Código-Fonte Python COLAB

O código-fonte Python completo está incluído abaixo. Ele é totalmente autônomo e pode ser executado diretamente, pois incorpora a geração de dados fictícios (mock data) necessários para simular o mapa da cidade e os pedidos, conforme o requisito.

O código implementa **K-Means** (baseado na **Unidade 3**) para agrupamento e **A^* (A-Star)** (baseado na **Unidade 2**) para roteamento.

Python

```
import heapq
import math
import numpy as np
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# --- Mock Data (Simulação de Dados) ---
# Em um projeto real, isso viria de CSVs ou um banco de dados.

# 1. Coordenadas dos Pontos (Nós)
# Mapeia cada local (nó) para uma coordenada (x, y)
# Usado pelo K-Means para clustering e pelo A* para a heurística.
coordenadas_pontos = {
    'Base': (0, 0),
    'A': (1, 5),
    'B': (3, 2),
    'C': (4, 8),
    'D': (5, 6),
    'E': (8, 9),
    'F': (10, 5),
    'G': (12, 8),
    'H': (11, 2),
    'I': (7, 1),
}

# 2. Grafo da Cidade (Arestas Ponderadas)
# Representa as ruas (arestas) e distâncias (pesos) entre bairros (nós).
```

```python
# Esta é a "Representação de problemas com grafos" da Unidade 2[cite: 748].
grafo_cidade = {
    'Base': {'A': 5, 'B': 4, 'I': 9},
    'A': {'Base': 5, 'C': 4},
    'B': {'Base': 4, 'D': 7, 'I': 3},
    'C': {'A': 4, 'D': 3, 'E': 5},
    'D': {'B': 7, 'C': 3, 'F': 8},
    'E': {'C': 5, 'G': 4},
    'F': {'D': 8, 'G': 6, 'H': 7, 'I': 5},
    'G': {'E': 4, 'F': 6},
    'H': {'F': 7, 'I': 3},
    'I': {'Base': 9, 'B': 3, 'F': 5, 'H': 3},
}

# 3. Lista de Pedidos (Desafio do Dia)
# Pontos de entrega que precisam ser atendidos.
pedidos_do_dia = ['C', 'D', 'E', 'G', 'H', 'I']

# ---------------------------------------------------------------------
# ETAPA 1: AGRUPAMENTO DE ENTREGAS (K-MEANS)
# Baseado na Unidade 3: Algoritmos Básicos em Machine Learning [cite: 7]
# ---------------------------------------------------------------------

def agrupar_pedidos_em_zonas(pedidos, coordenadas, num_zonas=2):
    """
    Usa K-Means para agrupar pedidos em zonas de entrega.
    Conceito da Unidade 3[cite: 183]: K-Means é um método de clustering
    não supervisionado que particiona dados em 'k' grupos (zonas)
    minimizando a variabilidade intra-grupo [cite: 184-186].
    """
    print("--- 1. Executando K-Means (Unidade 3) ---")

    # Prepara os dados: extrai as coordenadas (x,y) dos pedidos
    pontos_pedidos = np.array([coordenadas[ponto] for ponto in pedidos])

    # Padronização é uma boa prática (Unidade 3, Boas Práticas [cite: 281])
    scaler = StandardScaler()
    pontos_padronizados = scaler.fit_transform(pontos_pedidos)

    # Instancia e treina o modelo K-Means
    kmeans = KMeans(n_clusters=num_zonas, random_state=42, n_init=10)
    kmeans.fit(pontos_padronizados)

    # Obtém os rótulos (a qual zona cada pedido pertence)
```

```
    rotulos_zona = kmeans.labels_

    # Organiza os pedidos nas zonas
    zonas = [[] for _ in range(num_zonas)]
    for i, ponto in enumerate(pedidos):
        zona_idx = rotulos_zona[i]
        zonas[zona_idx].append(ponto)

    print(f"Pedidos agrupados em {num_zonas} zonas:")
    for i, zona in enumerate(zonas):
        print(f"  Zona {i+1}: {zona}")
    print("----------------------------------------\n")
    return zonas

# ---------------------------------------------------------------------
# ETAPA 2: OTIMIZAÇÃO DE ROTA (ALGORITMO A*)
# Baseado na Unidade 2: Algoritmos de Busca em Grafos [cite: 633]
# ---------------------------------------------------------------------

def heuristica_admissivel(ponto_a, ponto_b, coordenadas):
    """
    Heurística (h(n)) para o A*.
    Calcula a distância Euclidiana (linha reta) entre dois pontos.
    Conceito da Unidade 2[cite: 700]: Esta é uma "heurística admissível"
    porque a distância em linha reta NUNCA superestima o custo real
    de dirigir pelas ruas, garantindo que o A* encontre a rota ótima.
    """
    (x1, y1) = coordenadas[ponto_a]
    (x2, y2) = coordenadas[ponto_b]
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

def a_star_search(grafo, inicio, fim, coordenadas):
    """
    Implementação do Algoritmo A* (A-Star).
    Conceito da Unidade 2[cite: 696]: A* é uma busca informada que combina
    o custo real percorrido (g) e a heurística (h) para estimar o custo
    total (f): f(n) = g(n) + h(n)[cite: 700].
    """

    # Fila de prioridade (Open Set) [cite: 703]
    # Armazena (f_cost, g_cost, no_atual, caminho_percorrido)
    fila_prioridade = [(0 + heuristica_admissivel(inicio, fim, coordenadas), 0, inicio, [inicio])]

    # Conjunto de visitados (Closed Set) [cite: 701]
```

```
    visitados = set()

    while fila_prioridade:
        # Pega o nó com menor f(n) [cite: 707]
        (f_cost, g_cost, no_atual, caminho) = heapq.heappop(fila_prioridade)

        if no_atual in visitados:
            continue

        visitados.add(no_atual)

        # Objetivo alcançado [cite: 708]
        if no_atual == fim:
            return caminho, g_cost

        # Explora vizinhos
        if no_atual in grafo:
            for vizinho, peso_aresta in grafo[no_atual].items():
                if vizinho not in visitados:
                    # g(n) = custo do caminho até agora
                    novo_g_cost = g_cost + peso_aresta

                    # h(n) = heurística do vizinho até o fim
                    h_cost = heuristica_admissivel(vizinho, fim, coordenadas)

                    # f(n) = g(n) + h(n)
                    novo_f_cost = novo_g_cost + h_cost

                    heapq.heappush(fila_prioridade, (novo_f_cost, novo_g_cost, vizinho, caminho + [vizinho]))

    return None, 0 # Caminho não encontrado

# ----------------------------------------------------------------------
# EXCUÇÃO PRINCIPAL: ROTA INTELIGENTE
# ----------------------------------------------------------------------

def main():
    print("--- Iniciando Solução 'Rota Inteligente' para Sabor Express ---")

    # ETAPA 1: Agrupar pedidos
    # Vamos supor que temos 2 entregadores
    num_entregadores = 2
    zonas_de_entrega = agrupar_pedidos_em_zonas(pedidos_do_dia, coordenadas_pontos,
num_entregadores)
```

```
    # ETAPA 2: Calcular rotas para cada zona
    print("--- 2. Calculando Rotas com A* (Unidade 2) ---")

    custo_total_empresa = 0

    for i, zona in enumerate(zonas_de_entrega):
        if not zona:
            print(f"Entregador {i+1}: Sem pedidos hoje.")
            continue

        print(f"\nPlanejando Entregador {i+1} (Zona {i+1}: {zona}):")

        # Otimização simples: vai da Base ao primeiro ponto,
        # depois para o próximo, e assim por diante.
        # (Uma otimização TSP [Unidade 2, cite: 997] seria um próximo passo)

        ponto_partida = 'Base'
        custo_total_zona = 0

        for ponto_destino in zona:
            rota, custo_trecho = a_star_search(grafo_cidade, ponto_partida, ponto_destino,
coordenadas_pontos)

            if rota:
                print(f"  > Rota: {' -> '.join(rota)} (Custo: {custo_trecho:.2f})")
                custo_total_zona += custo_trecho
                ponto_partida = ponto_destino # O próximo trecho começa onde este terminou
            else:
                print(f"  > Rota de {ponto_ partida} para {ponto_destino} não encontrada!")

        # Adiciona a rota de volta para a Base
        rota_volta, custo_volta = a_star_search(grafo_cidade, ponto_partida, 'Base', coordenadas_pontos)
        if rota_volta:
            print(f"  > Volta: {' -> '.join(rota_volta)} (Custo: {custo_volta:.2f})")
            custo_total_zona += custo_volta

        print(f"  Custo Total da Rota {i+1}: {custo_total_zona:.2f}")
        custo_total_empresa += custo_total_zona

    print("-----------------------------------------")
    print(f"\nCUSTO OPERACIONAL TOTAL (SOMA DE ROTAS): {custo_total_empresa:.2f}")
    print("--- Solução 'Rota Inteligente' Concluída ---")

if __name__ == "__main__":
    main()
```

Código-Fonte Python VSCODE

```
import heapq
import math
import numpy as np

# Lightweight fallback implementations to avoid scikit-learn dependency
def simple_standard_scaler(X):
    mean = X.mean(axis=0)
    std = X.std(axis=0, ddof=0)
    std[std == 0] = 1.0
    return (X - mean) / std

def simple_kmeans(X, n_clusters, n_init=10, max_iter=100, random_state=42):
    rng = np.random.RandomState(random_state)
    best_labels = None
    best_inertia = np.inf
    n_samples = X.shape[0]
    for init in range(n_init):
        if n_clusters <= n_samples:
            indices = rng.choice(n_samples, n_clusters, replace=False)
            centers = X[indices].copy()
        else:
            # fallback: sample with replacement if clusters > samples
            centers = X[rng.choice(n_samples, n_samples, replace=True)][:n_clusters].copy()
        for it in range(max_iter):
            dists = np.linalg.norm(X[:, None, :] - centers[None, :, :], axis=2)
            labels = np.argmin(dists, axis=1)
            new_centers = np.array([X[labels==k].mean(axis=0) if np.any(labels==k) else centers[k]
for k in range(n_clusters)])
            if np.allclose(new_centers, centers):
                break
            centers = new_centers
        inertia = np.sum((X - centers[labels])**2)
        if inertia < best_inertia:
            best_inertia = inertia
            best_labels = labels.copy()
    return best_labels

# --- Mock Data (Simulação de Dados) ---
```

```
# Em um projeto real, isso viria de CSVs ou um banco de dados.

# 1. Coordenadas dos Pontos (Nós)
# Mapeia cada local (nó) para uma coordenada (x, y)
# Usado pelo K-Means para clustering e pelo A* para a heurística.
coordenadas_pontos = {
    'Base': (0, 0),
    'A': (1, 5),
    'B': (3, 2),
    'C': (4, 8),
    'D': (5, 6),
    'E': (8, 9),
    'F': (10, 5),
}

def agrupar_pedidos_em_zonas(pedidos, coordenadas, num_zonas=2):
    """
    Usa uma versão leve de K-Means para agrupar pedidos em zonas de entrega.
    Esta implementação usa simple_standard_scaler e simple_kmeans definidos acima.
    """
    print("--- 1. Executando K-Means (Unidade 3) ---")

    if not pedidos:
        return [[] for _ in range(num_zonas)]

    # Prepara os dados: extrai as coordenadas (x,y) dos pedidos
    pontos_pedidos = np.array([coordenadas[ponto] for ponto in pedidos])

    # Padronização simples (substitui StandardScaler)
    pontos_padronizados = simple_standard_scaler(pontos_pedidos)

    # Usa implementação leve de K-Means (substitui sklearn.KMeans)
    rotulos_zona = simple_kmeans(pontos_padronizados, n_clusters=num_zonas, n_init=10,
max_iter=100, random_state=42)

    # Se por algum motivo rotulos_zona for None (ex.: pontos == 0), coloca todos em uma zona
    if rotulos_zona is None:
        rotulos_zona = np.zeros(len(pedidos), dtype=int)

    # Organiza os pedidos nas zonas
    zonas = [[] for _ in range(num_zonas)]
    for i, ponto in enumerate(pedidos):
        zona_idx = int(rotulos_zona[i]) % num_zonas
```

```
            zonas[zona_idx].append(ponto)

    print(f"Pedidos agrupados em {num_zonas} zonas:")
    for i, zona in enumerate(zonas):
        print(f"  Zona {i+1}: {zona}")
    print("----------------------------------------\n")
    return zonas

# ---------------------------------------------------------------------
# ETAPA 2: OTIMIZAÇÃO DE ROTA (ALGORITMO A*)
# Baseado na Unidade 2: Algoritmos de Busca em Grafos [cite: 633]
# ---------------------------------------------------------------------

def heuristica_admissivel(ponto_a, ponto_b, coordenadas):
    """
    Heurística (h(n)) para o A*.
    Calcula a distância Euclidiana (linha reta) entre dois pontos.
    Conceito da Unidade 2[cite: 700]: Esta é uma "heurística admissível"
    porque a distância em linha reta NUNCA superestima o custo real
    de dirigir pelas ruas, garantindo que o A* encontre a rota ótima.
    """
    (x1, y1) = coordenadas[ponto_a]
    (x2, y2) = coordenadas[ponto_b]
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

def a_star_search(grafo, inicio, fim, coordenadas):
    """
    Implementação do Algoritmo A* (A-Star).
    Conceito da Unidade 2[cite: 696]: A* é uma busca informada que combina
    o custo real percorrido (g) e a heurística (h) para estimar o custo
    total (f): f(n) = g(n) + h(n)[cite: 700].
    """

    # Fila de prioridade (Open Set) [cite: 703]
    # Armazena (f_cost, g_cost, no_atual, caminho_percorrido)
    fila_prioridade = [(0 + heuristica_admissivel(inicio, fim, coordenadas), 0, inicio, [inicio])]

    # Conjunto de visitados (Closed Set) [cite: 701]
    visitados = set()

    while fila_prioridade:
        # Pega o nó com menor f(n) [cite: 707]
        (f_cost, g_cost, no_atual, caminho) = heapq.heappop(fila_prioridade)
```

```
        if no_atual in visitados:
            continue

        visitados.add(no_atual)

        # Objetivo alcançado [cite: 708]
        if no_atual == fim:
            return caminho, g_cost

        # Explora vizinhos
        if no_atual in grafo:
            for vizinho, peso_aresta in grafo[no_atual].items():
                if vizinho not in visitados:
                    # g(n) = custo do caminho até agora
                    novo_g_cost = g_cost + peso_aresta

                    # h(n) = heurística do vizinho até o fim
                    h_cost = heuristica_admissivel(vizinho, fim, coordenadas)

                    # f(n) = g(n) + h(n)
                    novo_f_cost = novo_g_cost + h_cost

                    heapq.heappush(fila_prioridade, (novo_f_cost, novo_g_cost, vizinho, caminho +
[vizinho]))

    return None, 0 # Caminho não encontrado

# ----------------------------------------------------------------------
# Dados de execução e grafo simples construído a partir das coordenadas
# ----------------------------------------------------------------------

# Pedidos de exemplo para o dia (podem ser carregados de arquivo em produção)
pedidos_do_dia = ['A', 'B', 'C', 'D']

# Constrói um grafo completo onde o peso da aresta é a distância Euclidiana
grafo_cidade = {}
for u, coord_u in coordenadas_pontos.items():
    vizinhos = {}
    for v, coord_v in coordenadas_pontos.items():
        if u == v:
            continue
        dist = math.sqrt((coord_u[0] - coord_v[0])**2 + (coord_u[1] - coord_v[1])**2)
```

```
        vizinhos[v] = dist
    grafo_cidade[u] = vizinhos

# ------------------------------------------------------------------
# EXCUÇÃO PRINCIPAL: ROTA INTELIGENTE
# ------------------------------------------------------------------

def main():
    print("--- Iniciando Solução 'Rota Inteligente' para Sabor Express ---")

    # ETAPA 1: Agrupar pedidos
    # Vamos supor que temos 2 entregadores
    num_entregadores = 2
    zonas_de_entrega = agrupar_pedidos_em_zonas(pedidos_do_dia, coordenadas_pontos,
num_entregadores)

    # ETAPA 2: Calcular rotas para cada zona
    print("--- 2. Calculando Rotas com A* (Unidade 2) ---")

    custo_total_empresa = 0

    for i, zona in enumerate(zonas_de_entrega):
        if not zona:
            print(f"Entregador {i+1}: Sem pedidos hoje.")
            continue

        print(f"\nPlanejando Entregador {i+1} (Zona {i+1}: {zona}):")

        # Otimização simples: vai da Base ao primeiro ponto,
        # depois para o próximo, e assim por diante.
        # (Uma otimização TSP [Unidade 2, cite: 997] seria um próximo passo)

        ponto_partida = 'Base'
        custo_total_zona = 0

        for ponto_destino in zona:
            rota, custo_trecho = a_star_search(grafo_cidade, ponto_partida, ponto_destino,
coordenadas_pontos)

            if rota:
                print(f"  > Rota: {' -> '.join(rota)} (Custo: {custo_trecho:.2f})")
                custo_total_zona += custo_trecho
                ponto_partida = ponto_destino # O próximo trecho começa onde este terminou
```

```
            else:
                print(f"  > Rota de {ponto_partida} para {ponto_destino} não encontrada!")

        # Adiciona a rota de volta para a Base
        rota_volta, custo_volta = a_star_search(grafo_cidade, ponto_partida, 'Base',
coordenadas_pontos)
        if rota_volta:
            print(f"  > Volta: {' -> '.join(rota_volta)} (Custo: {custo_volta:.2f})")
            custo_total_zona += custo_volta

        print(f"  Custo Total da Rota {i+1}: {custo_total_zona:.2f}")
        custo_total_empresa += custo_total_zona

    print("-----------------------------------------")
    print(f"\nCUSTO OPERACIONAL TOTAL (SOMA DE ROTAS): {custo_total_empresa:.2f}")
    print("--- Solução 'Rota Inteligente' Concluída ---")

if __name__ == "__main__":
    main()
```