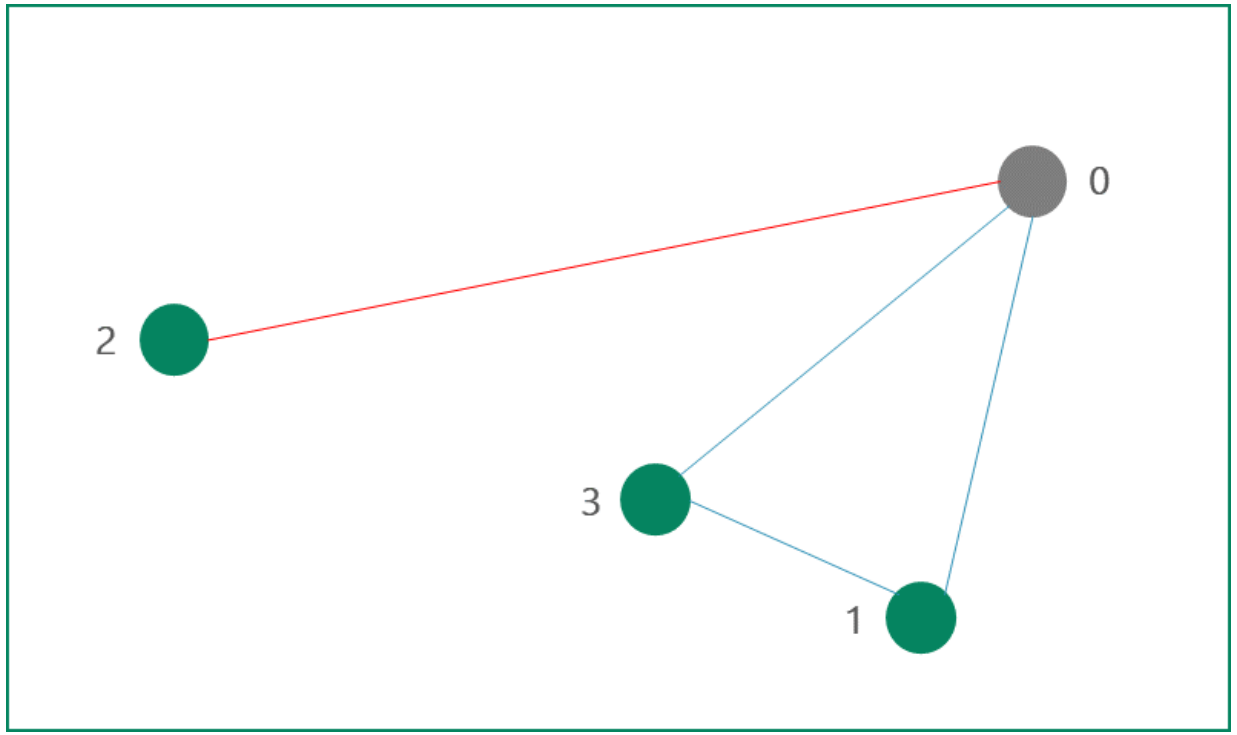
Vehicle Routing Problem

Grupo 2:
- Ana Pinto - 97168
- Ana Sofia Caetano - 96326
- Afonso Miao - 78167
- Afonso Meireles – 96018

Inteligência Computacional e Otimização
Profª Rosário Laureano
Profº Vítor Basto Fernandes

27 de Maio de 2021



O Grupo

Afonso Miao
MEI

Ana Pinto
MCD

Ana Sofia Caetano
MIG

Afonso Meireles
MSIAD

Índice

- Mockup inicial
- Interface
- Processamento no Backend
- Criação das soluções
- Avaliação das soluções
- Seleção das soluções

Mockup da interface: Vehicle Routing Problem

Centro de Fornecimento:

Long./Lat.: 9.87543 39.5471

Pontos de Entrega

1	Long./Lat.:	9.7590	38.1209	Carga: 15	Prioridade:	
2	Long./Lat.:	8.9656	39.0123	Carga: 25	Prioridade:	
3	Long./Lat.:	9.5231	38.3487	Carga: 30	Prioridade:	

+

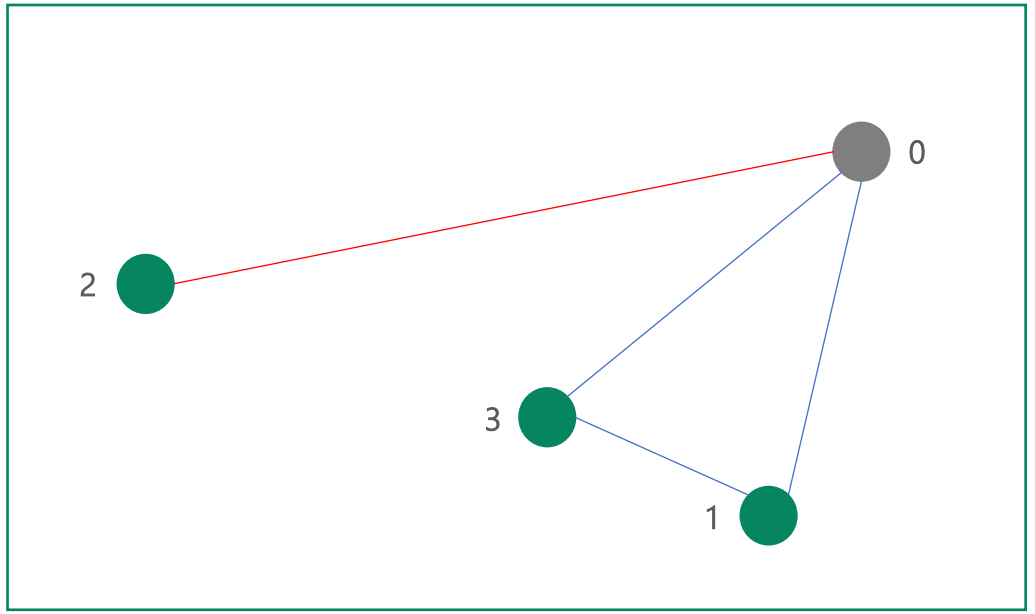
Veículos

	Capacidade:	Consumo:
Veículo A:	100	1.5
Veículo B:	80	1.8

Otimizar:

- ☐ Distância Percorrida
- ☒ Veículos Utilizados
- ☒ Custo
- ☐ Duração ao último ponto de entrega

Calcular



Veículo A: 0 – 2 – 0

Veículo B: 0 – 1 – 3 – 0

Definição dos objetivos, restrições e inputs do user

Objetivos:
- Minimizar distância percorrida
- Minimizar veículos utilizados
- Minimizar custo
- Minimizar duração ao ultimo cliente

Restrições:
- Número de veículos disponíveis
- Capacidade de cada veículo
- Só 1 visita por cliente
- Carga a entregar por cliente
- Prioridade entre clientes (facultativo)
- Rotas começam e acabam no Cen. Fornecimento

Inputs do User:
- Coordenadas do Centro de Fornecimento (Nó 0)
- Coordenadas dos Clientes (Nós 1, 2, 3, ….)
- Requisitos de cada cliente (facultativo)
- Prioridade entre clientes (facultativo)
- Número de veículos disponíveis
- Capacidade de cada veículo (facultativo)
- Consumo / Custo/KM de cada veículo (facultativo)

Matriz de Distâncias / Matriz de Durações

	0	1	2	3	4	5
0	0	1.5	3.3	2.2	0.5	1.4
1	2.1	0	0.2	1.1	1	2
2	3.3	2	0	4.5	9	3
3	1.1	5	5.4	0	8.1	0.2
4	3.9	7.2	6.8	11	0	14
5	5.3	2.3	4.9	3.2	6	0

Requisitos de cada cliente

1	2	3	4	5
15	24	32	54	90

Capacidade de cada veículo

A	B	C
100	180	150

Custo/KM de cada veículo

A	B	C
1.5	1.2	1.8

Algoritmo Genético

Soluções: Cromossomas em que cada gene é um nó, cada cromossoma representa uma rota
Terminam e começam em 0

					Distância	Custo	Carga
A	0	1	0		1.5+2.1 = 3.6	3.6*1.5 = 5.4	15 / 100
B	0	5	4	0	1.4+6+3.9 = 11.3	11.3*1.2 = 13.56	144 / 180
C	0	2	3	0	3.3+4.5+1.1 = 8.9	8.9*1.8 = 16.02	56 / 150
					Total = 23.8	Total = 34.98	

Duração do último cliente:

Veículo A: 01 - 1.5h
Veículo B: 054 - 7.4h
Veículo C: 023 - 7.8h -> este é o valor mais elevado que se quer minimizar

Interface

- Inputs do User:

- Coordenadas do Centro de Fornecimento (Nó 0)
- Coordenadas dos Clientes (Nós 1, 2, 3, ….)
- Requisitos de cada cliente (carga)
- Número de veículos disponíveis
- Capacidade de cada veículo
- Consumo / Custo/KM de cada veículo

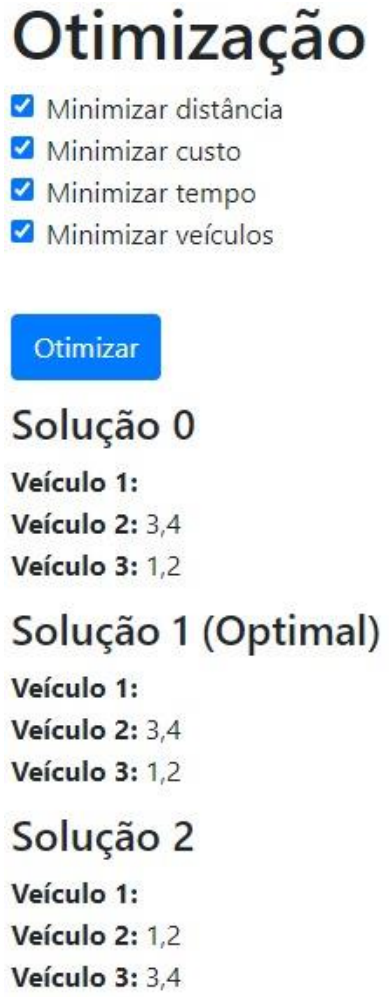
- API OpenRouteService

- Matriz de distâncias reais, com base nos percursos de estrada
- Matriz de tempo

Análise do CVRP – Solução atual

Pressupostos iniciais:

- Existe apenas 1 centro de Fornecimento
- Todos os veiculos começam e regressam ao centro de Fornecimento
- 4 Objetivos:
  - Minimizar a distância percorrida;
  - Minimizar o custo;
  - Minimizar o tempo;
  - Minimizar o número de veículos.

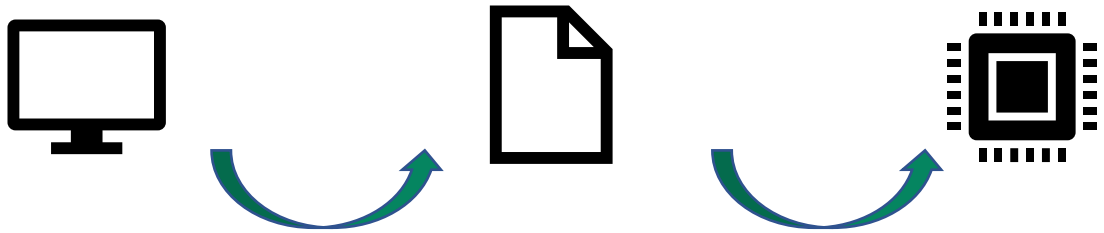


Processamento no Backend

Interface: JavaScript – React
Backend: Django
Integração com jMetalPy

Algoritmo

- Algoritmo Genético para *Single-Objective*
- NSGAII para multiobjetivo



interface json jMetalPy


```
{
    "optimization": [
        "1",
        "1",
        "0",
        "0"
    ],
    "data_vehicles": [
        {
            "id": "2",
            "capacity": "100",
            "cost": "0.5"
        },
        {
            "id": "3",
            "capacity": "50",
            "cost": "0.6"
        },
        {
            "id": "1",
            "capacity": "30",
            "cost": "1.5"
        }
    ],
    "data_nodes": [
        {
            "node": 0,
            "city": "Lisboa",
            "latitude": "38.7452",
            "longitude": "-9.1604",
            "depot": "true",
            "demand": "0"
        },
        {
            "node": 1,
            "city": "Porto",
            "latitude": "41.1495",
            "longitude": "-8.6108",
            "depot": "false",
            "demand": "10"
        },
        {
            "node": 2,
            "city": "Braga",
            "latitude": "41.5333",
            "longitude": "-8.4167",
            "depot": "false",
            "demand": "20"
```

Criação das soluções – single-objective TSP

TSP:

- Variável: número de cidades (nós);
- Objetivo: minimizar a distância.

```
def create_solution(self) -> PermutationSolution:
    new_solution = PermutationSolution(number_of_variables=self.number_of_variables,
                                       number_of_objectives=self.number_of_objectives)
    new_solution.variables = self.depot + random.sample(range(1,self.number_of_variables), k=self.number_of_variables-1)
    return new_solution
```

Criação da solução – multiobjetivo/single-objective

[3 2 -1 1 5 4] ⟶ Separamos as sub-rotas pelos negativos (-1, -2, -3…) consoante o número de veículos [3 2 -1 1 5 4] ⟶ [1 2 -1 3 5 4]

Veículo 1 Veículo 2

- **Sub-rota:** rota de cada veículo
- **Número de carros:** consoante o número de carros escolhidos pelo utilizador, inserimos no *array* números negativos

```
def create_solution(self) -> PermutationSolution:
    new_solution = PermutationSolution(number_of_variables=self.number_of_variables,
                                       number_of_objectives=self.number_of_objectives)

    random_nodes = random.sample(range(1,self.number_of_variables+1), k=self.number_of_variables)
    cars_to_add = self.num_vehicles - 1
    random_cars = random.sample(range(1, self.num_vehicles+1), k=cars_to_add)
    for i in range(cars_to_add):
        dummy_node = i + 1
        random_nodes.append(-dummy_node)

    random.shuffle(random_nodes)

    new_solution.variables = random_nodes
    return new_solution
```

Criação da solução – multiobjetivo/singleobjective

Algoritmo NSGA-II:

- População inicial: número de objetivos x número de veículos x pontos de entrega x 10
- Iterações: número de veículos x pontos de entrega x 1000
- Mutação: PermutationSwapMutation (Prob = 0.20)
- Crossover: PMXCrossover (Prob = 0.90)

```
max_evaluations = len(data['data_vehicles']) * len(data['data_nodes']) * 1000
print("Number of evaluations: ", max_evaluations)
dimension = number_of_objectives * len(data['data_vehicles']) * len(data['data_nodes']) * 10
print("population_size: ", dimension)
algorithm = NSGAII(
    problem=problem,
    population_size=dimension,
    offspring_population_size=dimension,
    mutation=PermutationSwapMutation(probability=0.2),
    crossover=PMXCrossover(probability=0.9),
    termination_criterion = StoppingByEvaluations(max_evaluations=max_evaluations)
)
```

Avaliação das soluções

- Fitness 1: variável que faz o somatório das distâncias
- Fitness 2: variável que faz o somatório do custo
- Fitness 3: variável correspondente à rota com o maior tempo
- Fitness 4: variável que faz o somatório de veículos que possuem rotas

```
fitness1 = 0 #distance
fitness2 = 0 #cost
fitness3 = 0 #tempo
fitness4 = 0 #veiculos

matrix_route = []
sub_route = []

#Creating matrix with subroutes of each vehicle
for i in range(len(solution.variables)):
    node = solution.variables[i]
    if i == 0 and node < 0:
        matrix_route.append([])
    elif node > 0: # nó positivo --> append subroute
        #print("Appending to sub_route: ", node)
        sub_route.append(node)
    else: # nó negativo --> append matrix
        #print("Appending subroute: ", sub_route)
        matrix_route.append(sub_route)
        sub_route = []
```

Matriz com a rota de cada veículo

```
#[
# [1,2,3]
# [5,6,8]
# []
#]
```

Avaliação das soluções

Distância:

- Calculamos a distância perocorrida em cada sub-rota e somamos o total;
- Ao primeiro e último nó soma-se a distância ao centro de fornecimento;

Custo:

- Distância percorrida em cada sub-rota multiplicada pelo custo (€/Km) do veículo que a percorre

Tempo:

- Da mesma forma que se calculam as distâncias de cada sub-rota são calculadas as durações e preserva-se a sub-rota com a maior duração.

Número de veículos:

- Contabilizam-se as sub-rotas que não estão vazias na solução (número de veículos utilizados)

```
for i in range(len(sub_route) - 1):
    x = sub_route[i]
    y = sub_route[i + 1]

    if i == 0:
        distance_to_warehouse = self.distance_to_warehouse[x-1]
        fitness1 += distance_to_warehouse
        fitness2 += distance_to_warehouse * self.vehicle_costs[car]
        time_compare += self.times_to_warehouse[x-1]
        route_demand += self.demand_section[x-1]
        #Changes here
        if route_demand > self.vehicle_capacities[car]:
            fitness1 += 99999999
            fitness2 += 99999999
            fitness3 += 99999999
        continue

    route_demand = self.demand_section[y-1]
    if route_demand > self.vehicle_capacities[car]:
            fitness1 += 99999999
            fitness2 += 99999999
            fitness3 += 99999999
            continue
    fitness1 += self.distance_matrix[x][y]
    fitness2 += self.distance_matrix[x][y] * self.vehicle_costs[car]
    time_compare += self.time_matrix[x][y]

#Soma nó final até à warehouse
fitness1 += self.distance_to_warehouse[y-1]
fitness2 += self.distance_to_warehouse[y-1] * self.vehicle_costs[car]
time_compare += self.times_to_warehouse[y-1]

if time_compare > fitness3:
    fitness3 = time_compare
```

Seleção das soluções

```
array_index = calculate_solutions_more_2_objectives(front)
for index in array_index:
    solutions_to_pass.append(front[index].variables)
```

```
def calculate_solutions_more_2_objectives(objectives_array: list):
    array_pd = []
    for solution in objectives_array:
        array_pd.append(solution.objectives)
    df = pd.DataFrame(array_pd)
    result = df.idxmin(axis=0, skipna=True)

    solution_index_each_objective = result.tolist()

    results_array = []
    for objective_array in objectives_array:
        total = 0
        for objective in objective_array.objectives:
            total += objective**2
        results_array.append(math.sqrt(total))

    front_pareto_index = results_array.index(min(results_array))
    solution_index_each_objective.append(front_pareto_index)
    print(solution_index_each_objective)


    print("Indexes of best solution each objective + frontpareto: ", solution_index_each_objective)
    return solution_index_each_objective
```

Soluções para *single-objective*:

- Apresenta-se a solução ótima para o objetivo selecionado

Soluções para *multiobjetivo:*

- Apresenta-se uma solução por objetivo (a solução que minimiza esse objetivo na frente de pareto) + a solução mais equilibrada, que minimiza a fórmula:

$$\sqrt{\boldsymbol{Objetivo\ 1}^{\,2} + \boldsymbol{Objetivo\ 2}^{\,2} + \boldsymbol{Objetivo\ 3}^{\,2} + \ \ldots}$$

```
def calculate_pareto(objectives_array: list):
    print("Calculating front pareto")

    results_array = []
    for objective_array in objectives_array:
        total = 0
        for objective in objective_array.objectives:
            total += objective**2
        results_array.append(math.sqrt(total))

    front_pareto_index = results_array.index(min(results_array))
    print("Front pareto index: ", front_pareto_index)
    return front_pareto_index
```

Apresentação da Interface