            if (dfsVisitedVertices.contains(nextVisitedVertex)) {
                continue;
            }

            DFS_execution(nextVisitedVertex, dfsVisitedVertices, true);
        }
    }

    /* Get all adjacency vertices from the list of edges of the current
    ↪  vertex,
       wich means all the destination vertices
     */
    private List<T> getAdjacencyVertices(T vertex) {

        return this.edges.get(vertex).stream().map(e ->
        ↪  e.destVertex).collect(Collectors.toList());
        //.collect(Collectors.toCollection(LinkedList::new));
    }
}
```

Código Fonte 34: Solução para o problema *UVa 10034 – Freckles* (pt. 9).