



ESCOLA  
SUPERIOR  
DE TECNOLOGIA  
E GESTÃO

## **Análise Algorítmica e Otimização**

**UFLP – Problema de localização de instalações sem restrições de capacidade**

**Guilherme Francisco Mieiro – 8180148**

# Índice

Índice .....	2
Índice de Figuras.....	3
Índice de Tabelas .....	5
Lista de Siglas e Acrónimos .....	6
1. Introdução .....	7
1.1 Contextualização.....	7
1.2 Apresentação do Caso de Estudo .....	7
1.3 Motivação e Objetivos .....	8
1.4 Estrutura do Relatório.....	8
2. Pesquisa bibliográfica sobre o Problema de Localização de Instalações sem Restrições de Capacidade.....	9
3. Recolha de Dados.....	11
3.1 Classe <i>WarehouseLocation</i> .....	11
3.2 Classe <i>Customer</i> .....	12
3.2 Classe <i>ProblemScenario</i> .....	13
3.2 Classe <i>ProblemScenarios</i> .....	15
3.2 Classe <i>Logger</i> .....	17
4. Implementação de algoritmos para a resolução do Problema de Localização de Instalações sem Restrições de Capacidade .....	18
4.1 Algoritmo Genético.....	18
4.2 Algoritmo <i>Greedy</i> (ou Ganancioso).....	29
4.3 Algoritmo <i>Simulated Annealing</i> .....	34
5. Análise do desempenho dos algoritmos implementados .....	41
6. Conclusões e Trabalho Futuro .....	45
Referências WWW.....	45

# Índice de Figuras

Figura 1 - Classe <i>WarehouseLocation</i> .....	11
Figura 2 - Classe <i>Customer</i> .....	12
Figura 3 - Classe <i>ProblemScenario</i> .....	13
Figura 4 - Método responsável por ler os dados relativos a um problema de um ficheiro .....	14
Figura 5 - Classe <i>ProblemScenarios</i> .....	15
Figura 6 - Método responsável por ler todos os cenários de um dado diretório .....	16
Figura 7 - Classe <i>Logger</i> .....	17
Figura 8 - Ilustração do Processo de Evolução.....	19
Figura 9 - Inicialização de Variáveis e Hiper Parâmetros.....	20
Figura 10 - Métodos Construtores .....	20
Figura 11 - Método para definir o cenário (definir os dados do problema), métodos de acesso e método auxiliar para fazer uma <i>deep copy</i> de um cromossoma.....	21
Figura 12 - Métodos para gerar respetivamente 1 cromossoma, 1 solução e X soluções aleatórias .....	21
Figura 13 - Método para calcular o custo da solução atual (calcular o <i>fitness</i> ) (Função de Avaliação).....	22
Figura 14 - Método para calcular o custo de cada uma das soluções de uma lista de soluções .....	22
Figura 15 - Método para efetuar a seleção das X melhores soluções (as com o menor custo). 23	
Figura 16 - Método para efetuar o cruzamento entre 2 soluções .....	24
Figura 17 - Método para efetuar o cruzamento entre as soluções de uma lista à exceção de Y “elites” (não queremos destruir as nossas melhores soluções) até atingir uma população de X soluções.....	24
Figura 18 - Método auxiliar que decide se dada a probabilidade de um evento acontecer (a mutação), ele acontece ou não. ....	25
Figura 19 - Métodos que efetuam a mutação de 1 gene, 1 cromossoma e uma solução respetivamente.....	25
Figura 20 - Método que efetua mutações nas soluções de uma lista à exceção das Y melhores soluções (as “elites”) até atingir uma população de X soluções .....	26
Figura 21 - Implementação dos métodos <i>compareTo</i> e <i>toString</i> .....	26
Figura 22 - Método que efetua o ciclo de evolução (população inicial, seleção, cruzamento, mutação, repetição) até que um dos critérios de paragem seja atingido, retornando a melhor solução encontrada.....	27
Figura 23 - Execução do algoritmo genético.....	28
Figura 24 - Output do algoritmo genético.....	28
Figura 25 - Inicialização de Variáveis e Hiper Parâmetros .....	30

Figura 26 - Métodos Construtores .....	30
Figura 27 - Método para definir o cenário (definir os dados do problema), métodos de acesso e método auxiliar para fazer uma <i>deep copy</i> da <i>solutionArray</i> .....	30
Figura 28 - Método para calcular o custo da solução atual (igual à função fitness do algoritmo genético) .....	31
Figura 29 - Implementação do método <i>toString</i> .....	31
Figura 30 - Método que iterativamente efetua a escolha <i>greedy</i> construindo assim a solução	32
Figura 31 - Execução do algoritmo <i>greedy</i> .....	33
Figura 32 - Output do algoritmo <i>greedy</i> .....	33
Figura 33 - Inicialização de Variáveis e Hiper Parâmetros .....	35
Figura 34 - Métodos Construtores .....	35
Figura 35 - Método para definir o cenário (definir os dados do problema), métodos de acesso e método auxiliar para fazer uma <i>deep copy</i> de um <i>StateArray</i> .....	36
Figura 36 - Método para gerar 1 <i>StateArray</i> aleatório (para quando não existe solução inicial)	36
Figura 37 - Método para calcular o custo da solução atual (semelhante à solução fitness do algoritmo genético) .....	37
Figura 38 - Método auxiliar que decide se dada a probabilidade de um evento acontecer (a perturbação de um elo individual do <i>StateArray</i> ), ele acontece ou não. ....	38
Figura 39 - Métodos que efetuam a perturbação de 1 elo de um <i>StateArray</i> , de 1 <i>StateArray</i> e de uma solução respetivamente .....	38
Figura 40 - Implementação do método <i>toString</i> .....	39
Figura 41 - Método que iterativamente efetua a perturbações e decide aceita-las ou não construindo assim a solução .....	39
Figura 42 - Execução do algoritmo Simulated Annealing .....	40
Figura 43 - Output do algoritmo <i>Simulated Annealing</i> .....	40

# Índice de Tabelas

Tabela 1 - Siglas e Acrónimos .....	6
Tabela 2 - Desempenho do Algoritmo Genético .....	41
Tabela 3 - Desempenho do Algoritmo <i>Greedy</i> .....	42
Tabela 4 - Desempenho do Algoritmo <i>Simulated Annealing</i> .....	43

## Lista de Siglas e Acrónimos

Sigla	Significado
FLP	<i>Facility Location Problem</i>
UFLP	<i>Uncapacitated Facility Location Problem</i>

**Tabela 1 - Siglas e Acrónimos**

# 1. Introdução

## 1.1 Contextualização

Este trabalho foi realizado no âmbito da disciplina de Análise Algorítmica e Otimização e tem como objetivo a implementação prática de alguns algoritmos heurísticos estudados nas aulas com vista à resolução do problema de localização de instalações sem restrições de capacidade (UFLP).

## 1.2 Apresentação do Caso de Estudo

O Problema de “Localização de Instalações sem Restrições de Capacidade” (UFLP, do inglês *Uncapacitated Facility Location Problem*) é uma simplificação do problema original “Problema de Localização de Instalações” (FLP), que surgiu naturalmente das necessidades práticas na área de logística e gestão de operações durante as décadas de 1950 e 1960 e foi formalizado ao longo do tempo na pesquisa operacional e ciência da computação.

Este é um problema clássico de otimização combinatória na área de pesquisa operacional.

Neste problema, havendo instalações cuja abertura tem um custo fixo e havendo clientes cuja procura precisa ser satisfeita tendo assim um custo de alocação (custo de transporte) dessa procura à instalação mais próxima, o objetivo é determinar a localização de um conjunto de facilidades (ou instalações) de forma a atender a toda a procura de todos os clientes minimizando o custo total.

## 1.3 Motivação e Objetivos

O problema de localização de instalações sem restrições de capacidade é um problema de natureza combinatória do tipo NP-Difícil, isto significa que a sua resolução se torna exponencialmente mais complicada à medida que o número de instalações e clientes aumenta, tornando assim para este tipo de problema o uso de algoritmos exatos pouco adequado (principalmente para instâncias maiores).

Em vez disso, é sugerido o uso de algoritmos heurísticos que embora nem sempre consigam obter a solução ótima, num menor espaço de tempo e com um menor custo de recursos computacionais conseguem obter uma solução “boa o suficiente”.

## 1.4 Estrutura do Relatório

Este relatório está dividido em 6 partes.

- Na primeira é feita a introdução ao tema, contextualização e apresentação das motivações e objetivos do trabalho.
- Na segunda é apresentada uma descrição mais detalhada do problema a abordar (o problema de localização de instalações sem restrições de capacidade) bem como as suas origens e aplicações.
- Na terceira é mostrada como foi efetuada a recolha de dados a utilizar nos algoritmos heurísticos.
- Na quarta é mostrada como foi efetuada a implementação dos diversos algoritmos heurísticos.
- Na quinta é feita uma análise sobre o desempenho dos diversos algoritmos heurísticos bem como uma reflexão dos mesmos.
- Na sexta é efetuada uma reflexão sobre o trabalho, pontos fortes e passíveis de serem melhorados, bem como trabalhos futuros.



## 2. Pesquisa bibliográfica sobre o Problema de Localização de Instalações sem Restrições de Capacidade

O **Problema de Localização de Facilidades Não-Capacitado** (UFLP) não é atribuído a um único inventor ou pesquisador, mas é um problema que emergiu naturalmente a partir das necessidades práticas na área de logística e gestão de operações e foi formalizado ao longo do tempo na pesquisa operacional e ciência da computação.

### Origem e Desenvolvimento

- **Décadas de 1950 e 1960:** O UFLP e problemas relacionados começaram a ganhar formalização e estudo sistemático durante este período, com o crescimento da pesquisa em otimização e teoria dos grafos. Pesquisadores como **Tjalling Koopmans** e **Martin Beckmann** exploraram problemas de localização e distribuição em contextos econômicos e logísticos.
- **1970s:** O problema ganhou mais formalismo matemático e foi amplamente estudado como um problema de otimização combinatória. Livros e artigos sobre pesquisa operacional começaram a incluir versões do problema de localização de facilidades.
- **Desenvolvimento Teórico:** O problema está intimamente ligado à teoria de programação linear inteira e combinatória, com contribuições importantes de muitos pesquisadores ao longo das décadas, como **George Dantzig**, **Richard M. Karp**, e **Jack Edmonds**, que trabalharam em algoritmos e técnicas de otimização relevantes.

### Descrição do Problema

No UFLP, existem as seguintes entidades:

- **Clientes:** Um conjunto de clientes, cada um com uma demanda (procura) específica que deve ser atendida por uma das facilidades.
- **Facilidades:** Um conjunto de possíveis localizações para as facilidades, cada uma com um custo fixo de abertura e operação.
- **Custo de Transporte (ou de alocação):** O custo associado ao atendimento de cada cliente por uma determinada facilidade, geralmente relacionado à distância entre eles.

## Objetivo

O objetivo do UFLP é decidir:

1. **Quais facilidades abrir:** Selecionar as localizações que devem ser usadas.
2. **Como atender os clientes:** Atribuir cada cliente à facilidade que minimiza o custo total, levando em consideração tanto o custo fixo de abertura das facilidades quanto os custos variáveis de transporte.

## Formulação Matemática

O problema pode ser formulado como um programa linear inteiro, onde:

- **Variáveis binárias** indicam se uma facilidade é aberta ou não.
- **Variáveis binárias adicionais** indicam se um cliente é atendido por uma facilidade específica.

## Aplicações

O UFLP tem inúmeras aplicações práticas, como:

- **Localização de armazéns:** Decidir onde construir armazéns para minimizar os custos de distribuição.
- **Planejamento de redes de serviços:** Como decidir onde instalar centros de atendimento ao cliente ou filiais de um banco.
- **Gestão de cadeia de suprimentos:** Otimizar a localização de centros de distribuição para minimizar os custos de transporte e operação.

## Soluções

Devido à sua complexidade, o UFLP é um problema NP-difícil, o que significa que soluções exatas podem ser computacionalmente inviáveis para instâncias grandes. Portanto, métodos heurísticos e meta heurísticos, como algoritmos genéticos, *simulated annealing*, e algoritmos de colônia de formigas, são frequentemente usados para encontrar soluções aproximadas de alta qualidade em tempo razoável.

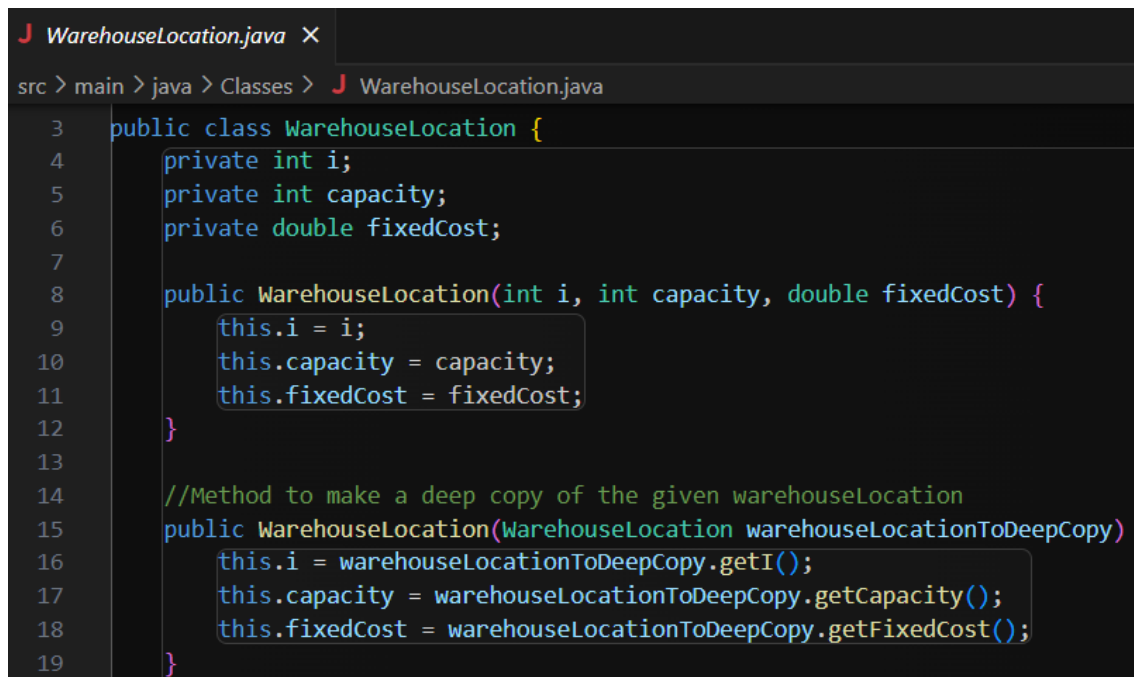
### 3. Recolha de Dados

Com vista a fornecer dados ilustrativos do problema aos algoritmos heurísticos é primeiro necessário lê-los dos ficheiros facultados pela professora e carregar os mesmos para memória.

Para isso, foram desenvolvidas algumas classes e métodos auxiliares.

#### 3.1 Classe *WarehouseLocation*

Esta classe tem como objetivo ilustrar e armazenar uma instalação. Para além dos parâmetros e métodos construtores, apenas possui métodos de acesso.

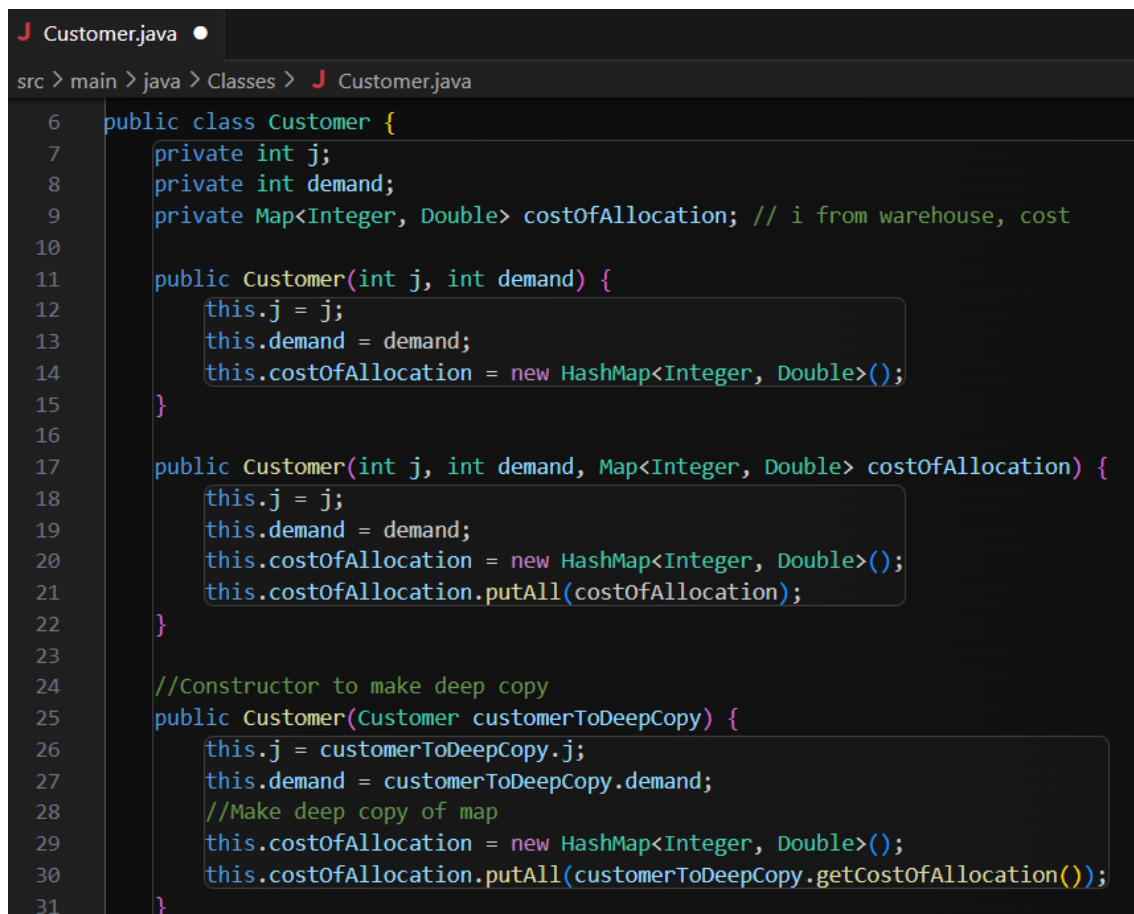


```
3 public class WarehouseLocation {
4     private int i;
5     private int capacity;
6     private double fixedCost;
7
8     public WarehouseLocation(int i, int capacity, double fixedCost) {
9         this.i = i;
10        this.capacity = capacity;
11        this.fixedCost = fixedCost;
12    }
13
14    //Method to make a deep copy of the given warehouseLocation
15    public WarehouseLocation(WarehouseLocation warehouseLocationToDeepCopy)
16    {
17        this.i = warehouseLocationToDeepCopy.getI();
18        this.capacity = warehouseLocationToDeepCopy.getCapacity();
19        this.fixedCost = warehouseLocationToDeepCopy.getFixedCost();
20    }
21 }
```

Figura 1 - Classe *WarehouseLocation*

## 3.2 Classe *Customer*

Esta classe tem como objetivo ilustrar e armazenar um cliente. Para além dos parâmetros e métodos construtores, apenas possui métodos de acesso.



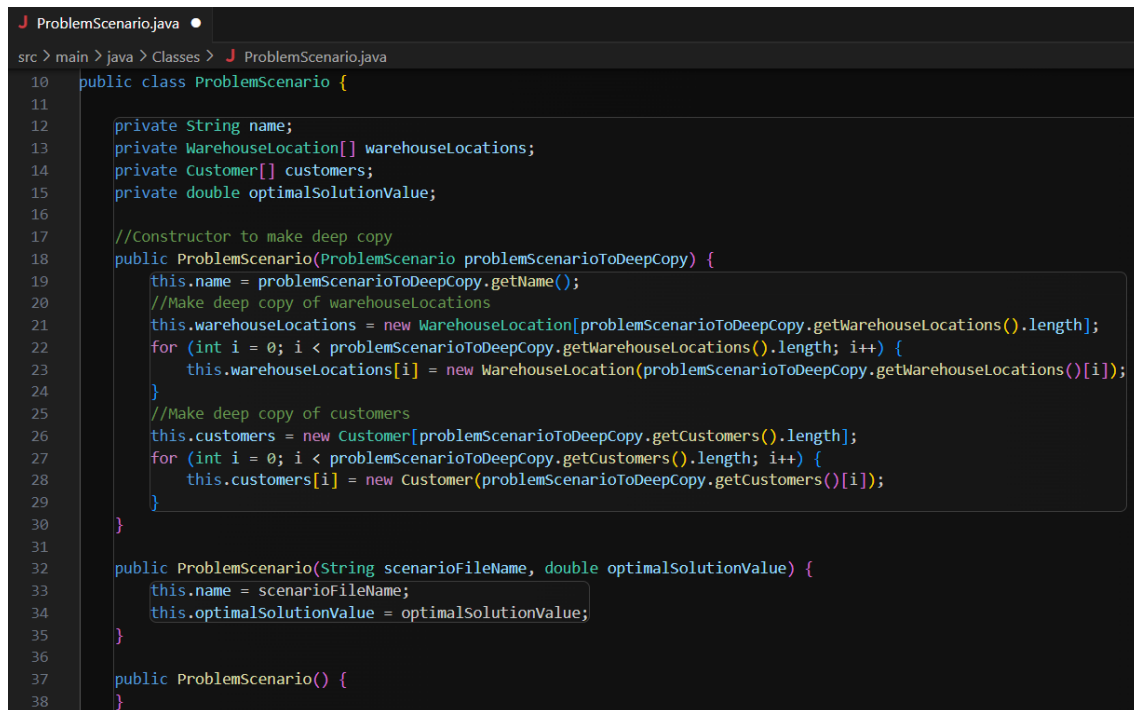
```
Customer.java
src > main > java > Classes > Customer.java

6 public class Customer {
7     private int j;
8     private int demand;
9     private Map<Integer, Double> costOfAllocation; // i from warehouse, cost
10
11     public Customer(int j, int demand) {
12         this.j = j;
13         this.demand = demand;
14         this.costOfAllocation = new HashMap<Integer, Double>();
15     }
16
17     public Customer(int j, int demand, Map<Integer, Double> costOfAllocation) {
18         this.j = j;
19         this.demand = demand;
20         this.costOfAllocation = new HashMap<Integer, Double>();
21         this.costOfAllocation.putAll(costOfAllocation);
22     }
23
24     //Constructor to make deep copy
25     public Customer(Customer customerToDeepCopy) {
26         this.j = customerToDeepCopy.j;
27         this.demand = customerToDeepCopy.demand;
28         //Make deep copy of map
29         this.costOfAllocation = new HashMap<Integer, Double>();
30         this.costOfAllocation.putAll(customerToDeepCopy.getCostOfAllocation());
31     }
}
```

Figura 2 - Classe *Customer*

## 3.2 Classe *ProblemScenario*

Esta classe tem como objetivo ilustrar e armazenar um cenário, isto é, um problema com o seu nome, localizações, clientes e custo ótimo. Para além dos parâmetros e métodos construtores, possui métodos de acesso e um método para dado o caminho de um ficheiro, ler os valores relativos ao problema desse ficheiro ficando assim com os mesmos carregados em memória, passíveis de serem usados pelos algoritmos heurísticos.



```
10 public class ProblemScenario {
11
12     private String name;
13     private WarehouseLocation[] warehouseLocations;
14     private Customer[] customers;
15     private double optimalSolutionValue;
16
17     //Constructor to make deep copy
18     public ProblemScenario(ProblemScenarioToDeepCopy problemScenarioToDeepCopy) {
19         this.name = problemScenarioToDeepCopy.getName();
20         //Make deep copy of warehouseLocations
21         this.warehouseLocations = new WarehouseLocation[problemScenarioToDeepCopy.getWarehouseLocations().length];
22         for (int i = 0; i < problemScenarioToDeepCopy.getWarehouseLocations().length; i++) {
23             this.warehouseLocations[i] = new WarehouseLocation(problemScenarioToDeepCopy.getWarehouseLocations()[i]);
24         }
25         //Make deep copy of customers
26         this.customers = new Customer[problemScenarioToDeepCopy.getCustomers().length];
27         for (int i = 0; i < problemScenarioToDeepCopy.getCustomers().length; i++) {
28             this.customers[i] = new Customer(problemScenarioToDeepCopy.getCustomers()[i]);
29         }
30     }
31
32     public ProblemScenario(String scenarioFileName, double optimalSolutionValue) {
33         this.name = scenarioFileName;
34         this.optimalSolutionValue = optimalSolutionValue;
35     }
36
37     public ProblemScenario() {
38     }
```

Figura 3 - Classe *ProblemScenario*

## Ler cenário de um ficheiro

```

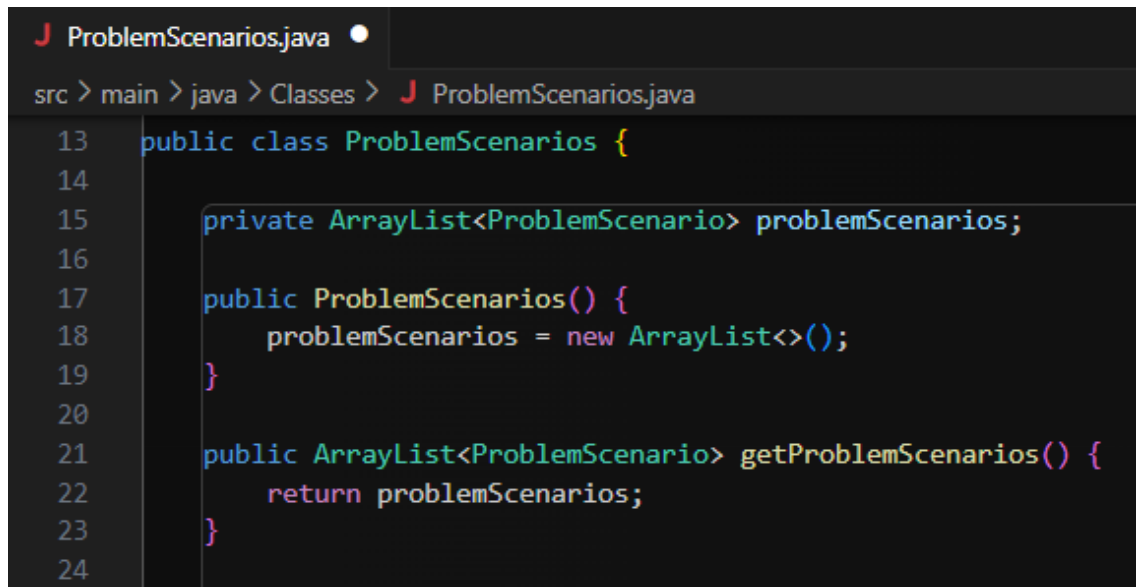
77  /**
78   * I want to read all the data about this scenario from a file
79   */
80  public void ReadScenarioFromFile(Path problemScenarioFilePath) {
81      System.out.println("Reading: " + problemScenarioFilePath);
82      try {
83          // Read all lines from the file
84          List<String> scenarioData = Files.readAllLines(problemScenarioFilePath)
85              .stream()
86              .filter(line -> !line.trim().isEmpty()) // Filters out empty or whitespace-only lines
87              .collect(Collectors.toList());
88
89          for (int i = 0; i < scenarioData.size(); i++) {
90              // Remove white spaces in the start and end of the line
91              String line = scenarioData.get(i).trim();
92              // Remove all the points (.) in the end of the line
93              scenarioData.set(i, line.replaceAll("\\.$", ""));
94          }
95
96          int numberOfWarehouseLocations = -1;
97          int numberOfCustomers = -1;
98          int currentCustomerIndex = -1;
99          int currentWarehouseIndex = -1;
100
101          // For each line in the scenario file
102          for (int i = 0; i < scenarioData.size(); i++) {
103
104              switch (i) {
105                  // If we are in line 0, get the number of warehouses and clients
106                  case 0: {
107                      // Get the number of warehouses and customers from splitting the line
108                      String[] lineParts = scenarioData.get(i).trim().split(" ");
109                      numberOfWarehouseLocations = Integer.parseInt(lineParts[0]);
110                      numberOfCustomers = Integer.parseInt(lineParts[lineParts.length - 1]);
111                      // Use those numbers to update the size of the arrays
112                      warehouseLocations = new WarehouseLocation[numberOfWarehouseLocations];
113                      customers = new Customer[numberOfCustomers];
114                      // Logger.WriteMessage("NumberOfWarehouseLocations: " + numberOfWarehouseLocations + "\t" + "NumberOfCustomers: " +
115                      break;
116                  }
117                  default: {
118                      // If we are reading the lines 1-numberOfWarehouseLocations, get the info about warehouses
119                      if (i >= 1 && i <= numberOfWarehouseLocations) {
120                          // Removes white spaces
121                          String[] lineParts = scenarioData.get(i).trim().split("\\s+");
122                          int warehouseCapacity = Integer.parseInt(lineParts[0]);
123                          double warehouseFixedCost = Double.parseDouble(lineParts[lineParts.length - 1]);
124                          // Store warehouse
125                          WarehouseLocation newWarehouseLocation = new WarehouseLocation(i - 1, warehouseCapacity, warehouseFixedCost);
126                          this.warehouseLocations[i - 1] = newWarehouseLocation;
127                          Logger.WriteMessage(newWarehouseLocation.toString());
128                          break;
129                      } else {
130                          // Get info about costumers
131                          // Trim the line and split by spaces
132                          String[] parts = scenarioData.get(i).trim().split("\\s+");
133                          // Check if the first part is an integer (likely demand)
134                          try {
135                              int demand = Integer.parseInt(parts[0]);
136                              // If so, create a new customer
137                              currentCustomerIndex++;
138                              currentWarehouseIndex = -1;
139                              Customer newCustomer = new Customer(currentCustomerIndex, demand);
140                              // Store it in the array
141                              customers[currentCustomerIndex] = newCustomer;
142                              Logger.WriteMessage(newCustomer.toString());
143                          } catch (NumberFormatException e) {
144                              // Not an integer, consider it as a cost line
145                              for (String cost : parts) {
146                                  currentWarehouseIndex++;
147
148                                  customers[currentCustomerIndex].addCostOfAllocation(currentWarehouseIndex, Double.parseDouble(cost));
149                                  // Logger.WriteMessage("warehouseLocation: " + currentWarehouseIndex + "\t" + "costOfAllocation: " + co
150                              }
151                          }
152                      }
153                  }
154              }
155          }
156      } catch (IOException e) {
157          e.printStackTrace();
158      }
159  }

```

Figura 4 - Método responsável por ler os dados relativos a um problema de um ficheiro

## 3.2 Classe *ProblemScenarios*

Esta classe tem como objetivo ilustrar e armazenar todos os cenários de um dado diretório, isto é todos os cenários de um dado diretório (70, 100, 130, a-c, M) bem como as suas soluções ótimas, para que estes possam posteriormente ser usados pelos algoritmos heurísticos.



```
J ProblemScenarios.java
src > main > java > Classes > J ProblemScenarios.java
13 public class ProblemScenarios {
14
15     private ArrayList<ProblemScenario> problemScenarios;
16
17     public ProblemScenarios() {
18         problemScenarios = new ArrayList<>();
19     }
20
21     public ArrayList<ProblemScenario> getProblemScenarios() {
22         return problemScenarios;
23     }
24 }
```

Figura 5 - Classe *ProblemScenarios*

## Ler cenários de um diretório

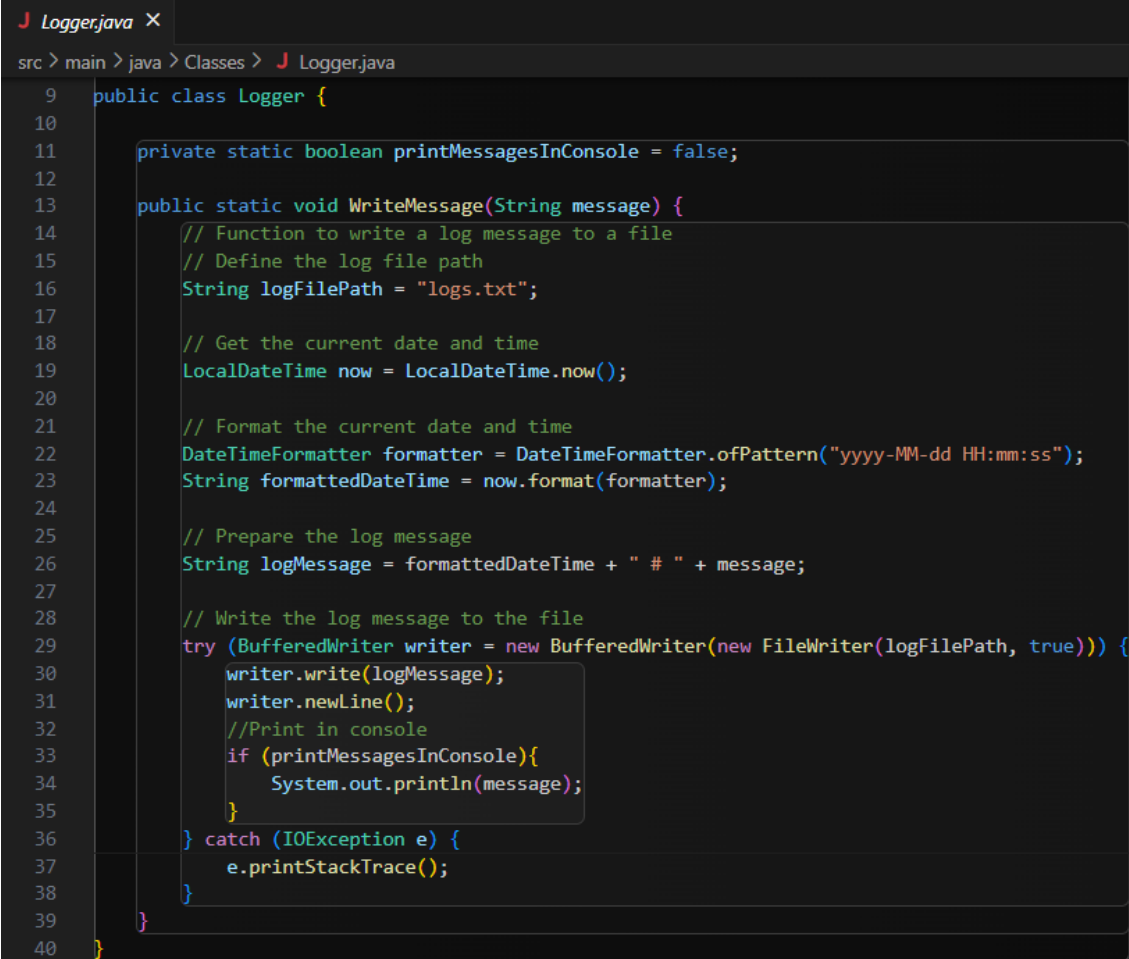
```
25  /**
26   * I want to read all the problem scenarios in a directory and store them in the array
27   */
28  public void ReadScenariosFromDir(String problemScenariosDirPartialPath){
29
30      // Get the user's project directory
31      String projectDir = System.getProperty("user.dir");
32
33      // Get the scenarios directory
34      String problemScenariosDirCompletePath = projectDir + problemScenariosDirPartialPath;
35
36      // Construct the absolute path to the file containing the scenarios list
37      Path scenariosFilePath = Paths.get(problemScenariosDirCompletePath + "files.lst");
38
39      Logger.WriteMessage("Reading scenarios from " + scenariosFilePath.toString());
40
41      // Construct the absolute path to the optimal solutions
42      Path optimalSolutionsFilePath = Paths.get(projectDir + "\\ProblemScenariosData\\optimal.txt");
43      System.out.println(optimalSolutionsFilePath.toString());
44      System.out.println(scenariosFilePath.toString());
45
46
47      try {
48          // Read all scenarios from the scenariosFilePath
49          List<String> scenarioFileNames = Files.readAllLines(scenariosFilePath);
50
51          // Read all optimalSolutions from the optimalSolutionsFilePath
52          List<String> optimalSolutions = Files.readAllLines(optimalSolutionsFilePath, Charset.forName("Windows-1252"));
53
54          // For each scenarioFile
55          for (String scenarioFileName : scenarioFileNames) {
56              Logger.WriteMessage("fileName:" + scenarioFileName);
57
58
59              //Get the optimal solution
60              double optimalSolution = -1;
61              for(String optSolLine : optimalSolutions){
62                  //Split the line
63                  String[] parts = optSolLine.trim().split("\\s+");
64                  //If the part 0 is equal to the name
65                  if ((parts[0]+".txt").equalsIgnoreCase(scenarioFileName)){
66                      //then part 1 is the optimal value
67                      optimalSolution = Double.parseDouble(parts[parts.length-1]);
68                      Logger.WriteMessage("Optimal Solution: " + optimalSolution);
69                      break;
70                  }
71              }
72
73              //Create a ProblemScenario
74              ProblemScenario problemScenario = new ProblemScenario(scenarioFileName, optimalSolution);
75              //Read the info from that problemScenario
76              problemScenario.ReadScenarioFromFile(Paths.get(problemScenariosDirCompletePath + scenarioFileName));
77              //Store problemScenario in the list
78              this.problemScenarios.add(problemScenario);
79          }
80      } catch (IOException e) {
81          e.printStackTrace();
82      }
83  }
```

Figura 6 - Método responsável por ler todos os cenários de um dado diretório



## 3.2 Classe *Logger*

Esta classe tem como objetivo registrar informações relevantes (*logs*) à cerca do funcionamento dos algoritmos em um ficheiro (*logs.txt*).



```
9 public class Logger {
10
11     private static boolean printMessagesInConsole = false;
12
13     public static void WriteMessage(String message) {
14         // Function to write a log message to a file
15         // Define the log file path
16         String logFilePath = "logs.txt";
17
18         // Get the current date and time
19         LocalDateTime now = LocalDateTime.now();
20
21         // Format the current date and time
22         DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
23         String formattedDateTime = now.format(formatter);
24
25         // Prepare the log message
26         String logMessage = formattedDateTime + " # " + message;
27
28         // Write the log message to the file
29         try (BufferedWriter writer = new BufferedWriter(new FileWriter(logFilePath, true))) {
30             writer.write(logMessage);
31             writer.newLine();
32             //Print in console
33             if (printMessagesInConsole){
34                 System.out.println(message);
35             }
36         } catch (IOException e) {
37             e.printStackTrace();
38         }
39     }
40 }
```

Figura 7 - Classe *Logger*

## 4. Implementação de algoritmos para a resolução do Problema de Localização de Instalações sem Restrições de Capacidade

### 4.1 Algoritmo Genético

Um algoritmo genético é uma técnica de otimização e busca inspirada nos processos da seleção natural e genética, como ocorre na biologia. Ele é usado para encontrar soluções aproximadas para problemas complexos onde outras técnicas de otimização podem não ser eficazes.

#### Estrutura Básica de um Algoritmo Genético

##### 1. População Inicial:

- O processo começa com uma população inicial de **indivíduos** (soluções candidatas), que são geralmente representados como sequências de genes (cromossomas), podendo ser cadeias binárias, inteiros, ou outros formatos dependendo do problema.

##### 2. Avaliação (Função de Fitness):

- Cada indivíduo na população é avaliado usando uma **função de fitness** que mede quão boa é a solução. Soluções melhores recebem uma pontuação mais alta (ou mais baixa caso se trate de um problema de minimização).

##### 3. Seleção:

- Indivíduos são selecionados para reproduzir com base na sua aptidão (fitness).

##### 4. Crossover (Recombinação):

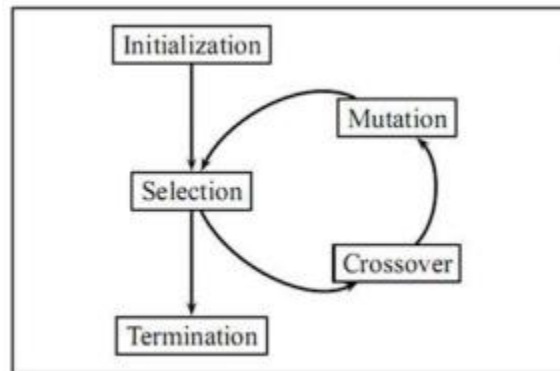
- Dois indivíduos selecionados (pais) combinam parte de seus genes para criar novos indivíduos (filhos). Isso simula a reprodução sexual, onde características de ambos os pais são combinadas, promovendo a diversidade na população.

##### 5. Mutação:

- Após o *crossover*, alguns genes dos novos indivíduos podem sofrer **mutações** aleatórias. A mutação introduz pequenas variações, evitando que a população convirja prematuramente para soluções sub ótimas.
- Isto é necessário pois a solução ótima pode conter genes não presentes na população inicial.

6. **Substituição:**

- A nova geração de indivíduos substitui parte ou toda a população antiga. Este ciclo de avaliação, seleção, *crossover* e mutação é repetido até que se atinja uma condição de paragem, como um número máximo de gerações ou a obtenção de uma solução satisfatória.



**Figura 8 - Ilustração do Processo de Evolução**

## Implementação (Algoritmo Genético)

```
J GeneticAlgorithmSolution.java ×
src > main > java > Classes > J GeneticAlgorithmSolution.java

8 public class GeneticAlgorithmSolution implements Comparable<GeneticAlgorithmSolution> {
9
10     private final static int numberOfSolutionsPerGeneration = 100;
11     private final static int xNumberOfBestSolutionsForSelectionFunction = 40; //Even Number if possible
12
13     private final static int xPopulationSizeGoalForCrossoverFunction = 100;
14     private final static int yNumberOfEliteNonCrossedSolutions = 6; //Even Number if possible
15
16     private final static int xPopulationSizeGoalForMutationFunction = 120;
17     private final static int yNumberOfEliteNonMutatedSolutions = 6;
18     private final static double chanceOfSolutionMutation = 0.60;
19     private final static double chanceOfGeneMutation = 0.20;
20
21     private final static int maxNumberOfGenerations = 200;
22     private final static long maxRunEvolutionDuration = 50000; //milliseconds
23
24
25     private static ProblemScenario problemScenario = null;
26
27     private int[] chromosome;
28     private double currentSolutionCost;
29 }
```

Figura 9 - Inicialização de Variáveis e Hiper Parâmetros

```
30 /**
31  * Makes a solution with the given chromosome
32  * Also calculates the total cost of the created solution
33  */
34 public GeneticAlgorithmSolution(int[] chromosome) {
35     this.chromosome = GeneticAlgorithmSolution.generateChromosomeDeepCopy(chromosome);
36     this.calculateTotalCost();
37 }
38
39 /**
40  * Makes a deep copy with the given solution
41  */
42 public GeneticAlgorithmSolution(GeneticAlgorithmSolution geneticAlgorithmSolutionToDeepCopy) {
43     this.chromosome = GeneticAlgorithmSolution.generateChromosomeDeepCopy(geneticAlgorithmSolutionToDeepCopy.getChromosome());
44     this.currentSolutionCost = geneticAlgorithmSolutionToDeepCopy.getCurrentSolutionCost();
45 }
```

Figura 10 - Métodos Construtores

```

47 //Sets the problem scenario ... needs to be set BEFORE the constructor
48 public static void setProblemScenario(ProblemScenario problemScenario) {
49     GeneticAlgorithmSolution.problemScenario = problemScenario;
50 }
51
52 public int[] getChromosome() {
53     return chromosome;
54 }
55
56 public double getCurrentSolutionCost() {
57     return currentSolutionCost;
58 }
59
60 /**
61  * Generates and returns a deep copy of the provided Chromosome
62  */
63 private static int[] generateChromosomeDeepCopy(int[] originalChromosome) {
64     //Make deep copy of the chromosome
65     int[] chromosomeDeepCopy = new int[originalChromosome.length];
66     for (int i = 0; i < originalChromosome.length; i++) {
67         chromosomeDeepCopy[i] = originalChromosome[i];
68     }
69     return chromosomeDeepCopy;
70 }

```

Figura 11 - Método para definir o cenário (definir os dados do problema), métodos de acesso e método auxiliar para fazer uma *deep copy* de um cromossoma

### População Inicial

```

76 private static int[] generateRandomChromosome() {
77     Random rand = new Random();
78     //Creates a list to store the generated genes
79     int[] generatedChromosome = new int[problemScenario.getWarehouseLocations().length];
80
81     for (int i = 0; i < generatedChromosome.length; i++)
82         generatedChromosome[i] = rand.nextInt(0, 2);
83     //Returns generated chromosome
84     return generatedChromosome;
85 }
86
87 /**
88  * Generates and returns a random solution
89  */
90 private static GeneticAlgorithmSolution generateRandomSolution() {
91     //Creates a solution with a random chromosome
92     return new GeneticAlgorithmSolution(GeneticAlgorithmSolution.generateRandomChromosome());
93 }
94
95 /**
96  * Generates and returns a list with X number of random solutions
97  */
98 private static List<GeneticAlgorithmSolution> generateXRandomSolutions(int numberOfRandomSolutions) {
99     //Creates a list to store the generated solutions
100     List<GeneticAlgorithmSolution> generatedSolutions = new ArrayList<>();
101     //Generates random solutions and stores them in the list
102     for (int i = 0; i < numberOfRandomSolutions; i++)
103         generatedSolutions.add(GeneticAlgorithmSolution.generateRandomSolution());
104     //Sorts the generated Solutions
105     generatedSolutions.sort(Comparator.naturalOrder());
106     //Returns the generated solutions
107     return generatedSolutions;
108 }

```

Figura 12 - Métodos para gerar respectivamente 1 cromossoma, 1 solução e X soluções aleatórias

## Avaliação

```
110  /**
111   * Calculate the total cost value of the current solution
112   */
113  private void calculateTotalCost() {
114      double totalCost = 0;
115      //For each warehouse with 1 value in the chromosome, add the fixed cost
116      //For each chromosome
117      for (int i = 0; i < this.chromosome.length; i++) {
118          //if the chromosome is 1 (warehouse is open)
119          if (this.chromosome[i] == 1) {
120              //Add the fixed cost of the warehouse to the total cost
121              totalCost += this.problemScenario.getWarehouseLocations()[i].getFixedCost();
122          }
123      }
124
125      //For each customer, add the costOfAllocation of the nearest opened warehouse
126      //For each customer
127      for (int i = 0; i < this.problemScenario.getCustomers().length; i++) {
128          //Get the cost of allocations
129          Map<Integer, Double> costOfAllocation = this.problemScenario.getCustomers()[i].getCostOfAllocation();
130          // Create a list of the map entries
131          List<Map.Entry<Integer, Double>> entries = new ArrayList<>(costOfAllocation.entrySet());
132
133          // Sort the list by values
134          entries.sort(Map.Entry.comparingByValue());
135
136          // Iterate over the sorted entries
137          for (Map.Entry<Integer, Double> entry : entries) {
138              //Logger.WriteLine("Key: " + entry.getKey() + ", Value: " + entry.getValue());
139              // Until we find the first opened warehouse
140              if (this.chromosome[entry.getKey()] == 1) {
141                  //If so ... Add the value to the total cost and break the loop
142                  totalCost += entry.getValue();
143                  break;
144              }
145          }
146      }
147
148      //If all the warehouses are closed (the cost is 0.0, then make the cost a very large number)
149      if (totalCost == 0) {
150          totalCost = Double.POSITIVE_INFINITY;
151      }
152
153      //All done ... set the new cost for the current solution
154      this.currentSolutionCost = totalCost;
155  }
```

Figura 13 - Método para calcular o custo da solução atual (calcular o *fitness*) (Função de Avaliação)

```
156  /**
157   * Calculates the total cost value of each solution in a list
158   */
159  private static void calculateTotalCosts(List<GeneticAlgorithmSolution> solutionsList) {
160      //For each solution in the solutionsList, it calculates the total cost
161      for (GeneticAlgorithmSolution solution : solutionsList)
162          solution.calculateTotalCost();
163  }
```

Figura 14 - Método para calcular o custo de cada uma das soluções de uma lista de soluções

## Seleção

```
165  /**
166   * Selects the X best solutions and returns a list with them
167   */
168  private static List<GeneticAlgorithmSolution> selectionFunctionXBest(
169      List<GeneticAlgorithmSolution> solutionsList,
170      int xNumberOfBestSolutions
171  ) {
172      //Creates a list to store the xBest solutions
173      List<GeneticAlgorithmSolution> xBestSolutions = new ArrayList<>();
174      //Sorts the solutionsList so that the best ones come first
175      solutionsList.sort(Comparator.naturalOrder());
176      //Adds the X first (and therefore best) solutions to the storage list
177      for (int i = 0; i < xNumberOfBestSolutions; i++)
178          xBestSolutions.add(solutionsList.get(i));
179      //Returns array with X best solutions
180      return xBestSolutions;
181  }
```

Figura 15 - Método para efetuar a seleção das X melhores soluções (as com o menor custo)

## Cruzamento (Crossover)

```
184  /**
185   * Given an array with 2 initial solutions, returns an array with the 2 resulting solutions from crossing of the 2 initial ones
186   */
187  private static GeneticAlgorithmSolution[] crossoverFunction(GeneticAlgorithmSolution initial2SolutionFirst, GeneticAlgorithmSolution initial2SolutionSecond) {
188      //Get initial chromosomes
189      int[] initialChromosome1 = initial2SolutionFirst.getChromosome();
190      int[] initialChromosome2 = initial2SolutionSecond.getChromosome();
191      //Get the random part where u will part them (from position 1 to the one before the last)
192      int randomPartPosition = (new Random()).nextInt(1, initialChromosome1.length - 1);
193
194      //Build the new crossed chromosomes
195      int[] crossedChromosome1 = new int[initialChromosome1.length];
196      int[] crossedChromosome2 = new int[initialChromosome1.length];
197
198      // Copy the first part from each parent into the new chromosomes
199      System.arraycopy(initialChromosome1, 0, crossedChromosome1, 0, randomPartPosition);
200      System.arraycopy(initialChromosome2, 0, crossedChromosome2, 0, randomPartPosition);
201
202      // Copy the second part from the other parent into the new chromosomes
203      System.arraycopy(initialChromosome2, randomPartPosition, crossedChromosome1, randomPartPosition, initialChromosome1.length - randomPartPosition);
204      System.arraycopy(initialChromosome1, randomPartPosition, crossedChromosome2, randomPartPosition, initialChromosome2.length - randomPartPosition);
205
206      //Build new SolutionArray
207      GeneticAlgorithmSolution[] crossed2SolutionArray = new GeneticAlgorithmSolution[2];
208      crossed2SolutionArray[0] = new GeneticAlgorithmSolution(crossedChromosome1);
209      crossed2SolutionArray[1] = new GeneticAlgorithmSolution(crossedChromosome2);
210
211      //All done, return
212      return crossed2SolutionArray;
213  }
```

Figura 16 - Método para efetuar o cruzamento entre 2 soluções

```
215  /**
216   * Crossover the existing solutions into a total of X population
217   * Elites are the Y number of solutions that we keep unchanged
218   */
219  private static List<GeneticAlgorithmSolution> crossoverFunctionIntoXPopulation(
220      List<GeneticAlgorithmSolution> originalSolutionsList,
221      int xPopulationSizeGoal,
222      int yNumberOfEliteUnchangedSolutions) {
223
224
225      Random rand = new Random();
226
227      //Creates a list to store the crossed solutions
228      List<GeneticAlgorithmSolution> crossedSolutions = new ArrayList<>();
229      //Adds the elites to the crossedSolutions
230      for (int i = 0; i < yNumberOfEliteUnchangedSolutions; i++)
231          crossedSolutions.add(new GeneticAlgorithmSolution(originalSolutionsList.get(i)));
232      //For each 2 non elite solution
233      for (int i = yNumberOfEliteUnchangedSolutions; i + 1 < originalSolutionsList.size(); i++) {
234          GeneticAlgorithmSolution originalSolution1 = new GeneticAlgorithmSolution(originalSolutionsList.get(i));
235          GeneticAlgorithmSolution originalSolution2 = new GeneticAlgorithmSolution(originalSolutionsList.get(i + 1));
236
237          //Makes crossover of the 2 original solutions
238          GeneticAlgorithmSolution[] crossed2SolutionsArray = GeneticAlgorithmSolution.crossoverFunction(originalSolution1, originalSolution2);
239
240          //Adds the 2 non elite crossed solutions to the list
241          crossedSolutions.add(crossed2SolutionsArray[0]);
242          crossedSolutions.add(crossed2SolutionsArray[1]);
243          //Increments a second time the counter (we modified 2 solutions)
244          i++;
245      }
246
247      //While number of solutions is below the population goal
248      for (int i = originalSolutionsList.size(); i + 1 < xPopulationSizeGoal; i++) {
249          //adds new crossed versions
250          GeneticAlgorithmSolution originalSolution1 = new GeneticAlgorithmSolution(originalSolutionsList.get(rand.nextInt(1, originalSolutionsList.size())));
251          GeneticAlgorithmSolution originalSolution2 = new GeneticAlgorithmSolution(originalSolutionsList.get(rand.nextInt(1, originalSolutionsList.size())));
252
253          //Makes crossover of the 2 original solutions
254          GeneticAlgorithmSolution[] crossed2SolutionsArray = GeneticAlgorithmSolution.crossoverFunction(originalSolution1, originalSolution2);
255
256          //Adds the 2 crossed solutions to the list
257          crossedSolutions.add(crossed2SolutionsArray[0]);
258          crossedSolutions.add(crossed2SolutionsArray[1]);
259          //Increments a second time the counter (we modified 2 solutions)
260          i++;
261      }
262
263      //Sorts the new list
264      crossedSolutions.sort(Comparator.naturalOrder());
265      //Returns the list with the mutated solutions
266      return crossedSolutions;
267  }
```

Figura 17 - Método para efetuar o cruzamento entre as soluções de uma lista à exceção de Y “elites” (não queremos destruir as nossas melhores soluções) até atingir uma população de X soluções.



## Mutação

```
271  /**
272   * Based on a eventSuccessChance, returns true or false if the event succeeds or not
273   */
274  private static boolean chanceEventRoulette(double eventSuccessChance) throws Exception {
275      if (eventSuccessChance < 0 || eventSuccessChance > 1)
276          throw new Exception("eventSuccessChance must be between 0 and 1");
277      if (eventSuccessChance == 1)
278          return true;
279      //Makes random number between 0 (inclusive) and 101(exclusive)
280      int randomNumber = (new Random()).nextInt(0, 101);
281      //Gets eventSuccessChance in % form
282      int eventSuccessChanceNumber = (int) (eventSuccessChance * 100);
283      //if eventSuccessChanceNumber >= randomNumber it means that the event had success, otherwise it failed
284      return (eventSuccessChanceNumber >= randomNumber);
285  }
```

Figura 18 - Método auxiliar que decide se dada a probabilidade de um evento acontecer (a mutação), ele acontece ou não.

```
288  /**
289   * Mutates a gene
290   */
291  private static int mutateGene(int originalGene) {
292
293      //Simple gene mutation ... if 0 becomes 1, if 1 becomes 0
294      if (originalGene == 0) {
295          return 1;
296      } else {
297          return 0;
298      }
299  }
300
301  /**
302   * Mutates a chromosome
303   */
304
305  private static int[] mutateChromosome(
306      int[] originalChromosome,
307      double chanceOfMutatedGene) throws Exception {
308      Random rand = new Random();
309      //Creates an array to store the mutated genes
310      int[] mutatedChromosome = new int[originalChromosome.length];
311
312      //For each gene in the originalChromosome
313      for (int i = 0; i < originalChromosome.length; i++) {
314          int mutatedGene = originalChromosome[i];
315          //If given the chance of gene mutation, it mutates
316          if (GeneticAlgorithmSolution.chanceEventRoulette(chanceOfMutatedGene)) {
317              mutatedGene = GeneticAlgorithmSolution.mutateGene(mutatedGene);
318              //With the mutation done, stores the mutated gene in the mutated chromosome
319              mutatedChromosome[i] = mutatedGene;
320          } else {
321              //If not stores the gene without mutation
322              mutatedChromosome[i] = mutatedGene;
323          }
324      }
325      //With all the mutations done, returns the mutated chromosome
326      return mutatedChromosome;
327  }
328
329  /**
330   * Mutates a Solution
331   */
332  private void mutateSolution(double chanceOfGeneMutation) throws Exception {
333      //Mutates the current solution
334      this.chromosome = GeneticAlgorithmSolution.mutateChromosome(this.chromosome, chanceOfGeneMutation);
335      this.calculateTotalCost();
336  }
```

Figura 19 - Métodos que efetuam a mutação de 1 gene, 1 cromossoma e uma solução respectivamente

```

338  /**
339   * Mutates the solutions in the array on a chance base except for X number of elites
340   * Obs: if the xPopulationSizeGoal is different from the originalSolutionsList size, it will add new mutation variations
341   * till the populationSizeGoal is achieved
342   */
343  private static List<GeneticAlgorithmSolution> mutationFunctionIntoXPopulationChanceBased(
344      List<GeneticAlgorithmSolution> originalSolutionsList,
345      int xPopulationSizeGoal,
346      int yNumberOfEliteUnchangedSolutions,
347      double chanceOfSolutionMutation,
348      double chanceOfGeneMutation) throws Exception {
349      Random rand = new Random();
350      //Creates a list to store the mutated solutions
351      List<GeneticAlgorithmSolution> mutatedSolutions = new ArrayList<>();
352      //Adds the elites to the mutatedSolutions
353      for (int i = 0; i < yNumberOfEliteUnchangedSolutions; i++)
354          mutatedSolutions.add(new GeneticAlgorithmSolution(originalSolutionsList.get(i)));
355      //For each non elite solution
356      for (int i = yNumberOfEliteUnchangedSolutions; i < originalSolutionsList.size(); i++) {
357          GeneticAlgorithmSolution mutatedSolution = new GeneticAlgorithmSolution(originalSolutionsList.get(i));
358          //If given the chanceOfSolutionMutation, it mutates
359          if (GeneticAlgorithmSolution.chanceEventRoulette(chanceOfSolutionMutation))
360              mutatedSolution.mutateSolution(chanceOfGeneMutation);
361          //Adds the non elite solution to the list
362          mutatedSolutions.add(mutatedSolution);
363      }
364      //While number of solutions is below the population goal
365      for (int i = originalSolutionsList.size(); i < xPopulationSizeGoal; i++) {
366          //adds new mutated versions
367          GeneticAlgorithmSolution mutatedSolution = new GeneticAlgorithmSolution(originalSolutionsList.get(rand.nextInt(originalSolutionsList.size())));
368          mutatedSolution.mutateSolution(chanceOfGeneMutation);
369          mutatedSolutions.add(mutatedSolution);
370      }
371      //Sorts the new list
372      mutatedSolutions.sort(Comparator.naturalOrder());
373      //Returns the list with the mutated solutions
374      return mutatedSolutions;
375  }

```

Figura 20 - Método que efetua mutações nas soluções de uma lista à exceção das Y melhores soluções (as “elites”) até atingir uma população de X soluções

```

447  @Override
448  public int compareTo(GeneticAlgorithmSolution o) {
449      //If the total cost of the current solution is bigger than the one from 0 solution, return 1, otherwise return -1.
450      if (this.currentSolutionCost == o.currentSolutionCost) {
451          return 0;
452      } else if (this.currentSolutionCost > o.currentSolutionCost) {
453          return 1;
454      } else {
455          return -1;
456      }
457  }
458
459  @Override
460  public String toString() {
461      return "GeneticAlgorithmSolution{" +
462          "currentSolutionCost=" + String.format("%.3f", currentSolutionCost) +
463          ", chromosome=" + Arrays.toString(chromosome) +
464          '}';
465  }
466  }

```

Figura 21 - Implementação dos métodos *compareTo* e *toString*

## Evolução

```
378  /**
379   * Runs the evolution, finds and returns the best solution
380   */
381  public static GeneticAlgorithmSolution runEvolution(boolean showProgressMessages) throws Exception {
382
383      System.out.println("\n#####");
384      System.out.println("### -- The Great Evolution -- ###");
385      System.out.println("#####");
386      System.out.println("ScenarioName: " + problemScenario.getName() + "\t" + "OptimalSolution: " + problemScenario.getOptimalSolutionValue());
387      System.out.println("NumberOfWarehouses: " + problemScenario.getWarehouseLocations().length + "\t" + "NumberOfCustomers: " + problemScenario.getCustomers().length);
388      System.out.println("MaxGeneration: " + maxNumberOfGenerations + "\t" + "PopulationSize: " + numberOfSolutionsPerGeneration + "\t" + "MaxTime(ms): " + maxRunEvolutionDuration);
389
390      //Generates X number of Random Solutions (initial population)
391      List<GeneticAlgorithmSolution> generatedSolutions = GeneticAlgorithmSolution.generateXRandomSolutions(numberOfSolutionsPerGeneration);
392
393      //While X, Selects, Mutates and repeat
394      int currentGeneration = 0;
395      long startTime = System.currentTimeMillis();
396      long maxDuration = maxRunEvolutionDuration; // 0.5 seconds in milliseconds
397
398      //Stops when you reach max number of solutions, max number of time or u get the optimal solution
399      while (currentGeneration < maxNumberOfGenerations
400             && (System.currentTimeMillis() - startTime) <= maxDuration
401             && !(String.format("%.2f", generatedSolutions.get(0).currentSolutionCost).equals(String.format("%.2f", problemScenario.getOptimalSolutionValue()))))
402      ) {
403          //Selects top solutions
404          generatedSolutions = GeneticAlgorithmSolution.selectionFunctionXBest(generatedSolutions, GeneticAlgorithmSolution.xNumberOfBestSolutionsForSelectionFunction);
405
406          //Crosses the solutions (not the elites)
407          generatedSolutions = GeneticAlgorithmSolution.crossoverFunctionIntoXPopulation(
408              generatedSolutions,
409              xPopulationSizeGoalForCrossoverFunction,
410              yNumberOfEliteNonCrossedSolutions);
411
412          //Mutates the solutions (not the elites)
413          generatedSolutions = GeneticAlgorithmSolution.mutationFunctionIntoXPopulationChanceBased(
414              generatedSolutions,
415              xPopulationSizeGoalForMutationFunction,
416              yNumberOfEliteNonMutatedSolutions,
417              chanceOfSolutionMutation,
418              chanceOfGeneMutation);
419
420          //Print Progress every few generations
421          if ((currentGeneration % (maxNumberOfGenerations / 40)) == 0 && showProgressMessages) {
422              System.out.println("\n#####");
423              System.out.println("CurrentGeneration: " + currentGeneration + "\t" + "CompilationTime(ms): " + (System.currentTimeMillis() - startTime) + " milliseconds");
424              System.out.println("BestSolution: " + generatedSolutions.get(0));
425              System.out.println("OptimalRatio: " + String.format("%.5f", (problemScenario.getOptimalSolutionValue()/generatedSolutions.get(0).currentSolutionCost)));
426              System.out.println("#####");
427          }
428
429          //Increase counters
430          currentGeneration++;
431      }
432
433      //returns Best Solution //the solutions all get sorted during the process
434      GeneticAlgorithmSolution bestSolution = generatedSolutions.get(0);
435      System.out.println("#####");
436      System.out.println("CurrentGeneration: " + (currentGeneration - 1) + "\t" + "CompilationTime(ms): " + (System.currentTimeMillis() - startTime) + " milliseconds");
437      System.out.println("BestSolution: " + bestSolution.toString());
438      System.out.println("OptimalRatio: " + String.format("%.5f", (problemScenario.getOptimalSolutionValue()/bestSolution.currentSolutionCost)));
439      //Sometimes u might get a very small decimal case of difference ... so lets round it up
440      if ((String.format("%.2f", bestSolution.currentSolutionCost).equals(String.format("%.2f", problemScenario.getOptimalSolutionValue()))))
441      {
442          System.out.println("U GOT THE OPTIMAL SOLUTION!");
443          System.out.println("#####");
444          return bestSolution;
445      }
```

Figura 22 - Método que efetua o ciclo de evolução (população inicial, seleção, cruzamento, mutação, repetição) até que um dos critérios de paragem seja atingido, retornando a melhor solução encontrada.

### **Execução (Algoritmo Genético)**

```
//Read the problem scenarios data
ProblemScenarios problemScenarios = new ProblemScenarios();
problemScenarios.ReadScenariosFromDir(path130s); // <-- Change the path to read the scenarios from

boolean feedGreedyNonOptimalSolutionsToTheSimulatedAnnealing = false;

for (ProblemScenario problemScenario : problemScenarios.getProblemScenarios()) {
    System.out.println("\n#####");
    System.out.println("#####\t" + "Problem: " + problemScenario.getName() + "\t####");
    System.out.println("#####");

    //Solve the scenario with the Genetic Algorithm
    GeneticAlgorithmSolution.setProblemScenario(problemScenario);
    GeneticAlgorithmSolution.runEvolution( showProgressMessages: false);
}
```

**Figura 23 - Execução do algoritmo genético**

### ***Output (Algoritmo Genético)***

```
#####  
### -- The Great Evolution -- ###  
#####  
ScenarioName: cap132.txt    OptimalSolution: 851495.325  
NumberOfWarehouses: 50    NumberOfCustomers: 50  
MaxGeneration: 200    PopulationSize: 100 MaxTime(ms): 50000  
#####  
CurrentGeneration: 177    CompilationTime(ms): 6182 milliseconds  
BestSolution: GeneticAlgorithmSolution{currentSolutionCost=851495.325, chromosome=[0, 0, 0, 0, 0,  
OptimalRatio: 1,00000  
U GOT THE OPTIMAL SOLUTION!  
#####
```

**Figura 24 - Output do algoritmo genético**

## 4.2 Algoritmo *Greedy* (ou Ganancioso)

Um algoritmo ***Greedy*** (ou ganancioso) é uma abordagem de resolução de problemas que constrói a solução passo a passo, fazendo uma série de escolhas locais que parecem ser as melhores no momento, na esperança de que essa estratégia leve a uma solução global ótima.

### Estrutura Básica de um Algoritmo *Greedy*

#### 1. Escolha *Greedy*:

- Em cada etapa, o algoritmo seleciona a opção que parece ser a melhor ou mais vantajosa com base em um critério específico (por exemplo, o menor custo, maior valor, etc.).

#### 2. Irrevogabilidade:

- A escolha feita é final e não é revisada. O algoritmo não reconsidera as decisões anteriores, não faz *backtracking*.

#### 3. Construção da Solução:

- A solução é construída iterativamente. A cada passo, o algoritmo adiciona uma nova parte à solução final, até que uma solução completa seja formada.

### Vantagens

- **Simplicidade:** Algoritmos *Greedy* são fáceis de entender e implementar.
- **Eficiência:** Muitas vezes são muito rápidos, com tempo de execução geralmente linear ou polinomial.

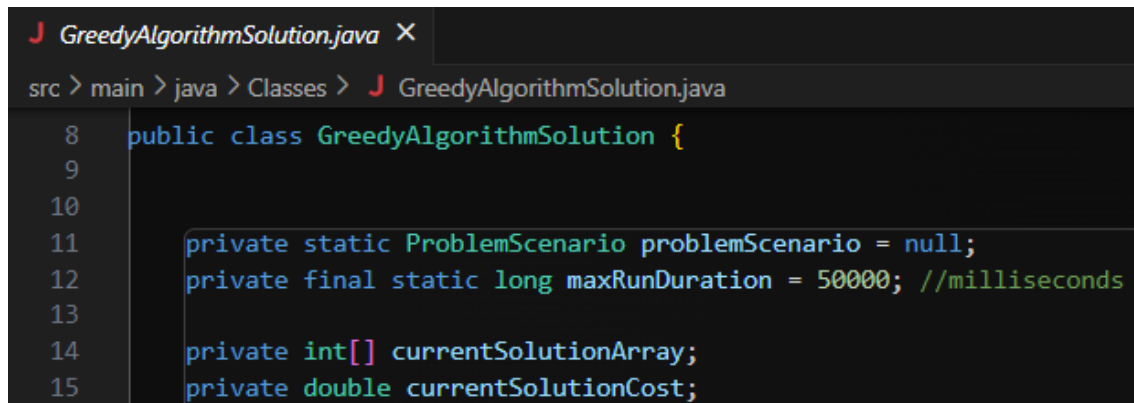
### Desvantagens

- **Ótimo Local vs. Ótimo Global:** Nem sempre garantem a solução globalmente ótima. Para alguns problemas, a solução *Greedy* pode ser sub ótima.
- **Aplicabilidade Limitada:** Não são adequados para todos os tipos de problemas. Funcionam bem para problemas que possuem a propriedade de "optimalidade de subestrutura" e "propriedade *Greedy*".

### Conclusão

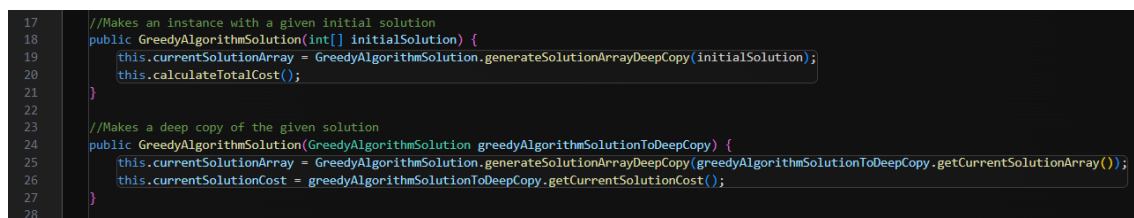
Um algoritmo *Greedy* é uma estratégia eficiente e direta para resolver problemas, onde as decisões locais feitas em cada etapa levam, na maioria dos casos, a uma boa solução global, embora nem sempre seja a ótima. É ideal para problemas em que uma solução rápida é preferível a uma solução exata.

## Implementação (Algoritmo Greedy)



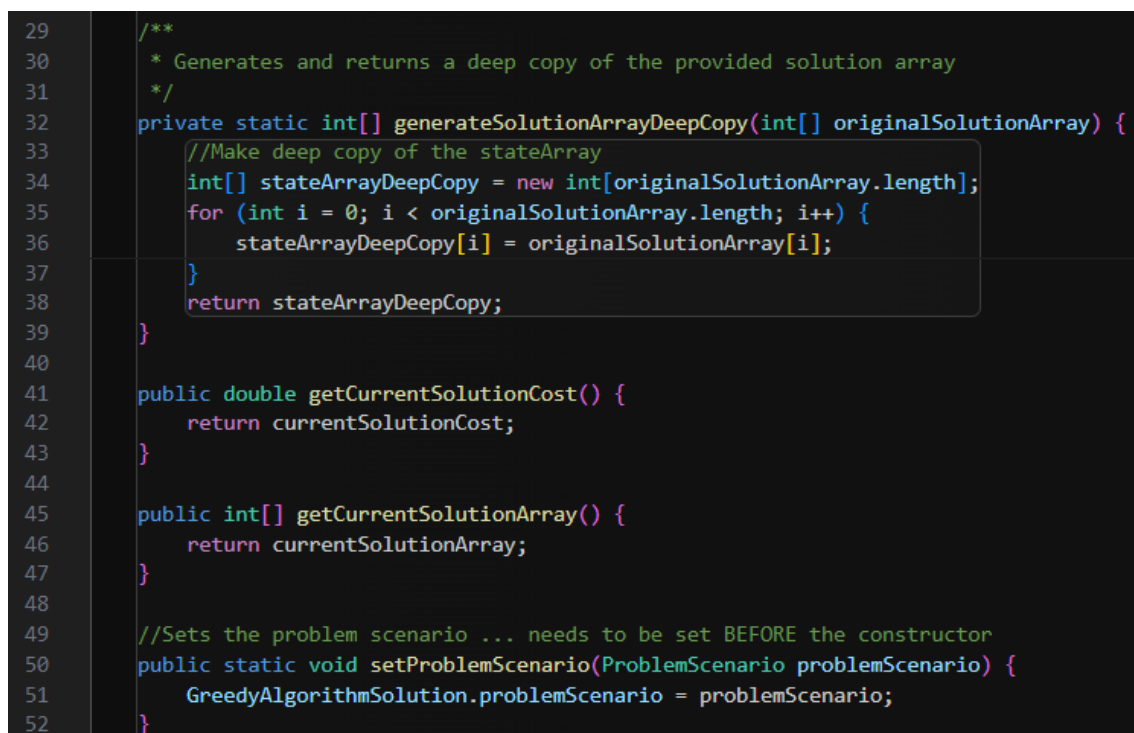
```
8 public class GreedyAlgorithmSolution {
9
10
11     private static ProblemScenario problemScenario = null;
12     private final static long maxRunDuration = 50000; //milliseconds
13
14     private int[] currentSolutionArray;
15     private double currentSolutionCost;
```

Figura 25 - Inicialização de Variáveis e Hiper Parâmetros



```
17 //Makes an instance with a given initial solution
18 public GreedyAlgorithmSolution(int[] initialSolution) {
19     this.currentSolutionArray = GreedyAlgorithmSolution.generateSolutionArrayDeepCopy(initialSolution);
20     this.calculateTotalCost();
21 }
22
23 //Makes a deep copy of the given solution
24 public GreedyAlgorithmSolution(GreedyAlgorithmSolution greedyAlgorithmSolutionToDeepCopy) {
25     this.currentSolutionArray = GreedyAlgorithmSolution.generateSolutionArrayDeepCopy(greedyAlgorithmSolutionToDeepCopy.getCurrentSolutionArray());
26     this.currentSolutionCost = greedyAlgorithmSolutionToDeepCopy.getCurrentSolutionCost();
27 }
28 }
```

Figura 26 - Métodos Construtores



```
29 /**
30  * Generates and returns a deep copy of the provided solution array
31  */
32 private static int[] generateSolutionArrayDeepCopy(int[] originalSolutionArray) {
33     //Make deep copy of the stateArray
34     int[] stateArrayDeepCopy = new int[originalSolutionArray.length];
35     for (int i = 0; i < originalSolutionArray.length; i++) {
36         stateArrayDeepCopy[i] = originalSolutionArray[i];
37     }
38     return stateArrayDeepCopy;
39 }
40
41 public double getCurrentSolutionCost() {
42     return currentSolutionCost;
43 }
44
45 public int[] getCurrentSolutionArray() {
46     return currentSolutionArray;
47 }
48
49 //Sets the problem scenario ... needs to be set BEFORE the constructor
50 public static void setProblemScenario(ProblemScenario problemScenario) {
51     GreedyAlgorithmSolution.problemScenario = problemScenario;
52 }
```

Figura 27 - Método para definir o cenário (definir os dados do problema), métodos de acesso e método auxiliar para fazer uma *deep copy* da *solutionArray*

## Avaliação

```
54  /**
55   * Calculate the total cost value of the current solution
56   */
57  private void calculateTotalCost() {
58      double totalCost = 0;
59      //For each warehouse with 1 value in the chromosome, add the fixed cost
60      //For each chromosome
61      for (int i = 0; i < this.currentSolutionArray.length; i++) {
62          //if the chromosome is 1 (warehouse is open)
63          if (this.currentSolutionArray[i] == 1) {
64              //Add the fixed cost of the warehouse to the total cost
65              totalCost += this.problemScenario.getWarehouseLocations()[i].getFixedCost();
66          }
67      }
68
69      //For each customer, add the costOfAllocation of the nearest opened warehouse
70      //For each customer
71      for (int i = 0; i < this.problemScenario.getCustomers().length; i++) {
72          //Get the cost of allocations
73          Map<Integer, Double> costOfAllocation = this.problemScenario.getCustomers()[i].getCostOfAllocation();
74          // Create a list of the map entries
75          List<Map.Entry<Integer, Double>> entries = new ArrayList<>(costOfAllocation.entrySet());
76
77          // Sort the list by values
78          entries.sort(Map.Entry.comparingByValue());
79
80          // Iterate over the sorted entries
81          for (Map.Entry<Integer, Double> entry : entries) {
82              //Logger.WriteLine("Key: " + entry.getKey() + ", Value: " + entry.getValue());
83              // Until we find the first opened warehouse
84              if (this.currentSolutionArray[entry.getKey()] == 1) {
85                  //If so ... Add the value to the total cost and break the loop
86                  totalCost += entry.getValue();
87                  break;
88              }
89          }
90      }
91
92      //If all the warehouses are closed (the cost is 0.0, then make the cost a very large number)
93      if (totalCost == 0) {
94          totalCost = Double.POSITIVE_INFINITY;
95      }
96
97      //All done ... set the new cost for the current solution
98      this.currentSolutionCost = totalCost;
99  }
```

Figura 28 - Método para calcular o custo da solução atual (igual à função fitness do algoritmo genético)

```
177  @Override
178  public String toString() {
179      return "GreedyAlgorithmSolution{" +
180          "currentSolutionCost=" + String.format("%.3f", currentSolutionCost) +
181          ", currentSolutionArray=" + Arrays.toString(currentSolutionArray) +
182          '}';
183  }
```

Figura 29 - Implementação do método *toString*

## Run

```

101 public static GreedyAlgorithmSolution run(boolean showProgressMessages) {
102
103     System.out.println("\n#####");
104     System.out.println("### --- The Greedy One --- ###");
105     System.out.println("#####");
106     System.out.println("ScenarioName: " + problemScenario.getName() + "\t" + "OptimalSolution: " + problemScenario.getOptimalSolutionValue());
107     System.out.println("NumberOfWarehouses: " + problemScenario.getWarehouseLocations().length + "\t" + "NumberOfCustomers: " + problemScenario.getCustomers().length);
108     System.out.println("#####");
109
110     //Save starting time
111     long startTime = System.currentTimeMillis();
112
113     //Initialize solution with all warehouses closed (default int is 0)
114     GreedyAlgorithmSolution bestSolution = new GreedyAlgorithmSolution(new int[problemScenario.getWarehouseLocations().length]);
115
116     int iterationCounter = 0;
117     int improvementCounter;
118
119     //Do while bestSolution != optimal one
120     while ((System.currentTimeMillis() - startTime) <= maxRunDuration
121         && !(String.format("%.2f", bestSolution.getCurrentSolutionCost()).equals(String.format("%.2f", problemScenario.getOptimalSolutionValue()))
122     ) {
123         //Store the best improvement difference
124         GreedyAlgorithmSolution newBestSolution = new GreedyAlgorithmSolution(bestSolution);
125         //Counts the improvements in each for cycle
126         improvementCounter = 0;
127
128         //For each position in the array (each warehouse)
129         for (int i = 0; i < problemScenario.getWarehouseLocations().length; i++) {
130             //Make a solutionArray (deep copy of the best one)
131             int[] newSolutionArray = GreedyAlgorithmSolution.generateSolutionArrayDeepCopy(bestSolution.currentSolutionArray);
132             //Change the current warehouse to open
133             newSolutionArray[i] = 1;
134             //And make a solution with it
135             GreedyAlgorithmSolution newSolution = new GreedyAlgorithmSolution(newSolutionArray);
136
137             //Now, check if the cost of the new solution is lower than the current new best solution
138             if (newSolution.getCurrentSolutionCost() < newBestSolution.getCurrentSolutionCost()) {
139                 //If so, make the current solution the new best one (use deep copy)
140                 newBestSolution = new GreedyAlgorithmSolution(newSolution);
141                 improvementCounter++;
142             }
143         }
144
145         //If improvements were made, make the best solution the new best one
146         if (improvementCounter != 0) {
147             bestSolution = new GreedyAlgorithmSolution(newBestSolution);
148         } else {
149             //Else, break the cycle (no new improvements are possible)
150             break;
151         }
152
153         //Print Progress every once in a while
154         if ((iterationCounter % 1) == 0 && showProgressMessages) {
155             System.out.println("\n#####");
156             System.out.println("CompilationTime(ms): " + (System.currentTimeMillis() - startTime) + " milliseconds");
157             System.out.println("BestSolution: " + bestSolution.toString());
158             System.out.println("OptimalRatio: " + String.format("%.5f", (problemScenario.getOptimalSolutionValue()/bestSolution.currentSolutionCost)));
159             System.out.println("#####");
160         }
161         iterationCounter++;
162     }
163
164     //All done
165     //Print the result and return the best solution
166
167     System.out.println("CompilationTime(ms): " + (System.currentTimeMillis() - startTime) + " milliseconds");
168     System.out.println("BestSolution: " + bestSolution.toString());
169     System.out.println("OptimalRatio: " + String.format("%.5f", (problemScenario.getOptimalSolutionValue()/bestSolution.currentSolutionCost)));
170     if ((String.format("%.2f", bestSolution.currentSolutionCost()).equals(String.format("%.2f", problemScenario.getOptimalSolutionValue()))
171     ) {
172         System.out.println("U GOT THE OPTIMAL SOLUTION!");
173     }
174     System.out.println("#####");
175     return bestSolution;
176 }

```

Figura 30 - Método que iterativamente efetua a escolha *greedy* construindo assim a solução



### Execução (Algoritmo Greedy)

```
//Read the problem scenarios data
ProblemScenarios problemScenarios = new ProblemScenarios();
problemScenarios.ReadScenariosFromDir(path130s); // <-- Change the path to read the scenarios from here

boolean feedGreedyNonOptimalSolutionsToTheSimulatedAnnealing = false;

for (ProblemScenario problemScenario : problemScenarios.getProblemScenarios()) {
    System.out.println("\n#####");
    System.out.println("#####\t" + "Problem: " + problemScenario.getName() + "\t#####");
    System.out.println("#####");

    //Solve the scenario with the Genetic Algorithm
    GeneticAlgorithmSolution.setProblemScenario(problemScenario);
    GeneticAlgorithmSolution.runEvolution( showProgressMessages: false);

    //Get a good solution with the GreedyAlgorithm
    GreedyAlgorithmSolution.setProblemScenario(problemScenario);
    GreedyAlgorithmSolution bestGreedySolution = GreedyAlgorithmSolution.run( showProgressMessages: false);
```

**Figura 31 - Execução do algoritmo *greedy***

### ***Output (Algoritmo Greedy)***

```
#####  
### ---- The Greedy One ---- ###  
#####  
ScenarioName: cap132.txt      OptimalSolution: 851495.325  
NumberOfWarehouses: 50   NumberOfCustomers: 50  
#####  
CompilationTime(ms): 114 milliseconds  
BestSolution: GreedyAlgorithmSolution{currentSolutionCost=852762,875, currentSolutionArray=[0, 0, 0, 0]  
OptimalRatio: 0,99851  
#####
```

**Figura 32 - Output do algoritmo *greedy***

## 4.3 Algoritmo *Simulated Annealing*

O ***Simulated Annealing*** (SA) é um algoritmo de otimização inspirado no processo físico de recozimento (*annealing*) de metais. Neste processo, um material é aquecido até uma temperatura elevada e depois arrefecido gradualmente, de forma a minimizar defeitos e alcançar uma estrutura cristalina estável, com energia mínima.

### Estrutura Básica de um Algoritmo *Simulated Annealing*

#### 1. Solução Inicial:

- O algoritmo começa com uma solução inicial (um estado inicial), que pode ser escolhida de forma aleatória ou baseada numa heurística.

#### 2. Perturbação (Vizinhança):

- Em cada iteração, gera-se uma nova solução a partir da solução atual, fazendo uma pequena modificação (uma mudança no estado). Esta modificação define a "vizinhança" da solução atual.

#### 3. Avaliação:

- A nova solução é avaliada através de uma função de custo ou objetivo (semelhante à função fitness do algoritmo genético).

#### 4. Aceitação da Solução:

- Se a nova solução for melhor (ou seja, tiver um custo inferior), é automaticamente aceite.
- Se a nova solução for pior, ainda pode ser aceite, mas com uma certa probabilidade, que diminui ao longo do tempo.

Esta probabilidade é dada pela função de aceitação

**$(\text{Math.exp}(-\text{costDifference} / \text{currentTemperature}) > \text{rand.nextDouble}())$** ,

onde quanto "pior" que a solução atual e quanto menor a temperatura, menor a probabilidade de a nova solução ser aceite.

#### 5. Atualização da Temperatura:

- A temperatura é gradualmente reduzida à medida que o algoritmo avança. No início, com a temperatura elevada, o algoritmo aceita pioras na solução com mais frequência, permitindo uma exploração ampla do espaço de soluções. À medida que a temperatura diminui, o algoritmo torna-se mais conservador, aceitando apenas pequenas pioras ou nenhuma, promovendo a convergência para uma solução final.

#### 6. Condições de Paragem:

- O processo continua até que a temperatura atinja um valor mínimo ou até que um número máximo de iterações seja atingido, momento em que a melhor solução encontrada é considerada a solução final.

## Implementação (*Simulated Annealing*)

```
J SimulatedAnnealingSolution.java X
src > main > java > Classes > J SimulatedAnnealingSolution.java
5 public class SimulatedAnnealingSolution {
6
7     private final static double initialTemperature = 100000;
8     private final static double coolingRate = 0.995;
9     private final static long maxRunDuration = 50000; //milliseconds
10    private final static double chanceOfIndividualPointStateChange = 0.20;
11
12    private static ProblemScenario problemScenario = null;
13
14    private double currentTemperature;
15    private int[] currentStateArray;
16    private double currentSolutionCost;
```

Figura 33 - Inicialização de Variáveis e Hiper Parâmetros

```
18 //Makes an instance with a random initial solution
19 public SimulatedAnnealingSolution() {
20     this.currentStateArray = SimulatedAnnealingSolution.generateRandomStateArray();
21     this.currentTemperature = initialTemperature;
22     this.calculateTotalCost();
23 }
24
25 //Makes an instance with a given initial solution
26 public SimulatedAnnealingSolution(int[] initialSolution) {
27     this.currentStateArray = SimulatedAnnealingSolution.generateCurrentStateArrayDeepCopy(initialSolution);
28     this.currentTemperature = initialTemperature;
29     this.calculateTotalCost();
30 }
31
32 //Makes a deep copy of the given solution
33 public SimulatedAnnealingSolution(SimulatedAnnealingSolution simulatedAnnealingSolutionToDeepCopy) {
34     this.currentStateArray = SimulatedAnnealingSolution.generateCurrentStateArrayDeepCopy(
35         simulatedAnnealingSolutionToDeepCopy.getCurrentSolutionArray());
36     this.currentSolutionCost = simulatedAnnealingSolutionToDeepCopy.getCurrentSolutionCost();
37     this.currentTemperature = simulatedAnnealingSolutionToDeepCopy.getCurrentTemperature();
38 }
```

Figura 34 - Métodos Construtores

```

40  /**
41   * Generates and returns a deep copy of the provided solution
42   */
43  private static int[] generateCurrentStateArrayDeepCopy(int[] originalStateArray) {
44      //Make deep copy of the stateArray
45      int[] stateArrayDeepCopy = new int[originalStateArray.length];
46      for (int i = 0; i < originalStateArray.length; i++) {
47          stateArrayDeepCopy[i] = originalStateArray[i];
48      }
49      return stateArrayDeepCopy;
50  }
51
52  //Sets the problem scenario ... needs to be set BEFORE the constructor
53  public static void setProblemScenario(ProblemScenario problemScenario) {
54      SimulatedAnnealingSolution.problemScenario = problemScenario;
55  }
56
57  public int[] getCurrentSolutionArray() {
58      return currentStateArray;
59  }
60
61  public double getCurrentSolutionCost() {
62      return currentSolutionCost;
63  }
64
65  public double getCurrentTemperature() {
66      return currentTemperature;
67  }

```

Figura 35 - Método para definir o cenário (definir os dados do problema), métodos de acesso e método auxiliar para fazer uma *deep copy* de um *StateArray*

### Solução Inicial Aleatória

```

70  /**
71   * Generates and returns a random StateArray
72   */
73  private static int[] generateRandomStateArray() {
74      Random rand = new Random();
75      //Creates a list to store the generated genes
76      int[] generatedStateArray = new int[problemScenario.getWarehouseLocations().length];
77
78      for (int i = 0; i < generatedStateArray.length; i++)
79          generatedStateArray[i] = rand.nextInt(0, 2);
80      //Returns generated StateArray
81      return generatedStateArray;
82  }

```

Figura 36 - Método para gerar 1 *StateArray* aleatório (para quando não existe solução inicial)

## Avaliação

```
84  /**
85   * Calculate the total cost value of the current solution
86   */
87  private void calculateTotalCost() {
88      double totalCost = 0;
89      //For each warehouse with 1 value in the chromosome, add the fixed cost
90      //For each chromosome
91      for (int i = 0; i < this.currentStateArray.length; i++) {
92          //if the chromosome is 1 (warehouse is open)
93          if (this.currentStateArray[i] == 1) {
94              //Add the fixed cost of the warehouse to the total cost
95              totalCost += this.problemScenario.getWarehouseLocations()[i].getFixedCost();
96          }
97      }
98
99      //For each customer, add the costOfAllocation of the nearest opened warehouse
100     //For each customer
101     for (int i = 0; i < this.problemScenario.getCustomers().length; i++) {
102         //Get the cost of allocations
103         Map<Integer, Double> costOfAllocation = this.problemScenario.getCustomers()[i].getCostOfAllocation();
104         // Create a list of the map entries
105         List<Map.Entry<Integer, Double>> entries = new ArrayList<>(costOfAllocation.entrySet());
106
107         // Sort the list by values
108         entries.sort(Map.Entry.comparingByValue());
109
110         // Iterate over the sorted entries
111         for (Map.Entry<Integer, Double> entry : entries) {
112             //Logger.WriteLine("Key: " + entry.getKey() + ", Value: " + entry.getValue());
113             // Until we find the first opened warehouse
114             if (this.currentStateArray[entry.getKey()] == 1) {
115                 //If so ... Add the value to the total cost and break the loop
116                 totalCost += entry.getValue();
117                 break;
118             }
119         }
120     }
121
122     //If all the warehouses are closed (the cost is 0.0, then make the cost a very large number)
123     if (totalCost == 0) {
124         totalCost = Double.POSITIVE_INFINITY;
125     }
126
127     //All done ... set the new cost for the current solution
128     this.currentSolutionCost = totalCost;
129 }
```

Figura 37 - Método para calcular o custo da solução atual (semelhante à solução fitness do algoritmo genético)

## Perturbação

```
130  /**
131   * Based on a eventSuccessChance, returns true or false if the event succeeds or not
132   */
133  private static boolean chanceEventRoulette(double eventSuccessChance) throws Exception {
134      if (eventSuccessChance < 0 || eventSuccessChance > 1)
135          throw new Exception("eventSuccessChance must be between 0 and 1");
136      if (eventSuccessChance == 1)
137          return true;
138      //Makes random number between 0 (inclusive) and 101(exclusive)
139      int randomNumber = (new Random()).nextInt(0, 101);
140      //Gets eventSuccessChance in % form
141      int eventSuccessChanceNumber = (int) (eventSuccessChance * 100);
142      //if eventSuccessChanceNumber >= randomNumber it means that the event had success, otherwise it failed
143      return (eventSuccessChanceNumber >= randomNumber);
144  }
```

Figura 38 - Método auxiliar que decide se dada a probabilidade de um evento acontecer (a perturbação de um elo individual do *StateArray*), ele acontece ou não.

```
146  /**
147   * Change a IndividualPointState
148   */
149  private static int changeIndividualPointState(int originalIndividualPointState) {
150
151      //Simple state change ... if 0 becomes 1, if 1 becomes 0
152      if (originalIndividualPointState == 0) {
153          return 1;
154      } else {
155          return 0;
156      }
157  }
158
159  /**
160   * Changes a stateArray
161   */
162  private static int[] changeStateArray(
163      int[] originalStateArray,
164      double chanceOfIndividualStateChange) throws Exception {
165      Random rand = new Random();
166      //Creates an array to store the changedStateArray
167      int[] changedStateArray = new int[originalStateArray.length];
168
169      //For each IndividualStatePoint in the originalStateArray
170      for (int i = 0; i < originalStateArray.length; i++) {
171          int changedIndividualPointState = originalStateArray[i];
172          //If given the chance of individualStatePointChange it changes
173          if (SimulatedAnnealingSolution.chanceEventRoulette(chanceOfIndividualStateChange)) {
174              changedIndividualPointState = SimulatedAnnealingSolution.changeIndividualPointState(changedIndividualPointState);
175              //With the change, stores the changed individualStatePoint in the changedStateArray
176              changedStateArray[i] = changedIndividualPointState;
177          } else {
178              //If not stores the individualStatePoint without change
179              changedStateArray[i] = changedIndividualPointState;
180          }
181      }
182      //With all the changes done, returns the changedStateArray
183      return changedStateArray;
184  }
185
186  /**
187   * Changes a Solution
188   */
189  private void changeSolution(double chanceOfIndividualStateChange) throws Exception {
190      //Changes the currentStateArray
191      this.currentStateArray = SimulatedAnnealingSolution.changeStateArray(this.currentStateArray, chanceOfIndividualStateChange);
192      //Updates the cost
193      this.calculateTotalCost();
194  }
```

Figura 39 - Métodos que efetuam a perturbação de 1 elo de um *StateArray*, de 1 *StateArray* e de uma solução respetivamente

```

273 @Override
274 public String toString() {
275     return "SimulatedAnnealingAlgorithm{" +
276         "currentSolutionCost=" + String.format("%.3f", currentSolutionCost) +
277         ", currentTemperature=" + currentTemperature +
278         ", currentSolutionArray=" + Arrays.toString(currentStateArray) +
279         '}';
280 }
281 }

```

Figura 40 - Implementação do método *toString*

## Run

```

201 public SimulatedAnnealingSolution run(boolean showProgressMessages) throws Exception {
202
203     System.out.println("\n*****");
204     System.out.println("### -- The Great Annealing -- ###");
205     System.out.println("*****");
206     System.out.println("ScenarioName: " + problemScenario.getName() + "\t" + "OptimalSolution: " + problemScenario.getOptimalSolutionValue());
207     System.out.println("NumberOfWarehouses: " + problemScenario.getWarehouseLocations().length + "\t" + "NumberOfCustomers: " + problemScenario.getCustomers().length);
208     System.out.println("*****");
209
210     //Saves starting time and creates random instance
211     long startTime = System.currentTimeMillis();
212     Random rand = new Random();
213
214     //Variable to store the best solution
215     SimulatedAnnealingSolution bestStateSolution = new SimulatedAnnealingSolution(this.currentStateArray);
216
217     //Iteration counter
218     int iterationCounter = 0;
219
220     //Runs simulation while temperature > 1 && time < maxRunDuration && current solution != optimal one
221     while (currentTemperature > 1 && (System.currentTimeMillis() - startTime) <= maxRunDuration
222           && ! (String.format("%.2f", currentSolutionCost).equals(String.format("%.2f", problemScenario.getOptimalSolutionValue()))))
223     {
224         //Create nextState solution from the current one
225         SimulatedAnnealingSolution nextStateSolution = new SimulatedAnnealingSolution(this.currentStateArray);
226         //Change it
227         nextStateSolution.changeSolution(chanceOfIndividualPointStateChange);
228
229         //Calculate costDifferences (negative value is GOOD (the changed state has a cost that is LOWER than the current one))
230         double costDifference = nextStateSolution.getCurrentSolutionCost() - this.currentSolutionCost;
231
232         //If the changed cost is lower, or if the worse cost is "acceptable" ... then make it the current one
233         if (costDifference < 0 || Math.exp(-costDifference / currentTemperature) > rand.nextDouble()) {
234             //Make the current solution the changed one (use deep copy)
235             this.currentStateArray = SimulatedAnnealingSolution.generateCurrentStateArrayDeepCopy(nextStateSolution.getCurrentSolutionArray());
236             this.calculateTotalCost();
237
238             //If it is a new best solution
239             if (this.currentSolutionCost < bestStateSolution.getCurrentSolutionCost()) {
240                 //Then make the best solution, the current one
241                 bestStateSolution = new SimulatedAnnealingSolution(this.currentStateArray);
242             }
243         }
244
245         //Print Progress every once in a while
246         if ((iterationCounter % 40) == 0 && showProgressMessages) {
247             System.out.println("\n*****");
248             System.out.println("CurrentTemperature: " + String.format("%.3f", this.currentTemperature) + "\t" + "CompilationTime(ms): " + (System.currentTimeMillis() - startTime) + " milliseconds");
249             System.out.println("BestSolution: " + bestStateSolution.toString());
250             System.out.println("OptimalRatio: " + String.format("%.5f", (problemScenario.getOptimalSolutionValue()/bestStateSolution.currentSolutionCost)));
251             System.out.println("*****");
252         }
253
254         //Decrease temperature //Increase counter
255         iterationCounter++;
256         this.currentTemperature *= coolingRate;
257     }
258
259     //returns Best Solution
260
261     System.out.println("CurrentTemperature: " + String.format("%.3f", this.currentTemperature) + "\t" + "CompilationTime(ms): " + (System.currentTimeMillis() - startTime) + " milliseconds");
262     System.out.println("BestSolution: " + bestStateSolution.toString());
263     System.out.println("OptimalRatio: " + String.format("%.5f", (problemScenario.getOptimalSolutionValue()/bestStateSolution.currentSolutionCost)));
264     if ((String.format("%.2f", bestStateSolution.currentSolutionCost).equals(String.format("%.2f", problemScenario.getOptimalSolutionValue()))))
265         System.out.println("U GOT THE OPTIMAL SOLUTION!");
266     System.out.println("*****");
267     return bestStateSolution;
268 }
269 }

```

Figura 41 - Método que iterativamente efetua a perturbações e decide aceita-las ou não construindo assim a solução

## Execução (Simulated Annealing)

```
//If the solution from the greedy algorithm is different from the optimal one, feed it to the SimulatedAnnealing
if (bestGreedySolution.getCurrentSolutionCost() != problemScenario.getOptimalSolutionValue()
    && feedGreedyNonOptimalSolutionsToTheSimulatedAnnealing) {
    //Solve the scenario with the SimulatedAnnealingAlgorithm // With the GREEDY sub optimal solution
    System.out.println("Feeding non optimal greedy solution to SimulatedAnnealing algorithm");
    SimulatedAnnealingSolution.setProblemScenario(problemScenario);
    SimulatedAnnealingSolution instance = new SimulatedAnnealingSolution(bestGreedySolution.getCurrentSolutionArray());
    instance.run(false);
} else {
    //Solve the scenario with the SimulatedAnnealingAlgorithm // With RANDOM initial solution
    SimulatedAnnealingSolution.setProblemScenario(problemScenario);
    SimulatedAnnealingSolution instance = new SimulatedAnnealingSolution();
    instance.run(false);
}
```

Figura 42 - Execução do algoritmo Simulated Annealing

## Output (Simulated Annealing)

```
#####
### -- The Great Annealing -- ###
#####
ScenarioName: cap132.txt    OptimalSolution: 851495.325
NumberOfWarehouses: 50    NumberOfCustomers: 50
#####
CurrentTemperature: 0,999    CompilationTime(ms): 1246 milliseconds
BestSolution: SimulatedAnnealingAlgorithm{currentSolutionCost=883180,038, currentSolutionArray=[0, 0,
OptimalRatio: 0,96412
#####
```

Figura 43 - Output do algoritmo *Simulated Annealing*



## 5. Análise do desempenho dos algoritmos implementados

Para testar o desempenho dos algoritmos, vamos usa-los para resolver as instancias de UFLP facultadas pela professora.

### Algoritmo Genético

Para a configuração do algoritmo genético usamos de principais híper parâmetros, 100 soluções por geração, máximo de 200 gerações ou 50 segundos de tempo de compilação (verifica no inicio de cada ciclo evolutivo), com 60% de probabilidade cada uma qualquer solução não pertencendo às 6 melhores sofrer mutação a cada uma qualquer geração com 20% de probabilidade de cada um qualquer gene individual ser mutado.

Cenário	N.º Instalações	N.º Clientes	S. Ótima	S. Obtida	Rácio*	Duração (ms)
cap71.txt	16	50	932615.75	932615,750	1,00000	125
cap72.txt	16	50	977799.4	977799,400	1,00000	37
cap73.txt	16	50	1010641.45	1010641,450	1,00000	62
cap74.txt	16	50	1034976.975	1034976,975	1,00000	44
cap101.txt	25	50	796648.437	796648,438	1,00000	293
cap102.txt	25	50	854704.2	854704,200	1,00000	225
cap103.txt	25	50	893782.112	893782,113	1,00000	203
cap104.txt	25	50	928941.75	928941,750	1,00000	159
cap131.txt	50	50	793439.562	798338,450	0,99386	6953
cap132.txt	50	50	851495.325	852257,975	0,99911	6554
cap133.txt	50	50	893076.712	894095,763	0,99886	6953
cap134.txt	50	50	928941.75	928941,750	1,00000	1173
capa.txt	100	1000	1.7156454478E7	24854325,232	0,69028	50689
capb.txt	100	1000	1.2979071582E7	16225708,361	0,79991	50073
capc.txt	100	1000	1.1505594329E7	13238828,197	0,86908	50121
Kcapmo1.txt	100	100	1156.909	1187,621	0,97414	31055
Kcapmo2.txt	100	100	1227.667	1430,156	0,85841	29916
Kcapmp1.txt	200	200	2460.101	7869,138	0,31263	50234
Kcapmp2.txt	200	200	2419.325	11255,604	0,21494	50030
Kcapmq1.txt	300	300	3591.273	37482,558	0,09581	51256
Kcapmq2.txt	300	300	3543.662	41413,533	0,08557	50198
Kcapmr1.txt	500	500	2349.856	62823,139	0,03740	52068
Kcapmr2.txt	500	500	2344.757	58533,301	0,04006	54666

**Tabela 2 - Desempenho do Algoritmo Genético**

\*Rácio = (Solução Ótima/Solução Obtida)

## Algoritmo Greedy

De hiper parâmetro para configurar o algoritmo *greedy* usamos apenas o tempo máximo de compilação de 50 segundos (verificado no início de cada iteração), contudo à exceção de nos 2 últimos testes este não foi um fator limitativo, tendo o algoritmo executado enquanto a próxima escolha *greedy* resultasse numa redução do custo total.

Cenário	N.º Instalações	N.º Clientes	S. Ótima	S. Obtida	Racio	Duração (ms)
cap71.txt	16	50	932615.75	932615,750	1,00000	37
cap72.txt	16	50	977799.4	981538,850	0,99619	17
cap73.txt	16	50	1010641.45	1012476,975	0,99819	6
cap74.txt	16	50	1034976.975	1034976,975	1,00000	5
cap101.txt	25	50	796648.437	797508,725	0,99892	123
cap102.txt	25	50	854704.2	855971,750	0,99852	42
cap103.txt	25	50	893782.112	895027,188	0,99861	22
cap104.txt	25	50	928941.75	928941,750	1,00000	10
cap131.txt	50	50	793439.562	794299,850	0,99892	279
cap132.txt	50	50	851495.325	852762,875	0,99851	189
cap133.txt	50	50	893076.712	894095,763	0,99886	125
cap134.txt	50	50	928941.75	928941,750	1,00000	59
capa.txt	100	1000	1.7156454478E7	17902353,241	0,95834	6705
capb.txt	100	1000	1.2979071582E7	13131893,837	0,98836	9020
capc.txt	100	1000	1.1505594329E7	11947717,759	0,96300	13371
Kcapmo1.txt	100	100	1156.909	1208,238	0,95752	604
Kcapmo2.txt	100	100	1227.667	1277,600	0,96092	488
Kcapmp1.txt	200	200	2460.101	2474,506	0,99418	5175
Kcapmp2.txt	200	200	2419.325	2549,136	0,94908	4254
Kcapmq1.txt	300	300	3591.273	3801,989	0,94458	15815
Kcapmq2.txt	300	300	3543.662	3622,431	0,97826	16277
Kcapmr1.txt	500	500	2349.856	2678,367	0,87735	50025
Kcapmr2.txt	500	500	2344.757	2543,372	0,92191	66680

**Tabela 3 - Desempenho do Algoritmo Greedy**

## Algoritmo *Simulated Annealing*

Para a configuração do algoritmo *Simulated Annealing* usamos de hiper parâmetros uma temperatura inicial de 100000, uma taxa de arrefecimento de 0.995 (a cada ciclo a temperatura é 0.995 vezes a temperatura do ciclo anterior), uma duração máxima de 50 segundos e uma probabilidade de 20% de um qualquer elo individual sofrer uma perturbação aquando da perturbação do estado da solução.

Cenário	N.º Instalações	N.º Clientes	S. Ótima	S. Obtida	Racio	Duração (ms)
cap71.txt	16	50	932615.75	932615,750	1,00000	152
cap72.txt	16	50	977799.4	977799,400	1,00000	205
cap73.txt	16	50	1010641.45	1012476,975	0,99819	278
cap74.txt	16	50	1034976.975	1034976,975	1,00000	154
cap101.txt	25	50	796648.437	798325,938	0,99790	494
cap102.txt	25	50	854704.2	855781,100	0,99874	409
cap103.txt	25	50	893782.112	898149,250	0,99514	322
cap104.txt	25	50	928941.75	935671,775	0,99281	315
cap131.txt	50	50	793439.562	814209,900	0,97449	1047
cap132.txt	50	50	851495.325	892430,263	0,95413	908
cap133.txt	50	50	893076.712	938982,613	0,95111	862
cap134.txt	50	50	928941.75	1023957,525	0,90721	815
capa.txt	100	1000	1.7156454478E7	44036771,894	0,38959	41273
capb.txt	100	1000	1.2979071582E7	22255515,118	0,58318	42917
capc.txt	100	1000	1.1505594329E7	17012160,213	0,67632	42761
Kcapmo1.txt	100	100	1156.909	3974,212	0,29110	5346
Kcapmo2.txt	100	100	1227.667	4607,490	0,26645	4902
Kcapmp1.txt	200	200	2460.101	19428,476	0,12662	22450
Kcapmp2.txt	200	200	2419.325	21894,242	0,11050	22862
Kcapmq1.txt	300	300	3591.273	55980,845	0,06415	50012
Kcapmq2.txt	300	300	3543.662	56170,975	0,06309	50014
Kcapmr1.txt	500	500	2349.856	78300,722	0,03001	50062
Kcapmr2.txt	500	500	2344.757	69912,912	0,03354	50005

**Tabela 4 - Desempenho do Algoritmo *Simulated Annealing***

## **Análise dos resultados**

Como podemos observar nos resultados à medida que o número de instalações e clientes aumenta, a qualidade das soluções no algoritmo genético e *simulated annealing* tende a diminuir. Isto deve-se em parte ao aumento do espaço de solução do problema e em parte também ao elevado custo do cálculo das funções de avaliação, isto é, para cada solução, para cada  $j$  cliente são necessários comparar  $i$  (número de instalações) custos de alocação ( $O(s \cdot j \cdot i)$ ) o que resulta num incremento bastante significativo nos recursos necessários para obter uma boa solução.

Assim, nestas circunstâncias para obter uma melhor solução usando estes algoritmos é necessário mais tempo e gerações no caso do algoritmo genético e mais tempo e uma taxa de arrefecimento mais baixa no caso do *simulated annealing*.

Já o algoritmo *greedy* surpreendeu muito pela positiva uma vez que com os mesmos recursos computacionais conseguiu obter soluções bastante superiores.

Para concluir, embora os algoritmos genéticos e *simulated annealing* tenham uma natureza mais robusta e tendam a encontrar soluções melhores em problemas complexos devido à sua capacidade de explorar amplamente o espaço de soluções, em várias situações o algoritmo *greedy* pode ser preferível uma vez que permite encontrar boas soluções rapidamente com menos recursos computacionais.

## 6. Conclusões e Trabalho Futuro

Neste trabalho foi-me proposta a implementação de algoritmos heurísticos e o seu uso para a resolução do famoso problema de localização de instalações sem restrições de capacidade.

Pessoalmente creio que consegui cumprir os objetivos do trabalho com bastante sucesso tendo implementado 3 algoritmos e feito uso dos mesmos. Tive especial atenção ao encapsulamento, abstração e extenso uso de comentários no código aquando do desenvolvimento dos mesmos permitindo assim que estes mesmos algoritmos possam ser facilmente reutilizados e adaptados a outros problemas no futuro. Uma possível melhoria, mas que não considere fazer parte do âmbito do projeto, foi colocar os algoritmos a suportar execução paralela fazendo uso de diferentes *threads*, permitindo assim um tempo de compilação e por consequência tempo de resolução dos problemas mais curto.

Apesar de alguns percalços ao longo do desenvolvimento (nomeadamente a recolher a informação dos ficheiros facultados), estou bastante satisfeito com o resultado final já que vejo a imensa aplicabilidade dos algoritmos e técnicas de otimização estudados e desenvolvidos.

Para trabalho futuro, pretendo estudar o algoritmo da colónia de formigas que embora não implementado neste trabalho, li a descrição do seu comportamento e o mesmo me cativou bastante.

## Referências WWW

[01] <https://antigo.moodle2.estg.ipp.pt/>

Aqui podemos encontrar os slides da disciplina de Análise de Algoritmos e Otimização onde uma descrição aos algoritmos desenvolvidos e do seu comportamento é feita.

[02] <https://www.youtube.com/>

Aqui podemos encontrar vídeos visualmente ilustrativos que podem ajudar a compreender o funcionamento e implementação dos algoritmos.

[03] <https://chatgpt.com/>

Aqui podemos encontrar explicações personalizadas bem como rápida informação sobre temas específicos. Muito útil para rapidamente mitigar pequenas falhas nos nossos conhecimentos que nos possam estar a impedir de progredir.