Orientações para Entrega do Código

ELE634 – Laboratório de Sistemas II

André Costa Batista

27 de outubro de 2025

1 Introdução

Este documento estabelece o padrão que deve ser seguido por todos os grupos para a entrega da implementação de seus algoritmos de otimização. O objetivo é garantir que todas as implementações possam ser executadas de forma padronizada e comparadas adequadamente.

2 Critério de Parada

O critério de parada adotado para todos os algoritmos será o **número de avaliações da função objetivo**. Isso garante uma comparação justa entre diferentes abordagens. Uma avaliação da função objetivo é uma execução da função do seu código que calcula o valor da função objetivo para uma solução candidata. Então, toda vez que seu código chamar essa função, deve ser contado como uma avaliação. O algoritmo deve parar quando atingir o número máximo de avaliações especificado e retornar a melhor solução encontrada até então. Eu vou ler o código das implementações para verificar se esse critério está sendo respeitado corretamente.

Quem for utilizar o Gurobi como forma de busca local, deve somar ao número de avaliações os seguintes parâmetros do objeto de modelo do Gurobi: `NodeCount` e `IterCount`. Para controlar uma execução do Gurobi de modo que respeite um número máximo para esses parâmetros, você pode utilizar o seguinte código:

```
from gurobipy import Model, GRB
model = Model()

# Configurar o modelo aqui

# Definir limites maximos
model.setParam(GRB.Param.NodeLimit, max_node_count)
model.setParam(GRB.Param.IterationLimit, max_iter_count)

# Otimizar
model.optimize()

# Apos a otimizacao, recuperar os valores usados
nos_explorados = modelo.NodeCount
iteracoes_usadas = modelo.IterCount

```

```python
# Calcular total de avaliacoes equivalentes
total_avaliacoes = nos_explorados + iteracoes_usadas
```

3 Estrutura da Entrega

3.1 Formato do Arquivo

1. Cada grupo deve entregar sua implementação em uma **pasta zipada**.
2. O arquivo ZIP deve ter o nome do grupo, por exemplo: `reginaldorossi.zip`
3. Dentro da pasta descompactada, deve existir um arquivo Python com o **nome do grupo**, por exemplo: `reginaldorossi.py`
4. Este arquivo principal deve conter um método chamado `resolva` que executa o algoritmo implementado.
5. A pasta pode conter outros arquivos auxiliares (módulos, classes, funções auxiliares, etc.), mas a **interface de execução** será sempre o método `resolva` do arquivo principal.

3.2 Exemplo de Estrutura

Estrutura de Diretórios

```
reginaldorossi.zip
|-- reginaldorossi.py        # Arquivo principal (obrigatorio)
|-- heuristicas.py           # Modulo auxiliar (opcional)
|-- operadores.py            # Modulo auxiliar (opcional)
|-- utils.py                 # Utilitarios (opcional)
`-- README.md                # Opcional
```

4 Especificação do Método `resolva`

4.1 Assinatura da Função

O método `resolva` deve ter a seguinte assinatura:

```python
def resolva(dados: Dados, numero_avaliacoes: int) -> Solucao:
    """
    Executa o algoritmo de otimizacao.

    Parametros:
    -----------
    dados : Dados
        Objeto contendo os dados da instancia do problema
        (requisicoes, onibus, janelas de tempo, etc.)

    numero_avaliacoes : int
        Numero maximo de avaliacoes da funcao objetivo permitidas

    Retorna:
```

```
    --------
    Solucao
        Objeto contendo:
        - Rotas de cada viagem de cada onibus
        - Instante de chegada de cada requisicao
        - Valor da funcao objetivo
    """
    # Implementacao do algoritmo aqui
    pass
```

4.2 Parâmetros de Entrada

`dados` Objeto da classe `Dados`[1] contendo todos os parâmetro de uma determinada instância do problema, conforme estão a modelagem.

`numero_avaliacoes` Valor inteiro que define o critério de parada. O algoritmo deve parar quando atingir este número de avaliações da função objetivo.

4.3 Valor de Retorno

O método deve retornar um objeto da classe `Solucao`[2] contendo:

- **Rotas**: Sequência de requisições atendidas por cada viagem de cada ônibus.
- **Tempos de chegada**: Instante em que cada ônibus chega em cada requisição de cada viagem.
- **Função objetivo**: Valor calculado da função objetivo.

4.4 Estrutura dos Atributos da Classe `Solucao`

A classe `Solucao` possui três atributos principais que devem ser preenchidos:

`rota` Dicionário hierárquico indexado por ônibus e viagem: `rota[k][v]` retorna uma lista de requisições visitadas pelo ônibus `k` na viagem `v`. Cada rota **sempre inicia e termina na garagem** (índice 0).

`chegada` Dicionário hierárquico indexado por ônibus e viagem: `chegada[k][v]` retorna uma lista de tempos (float) correspondentes aos instantes de chegada em cada ponto da rota.

`fx` Valor da função objetivo (float) calculado para a solução.

Veja a seguir um exemplo ilustrativo de como preencher esses atributos. Considere uma solução para a instância `pequena.json` com 14 requisições, 3 ônibus e até 4 viagens por ônibus:

[1] A mesma que está presente no arquivo `dados.py` no repositório da disciplina.
[2] A mesma que está no arquivo `solucao.py` do repositório da disciplina.

```python
from solucao import Solucao

# Criar objeto de solucao
solucao = Solucao()

# Configurar rotas
# Onibus 1 realiza 3 viagens
solucao.rota[1] = {}
solucao.rota[1][1] = [0, 2, 4, 0]      # Viagem 1: garagem -> req 2 -> req 4 -> garagem
solucao.rota[1][2] = [0, 6, 8, 11, 0] # Viagem 2: garagem -> req 6 -> req 8 -> req 11 -> garagem
solucao.rota[1][3] = [0, 13, 0]        # Viagem 3: garagem -> req 13 -> garagem
solucao.rota[1][4] = []                # Viagem 4: nao utilizada

# Onibus 2 realiza 2 viagem
solucao.rota[2] = {}
solucao.rota[2][1] = [0, 3, 0]          # Viagem 1: garagem -> req 3 -> garagem
solucao.rota[2][2] = [0, 7, 9, 10, 0]  # Viagem 2: garagem -> req 7 -> req 9 -> req 10 -> garagem
solucao.rota[2][3] = []                 # Viagem 3: nao utilizada
solucao.rota[2][4] = []                 # Viagem 4: nao utilizada

# Onibus 3 realiza 4 viagens
solucao.rota[3] = {}
solucao.rota[3][1] = [0, 1, 0]   # Viagem 1: garagem -> req 1 -> garagem
solucao.rota[3][2] = [0, 5, 0]   # Viagem 2: garagem -> req 5 -> garagem
solucao.rota[3][3] = [0, 12, 0]  # Viagem 3: garagem -> req 12 -> garagem
solucao.rota[3][4] = [0, 14, 0]  # Viagem 4: garagem -> req 14 -> garagem

# Configurar tempos de chegada (correspondentes as rotas)
solucao.chegada[1] = {}
solucao.chegada[1][1] = [3.89, 17.0, 31.08, 80.2]          # Tempos para rota[1][1]
solucao.chegada[1][2] = [80.2, 92.0, 132.5, 185.0, 200.2]  # Tempos para rota[1][2]
solucao.chegada[1][3] = [202.12, 215.0, 223.0]             # Tempos para rota[1][3]
solucao.chegada[1][4] = []                                 # Viagem nao utilizada

solucao.chegada[2] = {}
solucao.chegada[2][1] = [13.89, 27.0, 80.2]               # Tempos para rota[2][1]
solucao.chegada[2][2] = [80.2, 92.0, 132.5, 175.0, 200.2] # Tempos para rota[2][2]
solucao.chegada[2][3] = []                                 # Viagem nao utilizada
solucao.chegada[2][4] = []                                # Viagem nao utilizada

solucao.chegada[3] = {}
solucao.chegada[3][1] = [1.7, 14.81, 23.89]      # Tempos para rota [3][1]
```

```
solucao.chegada[3][2] = [23.89, 37.0, 45.97]      # Tempos para rota
    [3][2]
solucao.chegada[3][3] = [182.12, 195.0, 204.13]   # Tempos para rota
    [3][3]
solucao.chegada[3][4] = [204.13, 217.01, 224.18] # Tempos para rota
    [3][4]

# Configurar funcao objetivo
solucao.fx = 33486.4
```

Pontos Importantes

- **Índice 0 é a garagem**: Toda rota deve começar e terminar com 0
- **Correspondência**: O tamanho de `rota[k][v]` e `chegada[k][v]` deve ser **exatamente o mesmo**
- **Viagens não utilizadas**: Representadas por listas vazias `[]`
- **Saída da garagem (primeiro valor de chegada)**: Instante em que o ônibus começa o preparo para a viagem. Ou seja, é igual ao instante em que o ônibus chega na primeira requisição da viagem menos o tempo de deslocamento da garagem até essa requisição e menos o tempo de preparo inicial (que o tempo de serviço relativo à garagem).
- **Chegada na garagem (último valor de chegada)**: Instante em que o ônibus retorna à garagem após a última requisição da viagem.
- **Todos os ônibus e viagens**: Mesmo que não utilizados, devem ter entradas no dicionário
- **IMPORTANTE**: Dentro da execução do seu algoritmo, você pode utilizar a estrutura de dados que quiser. A estrutura apresentada aqui é apenas para a **saída final** do método `resolva` através do objeto `Solucao`.

4.5 Número de avaliações por instância

A regra que vou adotar para definir o número máximo de avaliações (N_{av}^{max}) da função objetivo por instância é a seguinte:

$$N_{av}^{max} = 10 \times n \times K \times r \tag{1}$$

onde, conforme a modelagem, n é o número de requisições, K é o número de ônibus disponíveis, e r é o número máximo de viagens que cada ônibus pode realizar. A tabela abaixo apresenta os valores de N_{av}^{max} para cada instância fornecida:

Número Máximo de Avaliações por Instância

Instância	n (requisições)	K (ônibus)	r (viagens/ônibus)	N_{av}^{max}
pequena	14	3	5	2100
média	67	6	12	48240
grande	108	11	10	118800
rush	108	11	10	118800

5 Validação e Avaliação

5.1 Processo de Validação

Após a execução de cada algoritmo, será executada uma função de validação que verificará:

1. **Viabilidade da solução**: Nenhuma restrição do problema foi violada
2. **Cálculo correto da função objetivo**: Conferência do valor reportado

Atenção

Uma solução final de cada execução que violar qualquer restrição será considerada **infactível** e será descartada da amostra dos resultados.

5.2 Protocolo de Execução para Comparação

1. Cada algoritmo será executado em cada instância **30 vezes**
2. Cada chamada da função `resolva` executa o algoritmo **uma única vez**
3. A comparação estatística entre os algoritmos será feita com base na média (ou mediana) dos valores da função objetivo obtidos nas 30 execuções (considerando apenas soluções viáveis). Gráficos de boxplot também serão apresentados.

6 Recomendações

- **Controle rigoroso do número de avaliações**: Certifique-se de que seu algoritmo respeita exatamente o limite especificado
- **Tratamento de erros**: Implemente tratamento adequado de exceções para evitar crashes
- **Documentação**: Comente seu código adequadamente, especialmente o método `resolva`
- **Teste local**: Teste sua implementação com as instâncias fornecidas antes da entrega