# Problema do Transporte - Rotas de ônibus

December 13, 2023

Aluno: Gustavo Becelli do Nacimento

## 1 Problema

A programação linear desempenha um papel importante na pesquisa operacional, sendo uma ferramenta valiosa para a otimização da alocação de recursos em diversas áreas, incluindo o transporte público. Neste estudo, nosso foco se concentra em aprimorar a eficiência do transporte público por meio da aplicação do Problema do Transporte, que é um problema de programação linear.

Os principais pontos abordados são os seguintes:

- O objetivo principal é otimizar a alocação de recursos em rotas de transporte público, visando à minimização de custos.
- Para atingir esse objetivo, considera-se vários fatores, incluindo as rotas disponíveis, suas capacidades, a demanda em cada bairro e os custos operacionais associados.
- A meta é equilibrar a distribuição de recursos de forma a atender à demanda de maneira eficiente, ao mesmo tempo em que se reduz os custos operacionais.
- Isso se mostra particularmente relevante no contexto do planejamento do transporte público, onde a necessidade é fornecer serviços eficazes dentro de um orçamento limitado.

Neste estudo, trabalha-se com a suposição de que a demanda pode ser atendida pelas rotas disponíveis, mas reconhece-se que a realidade pode variar. Abaixo são fornecidas informações sobre as rotas disponíveis, suas capacidades, os bairros atendidos, as demandas específicas e os custos operacionais estimados por passageiro.

## 2 Parâmetros

**Rotas Disponíveis e Capacidades:** - Linha Central: Capacidade de 1250 passageiros por dia. - Rota do Parque: Capacidade de 1550 passageiros por dia. - Via Expressa: Capacidade de 1400 passageiros por dia.

**Bairros Atendidos e Demandas:** - Centro: Demanda de 1200 passageiros por dia. - Parque das Flores: Demanda de 950 passageiros por dia. - Vila Nova: Demanda de 750 passageiros por dia. - Jardim Oceânico: Demanda de 1300 passageiros por dia.

**Custos Operacionais por Passageiro (estimativa em reais):** A tabela abaixo apresenta os custos operacionais estimados por passageiro para cada rota e bairro em reais (R$).

| Rota | Centro | Parque das Flores | Vila Nova | Jardim Oceânico |
|---|---|---|---|---|
| Linha Central | 3,50 | 4,00 | 3,75 | 4,50 |
| Rota do Parque | 3,00 | 3,50 | 4,00 | 3,60 |
| Via Expressa | 3,20 | 4,20 | 3,90 | 4,10 |

Esses dados serão utilizados para a resolução do problema de transporte, que visa alocar recursos de forma eficiente para atender à demanda, reduzindo custos operacionais. Para isso, é necessário considerar os parâmetros mencionados acima, que representam as limitações do problema.

## 3 Solução

[ ]:
```python
import numpy as np
import pandas as pd
from typing import Literal

PENALTY_COST = 1e12  # A demanda alta é usada para marcar as linhas ou colunas que já foram alocadas.


class TransportationOptimizerHelper:
    """
    Uma classe para otimizar o problema do transporte.

    Esta classe usa uma abordagem baseada em custos para alocar as capacidades de produção de forma a atender às demandas com custos mínimos. Ela funciona com base no princípio do Método de Aproximação de Vogel (VAM).
    """

    def __init__(
        self, production: np.ndarray, demand: np.ndarray, cost_matrix: np.ndarray
    ):
        """
        Inicializa o TransportationOptimizerHelper com capacidades de produção, demanda e matriz de custos.

        Args:
            production (np.ndarray): Capacidades de produção para cada fornecedor.
            demand (np.ndarray): Demanda requerida para cada consumidor.
            cost_matrix (np.ndarray): Custo de transporte de cada fornecedor para cada consumidor.
        """
```

```
        self._production = production
        self._demand = demand
        self._cost_matrix = cost_matrix
        self._total_cost = 0
        self._allocation_matrix = np.zeros(cost_matrix.shape)

    def optimize(self) -> tuple[float, pd.DataFrame]:
        """
        Otimiza o plano de transporte para minimizar o custo total.

        Ele aloca iterativamente o suprimento para a demanda com base na␣
↪abordagem de menor custo até que
        toda a demanda e suprimento sejam alocados.

        Returns:
            Uma tupla contendo o custo total e um DataFrame Pandas␣
↪representando a matriz de alocação.
        """
        if np.sum(self._production) != np.sum(self._demand):
            raise ValueError("Production and demand must be equal.")

        while np.sum(self._production) > 0 and np.sum(self._demand) > 0:
            self._process_cost_allocation()

        return self._total_cost, pd.DataFrame(self._allocation_matrix)

    def _process_cost_allocation(self):
        """
        Processa uma única alocação de suprimento para demanda com base na␣
↪diferença de custo.

        Ele determina se a alocação deve ser feita por linha ou coluna com base␣
↪na maior diferença de custo.
        """
        row_diff, col_diff = self._calculate_cost_difference(self._cost_matrix)
        max_row_diff_index = np.argmax(row_diff)
        max_col_diff_index = np.argmax(col_diff)

        if row_diff[max_row_diff_index] >= col_diff[max_col_diff_index]:
            self._allocate("row", max_row_diff_index)
        else:
            self._allocate("column", max_col_diff_index)

    def _calculate_cost_difference(
        self, matrix: np.ndarray
    ) -> tuple[np.ndarray, np.ndarray]:
        """
```

```
        Calcula a diferença de custo para cada linha e coluna na matriz de␣
↪custos.

        Args:
            matrix (np.ndarray): A matriz de custos.

        Returns:
            Uma tupla de arrays contendo as diferenças de custo para as linhas␣
↪e colunas.
        """
        row_diff = np.array([np.diff(np.partition(row, 1)[:2]) for row in␣
↪matrix])
        col_diff = np.array([np.diff(np.partition(col, 1)[:2]) for col in␣
↪matrix.T])
        return row_diff, col_diff

    def _allocate(self, axis: Literal["row", "column"], index: int):
        """
        Aloca o suprimento para a demanda, seja por linha ou coluna.

        Args:
            axis (Literal["row", "column"]): O eixo a ser alocado.
            index (int): O índice da linha ou coluna a ser alocada.
        """

        if axis == "row":
            self._allocate_row(index)
        else:
            self._allocate_column(index)

    def _allocate_row(self, row_index: int):
        """
        Aloca o suprimento para a demanda de uma linha específica.

        Args:
            row_index (int): O índice da linha a ser alocada.
        """
        min_cost_index = np.argmin(self._cost_matrix[row_index])
        allocated_amount = min(self._production[row_index], self.
↪_demand[min_cost_index])
        self._update_costs_and_allocation(row_index, min_cost_index,␣
↪allocated_amount)

    def _allocate_column(self, col_index: int):
        """
        Aloca o suprimento para a demanda de uma coluna específica.
```

```
        Args:
            col_index (int): O índice da coluna a ser alocada.
        """
        min_cost_index = np.argmin(self._cost_matrix[:, col_index])
        allocated_amount = min(self._production[min_cost_index], self.
↪_demand[col_index])
        self._update_costs_and_allocation(min_cost_index, col_index,␣
↪allocated_amount)

    def _update_costs_and_allocation(
        self, row_index: int, col_index: int, amount: float
    ):
        """
        Atualiza o custo total, produção e demanda após uma alocação.

        Args:
            row_index (int): O índice da linha onde a alocação é feita.
            col_index (int): O índice da coluna onde a alocação é feita.
            amount (float): A quantidade alocada.
        """
        self._total_cost += self._cost_matrix[row_index, col_index] * amount
        self._production[row_index] -= amount
        self._demand[col_index] -= amount
        self._allocation_matrix[row_index, col_index] += amount

        self._update_cost_matrix(row_index, col_index)

    def _update_cost_matrix(self, row_index: int, col_index: int):
        """
        Atualiza a matriz de custos após uma alocação.
        Args:
            row_index (int): O índice da linha onde a alocação é feita.
            col_index (int): O índice da coluna onde a alocação é feita.
        """
        if self._production[row_index] == 0:
            self._cost_matrix[row_index, :] = PENALTY_COST
        if self._demand[col_index] == 0:
            self._cost_matrix[:, col_index] = PENALTY_COST


class TransportationOptimizer:
    def __init__(self, supply, demand, costs):
        self.supply = supply
        self._demand = demand
        self.costs = costs
```

```
    def optimize(self) -> tuple[pd.DataFrame, float]:
        supply = np.array(list(self.supply.values()))
        demand = np.array(list(self._demand.values()))
        costs = np.array([list(self.costs[i].values()) for i in self.costs.
 ↪keys()])
        total_cost, df = TransportationOptimizerHelper(supply, demand, costs).
 ↪optimize()
        df.columns = self._demand.keys()
        df.index = self.supply.keys()
        return df, total_cost
```

[ ]:
```
supply = {
    "Linha Central": 1250,
    "Rota do Parque": 1550,
    "Via Expressa": 1400,
}

demand = {
    "Centro": 1200,
    "Parque das Flores": 950,
    "Vila Nova": 750,
    "Jardim Oceânico": 1300,
}

costs = {
    "Linha Central": {
        "Centro": 3.50,
        "Parque das Flores": 4.00,
        "Vila Nova": 3.75,
        "Jardim Oceânico": 4.50,
    },
    "Rota do Parque": {
        "Centro": 3.00,
        "Parque das Flores": 3.50,
        "Vila Nova": 4.00,
        "Jardim Oceânico": 3.60,
    },
    "Via Expressa": {
        "Centro": 3.20,
        "Parque das Flores": 4.20,
        "Vila Nova": 3.90,
        "Jardim Oceânico": 4.10,
    },
}


df, total_cost = TransportationOptimizer(supply, demand, costs).optimize()
```

```
print(f"O custo total é de R$ {total_cost:.2f}".replace(".", ","))
df
```

```
O custo total é de R$ 15207,50
```

[ ]:

| | Centro | Parque das Flores | Vila Nova | Jardim Oceânico |
|---|---|---|---|---|
| Linha Central | 0.0 | 0.0 | 750.0 | 500.0 |
| Rota do Parque | 0.0 | 950.0 | 0.0 | 600.0 |
| Via Expressa | 1200.0 | 0.0 | 0.0 | 200.0 |

## 4 Resultados e Conclusões

Neste estudo, utilizou-se o Método de Vogel para melhorar a forma como os recursos são distribuídos no transporte público, visando economizar dinheiro. Encontrou-se uma solução inicial que pode ser usada como ponto de partida para melhorar o transporte público.

Aqui está a solução inicial:

| Rota | Centro | Parque das Flores | Vila Nova | Jardim Oceânico |
|---|---|---|---|---|
| Linha Central | 0 | 0 | 750 | 500 |
| Rota do Parque | 0 | 950 | 0 | 600 |
| Via Expressa | 1200 | 0 | 0 | 200 |

É importante saber que essa solução inicial pode não ser a melhor possível. Para confirmar se é a melhor, seria preciso usar técnicas mais avançadas, como o Dual, mas isso estava fora do escopo deste trabalho devido à sua complexidade.

No entanto, nossos resultados têm potencial para melhorar a forma como os recursos são usados no transporte público, economizando dinheiro público e melhorando os serviços oferecidos à população.