Algoritmos II - Trabalho Prático I

Bruno Buti Ferreira Guilherme
Ricardo Shen

1 Introdução

Neste projeto, foi apresentado o problema de implementar, em Python, um sistema consulta de bares e restaurantes em Belo Horizonte programado, a partir de dados públicos oferecidos pela prefeitura. Neste caso, as buscas são feitas a partir de um sistema interativo a partir do mapa do município, em que o usuário seleciona uma determinada região retangular e recebe de volta todos os estabelecimentos registrados naquela área. Para tal, foi implementada e utilizada uma estrutura de dados conhecida como árvore K-dimensional (Kd Tree), especializada para armazenamento e busca de dados com coordenadas espaciais, de modo a garantir uma execução mais rápida e eficiente do programa

2 Implementação

2.1 Estruturas Auxiliares

Antes de começar programar a árvore, são necessárias algumas estruturas com o objetivo de modularizar e facilitar a implementação da classe principal da Kd Tree.

2.1.1 Info

Representação dos dados importantes do estabelecimento, como nome registrado, nome fantasia e seu endereço

```
class Info:
  def __init__(self, date, addr, name, ficname, has_license,
      id_ativ_econ_estabelecimento = None):
    self.date = date
    self.address = addr
    self.name = name
    self.ficname = ficname
    self.has_license = has_license
    self.id_ativ_econ_estabelecimento = id_ativ_econ_estabelecimento
```

2.1.2 Point

Representação de um estabelecimento no espaço geográfico bidimensional. Nesta e em outras classes foi usada a biblioteca Numpy (np) para otimizar a implementação dessas estruturas. Apresenta métodos x e y para retornar as respectivas coordenadas do ponto

```
class Point:
  def __init__(self, data, x = 0.0, y = 0.0):
    self.data = data
    self.point = np.array([x, y], dtype = np.float64)

  @property
  def x(self):
    return self.point[0]

  @property
```

```
  def y(self):
    return self.point[1]
```

2.1.3 Rectangle

Representação de uma área retangular no espaço, utilizando como base um vértice inferior esquerdo e um vértice superior direito. Apresenta métodos *contains* para determinar se um determinado ponto P está contido no retângulo e *intersect_axis* para determinar se ele intersecciona algum dos eixos de separação dos nós da árvore.

```
class Rectangle:
  def __init__(self, p1, p2):
    self.ll = np.array([min(p1.x, p2.x), min(p1.y, p2.y)], dtype = np.float64)
    self.ur = np.array([max(p1.x, p2.x), max(p1.y, p2.y)], dtype = np.float64)

  def contains(self, p):
    return self.ll[0] <= p.x <= self.ur[0] and self.ll[1] <= p.y <= self.ur[1]

  def intersect_axis(self, p, axis):
    return self.ll[axis] <= p, self.ur[axis] >= p
```

2.1.4 Node

Representação da estrutura fundamental que armazena os dados e a organização hierárquica da árvore. São guardados ponteiros para as subárvores da esquerda e da direita, caso existam, assim como informações do eixo de divisão e o valor da coordenada em que a segmentação foi realizada.

```
class Node:
  def __init__(self, point, axis, left = None, right = None):
    self.point = point
    self.axis = axis
    self.left = left
    self.right = right
    self.split = point.point[axis]
```

2.2 KdTree

Estrutura de organização espacial dos estabelecimentos, permitindo buscas rápidas a partir de uma região retangular. É armazenado apenas um único nó, a raíz da árvore, a partir de onde todas as inserções e buscas serão realizadas. Apresenta o método *__len__* para retornar o tamanho da árvore em uma chamada usando a função *len()*

```
class KdTree:
  def __init__(self, points, depth = 0):
    self.root, self.len = self.build_tree(list(points), depth)

  def __len__(self):
    return self.len
```

2.2.1 Método build_tree

Método utilizado para construir a árvore recursivamente a partir de uma lista de pontos. As inserções são iniciadas pela raíz e a cada chamada, a lista de pontos é particionada a partir da sua mediana, utilizada como o ponto armazenado pelo nó atual, com cada sublista sendo passada como parâmetro para a construção das subárvores da direita e da esquerda. As segmentações recursivas são encerradas quando a lista de pontos é totalmente consumida.

```python
def build_tree(self, points, depth):
  if not points:
    return None, 0

  k = 2
  axis = depth % k

  coords = np.array([p.point[axis] for p in points])
  median_idx = len(points) // 2

  partition_idxs = np.argpartition(coords, median_idx)
  points = [points[i] for i in partition_idxs]
  median = points[median_idx]

  left_node, left_len = self.build_tree(points[ : median_idx], depth + 1)
  right_node, right_len = self.build_tree(points[median_idx + 1 : ], depth +
      1)
  node = Node(median, axis, left = left_node, right = right_node)

  return node, left_len + right_len + 1
```

2.2.2 Método search

Método utilizado para realizar a consulta ortogonal dos pontos baseados em uma área retangular. A busca é feita recursivamente a partir de cada subárvore. Se um determinado nó pertence à região dada, ele é adicionado à uma lista de pontos encontrados. Em seguida, os nós da direita e da esquerda são analisados recursivamente, até que não hajam mais pontos a serem explorados. Diferentemente de uma busca linear por um vetor ou por uma lista encadeada, a estrutura da Kd Tree permite que a consulta seja guiada a partir dos dados geográficos, evitando comparações desnecessárias com pontos claramente fora do escopo fornecido. Essa característica da árvore garante operações de busca rápidas e eficientes, especialmente em situações com milhares de pontos, como neste trabalho. O tamanho da árvore também é feito recursivamente neste método.

```python
def search(self, search_area):
  in_range = []

  def search_recursive(node):
    if node is None:
      return

    if search_area.contains(node.point):
      in_range.append(node.point)

    left_intersect, right_intersect = search_area.intersect_axis(node.split,
        node.axis)

    if left_intersect:
      search_recursive(node.left)

    if right_intersect:
      search_recursive(node.right)

  search_recursive(self.root)
  return in_range
```

3 Filtragem de dados

Para otimizar o tempo de processamento anterior à inserção dos dados na árvore, foi realizado uma pré-filtragem de todos os estabelecimentos, armazenando em um arquivo CSV apenas os pontos de interesse, bares e restaurantes. Primeiramente, os dados foram baixados diretamente do link fornecido pela prefeitura de Belo Horizonte. A partir de lá, o CSV foi separado em linhas contendo informações dos estabelecimentos e a linha específica contendo a representação de cada coluna. Em seguida, os valores únicos na coluna de *DESCRICAO_CNAE_PRINCIPAL* foram separados e filtrados para manter apenas aqueles que se referem a restaurantes ou bares. Por fim, o dataset original foi novamente filtrado em um novo, contendo apenas os estabelecimentos que se encaixam na descrição, e salvo em um novo arquivo. Todas as operações na árvore são realizadas a partir do CSV contendo apenas os dados já tratados. Ao total, o número de linhas após a manipulação desceu de cerca de 534 mil para menos de 18 mil.

4 Aplicativo Web

O sistema de consultas foi programado utilizando principalmente o framework Dash, designado para a criação de sistemas web interativos. A visualização principal é de um mapa de Belo Horizonte, com ferramentas de movimentação e seleção de regiões retangulares. A partir desta, é possível desenhar a área sobre a qual deseja-se realizar a busca ortogonal dos estabelecimentos, utilizando a estrutura da Kd Tree demonstrada anteriormente

Os dados são carregados e a árvore de consulta é construída

```
points = kdtree.parse_csv('dados.csv')
tree = kdtree.KdTree(points)
bares_info = kdtree.parse_bares_completos_csv('bares.csv')
```

Utilizando o mapa interativo da cidade de BH, o usuário pode desenhar retângulos, que filtrarão e exibirão todos os bares e restaurantes cadastrados dentro daquele espaço

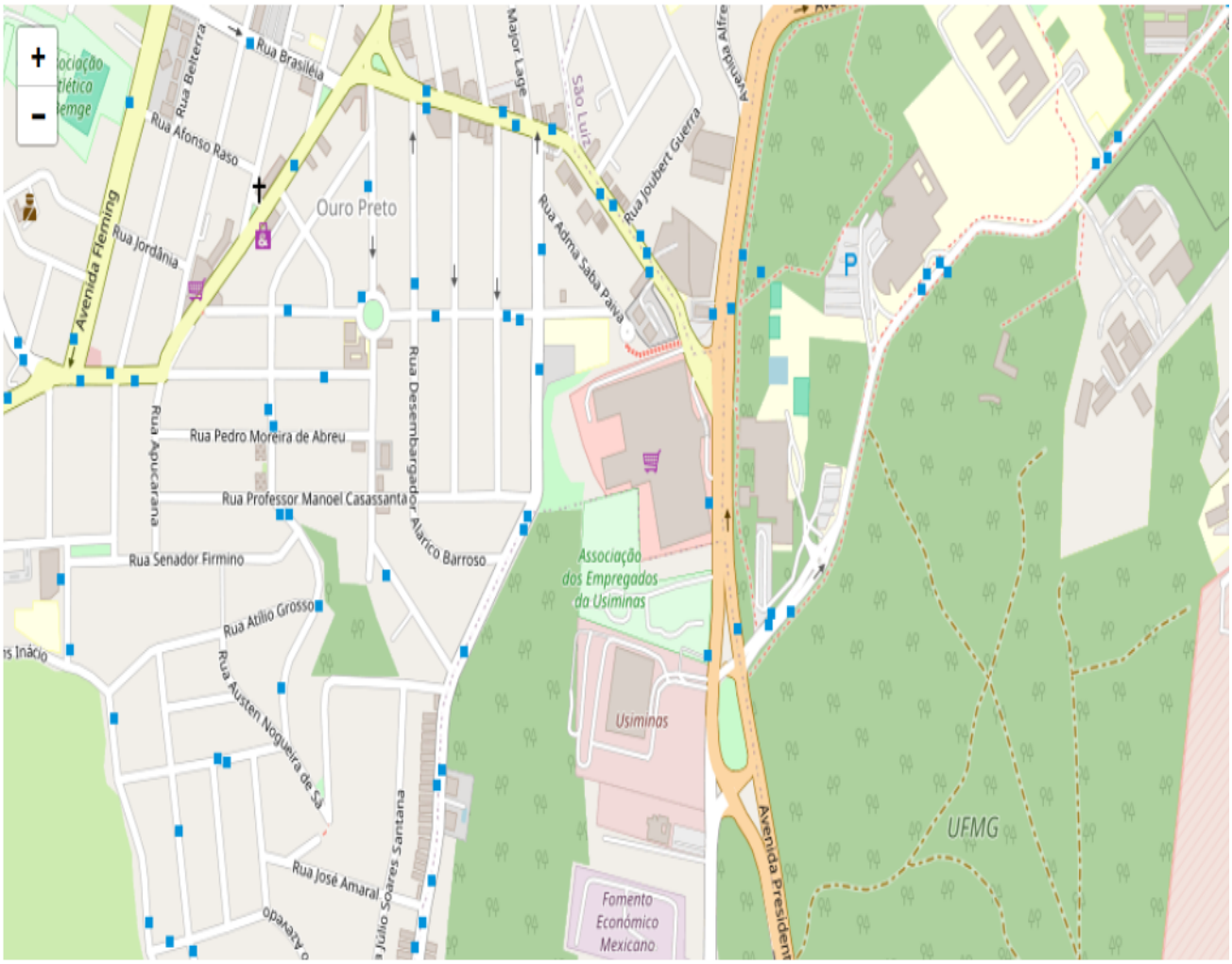
```
def filter_points(geojson):
    if not geojson or not geojson.get("features"):
        return [point_to_dict(p) for p in points], make_markers(points)
    coords = geojson["features"][0]["geometry"]["coordinates"][0]
    lats = [pt[1] for pt in coords]
    lons = [pt[0] for pt in coords]
    lat_min, lat_max = min(lats), max(lats)
    lon_min, lon_max = min(lons), max(lons)
    ll = kdtree.Point(None, lon_min, lat_min)
    ur = kdtree.Point(None, lon_max, lat_max)
    filtered = tree.search(kdtree.Rectangle(ll, ur))

    return [point_to_dict(p) for p in filtered], make_markers(filtered)
```

Além disso, ao passar o cursor sobre certos pontos, são exibidas informações referentes àquele estabelecimento na campanha Comida di Buteco

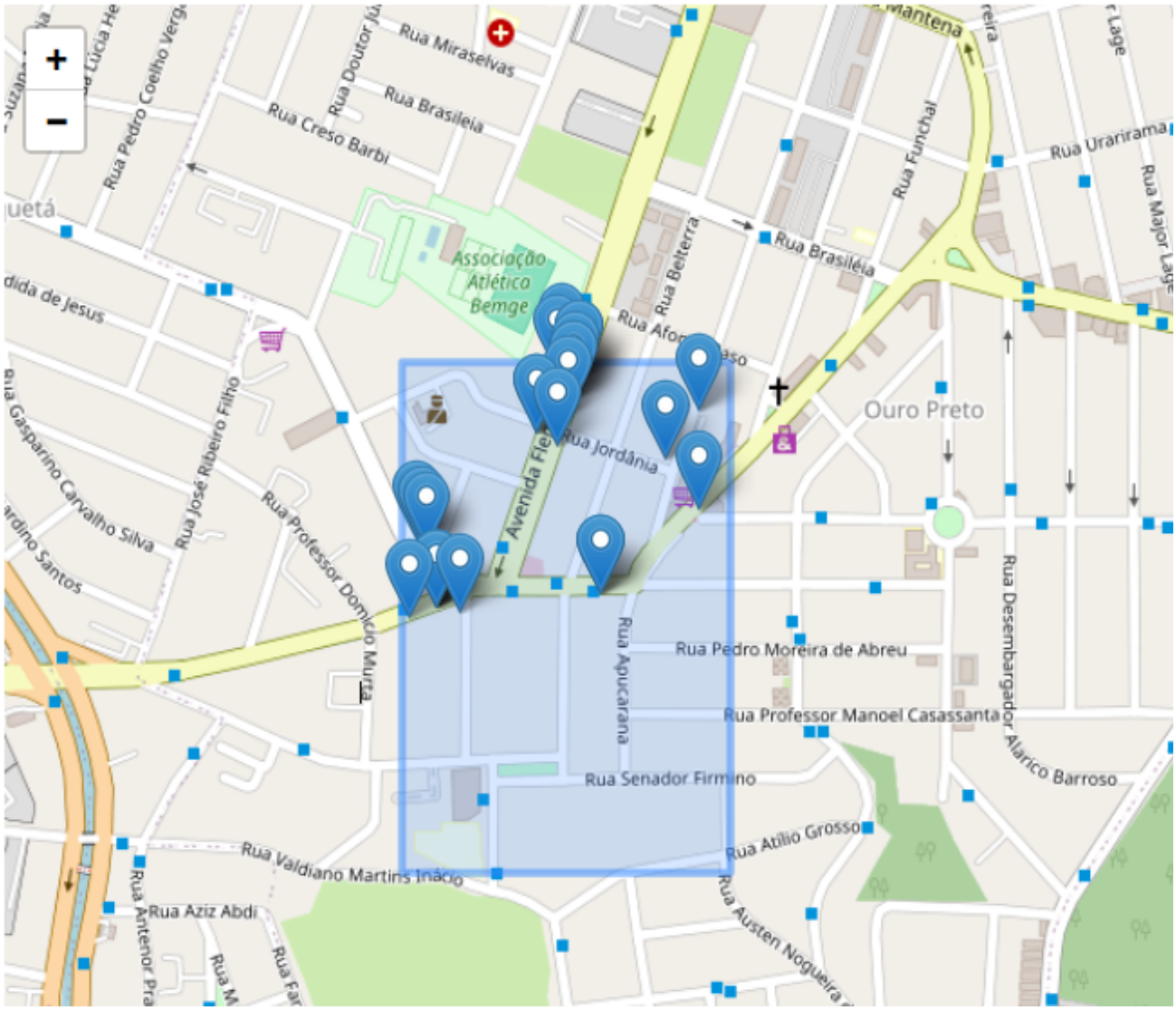
```
def get_bar_extra_info(point: kdtree.Point) -> str:
    bar_id = getattr(point.data, "id_ativ_econ_estabelecimento", None)
    info = bares_info.get(bar_id)
    if info:
        return (
            f"<b>{info['Nome']}</b><br>"
            f"<a href=\"{info['Link Detalhes']}\" target=\"_blank\">Ver
                detalhes</a><br>"
            f"<b>Prato:</b> {info['Nome Petisco']}<br>"
            f"<b>Descricao:</b> {info['Descricao']}<br>"
            f"<b>Endereco:</b> {info['Endereco']}<br>"
            f"<img src=\"{info['Link Imagem']}\" width=\"150\">"
        )
    return point.data.name
```

Buscador de bares e restaurantes BH



Nome	Data de início	Possui alvará

Buscador de bares e restaurantes BH



Buteco do Rod

Ver detalhes

Prato: Canela di Butequêro

Descrição: Cambito suíno ao molho de cerveja preta, acima de uma polenta temperada com páprica defumada e espinafre,

Endereço: Rua José Moura Peçanha, 12 | Ouro Preto, Belo Horizonte – MG