

Trabalho prático: Relatório

PERT/CPM Project

Estruturas de Informação (ESINF)

Docente: Ana Maria Madureira (AMD)

Guilherme Sousa nº1130638

João Oliveira nº1130537

Data: 6/10/2015

# Índice

1

1ªa Parte, 4

B

Bibliografia, 9

C

Complexidade Temporal (Big-Oh), 5

Conclusão, 9

D

Desenvolvimento, 4

Diagrama de Classes, 4

Diagrama de Sequência, 5

I

Introdução, 3

T

Testes e Resultados, 8

# Introdução

Neste semestre, na cadeira de Estruturas de Informação vamos desenvolver um projeto usando a “ferramenta” de PERT (Project Evaluation and Review Technique) em conjunto com um algoritmo de agendamento de um conjunto de atividades do projeto, também conhecido por CPM (Critical Path Method). O objetivo deste trabalho é especificar e modular uma aplicação de gestão de atividades de projeto, assim como, providenciar um conjunto particular de funcionalidades típicas destas aplicações. Nesta segunda parte do trabalho vamos tirar partido da programação orientada a objetos, para conseguirmos construir um gráfico PERT/CPM a partir das atividades “carregadas “ na primeira parte.

# Desenvolvimento

# 1ªa Parte

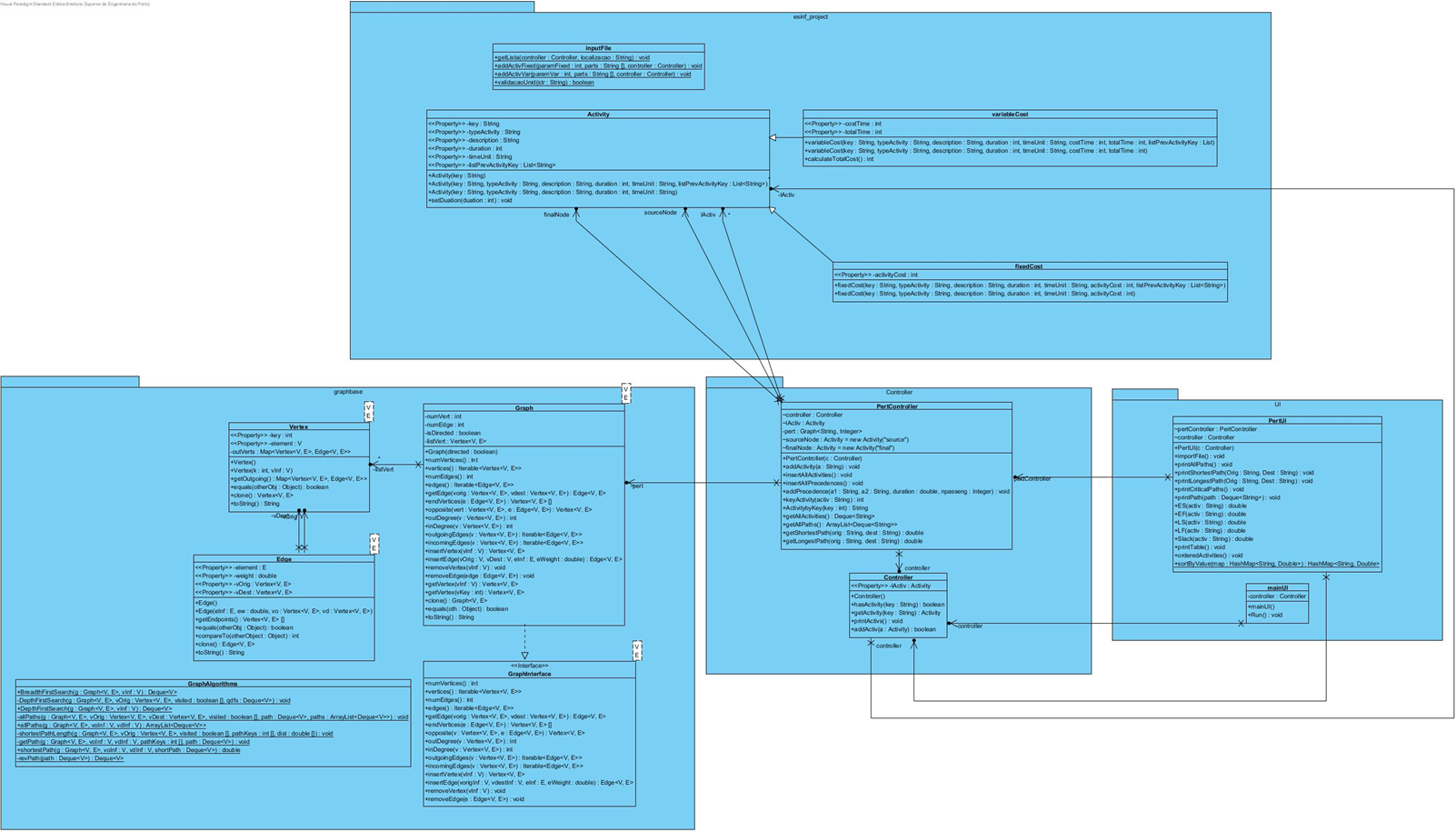
Nesta primeira parte tivemos como objetivo a criação das classes atividade, atividade fixa e actividade variavel. Consideramos uma relação relação de polimorfismo já que atividade fixa e variável partilham alguns atributos e são ambas atividades.

Além disso também tivemos de implementar a leitura de atividades de um ficheiro de texto tendo como base um exemplo dado.

# 2ªa Parte

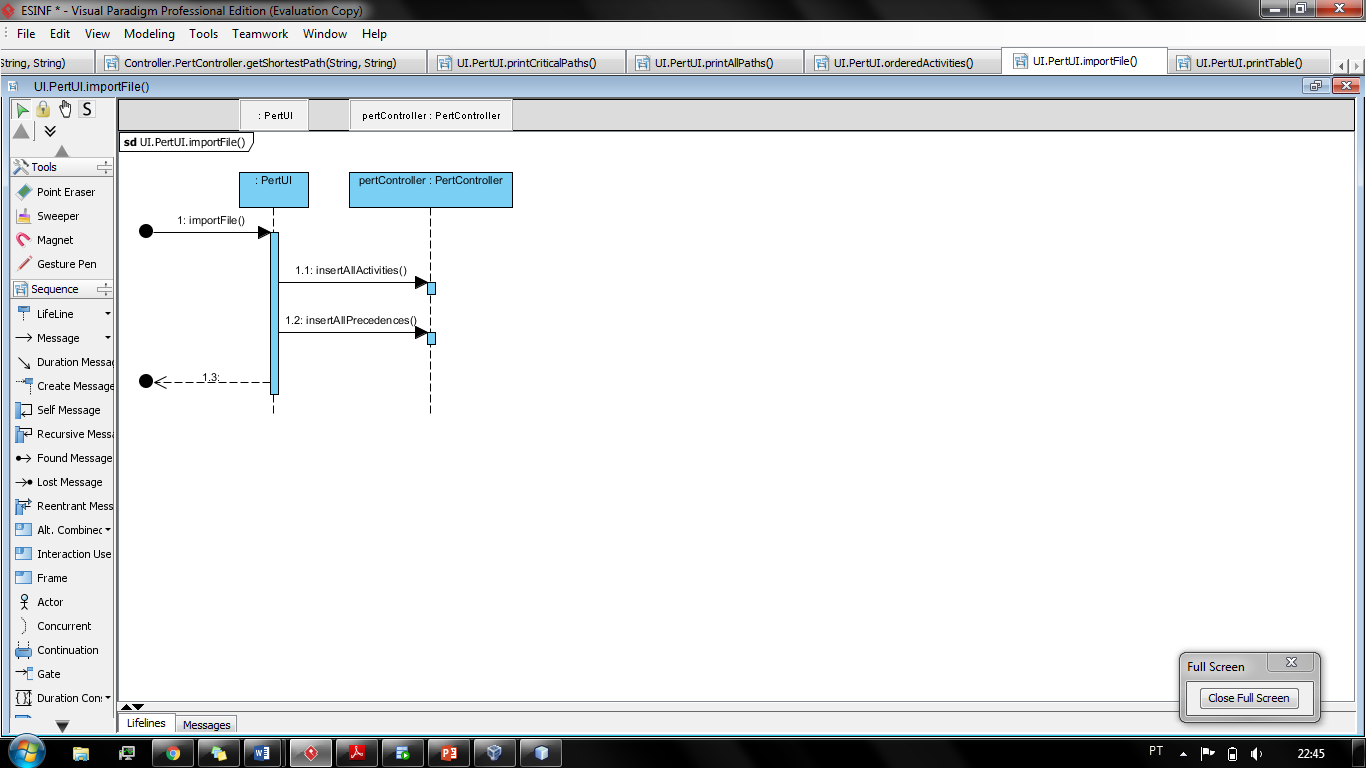
Nesta segunda parte tivemos como objetivo a criação do gráfico PERT/CPM a partir dos objetos carregados na 1ª parte, para tal implementamos novas classes controllers e UI para em primeira instância carregarmos toda a informação para o grafo e depois imprimi-la apresentando depois resposta aos problemas colocados, nomeadamente, a ordem por qual cada atividade é completada, todos os caminhos desde o “source node” ao “finish node” e apresentar a informação das atividades relativa ao “Earliest Start Time”, “Earliest Finish Time”, “Latest Start Time”, “Latest Finish Time” e “Slack”.No entanto, as classes que mais importância tiveram nesta 2ªa parte foram as que estão relacionadas com os grafos em si, pois sem estas classes não tínhamos conseguido finalizar o trabalho.

# Diagrama de Classes

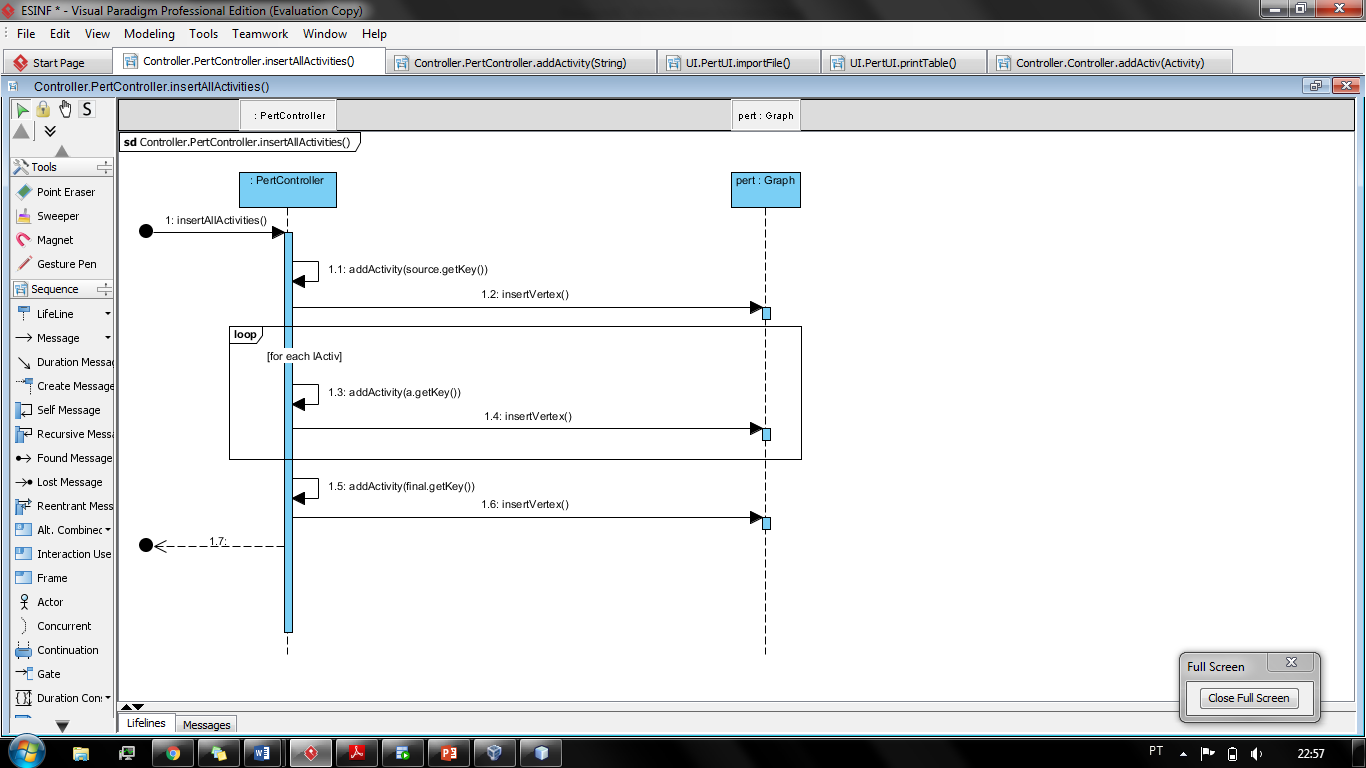
****

# Diagrama de Sequência

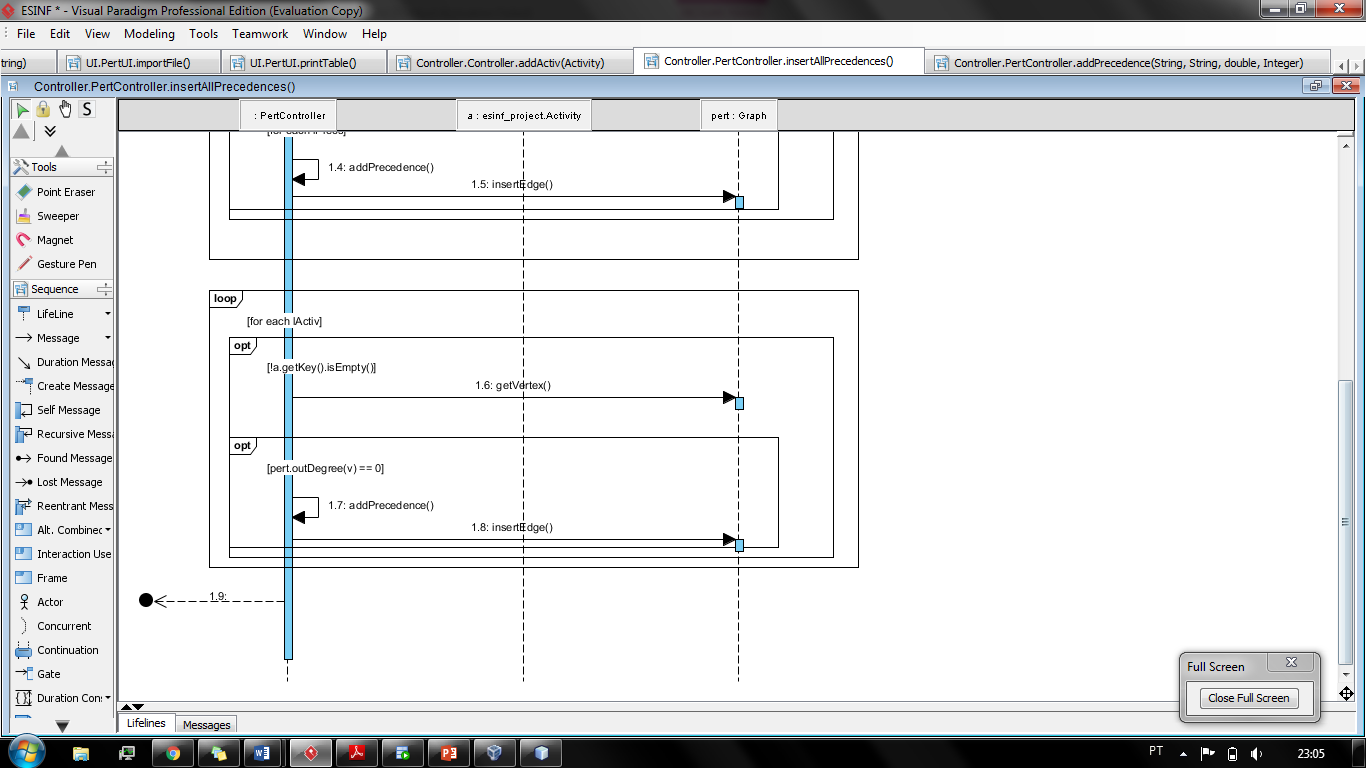
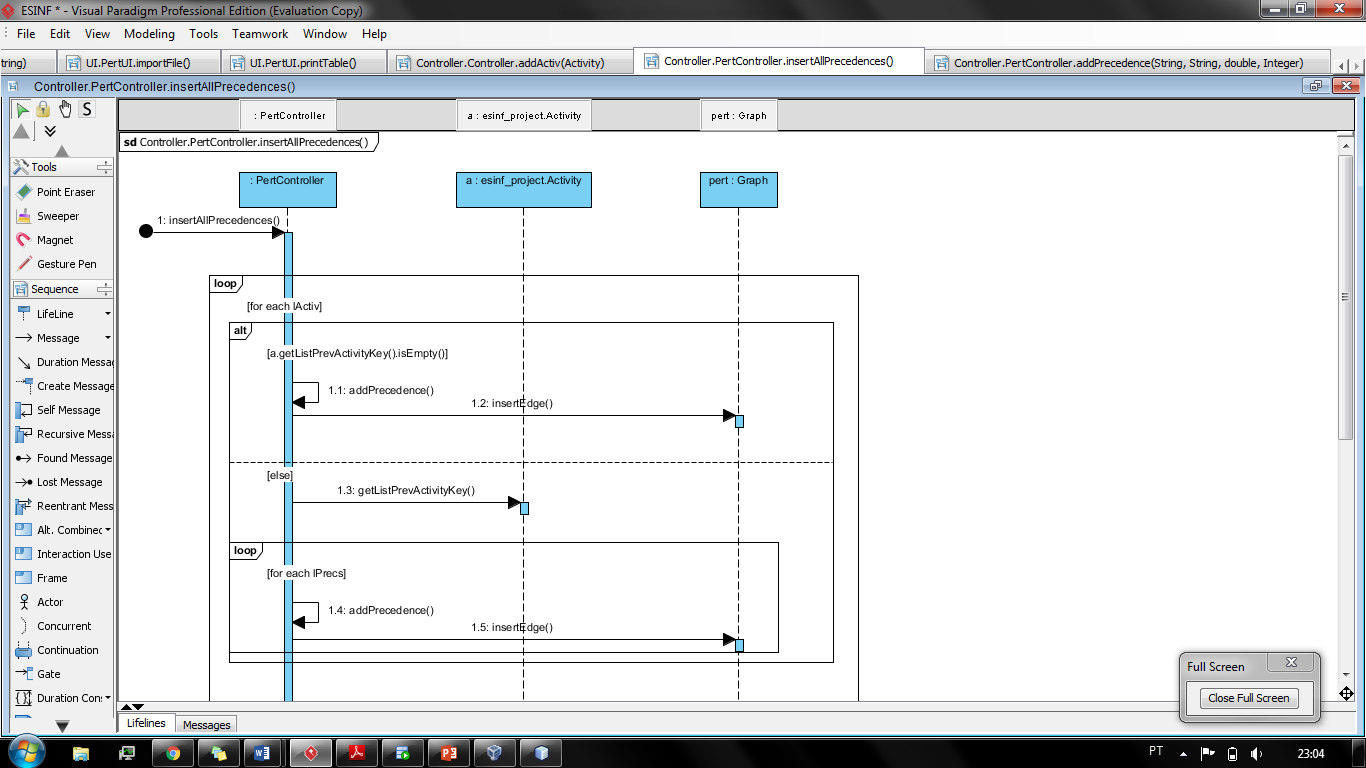
**importFile()** – Método que adiciona ao grafo as atividades obtidas através da leitura de um ficheiro.

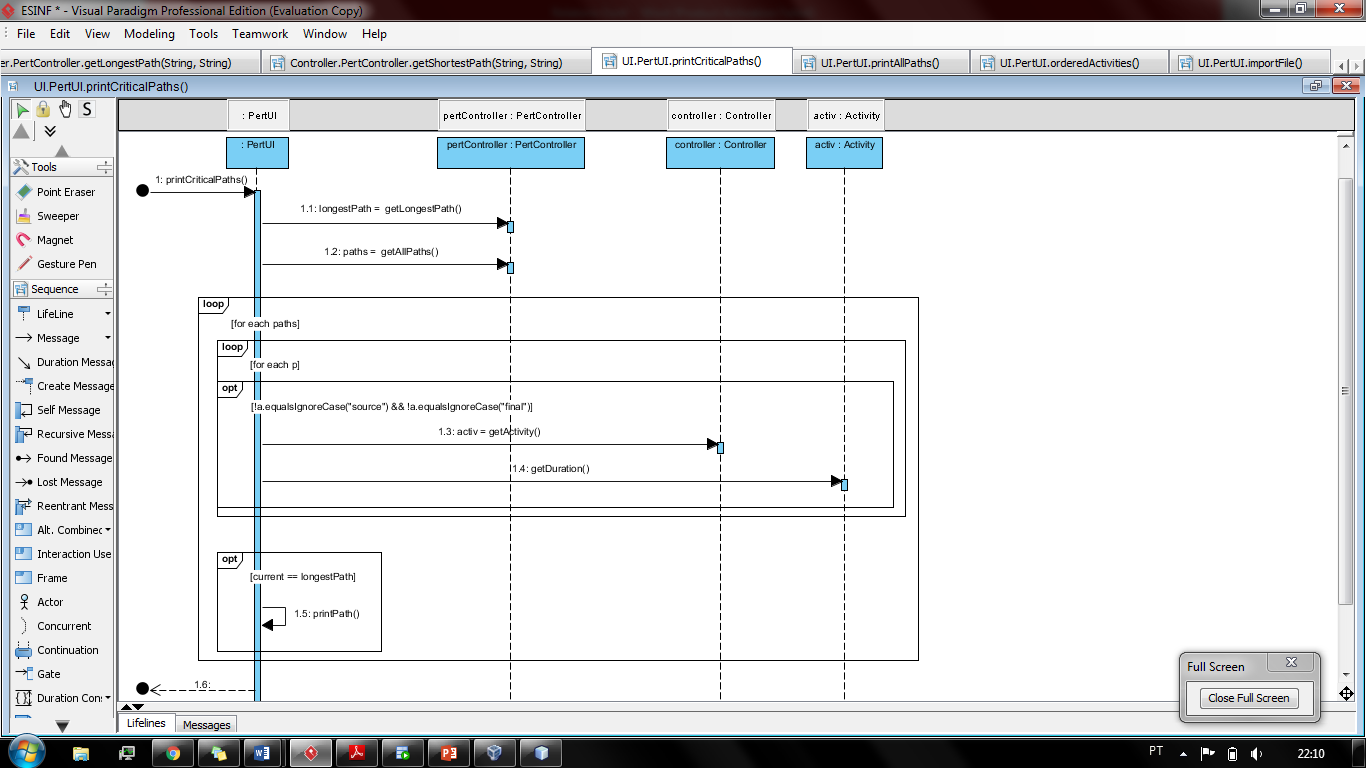


**insertAllActivities()** – Método que insere no grafo todas as atividades como vértices.

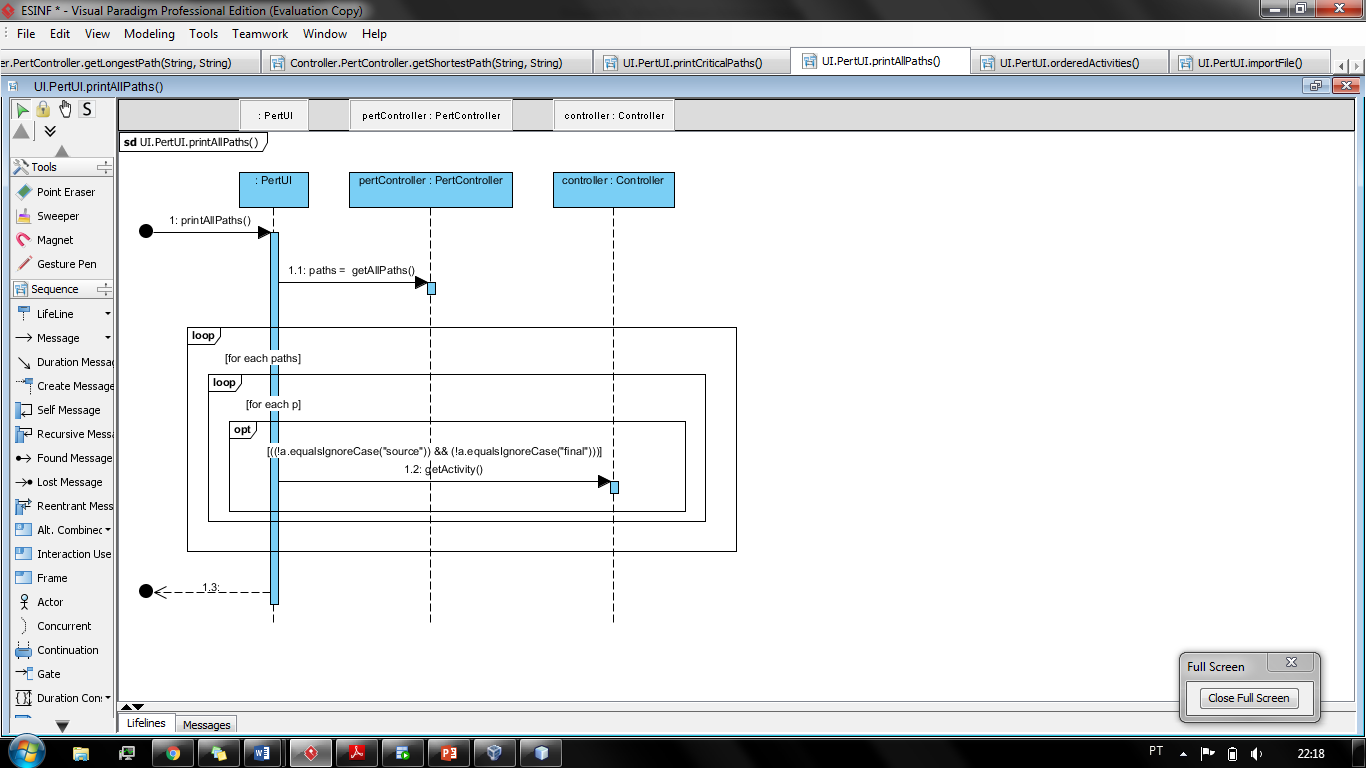


**insertAllPrecedences()** – Método que através das listas de precedências de cada atividade cria todos os edges, com a respectiva duração.

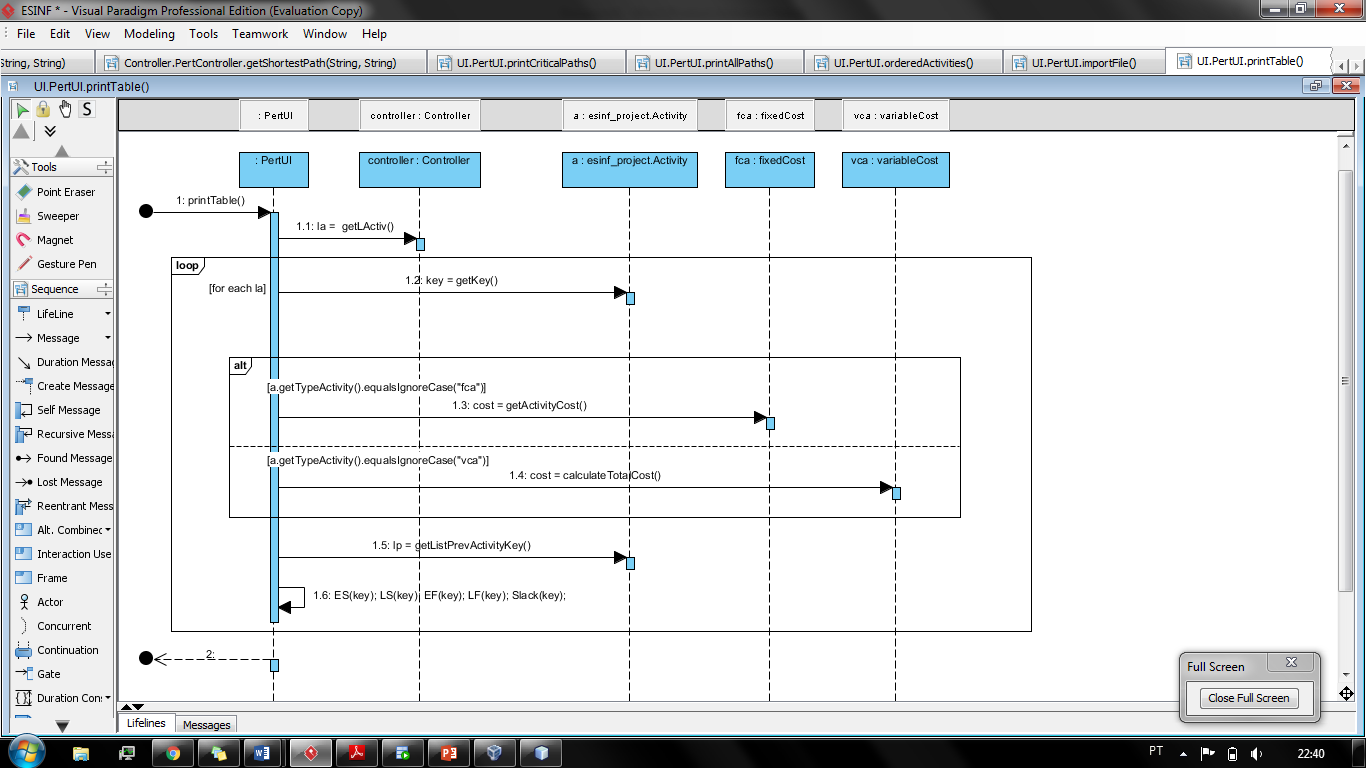


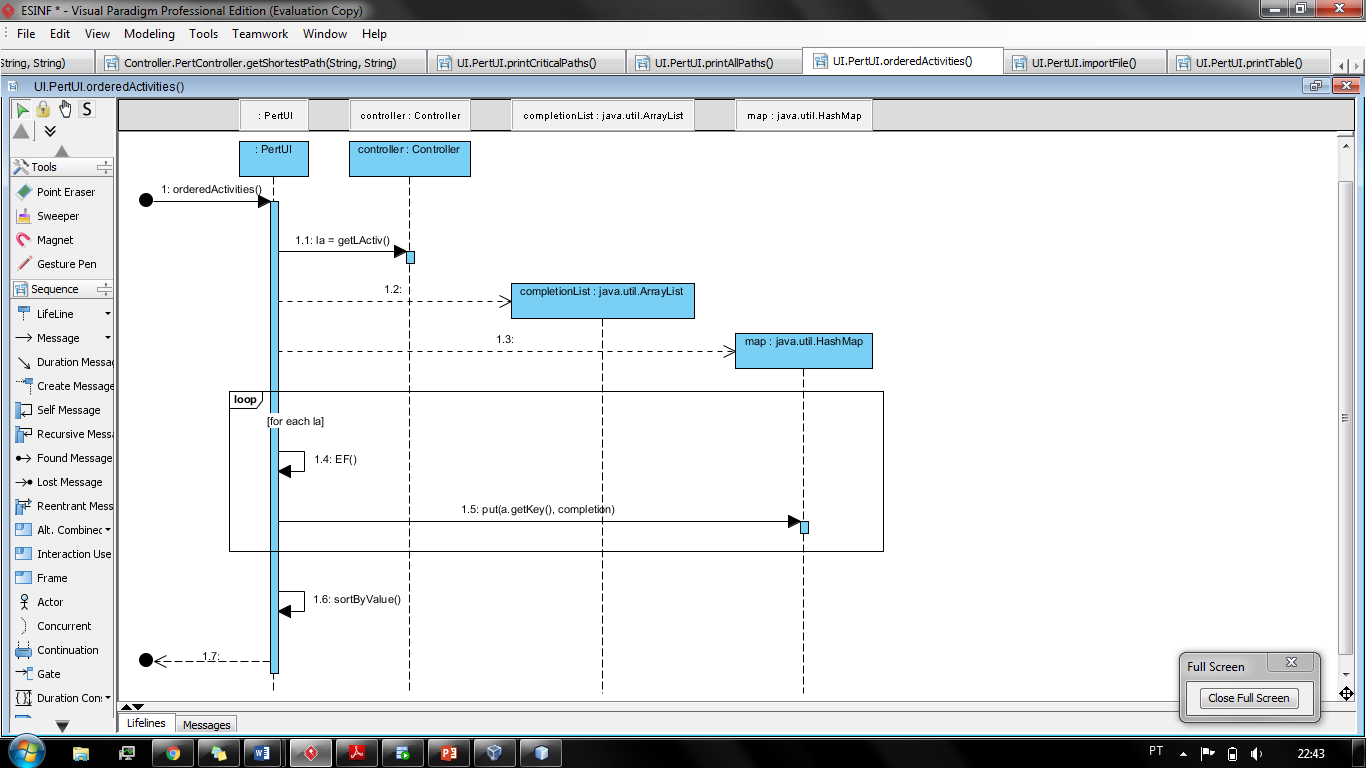
**printCriticalPaths()** – Método que obtém e imprime todos os caminhos críticos, isto é, todos os caminhos que tenham a mesma duração que o caminho mais longo do projecto.

**printAllPaths()** – Método que obtém e imprime todos os caminhos possíveis entre o “source node” e o “final node com a respetiva duração.

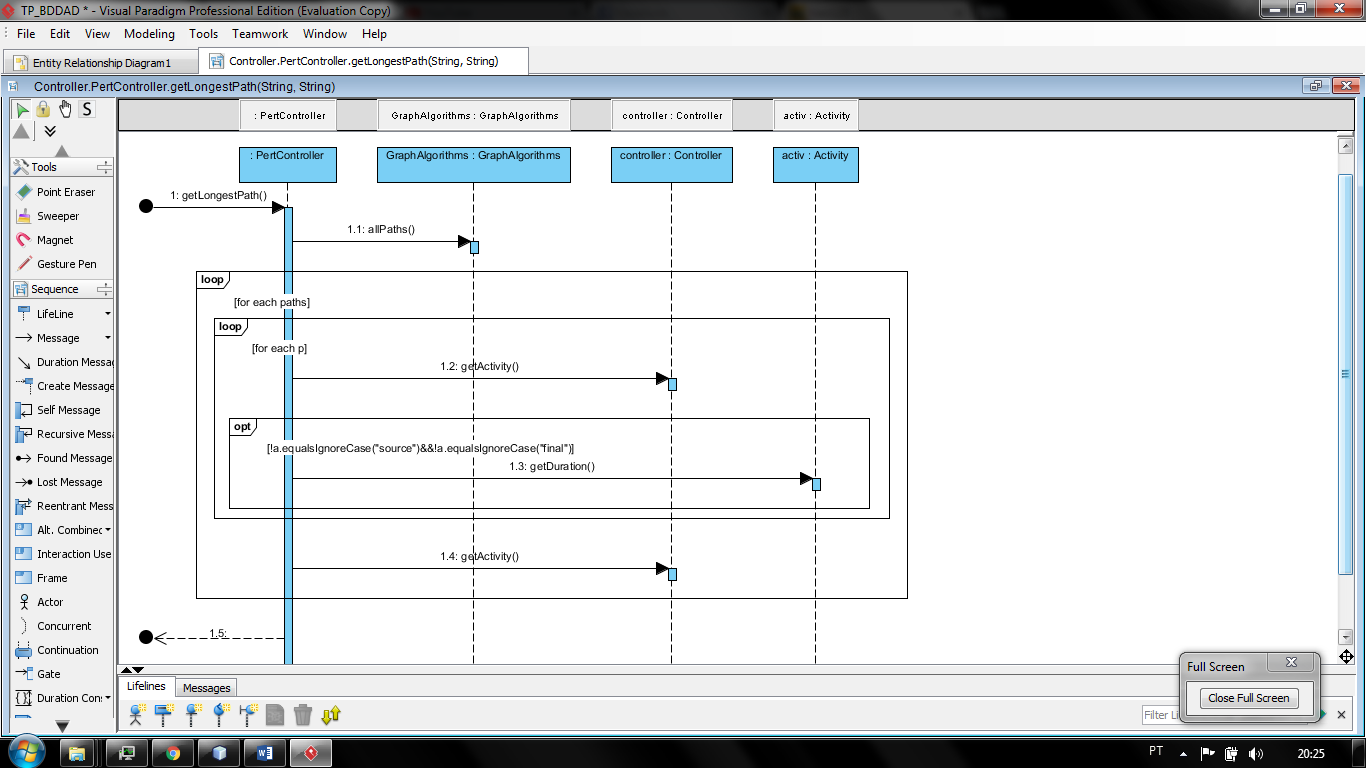


**printTable()** – Método que imprime cada atividade e as respectivas informações, tais como o seu custo, a duração, as atividades precedentes, o earliest start, o latest finish, o earliest finish, o latest finish e a slack da atividade.



**orderedActivities()** – Método que imprime cada atividade e as respectivas informações, tais como o seu custo, a duração, as atividades precedentes, o earliest start, o latest finish, o earliest finish, o latest finish e a slack da atividade.

**getLongestPath()** – Método que obtém o valor do caminho mais longo entre dois vertíces.



# Complexidade Temporal (Big-Oh)

|  |  |
| --- | --- |
| **Método getLista() da classe inputFile** | **O()** |
| int paramFixed = 6; | 1 |
| int paramVar=7; | 1 |
|  |  |
| BufferedReader br = new BufferedReader(new FileReader(localizacao)); | 1 |
| try { |  |
|  |  |
| String line = br.readLine(); | 1 |
|  |  |
| while (line != null) { | **n** |
| String[] parts = line.split(","); | n |
|  |  |
| if (parts[1].equalsIgnoreCase("fca")) { | n |
|  |  |
| if (parts.length > paramFixed) { | n |
| **addActivFixed(paramFixed, parts, controller);** | n**\*n** |
| } else { |  |
| fixedCost fixed = new fixedCost(parts[0], parts[1], parts[2], Integer.parseInt(parts[3]),parts[4], Integer.parseInt(parts[5])); | n |
| **controller.addActiv(fixed);** | **n\*n** |
| } |  |
| } |  |
| if (parts[1].equalsIgnoreCase("vca")) { | n |
| **addActivVar(paramVar, parts, controller);** | **n\*n** |
| }else{ |  |
| variableCost variable = new variableCost(parts[0], parts[1], parts[2], Integer.parseInt(parts[3]), parts[4], Integer.parseInt(parts[5]), Integer.parseInt(parts[6])); | n |
| **controller.addActiv(variable);** | **n\*n** |
| } |  |
| } |  |
| line = br.readLine(); | n |
| } |  |
| } finally { |  |
| br.close(); | 1 |
| } |  |
| **Complexidade** | **n2** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n2** pois implementa um while e dentro desse while vão ser executados métodos para adicionar uma variável (addActivVar/addActivFixed ou controller.addActiv()), e esses métdos têm complexidade n. Logo a complexidade é de **n\*n = n2**.

|  |  |
| --- | --- |
| **Método addActivFixed da casse inputFile** | **O()** |
| List<String> listPrevActKeys = new ArrayList(); | 1 |
| for (int i = paramFixed; i < parts.length; i++) { | **1, n+1, n** |
| String prevKey = parts[i]; | n |
| if (controller.hasActivity(prevKey) && (!listPrevActKeys.contains(prevKey))) { | n |
| listPrevActKeys.add(prevKey); | n |
| } else { |  |
| System.out.println("A prevKey '" + prevKey + "' da Actividade '" + parts[0] + "' n existe!"); |  |
| } |  |
| } |  |
|  |  |
| fixedCost fixed = new fixedCost(parts[0], parts[1], parts[2], Integer.parseInt(parts[3]), parts[4], Integer.parseInt(parts[5]), listPrevActKeys); | 1 |
| **controller.addActiv(fixed);** | **n** |
| } |  |
| **Complexidade** | **n** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n** pois implementa um “for” que percorre os ‘n’ parametros que contêm prevKeys.

|  |  |
| --- | --- |
| **Método addActivVar da casse inputFile** | **O()** |
| List<String> listPrevActKeys = new ArrayList(); | 1 |
| for (int i = paramVar; i < parts.length; i++) { | **1, n+1, n** |
| String prevKey = parts[i]; | n |
| if (controller.hasActivity(prevKey) && (!listPrevActKeys.contains(prevKey))) { | n |
| listPrevActKeys.add(prevKey); | n |
| } else { |  |
| System.out.println("A prevKey '" + prevKey + "' da Actividade '" + parts[0] + "' n existe!"); | 1 |
| } |  |
| } |  |
| if (validacaoUnid(parts[4])) { | 1 |
| variableCost variable = new variableCost(parts[0], parts[1], parts[2], Integer.parseInt(parts[3]), parts[4], Integer.parseInt(parts[5]), Integer.parseInt(parts[6]), listPrevActKeys); | 1 |
| **controller.addActiv(variable);** | **n** |
| } |  |
| Complexidade | **n** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n** pois implementa um “for” que percorre os ‘n’ parametros que contêm prevKeys.

|  |  |
| --- | --- |
| **Método addActiv da classe Controller** | **O()** |
| if (!hasActivity(a.getKey())) { | **n** |
| lActiv.add(a); | 1 |
| return true; | 1 |
| } else { |  |
| return false | 1 |
| } |  |
| **Complexidade** | **n** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n** pois implementa o método hasActivity() descrito a seguir.

|  |  |
| --- | --- |
| **Método hasActivity da classe Controller** | **O()** |
| for (int i = 0; i < lActiv.size(); i++) { | **1, n+1, n** |
| if(key.equalsIgnoreCase(lActiv.get(i).getKey())){ | **n** |
| return true; | 1 |
| } |  |
| } |  |
| return false | 1 |
| **Complexidade** | **n** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n** pois implementa um “for” que percorre as ‘n’ actividades existentes na lista.

|  |  |
| --- | --- |
| **Método allPaths(tipo void) da classe GraphAlgorithms** | **O()** |
| path.push(vOrig.getElement()); | 1 |
| int vKey = vOrig.getKey(); | 1 |
| visited[vKey] = true; | 1 |
|  |  |
| for (Edge<V, E> edge : g.outgoingEdges(vOrig)) { | n |
| Vertex<V, E> vAdj = g.opposite(vOrig, edge); | n\*n |
| String line = br.readLine(); | n\*1 |
|  |  |
| if (vAdj.getElement() == (vDest.getElement())) { | n\*1 |
| path.push(vAdj.getElement()); | n\*1 |
| Deque<V> revpath = revPath(path); | n\*1 |
| paths.add(new LinkedList(revpath)); | n\*1 |
| If (!path.isEmpty()) { | n\*1 |
| path.pop(); | 1 |
| } |  |
| } else { | 1 |
|  |  |
| if (!visited[vAdj.getKey()]) { | 1 |
| allPaths(g, vAdj, vDest, visited, path, paths); | n\*n |
| } |  |
| } |  |
| } |  |
| vKey = vOrig.getKey(); | 1 |
| visited[vKey] = false; | 1 |
| if (!path.isEmpty()){ | 1 |
| path.pop(); | 1 |
| } |  |
| } |  |
| **Complexidade** | **O(n2)** |

|  |  |
| --- | --- |
| **Método allPaths(tipo <V,E>ArrayList<Deque<V>>) da classe GraphAlgorithms** | **O()** |
| Deque<V> path = new LinkedList<>(); | 1 |
| ArrayList<Deque<V>> paths = new ArrayList<>(); | 1 |
| boolean[] visited = new boolean[g.numVertices()]; | 1 |
| Vertex<V, E> vOrig = g.getVertex(voInf); | 1 |
| Vertex<V, E> vDest = g.getVertex(vdInf); | 1 |
| if (vOrig != null && vDest != null) { | 1 |
| allPaths(g, vOrig, vDest, visited, path, paths); | n2 |
| } |  |
| return paths; | 1 |
|  |  |
| } |  |
| **Complexidade** | **O(n2)** |

|  |  |
| --- | --- |
| **Método revPath(tipo Deque<V>) da classe GraphAlgorithms** | **O()** |
| Deque<V> pathcopy = new LinkedList<>(path); | 1 |
|  |  |
| Deque<V> pathrev = new LinkedList<>(); | 1 |
| while (!pathcopy.isEmpty()) { | n |
| pathrev.push(pathcopy.pop()); | n\*1 |
| }  return pathrev; | 1 |
|  |  |
| **Complexidade** | **O(n)** |

|  |  |
| --- | --- |
| **Método addActivity(tipo boolean) da classe PertController** | **O()** |
| if(!a.isEmpty()){ | 1 |
| getPert().insertVertex(a); | 1 |
| return true;  } | 1 |
| else{  System.out.println("Activity doesn't exist"); | 1 |
| return false;  } | 1 |
| **Complexidade** | **O(1)** |

|  |  |
| --- | --- |
| **Método insertAllActivities(tipo boolean)da classe PertController** | **O()** |
| if (!lActiv.isEmpty() && sourceNode.getKey().equalsIgnoreCase("Source") && finalNode.getKey().equalsIgnoreCase("Final")) { | n |
| addActivity(sourceNode.getKey()); | 1 |
| for (Activity a : lActiv) { | n |
| addActivity(a.getKey());  } | n\*1 |
| addActivity(finalNode.getKey());  return true;  } | 1 |
| else {  return false;  } | 1 |
| **Complexidade** | **O(n)** |

|  |  |
| --- | --- |
| **Método insertAllPrecedences(tipo boolean) da classe PertController** | **O()** |
| if(!lActiv.isEmpty() && !sourceNode.getKey().isEmpty() && !finalNode.getKey().isEmpty() && getPert().numVertices()>0){ | 1 |
| for (Activity a : lActiv) { | n |
| if (a.getListPrevActivityKey().isEmpty()) {  addPrecedence(sourceNode.getKey(), a.getKey(), 0, 0); | 1 |
| } else {  List<String> lPrecs = a.getListPrevActivityKey(); | 1 |
| for (String p : lPrecs) { | n\*n |
| addPrecedence(p, a.getKey(), a.getDuration(), 0);  }  }  } | n\*n\*1 |
| for (Activity a : lActiv) { | n |
| if (!a.getKey().isEmpty()) { | n\*1 |
| Vertex<String, Integer> v = getPert().getVertex(a.getKey()); | n\*n |
| if (getPert().outDegree(v) == 0) { | n\*n |
| addPrecedence(a.getKey(), finalNode.getKey(), a.getDuration(), 0); | n\*1 |
| }  } |  |
| }  return true; | 1 |
| }  else{  return false; | 1 |
| **Complexidade** | **O(n2)** |

|  |  |
| --- | --- |
| **Método insertAllPrecedences(tipo boolean) da classe PertController** | **O()** |
| if(!lActiv.isEmpty() && !sourceNode.getKey().isEmpty() && !finalNode.getKey().isEmpty() && getPert().numVertices()>0){ | 1 |
| for (Activity a : lActiv) { | n |
| if (a.getListPrevActivityKey().isEmpty()) {  addPrecedence(sourceNode.getKey(), a.getKey(), 0, 0); | n\*1 |
| } else {  List<String> lPrecs = a.getListPrevActivityKey(); | n\*1 |
| for (String p : lPrecs) { | n\*n |
| addPrecedence(p, a.getKey(), a.getDuration(), 0);  }  } | n\*n\*1 |
| for (Activity a : lActiv) { | n |
| if (!a.getKey().isEmpty()) { | n\*1 |
| Vertex<String, Integer> v = getPert().getVertex(a.getKey()); | n\*n |
| if (getPert().outDegree(v) == 0) { | n\*n |
| addPrecedence(a.getKey(), finalNode.getKey(), a.getDuration(), 0); | n |
| }  } |  |
| }  return true; | 1 |
| }else{  return false; | 1 |
| **Complexidade** | **O(n2)** |

|  |  |
| --- | --- |
| **Método getLongestPath(tipo double) da classe PertController** | **O()** |
| ArrayList<Deque<String>> paths = GraphAlgorithms.allPaths(getPert(), orig, dest); | n2 |
| double maxDuration = 0; | 1 |
| for (Deque<String> p : paths) { | n |
| int current = 0; | n\*1 |
| for (String a : p) { | n\*n |
| Activity activ = controller.getActivity(a); | n\*n\*n |
| if (!a.equalsIgnoreCase("source")&&!a.equalsIgnoreCase("final")) { | n\*n\*n |
| current += activ.getDuration();  } | n\*n |
| }  Activity activ2 = controller.getActivity(dest); | n\*n |
| if (current > (maxDuration)) { | n\*1 |
| maxDuration = current;  }  } | n\*1 |
| return maxDuration;  } | 1 |
| **Complexidade** | **O(n3)** |

|  |  |
| --- | --- |
| **Método addPrecedence(tipo boolean) da classe PertController** | **O()** |
| if(!a1.isEmpty() && !a2.isEmpty() && duration>=0){ | 1 |
| getPert().insertEdge(a1, a2, n, duration); | 1 |
| return true;  } | 1 |
| else{  return false;  } | 1 |
|  |  |
| **Complexidade** | **O(1)** |

|  |  |
| --- | --- |
| **Método getAllPaths(tipo Deque<String>) da classe PertController** | **O()** |
| return GraphAlgorithms.allPaths(getPert(), sourceNode.getKey(), finalNode.getKey()); | n2 |
|  |  |
| **Complexidade** | **O(n2)** |

|  |  |
| --- | --- |
| **Método getPert(tipo Graph<String,Integer>) da classe PertController** | **O()** |
| return pert; | 1 |
| **Complexidade** | **O(1)** |

|  |  |
| --- | --- |
| **Método importFile(tipo boolean) da classe PertUI** | **O()** |
| boolean flag1=pertController.insertAllActivities(); | n |
| if(flag1==false){ | 1 |
| System.out.println("Erro, não foram introduzidas atividades");  } | 1 |
| boolean flag2=pertController.insertAllPrecedences(); | 1 |
| if(flag2==false){ | 1 |
| System.out.println("Não foram introduzidas edges");  } | 1 |
|  |  |
| **Complexidade** | **O(n)** |

|  |  |
| --- | --- |
| **Método printAllPaths(tipo boolean) da classe PertUI** | **O()** |
| ArrayList<Deque<String>> paths = pertController.getAllPaths(); | n2 |
| for (Deque<String> p : paths) { | n |
| int duration = 0; | n\*1 |
| for (String a : p) { | n\*n |
| if (((!a.equalsIgnoreCase("source")) && (!a.equalsIgnoreCase("final")))) { | n\*n\*n |
| System.out.print(a + " "); | n\*n\*1 |
| duration += controller.getActivity(a).getDuration();  } | n\*n\*n |
| }  System.out.print("Length: " + duration + "\n");  } | n\*n\*1 |
| **Complexidade** | **O(n3)** |

|  |  |
| --- | --- |
| **Método printCriticalPaths(tipo void) da classe PertUI** | **O()** |
| double longestPath = pertController.getLongestPath("source", "final"); | O(n3) |
| ArrayList<Deque<String>> paths = pertController.getAllPaths(); | n2 |
| for (Deque<String> p : paths) { | n |
| int current = 0; | n\*1 |
| for (String a : p) { | n\*n |
| if (!a.equalsIgnoreCase("source") && !a.equalsIgnoreCase("final")) { | n\*n\*n |
| Activity activ = controller.getActivity(a); | n\*n\*n |
| current += activ.getDuration();  }  } | n\*n\*1 |
| if (current == longestPath) { | n\*1 |
| printPath(p);  } | n\*1 |
| } |  |
| } |  |
| **Complexidade** | **O(n3** |

|  |  |
| --- | --- |
| **Método printPath(tipo void) da classe PertUI** | **O()** |
| for (String a : path) { | n |
| System.out.print(a + " ");  } | n\*1 |
| System.out.println(); | 1 |
|  |  |
|  |  |
| **Complexidade** | **O(n)** |

|  |  |
| --- | --- |
| **Método ES(tipo double) da classe PertUI** | **O()** |
| Activity a = controller.getActivity(activ); | n |
| return (pertController.getLongestPath("source", activ) - a.getDuration()); | n3 |
| **Complexidade** | **O(n3)** |

|  |  |
| --- | --- |
| **Método EF(tipo double) da classe PertUI** | **O()** |
| double es = ES(activ); | n3 |
| double ef = es + controller.getActivity(activ).getDuration(); | n |
| return ef; | 1 |
|  |  |
| **Complexidade** | **O(n3)** |

|  |  |
| --- | --- |
| **Método LS(tipo double) da classe PertUI** | **O()** |
| double maxPath = pertController.getLongestPath("source", "final"); | n3 |
| double path = pertController.getLongestPath(activ, "final"); | n3 |
| return (maxPath - path); | 1 |
|  |  |
|  |  |
| **Complexidade** | **O(n3)** |

|  |  |  |
| --- | --- | --- |
| **Método LS(tipo double) da classe PertUI** | **O()** | |
| Activity a = controller.getActivity(activ); | n |  |
| return (LS(activ) + a.getDuration()); | n3 | |
| **Complexidade** | **O(n3)** | |

|  |  |
| --- | --- |
| **Método Slack(tipo double) da classe PertUI** | **O()** |
| double lf = LF(activ); | n3 |
| double ef = EF(activ); | n3 |
| return (lf - ef); | 1 |
| **Complexidade** | **O(n3)** |

|  |  |
| --- | --- |
| **Método printTable(tipo void) da classe PertUI** | **O()** |
| List<Activity> la = controller.getlActiv(); | 1 |
| int cost = 0; | 1 |
| System.out.println("Activity | Cost | Duration | Predecessors | ES | LS | EF | LF | Slack |"); | 1 |
| for (Activity a : la) { | n |
| String key = a.getKey(); | n\*1 |
| if (a.getTypeActivity().equalsIgnoreCase("fca")) { | 1 |
| fixedCost fca = (fixedCost) a; | n\*1 |
| cost = fca.getActivityCost(); | n\*1 |
| } else if (a.getTypeActivity().equalsIgnoreCase("vca")) { | n\*n |
| variableCost vca = (variableCost) a; | n\*1 |
| cost = vca.calculateTotalCost();  } | n\*1 |
| System.out.format(" %-8s| %-5s| %-9s| ", key, cost, a.getDuration()); | n\*1 |
| List<String> lp = a.getListPrevActivityKey(); | n\*1 |
| int cont = 0; | n\*1 |
| for (String p : lp) { | n\*n |
| System.out.print(p); |  |
|  | n\*n\*1 |
| cont++;  } | n\*n\*1 |
| for (int i = 0; i < (13-cont); i++) { | n\*n |
| System.out.print(" ");  } | n\*n\*1 |
| System.out.format("| %-4s | %-4s | %-4s | %-4s | %-6s|", ES(key), LS(key), EF(key), LF(key), Slack(key)); | n\*1 |
|  |  |
| **Complexidade** | **O(n2)** |

|  |  |  |
| --- | --- | --- |
| **Método orderedActivities(tipo void) da classe PertUI** | | **O()** |
| List<Activity> la = controller.getlActiv(); | | 1 |
| HashMap<String, Double> map = new HashMap<String, Double>(); | | 1 |
| for (Activity a : la) { | | n |
| double completion = EF(a.getKey()); | | n\*1 |
| map.put(a.getKey(), completion);  } | | n\*1 |
| map = sortByValue(map); | | n |
| for (Entry<String, Double> entry : map.entrySet()) { | | n |
| System.out.println("Activity: " + entry.getKey() + " completion time:"  + entry.getValue());  } | | n\*1 |
|  | |  |
| **Complexidade** |  | **O(n)** |
| **Método sortByValue(tipo HashMap<String,Double>) da classe PertUI** | | **O()** |
| List list = new LinkedList(map.entrySet()); | | 1 |
| Collections.sort(list, new Comparator() { | |  |
| @Override  public int compare(Object o2, Object o1) {  return ((Comparable) ((Map.Entry) (o2)).getValue()).compareTo(((Map.Entry) (o1)).getValue());  }  }); | | 1 |
| HashMap result = new LinkedHashMap(); | |  |
| for (Iterator it = list.iterator(); it.hasNext();) { | | n+1 |
| Map.Entry entry = (Map.Entry) it.next(); | | 1 |
| result.put(entry.getKey(), entry.getValue()); | | 1 |
| }  return result; | | 1 |
|  | |  |
| **Complexidade** | | **O(n)** |

|  |  |
| --- | --- |
| **Método hasActivity(tipo boolean) da classe Controller** | **O()** |
| for (int i = 0; i < lActiv.size(); i++) { | n+1 |
| if (key.equalsIgnoreCase(lActiv.get(i).getKey())) { | n\*n |
| return true;  }  } | 1 |
| return false; | 1 |
| **Complexidade** | **O(n2)** |

|  |  |
| --- | --- |
| **Método getActivity(tipo Activity) da classe Controller** | **O()** |
| for (int i = 0; i < lActiv.size(); i++) { | n+1 |
| if (key.equalsIgnoreCase(lActiv.get(i).getKey())) { | n\*n |
| return lActiv.get(i);  }  } | 1 |
| return null; | 1 |
|  |  |
|  |  |
| **Complexidade** | **O(n2)** |

|  |  |
| --- | --- |
| **Método printActivs(tipo void) da classe Controller** | **O()** |
| System.out.println("\nLista de Actividades: "); | 1 |
| for (int i = 0; i < lActiv.size(); i++) { | n+1 |
| System.out.print(lActiv.get(i).getKey() + "," + lActiv.get(i).getDescription() + "," +lActiv.get(i).getTypeActivity() + ','); | 1 |
| Activity var = lActiv.get(i); | 1 |
| List<String> lPrevs = var.getListPrevActivityKey(); | 1 |
| for (int j = 0; j < lPrevs.size(); j++) { | n\*(n+1) |
| System.out.print(lPrevs.get(j).toString());  } | n\*n |
| System.out.println();  } | n |
|  |  |
|  |  |
| **Complexidade** | **O(n2)** |

|  |  |
| --- | --- |
| **Método addActiv(tipo Activity) da classe Controller** | **O()** |
| if (!hasActivity(a.getKey())) { | 1 |
| lActiv.add(a); | n |
| return true; | 1 |
| } else {  return false;  } |  |
|  |  |
|  |  |
| **Complexidade** | **O(n)** |

|  |  |
| --- | --- |
| **Método getlActiv(tipo ArrayList<Activity>) da classe Controller** | **O()** |
| return lActiv; | 1 |
| **Complexidade** | **O(1)** |

|  |  |
| --- | --- |
| **Método setlActiv(tipo void) da classe Controller** | **O()** |
| this.lActiv = lActiv; | 1 |
| **Complexidade** | **O(1)** |

# Testes e Resultados

Foram realizados testes a todas as classes implementadas, de maneira a verificar a eficácia da nossa aplicação. Como se pode verificar ao testar todo o projeto, os resultados são excelentes já que todos os testes são aprovados, sinal que evidencia a boa codificação do projeto.

# Conclusão

Após a realização da segunda parte do projeto, concluímos que as tarefas por nós desenvolvidas nesta segunda parte do trabalho, ligeiramente mais complicada, no entanto com o apoio das classes baseadas nos algoritmos dos grafos tudo se tornou mais fácil. Apesar de encontrarmos alguns desafios na codificação da aplicação ,pensamos que esta foi codificada com muito sucesso e eficácia.Apoiamos esta afirmação nos resultados apresentados em cima, e que estão relacionados com a complexidade temporal dos métodos que codificamos.

# Bibliografia

<https://moodle.isep.ipp.pt/acesso/>