Algoritmo A* - Relatório

- **Aluno**: João Victor Carrijo Pereira
- **Professor:** Leonardo Raiz
- **Disciplina:** Inteligência Artificial
- **Curso**: Análise e Desenvolvimento De Sistemas (Noturno - 6º Semestre - 2025/2)

1. Introdução ao Algoritmo A*

O algoritmo A* (A-estrela) é um dos mais importantes e eficientes algoritmos de busca de caminho em Inteligência Artificial. Sua principal característica é a capacidade de encontrar o caminho mais curto e de menor custo entre um ponto de partida e um ponto de destino em um grafo.

O A* se destaca por combinar o melhor de dois métodos de busca: a garantia de otimização do algoritmo de Dijkstra e a velocidade de um algoritmo de busca gulosa. Ele faz isso usando uma função de custo: $f(n) = g(n) + h(n)$ para avaliar cada nó a ser explorado.

- $g(n)$ é o custo real e acumulado para chegar da origem ao nó atual (custo do caminho).
- $h(n)$ é uma estimativa do custo para ir do nó atual (n) até o destino (heurística).

Ao focar no nó com menor valor, o algoritmo equilibra a exploração de caminhos que já são curtos com a priorização daqueles que parecem promissores, garantindo uma solução ideal de forma eficiente.

2. Descrição do Contexto Escolhido

Para aplicar o algoritmo A* em um cenário do mundo real, escolhi o problema de planejamento de rotas em um mapa, similar ao que serviços como o Google Maps fazem. O objetivo é encontrar o trajeto de menor distância entre duas cidades, sendo elas Franca-SP (Origem) e São Paulo-SP (Destino).

O "mapa" foi modelado como um grafo, onde cada cidade é um nó e as rodovias que as conectam são as arestas. O peso de cada aresta representa a distância entre as cidades, funcionando como o custo $g(n)$.

A heurística $h(n)$ foi calculada como a distância em linha reta (distância euclidiana) entre a cidade atual e o destino final. Essa é uma heurística admissível, pois a distância em linha reta é sempre a menor distância possível entre dois pontos, garantindo que o algoritmo A* não superestime o custo e encontre o caminho ideal.

3. Visão Geral da Implementação

A implementação foi desenvolvida em Python (Jupyter Notebook), utilizando a biblioteca NetworkX para a criação e manipulação do grafo e Matplotlib para a visualização.

Abaixo estão os trechos de código que demonstram a estrutura do mapa, a lógica do algoritmo A* e a visualização do resultado.

Criação do Grafo

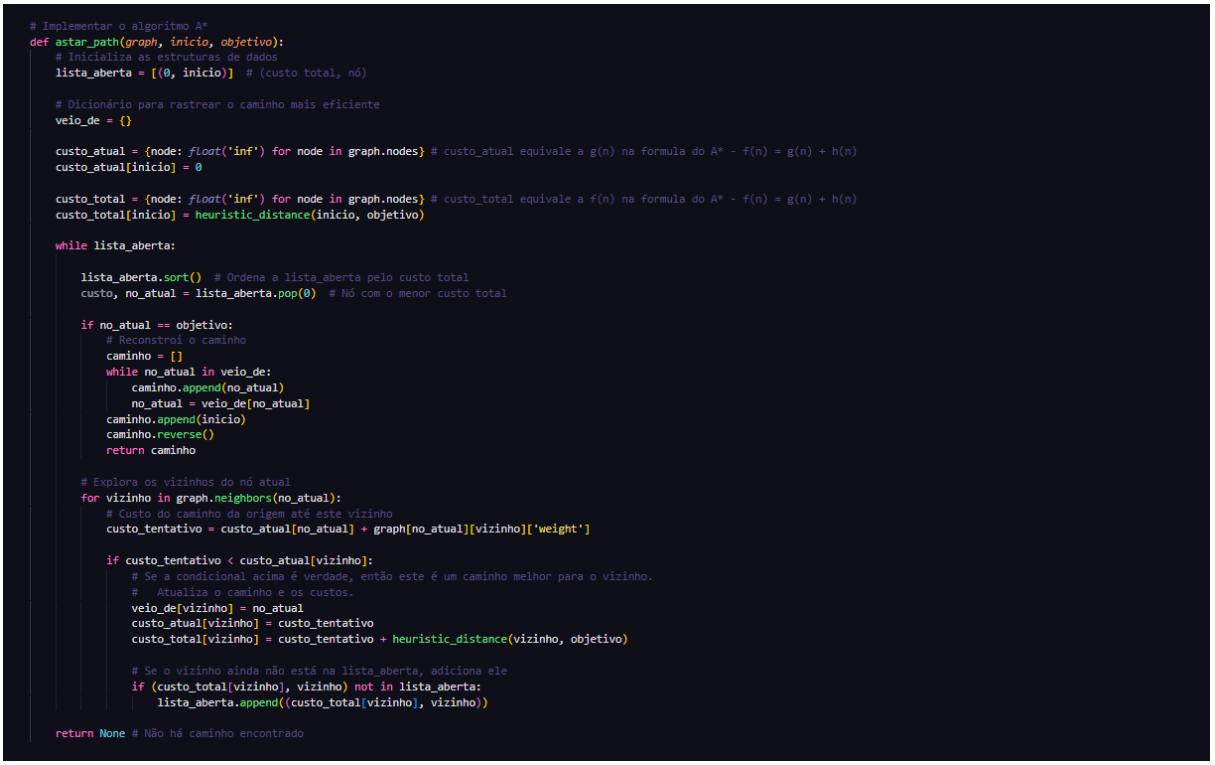
```python
# Criação do mapa de cidades (nós) e suas conexões (arestas)
cidades = {
  'Franca': (10, 100),
  'Ribeirão Preto': (50, 80),
  'Araraquara': (40, 60),
  'Pirassununga': (60, 50),
  'São Carlos': (50, 40),
  'Limeira': (70, 45),
  'Campinas': (80, 20),
  'Jundiaí': (95, 25),
  'Itatiba': (80, 5),
  'São Paulo': (100, 0)
}

# Instancia do grafo
graph = nx.Graph()

# Adicionando nós (cidades) ao grafo
for cidade, coords in cidades.items():
  graph.add_node(cidade, pos=coords)

# Adicionando arestas (conexões entre as cidades)
#  Arestas do caminho principal
graph.add_edge('Franca', 'Ribeirão Preto', weight=90)
graph.add_edge('Ribeirão Preto', 'São Carlos', weight=100)
graph.add_edge('São Carlos', 'Campinas', weight=115)
graph.add_edge('Campinas', 'São Paulo', weight=95)

#  Arestas de caminhos alternativos
graph.add_edge('Franca', 'Araraquara', weight=150)
graph.add_edge('Araraquara', 'Campinas', weight=160)
graph.add_edge('Ribeirão Preto', 'Pirassununga', weight=120)
graph.add_edge('Pirassununga', 'Limeira', weight=60)
graph.add_edge('Limeira', 'Jundiaí', weight=100)
graph.add_edge('Campinas', 'Jundiaí', weight=45)
graph.add_edge('Campinas', 'Itatiba', weight=30)
graph.add_edge('Itatiba', 'Jundiaí', weight=40)
graph.add_edge('Jundiaí', 'São Paulo', weight=60)
```

Função de Busca A*

A função `astar_path` implementa a lógica do algoritmo. Ela utiliza as seguintes estruturas de dados:

- **`lista_aberta`**: Uma fila de prioridade que armazena os nós a serem explorados.
- **`veio_de`**: Um dicionário que rastreia o caminho, armazenando o nó anterior para cada nó visitado.
- **`custo_atual`**: Um dicionário que guarda o custo real (`g_score`) da origem até o nó atual.
- **`custo_total`**: Um dicionário que guarda o custo total estimado (`f_score`) de cada nó.

O algoritmo funciona em um loop que, a cada iteração, retira o nó com o menor `custo_total` da `lista_aberta` e explora seus vizinhos, atualizando seus custos se encontrar um caminho mais eficiente.



```python
# Implementar o algoritmo A*
def astar_path(graph, inicio, objetivo):
    # Inicializa as estruturas de dados
    lista_aberta = [(0, inicio)]  # (custo total, nó)

    # Dicionário para rastrear o caminho mais eficiente
    veio_de = {}

    custo_atual = {node: float('inf') for node in graph.nodes} # custo_atual equivale a g(n) na formula do A* - f(n) = g(n) + h(n)
    custo_atual[inicio] = 0

    custo_total = {node: float('inf') for node in graph.nodes} # custo_total equivale a f(n) na formula do A* - f(n) = g(n) + h(n)
    custo_total[inicio] = heuristic_distance(inicio, objetivo)

    while lista_aberta:

        lista_aberta.sort()  # Ordena a lista_aberta pelo custo total
        custo, no_atual = lista_aberta.pop(0)  # Nó com o menor custo total

        if no_atual == objetivo:
            # Reconstroi o caminho
            caminho = []
            while no_atual in veio_de:
                caminho.append(no_atual)
                no_atual = veio_de[no_atual]
            caminho.append(inicio)
            caminho.reverse()
            return caminho

        # Explora os vizinhos do nó atual
        for vizinho in graph.neighbors(no_atual):
            # Custo do caminho da origem até este vizinho
            custo_tentativo = custo_atual[no_atual] + graph[no_atual][vizinho]['weight']

            if custo_tentativo < custo_atual[vizinho]:
                # Se a condicional acima é verdade, então este é um caminho melhor para o vizinho.
                #   Atualiza o caminho e os custos.
                veio_de[vizinho] = no_atual
                custo_atual[vizinho] = custo_tentativo
                custo_total[vizinho] = custo_tentativo + heuristic_distance(vizinho, objetivo)

                # Se o vizinho ainda não está na lista_aberta, adiciona ele
                if (custo_total[vizinho], vizinho) not in lista_aberta:
                    lista_aberta.append((custo_total[vizinho], vizinho))

    return None # Não há caminho encontrado
```

4. Análise de Resultados

A execução do algoritmo A* com o mapa de cidades fornecido resultou na identificação do caminho mais curto de Franca a São Paulo.

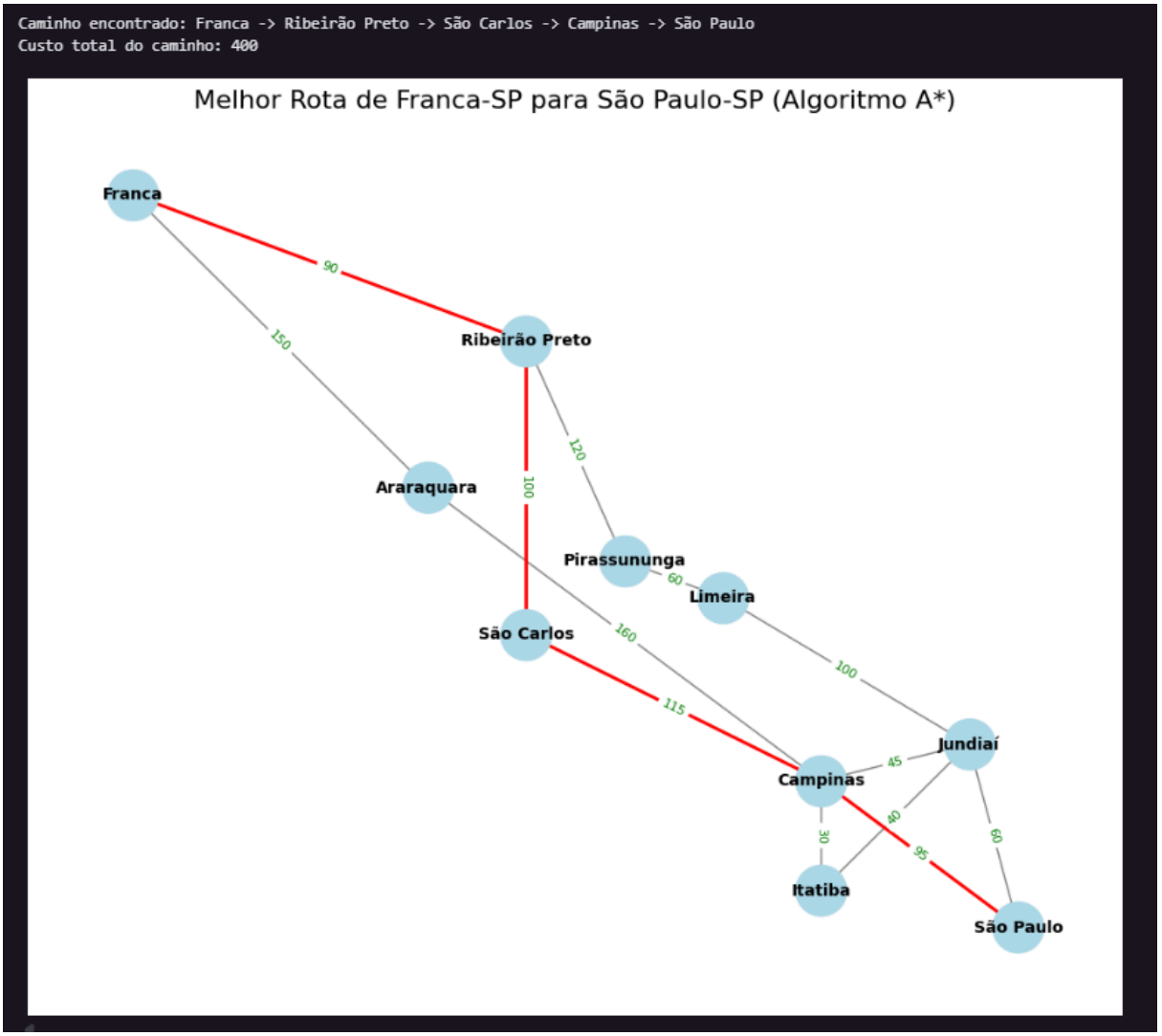
Caso de Teste:

- **Origem:** Franca
- **Destino:** São Paulo

Saída do Algoritmo:

- **Caminho Encontrado:** `Franca → Ribeirão Preto → São Carlos → Campinas → São Paulo`
- **Custo Total do Caminho:** 400

Visualização do Grafo



A visualização do grafo ilustra claramente o caminho encontrado. As arestas em cinza representam os caminhos alternativos, enquanto a linha vermelha destacada indica a rota ótima calculada pelo algoritmo A*. Os valores numéricos sobre as arestas mostram o custo (distância) de cada segmento da “rodovia”.

A imagem mostra que o A* ignorou rotas mais longas, como a que passa por Araraquara, e encontrou o caminho principal, que tem o menor custo total, demonstrando a eficácia da heurística em guiar a busca de forma inteligente.

5. Conclusão

A implementação do algoritmo A* foi bem-sucedida, cumprindo todos os objetivos propostos. A escolha de um problema de planejamento de rotas

demonstrou como um algoritmo de busca pode ser aplicado de forma prática para resolver desafios do dia a dia.

Pontos Fortes

- **Eficiência:** A heurística de distância euclidiana permitiu que o algoritmo evitasse a exploração de ramificações de caminhos longos e ineficientes, convergindo rapidamente para a solução ideal.
- **Precisão:** A* garantiu a descoberta do caminho de menor custo total, o que é crucial em aplicações como navegação GPS.
- **Estrutura de Código:** O uso de dicionários e listas para armazenar os custos e rastrear o caminho tornou a lógica clara.

Pontos Fracos

- **Performance da Lista Aberta:** A implementação usa a função `list.sort()` dentro do loop `while`, isto causará um desempenho ruim em grafos muito grandes. A cada iteração, o algoritmo precisa reordenar a lista inteira, o que é computacionalmente caro.