PROCC0083

Problema do Carteiro Chinês

Apresentação do problema e solução

José Wilson Martins Filho
Ian Sandes Alves

Gravação no Youtube | Código no Github

01

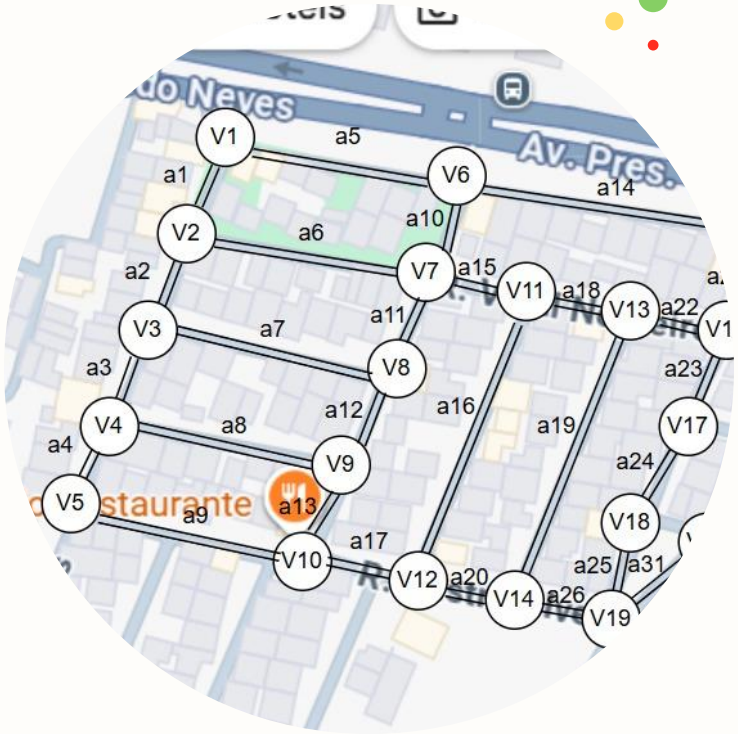
Introdução

Breve explicação do Carteiro Chinês

Problema do Carteiro Chinês

Busca a menor rota que percorra todas as arestas de um grafo ao menos uma vez, retornando ao ponto de partida. É uma modelagem clássica para otimizar percursos em que cada rua precisa ser visitada.

- **Objetivo:** minimizar a distância ou tempo para cobrir todas as ruas.
- **Aplicações:** coleta de lixo, varredura de ruas, inspeção de redes, entrega porta a porta.



Diferenças do Caixeiro Viajante

Característica	Carteiro Chinês (não direcionado)	Caixeiro Viajante
Elementos a visitar	Todas as arestas	Todos os vértices
Objetivo	Minimizar percurso total	Minimizar percurso total
Retorno ao ponto inicial?	Sim	Sim
Complexidade	Polinomial	NP-completo

02

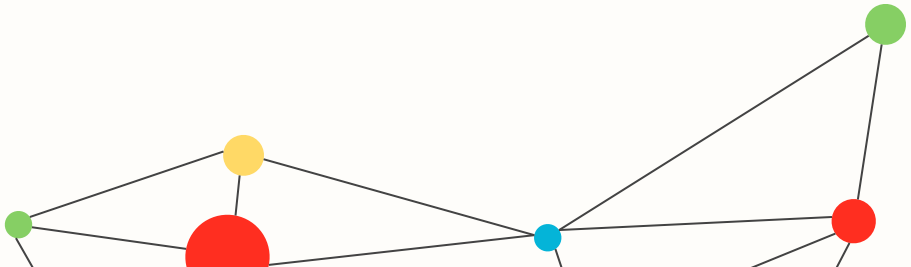
Problema

Descrição do problema da coleta de lixo

Descrição

O **Problema da Coleta de Lixo** trata do planejamento da melhor rota para um caminhão que precisa percorrer todas as ruas de uma região, coletando resíduos de cada ponto.
O objetivo é garantir que nenhuma rua fique sem atendimento, ao mesmo tempo em que se minimiza o custo total do percurso, seja em distância percorrida, tempo de execução ou consumo de combustível.

Matematicamente, o problema pode ser modelado como um grafo, onde os cruzamentos são vértices e as ruas são arestas com pesos correspondentes ao seu comprimento ou tempo de deslocamento. A solução ideal é encontrar uma rota fechada que percorra todas as arestas ao menos uma vez, retornando ao ponto inicial.



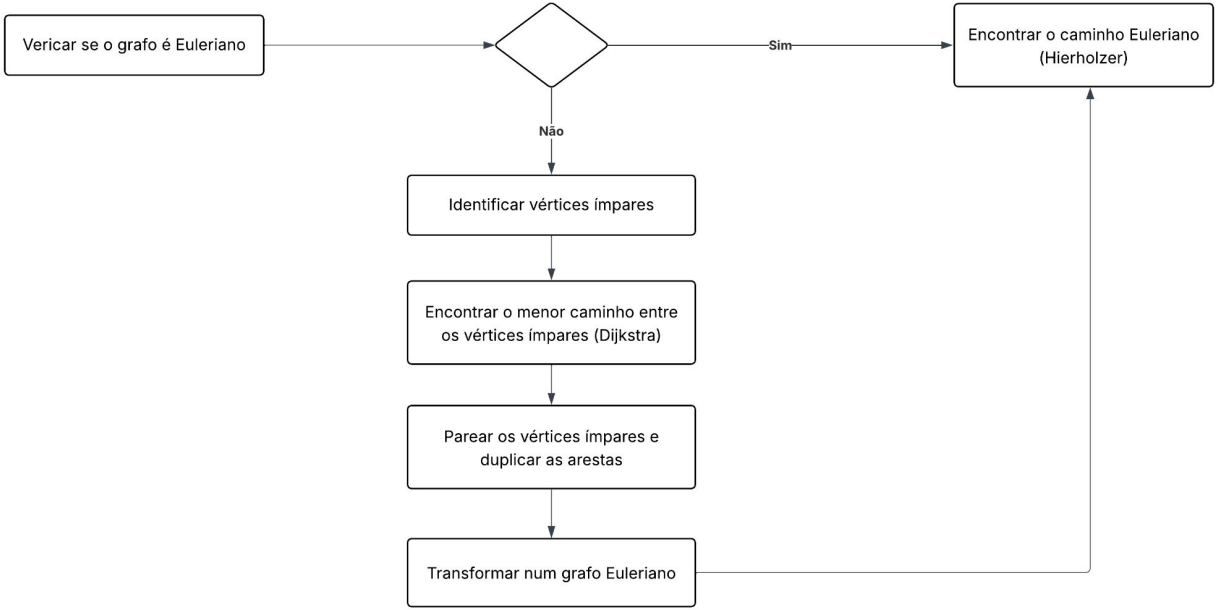
Aplicabilidade em Aracaju

03

Solução

Como o Carteiro Chinês resolve o problema

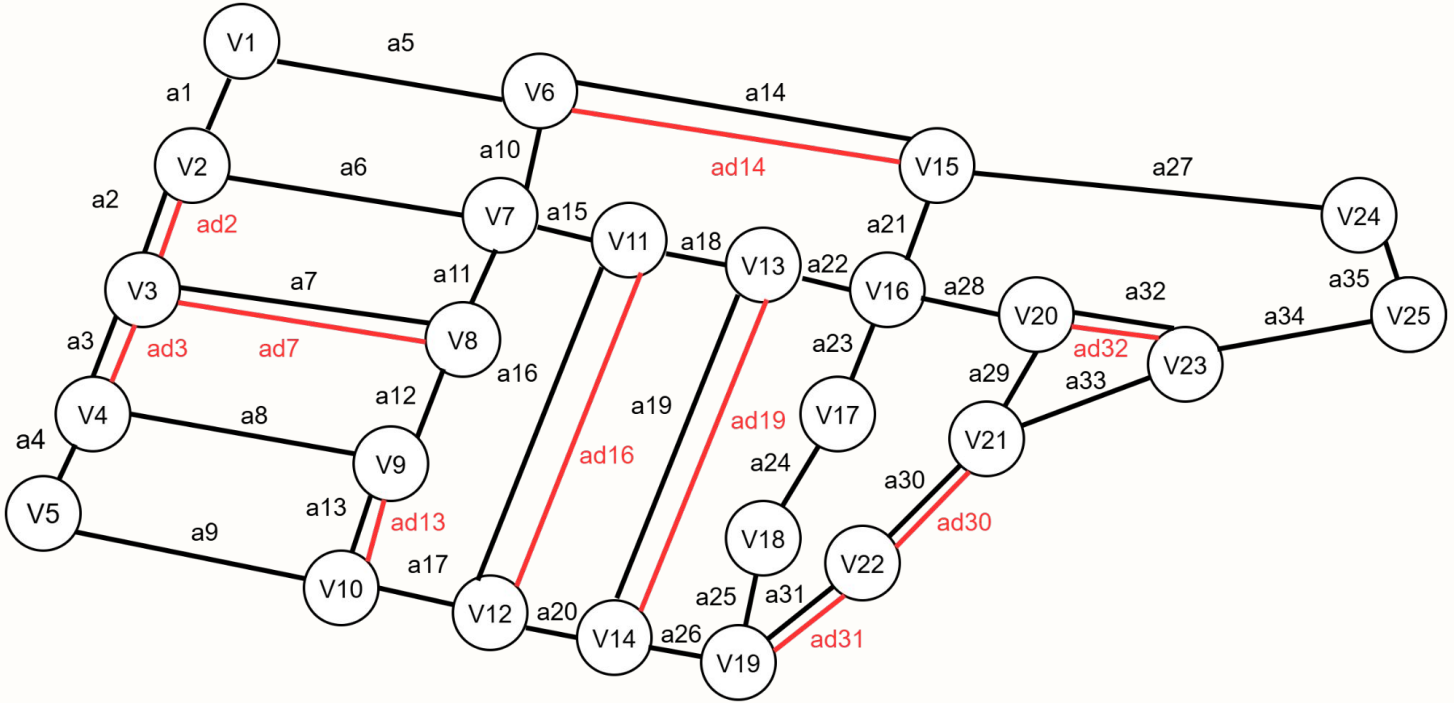
Solução – Fluxograma



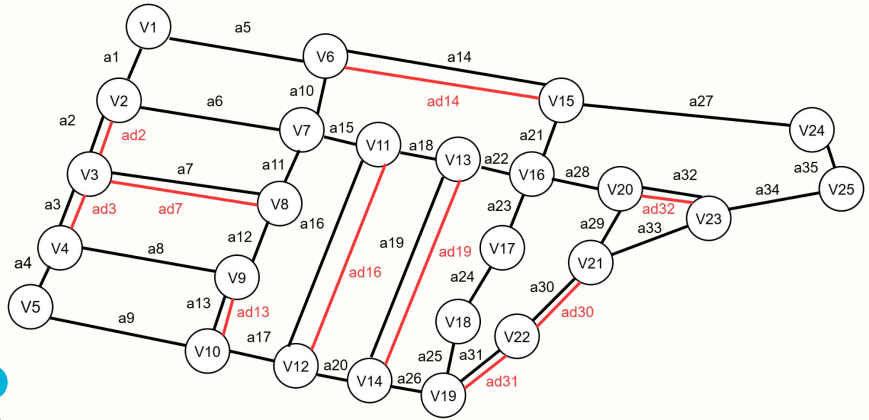
Análise do Grafo

- Total de vértices: 25
- Total de arestas: 35
- Quantidade de vértices de grau par: 9
- Quantidade de vértices de grau ímpar: 16

Grafo euleriano



Grafo euleriano



- Tour percorrido:

  V1 → V2 → V3 → V4 → V5 → V10 → V9 → V4 → V3 → V8 → V7 → V2 → V3 → V8 → V9 → V10 → V12 → V11 → V7 → V6 → V15 → V16 → V13 → V11 → V12 → V14 → V13 → V14 → V19 → V18 → V17 → V16 → V20 → V21 → V22 → V19 → V22 → V21 → V23 → V20 → V23 → V25 → V24 → V15 → V6 → V1

- Total de arestas: 45
- Vértice inicial: V1

04

Código

Explicação detalhada do código

Funções – Construir Lista de Adjacência

```python
def build_adjacency_list(edges_list):
    """
    Builds adjacency list from edges.

    Args:
        edges_list: List of tuples (source, target, label)

    Returns:
        defaultdict: Adjacency list where each vertex maps to list of neighbors
    """
    adjacency_list = defaultdict(list)
    for source_vertex, target_vertex, edge_label in edges_list:
        adjacency_list[source_vertex].append((target_vertex, 1, edge_label))
        adjacency_list[target_vertex].append((source_vertex, 1, edge_label))
    return adjacency_list
```

Link do código completo no Github

Funções – Implementação do Dijkstra

```python
def dijkstra(adjacency_list, source_vertex):
    """
    Calculates shortest distances from a source vertex to all others.

    Args:
        adjacency_list: Graph adjacency list
        source_vertex: Source vertex

    Returns:
        tuple: (distances, predecessors)
    """
    distances = {vertex: float("inf") for vertex in adjacency_list}
    predecessors = {vertex: None for vertex in adjacency_list}
    distances[source_vertex] = 0

    priority_queue = [(0, source_vertex)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance != distances[current_vertex]:
            continue

        for neighbor_vertex, edge_weight, _ in adjacency_list[current_vertex]:
            new_distance = current_distance + edge_weight
            if new_distance < distances[neighbor_vertex]:
                distances[neighbor_vertex] = new_distance
                predecessors[neighbor_vertex] = current_vertex
                heapq.heappush(priority_queue, (new_distance, neighbor_vertex))

    return distances, predecessors
```

Funções – Implementação do Dijkstra

```python
def dijkstra(adjacency_list, source_vertex):
    """
    Calculates shortest distances from a source vertex to all others.

    Args:
        adjacency_list: Graph adjacency list
        source_vertex: Source vertex

    Returns:
        tuple: (distances, predecessors)
    """
    distances = {vertex: float("inf") for vertex in adjacency_list}
    predecessors = {vertex: None for vertex in adjacency_list}
    distances[source_vertex] = 0

    priority_queue = [(0, source_vertex)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance != distances[current_vertex]:
            continue

        for neighbor_vertex, edge_weight, _ in adjacency_list[current_vertex]:
            new_distance = current_distance + edge_weight
            if new_distance < distances[neighbor_vertex]:
                distances[neighbor_vertex] = new_distance
                predecessors[neighbor_vertex] = current_vertex
                heapq.heappush(priority_queue, (new_distance, neighbor_vertex))

    return distances, predecessors
```

Link do código completo no Github

Funções – Implementação do Hierholzer

```python
def hierholzer(adjacency_list, starting_vertex):
    """
    Finds Eulerian circuit using Hierholzer's algorithm.

    Args:
        adjacency_list: Graph adjacency list (must be Eulerian)
        starting_vertex: Starting vertex of the circuit

    Returns:
        list: Eulerian circuit
    """
    vertex_stack = [starting_vertex]
    eulerian_circuit = []
    used_edges = set()

    next_edge_index = {vertex: 0 for vertex in adjacency_list}

    while vertex_stack:
        current_vertex = vertex_stack[-1]

        while (next_edge_index[current_vertex] < len(adjacency_list[current_vertex]) and
               (current_vertex,
                adjacency_list[current_vertex][next_edge_index[current_vertex]][0],
                adjacency_list[current_vertex][next_edge_index[current_vertex]][2]) in used_edges):
            next_edge_index[current_vertex] += 1

        if next_edge_index[current_vertex] == len(adjacency_list[current_vertex]):
            eulerian_circuit.append(current_vertex)
            vertex_stack.pop()
        else:
            neighbor_vertex, _, edge_label = adjacency_list[current_vertex][next_edge_index[current_vertex]]

            if (current_vertex, neighbor_vertex, edge_label) in used_edges:
                next_edge_index[current_vertex] += 1
                continue

            used_edges.add((current_vertex, neighbor_vertex, edge_label))
            used_edges.add((neighbor_vertex, current_vertex, edge_label))
            vertex_stack.append(neighbor_vertex)

    return eulerian_circuit[::-1]
```

codesnap.dev

[Link do código completo no Github]

Funções – Reconstruir o menor caminho

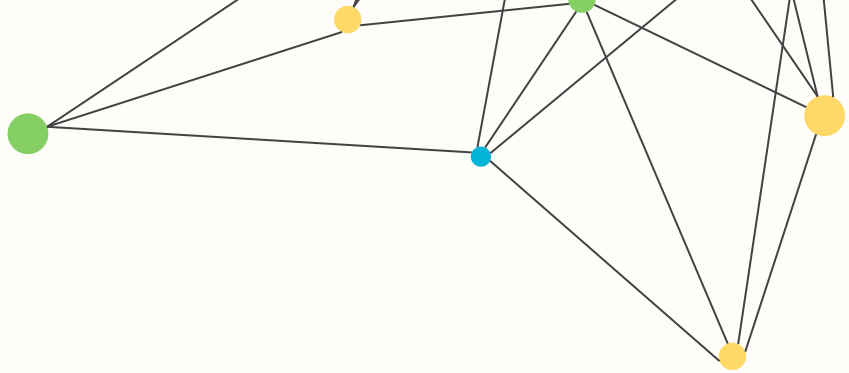
```python
def reconstruct_shortest_path(predecessors, start_vertex, end_vertex):
    """
    Reconstructs the shortest path between two vertices.

    Args:
        predecessors: Dijkstra predecessors dictionary
        start_vertex: Starting vertex
        end_vertex: Ending vertex

    Returns:
        list: Path from start to end
    """
    path = []
    current_vertex = end_vertex

    while current_vertex is not None:
        path.append(current_vertex)
        if current_vertex == start_vertex:
            break
        current_vertex = predecessors[current_vertex]

    return list(reversed(path))
```

Link do código completo no Github

Funções – Integração dos algoritmos

Complexidade de Tempo: $O(V^3 + E \cdot V + 2^k \cdot k^2)$

Complexidade de Espaço: $O(V^2 + E + 2^k)$



Obrigado!

José Wilson Martins Filho – 202521001188
Ian Sandes Alves – 202521001393