

Relatório - TP 1: Consulta Interativa de Estabelecimentos com KD-Tree e Dash Leaflet

DCC207: Algoritmos II, UFMG

Ester Sara Assis 2021031785
Júlia Paes de Viterbo 2021032137

Link de acesso ao Github Pages: <https://estersassis.github.io/dcc207-tp1-kdtree/>

0. Introdução

Este trabalho tem como objetivo explorar e aplicar os conceitos de geometria computacional, utilizando-se de árvores k-dimensionais (*KD-Trees*) para realizar buscas ortogonais em conjuntos de pontos. O resultado consiste em um sistema interativo para consulta de estabelecimentos comerciais em Belo Horizonte, mais especificamente, bares e restaurantes.

Nesse contexto, foram utilizados dados disponibilizados pela Prefeitura de Belo Horizonte para a seleção desses estabelecimentos, cujas coordenadas geográficas foram extraídas por meio do acesso à API do *OpenStreetMaps*¹. Esses estabelecimentos foram dispostos em um **mapa interativo** que permite ao usuário filtrar sua busca por uma **seleção espacial retangular** que delimita pontos para os quais são mostradas informações complementares em uma **tabela na parte inferior da tela**. Seguindo as orientações da tarefa, as *KD-Trees* foram utilizadas para retornar os pontos dentro da área selecionada de forma eficiente.

Para tanto, o projeto exige tanto a compreensão teórica dos algoritmos de geometria computacional quanto a aplicação prática de técnicas para a construção de uma ferramenta interativa à base de estruturas de dados específicas. O sistema foi desenvolvido em *Python*, em acordo com a especificação do trabalho, e suas bibliotecas associadas, sobretudo *Dash* e *Dash-Leaflet*, para garantir interatividade e facilidade de uso.

Como extra, foi implementada a funcionalidade de que, ao clicar em um estabelecimento que faz parte do festival Comida di Buteco, são exibidas informações sobre o prato concorrente desse local em 2025. Os requisitos (*requirements.txt*) e especificações necessários para rodar o projeto estão descritos no arquivo *README.md*.

1. Desenvolvimento

O projeto ficou com a seguinte divisão de arquivos:

```
└── app
    ├── __init__.py
    └── callbacks.py
```

```

geojson_utils.py
layout.py
server.py
data
└── bares_restaurantes_geocodificados.csv
    └── comida_di_buteco_2025.csv
        └── comida_di_buteco_corrigido.csv
kdtree
└── __init__.py
    └── kdtree.py
notebooks
└── data_processing.ipynb
scripts
└── __init__.py
    └── scrap_buteco.py
tests
└── __init__.py
    └── test_kdtree.py
.gitignore
index.html
LICENSE
Makefile
README.md
requirements.txt

```

A seguir, descrevem-se as principais decisões de projeto e a estrutura do sistema, separando a lógica em três camadas principais: pré-processamento, estrutura de dados e interface interativa. A funcionalidade extra é melhor detalhada em uma quarta subseção, mas foi implementada junto dos arquivos das camadas principais.

a. Pré-processamento de dados

```

notebooks
└── data_processing.ipynb
data
└── bares_restaurantes_geocodificados.csv

```

A partir dos dados de estabelecimentos em Belo Horizonte disponibilizados pela prefeitura², foi necessário primeiro selecionar os estabelecimentos da natureza correta. Para tanto, o arquivo *data_processing.ipynb*, na pasta *notebooks* realiza as tarefas de:

- seleção de atributos: são mantidos apenas os campos necessários para visualização: nome, endereço, data de abertura e presença de alvará;

```

Index(['ID_ATIV_ECON_ESTABELECIMENTO', 'CNAE_PRINCIPAL',
       'DESCRICAO_CNAE_PRINCIPAL', 'CNAE', 'DATA_INICIO_ATIVIDADE',
       'NATUREZA_JURIDICA', 'PORTE_EMPRESA', 'AREA_UTILIZADA', 'IND_SIMPLES',
       'IND_MEI', 'IND_POSSE_ALVARA', 'TIPO_UNIDADE', 'FORMA_ATUACAO',
       'DESC_LOGRADOURO', 'NOME_LOGRADOURO', 'NUMERO_IMOVEL', 'COMPLEMENTO',
       'NOME_BAIRRO', 'NOME', 'NOME_FANTASIA', 'CNPJ', 'GEOMETRIA'],
      dtype='object')
Registros iniciais: 534880

```

Figura 1: campos de descrição iniciais dos estabelecimentos.

```
df_final = df_final[['NOME', 'DATA_INICIO_ATIVIDADE', 'IND_POSSE_ALVARA', 'ENDERECO']]
```

Figura 2: filtragem de campos.

- filtragem por CNAE: apenas estabelecimentos com descrição relacionada a *bares* ou *restaurantes* são mantidos (de 534880 registros iniciais - ver Figura 1 -, 13801 restaram após essa filtragem);

```
keywords = ["BAR", "RESTAURANTE", "BARES", "RESTAURANTES"]  
filter = df['DESCRICAO_CNAE_PRINCIPAL'].apply(lambda x: any(k in x for k in keywords))  
df_final = df[filter].copy()
```

Figura 3: filtragem por CNAE.

- geocodificação: endereços são convertidos em coordenadas geográficas (latitude e longitude) com a API do *OpenStreetMap* (os valores nulos, como os mostrados na Figura 3, foram removidos);

```
df_final[['LATITUDE', 'LONGITUDE']] = df_final['ENDERECO'].apply(convert_coordinates)  
print(df_final.head())
```

Figura 4: conversão com coordenadas geográficas.

	NOME	DATA_INICIO_ATIVIDADE	ENDERECO
17	APARECIDA MARIA DE SOUZA	01-07-1993	
19	PIZZARIA E CHURRASCARIA VARANDA	15-10-1993	
29	RABBIT BURGER	02-05-1994	
172	TATU REI DO ANGU A BAHIANA	24-11-1993	
196	ROD BITS COMERCIO E REPRESENTACAO LTDA	01-06-1994	
	IND_POSSE_ALVARA		
17	NÃO RUA DESEMBARGADOR REIS ALVES, 90 - BAIRRO DAS ...		
19	NÃO RUA LUIZ PONGELUPE, 290 - CARDOSO , Belo Horiz...		
29	NÃO AVE RESSACA, 118 - PADRE EUSTACIO , Belo Hori...		
172	SIM RUA DESEMBARGADOR RIBEIRO DA LUZ, 135 - BARREI...		
196	NÃO AVE ELIAS ANTONIO ISSA, 288 - LETICIA , Belo H...		
	LATITUDE	LONGITUDE	
17	-19.962039	-44.000033	
19	NaN	NaN	
29	NaN	NaN	
172	-19.973201	-44.014192	
196	NaN	NaN	

Figura 5: *head* do *DataFrame* criado com os estabelecimentos filtrados + geolocalização.

- armazenamento do resultado salvo como *.csv*, *bares_restaurantes_geocodificados.csv*, cujos dados são usados para a construção da árvore *KD-Tree*.

Tendo esses dados filtrados e separados, é hora de construir a árvore bidimensional (dimensões: latitude e longitude), que será usada para permitir buscas eficientes por pontos dentro de uma região retangular no mapa definida pelo usuário, como especifica o trabalho.

b. Estrutura de dados (*KD-Tree*)

```
kd-tree
| __init__.py
| kd-tree.py
```

Para a estruturação interna do mecanismo de busca ortogonal, foi implementada uma árvore 2-dimensional, descrita em *kd-tree.py*, na pasta de mesmo nome, com as seguintes decisões de projeto:

- os pontos são armazenados como objetos *Point*, contendo posição (as coordenadas extraídas na etapa anterior) e metadados: nome do estabelecimento (*string*), indicador de existência de alvará (*bool*), data de início de atividade (tipo *data*) e endereço (*string*);

```
class Point:
    def __init__(self, point: tuple, name: str, alvara: str, date: str, address: str):
        self.point = point
        self.name = name
        self.alvara = alvara
        self.date = date
        self.address = address
```

Figura 6: definição de *Point*.

Cada nó da árvore possui um objeto desse tipo *Point* e ponteiros para os nós da esquerda e da direita.

```
class KDNode:
    def __init__(self, point: Point):
        self.point = point
        self.left = None
        self.right = None
```

Figura 7: definição do nós.

- a construção recursiva da árvore (função *build*) é feita por meio de chamada recursiva na lista de pontos *Point* alternando o eixo (x/y) para gerar balanceamento;

A cada chamada, verifica-se se a lista de pontos está vazia. Se sim, retorna-se *None*, indicando uma subárvore vazia. Caso haja pontos, escolhe-se um eixo de divisão com base na profundidade atual: se *depth % 2 == 0*, usa longitude (eixo x), se *depth % 2 == 1*, usa latitude (eixo y). Essa alternância permite dividir o plano como um “quebra-cabeças binário”, variando o critério a cada nível da árvore.

Dessa forma, a lista de pontos é ordenada com base no eixo escolhido e o ponto central (mediano) da lista é escolhido para ser o nó atual da árvore. Esse nó será a raiz de uma subárvore que contém: à esquerda, pontos “menores” que o mediano (no eixo atual), e à direita, pontos “maiores”.

O processo termina quando cada sublista tem 0 ou 1 elemento, e a árvore resultante é uma estrutura binária balanceada que divide o plano em retângulos hierárquicos e, assim permite buscas muito mais rápidas do que uma varredura simples na lista, atingindo eficiência média de $O(\sqrt{n})$ por busca retangular quando executada a próxima função: `search()`. Isso é interessante pela grande quantidade de pontos utilizadas no trabalho, da ordem de milhares.

```
def build(self, points: list[Point], depth=0):
    if not points:
        return None

    axis = depth % self.k
    sorted_points = sorted(points, key=lambda p: p.point[axis])
    median_idx = len(sorted_points) // 2
    median_point = sorted_points[median_idx]

    node = KDNode(median_point)
    node.left = self.build(sorted_points[:median_idx], depth + 1)
    node.right = self.build(sorted_points[median_idx + 1:], depth + 1)

    return node
```

Figura 8: função `build()`.

- a busca eficiente de todos os pontos dentro de um retângulo geográfico (`search()`, usada na interação com o mapa).

O objetivo é retornar todos os pontos dentro de uma área retangular `region`, passada como parâmetro para `search()` e delimitada por latitudes e longitudes $[(lon_{min}, lat_{min}), (lon_{max}, lat_{max})]$, como mostra a Figura 9.

```
(xmin, ymin), (xmax, ymax) = region
```

Figura 9: como a região delimitada é explicitada dentro de `search()`.

Começando da raiz da árvore, verifica-se se esse ponto está dentro da região explicitada. Se sim, adiciona-se o ponto aos resultados. Ao decidir qual próximo ramo da árvore explorar, só se entra no filho esquerdo se o ponto atual é maior ou igual ao limite inferior da região no eixo `cd` - ver Figura 10. No direito, só se entra se o ponto atual está abaixo do limite superior da região no eixo atual.

```
if node.left and p.point[cd] >= region[0][cd]:
    _search(node.left, depth + 1)
if node.right and p.point[cd] <= region[1][cd]:
    _search(node.right, depth + 1)
```

Figura 10: condições de visitação de `search()`.

A busca segue recursivamente para os ramos possivelmente relevantes, ignorando ramos que não podem conter nenhum ponto dentro da região.

Testes automatizados em `test_kdtree.py`, na pasta `tests`, garantem o funcionamento correto da estrutura, incluindo casos de borda, busca vazia e região total. Esses testes serão melhor discutidos na seção 2. deste documento.

c. Interface interativa (*Dash + Leaflet*)

```
app
|__ __init__.py
|__ callbacks.py
|__ geojson_utils.py
|__ layout.py
|__ server.py
```

O *front-end* do sistema foi construído com *Dash* e *dash-leaflet* nos arquivos explicitados abaixo, segundo se explica:

- *layout.py*: define o mapa com a ferramenta de seleção retangular e a tabela de resultados na parte inferior da tela (ver Figura 11), além do botão de “Restaurantes Comida di Buteco” - que aciona a tabela correspondente aos estabelecimentos que fazem parte do festival (Figuras 23, 24 e 25, melhor explicadas na seção 1.d.);

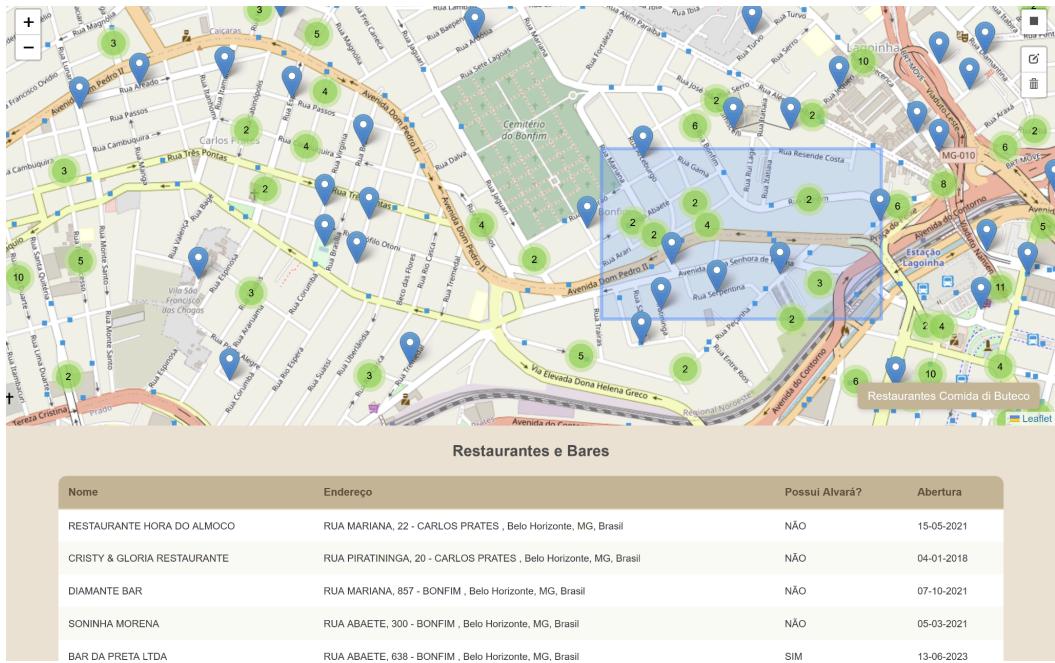


Figura 11: *front-end* do projeto mostrando ferramenta de seção retangular.

Ao acessar o projeto, a tela do usuário se assemelha à Figura 12, em se podem ver agrupamentos de estabelecimentos representados por círculos com números dentro (21, 6, 32 etc), os quais indicam as quantidades de bares e restaurantes de BH resumidos pelo círculo. A tabela na parte inferior da tela (Figura 13) lista todos os estabelecimentos

presentes na base filtrada nas etapas anteriores, até o usuário fazer uma seleção espacial, cuja funcionalidade é explicada mais adiante.

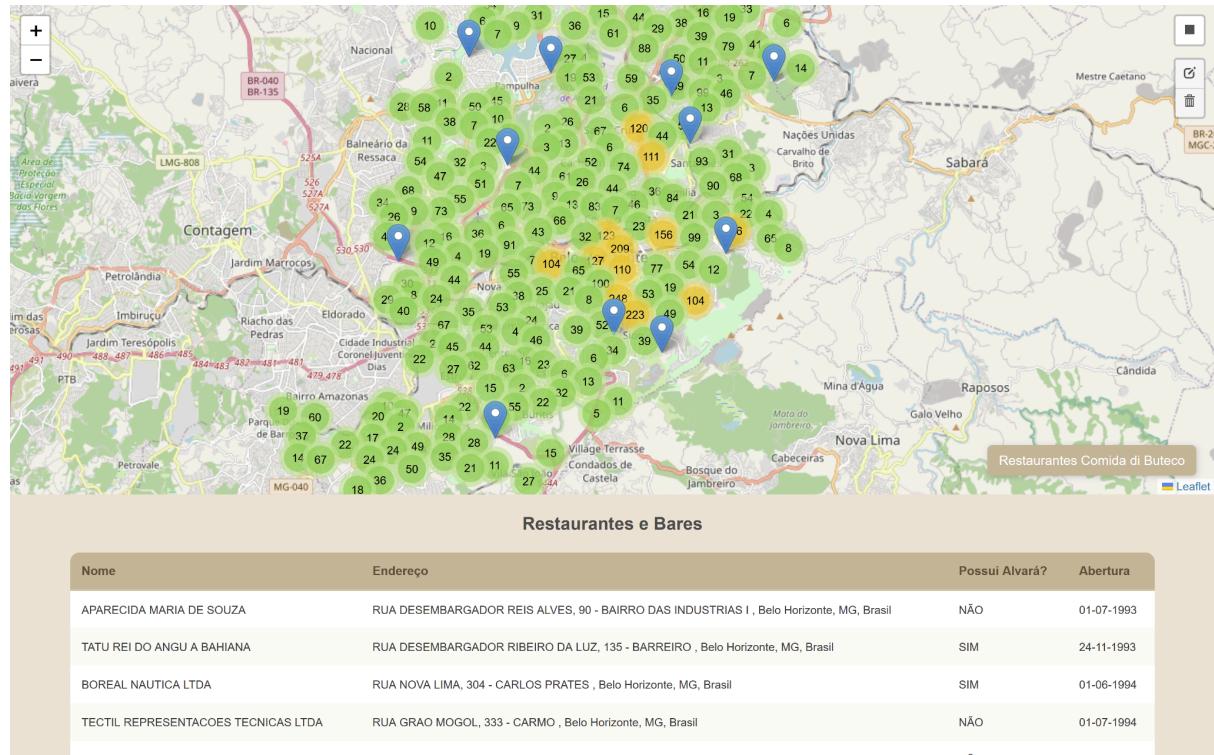


Figura 12: tela de abertura da aplicação.

Restaurantes e Bares			
Nome	Endereço	Possui Alvará?	Abertura
APARECIDA MARIA DE SOUZA	RUA DESEMBARGADOR REIS ALVES, 90 - BAIRRO DAS INDUSTRIAS I , Belo Horizonte, MG, Brasil	NÃO	01-07-1993
TATU REI DO ANGU A BAHIANA	RUA DESEMBARGADOR RIBEIRO DA LUZ, 135 - BARREIRO , Belo Horizonte, MG, Brasil	SIM	24-11-1993
BOREAL NAUTICA LTDA	RUA NOVA LIMA, 304 - CARLOS PRATES , Belo Horizonte, MG, Brasil	SIM	01-06-1994
TECTIL REPRESENTACOES TECNICAS LTDA	RUA GRAO MOGOL, 333 - CARMO , Belo Horizonte, MG, Brasil	NÃO	01-07-1994
AECIO CARLOS MIGUEL LEMOS 99648725691	RUA LUZIA SALOMAO, 50 - MANTIQUEIRA , Belo Horizonte, MG, Brasil	NÃO	15-12-2020

Figura 13: em detalhe, tabela inicial com todos os pontos do mapa.

A navegação no mapa ocorre pela interação por meio de clique-e-arraste com o mouse sobre a interface, e o usuário ainda pode dar *zoom in* e *zoom out* para mudar a escala de visualização do mapa, reorganizando os clusters de estabelecimentos agrupados:

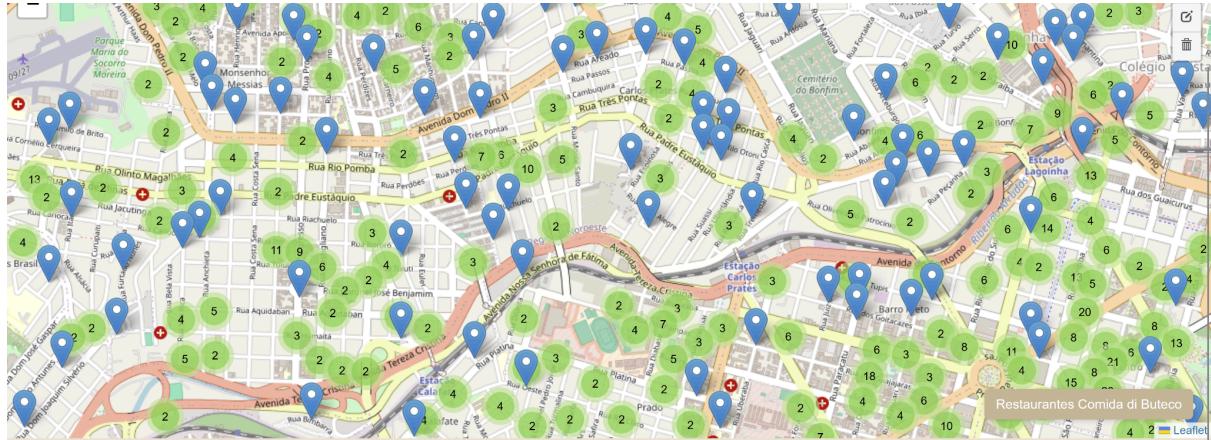


Figura 14: *zoom in* em relação à tela inicial.

- *geojson_utils.py*: define função que converte o *DataFrame* geocodificado para um objeto *geojson* (formato mais adequado para a integração);

```

features = [
    {
        "type": "Feature",
        "geometry": {
            "type": "Point",
            "coordinates": [row["LONGITUDE"], row["LATITUDE"]],
        },
        "properties": {
            "NOME": row["NOME"],
            "ENDERECO": row["ENDERECO"],
            "popup": f"<b>{row['NOME']}</b><br>{row['ENDERECO']}"
        }
    }
    for i, row in df.iterrows()
]
geojson = {"type": "FeatureCollection", "features": features}

```

Figura 15: formato geojson.

- *callbacks.py*: processa o evento de seleção no mapa, extrai a região e busca os pontos via *KD-Tree*;

Quando o usuário desenha um retângulo no mapa, o *callback* em *callbacks.py* captura os limites da área e retorna os pontos dela.

```

lons = [p[0] for p in coords]
lats = [p[1] for p in coords]
lon_min, lon_max = min(lons), max(lons)
lat_min, lat_max = min(lats), max(lats)

pontos = tree.search([(lon_min, lat_min), (lon_max, lat_max)])
return [{"NOME": p.name, "ENDERECO": p.address, "IND_POSSE_ALVARA": p.alvara, "DATA_INICIO_ATIVIDADE": p.date} for p in
pontos]

```

Figura 16: extração de busca dos pontos na área selecionada.

Para fazer a seleção da área retangular, o usuário clica no ícone mais superior no canto superior direito da tela apresentado na Figura 17:

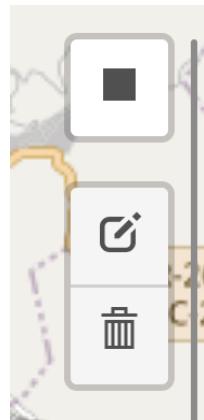


Figura 17: ferramentas de seleção.

Isso permite que a área seja desenhada com o mouse e a lista de estabelecimentos da tabela muda para exibir os locais delimitados.



Figura 18: seleção de área retangular.

Uma vez selecionada a área, clicando no botão central das ferramentas de seleção, o usuário pode editar o retângulo desenhado, movendo-o ou arrastando suas bordas para desenharem novos limites. Uma vez terminada a edição, o usuário pressiona “Save”, ou “Cancel” para abandonar a operação.

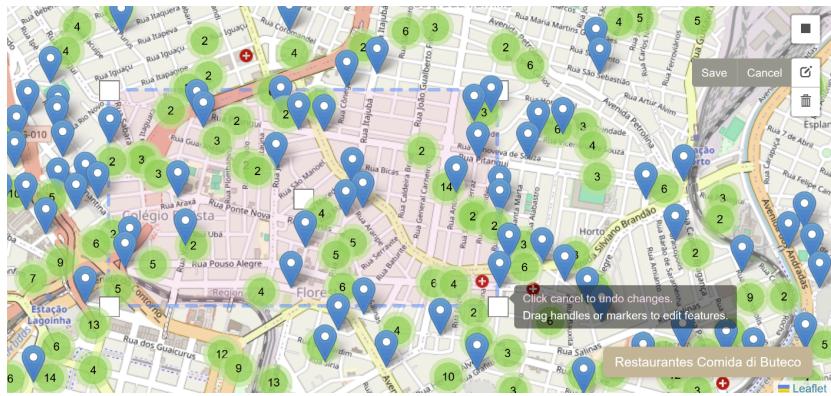


Figura 19: modo de edição (o retângulo desenhado fica pontilhado e as áreas com interatividade de edição aparecem como quadrados brancos).

Para apagar a seleção feita, pressiona-se o botão mais inferior dentre as ferramentas de seleção, o que abre uma aba de opções:

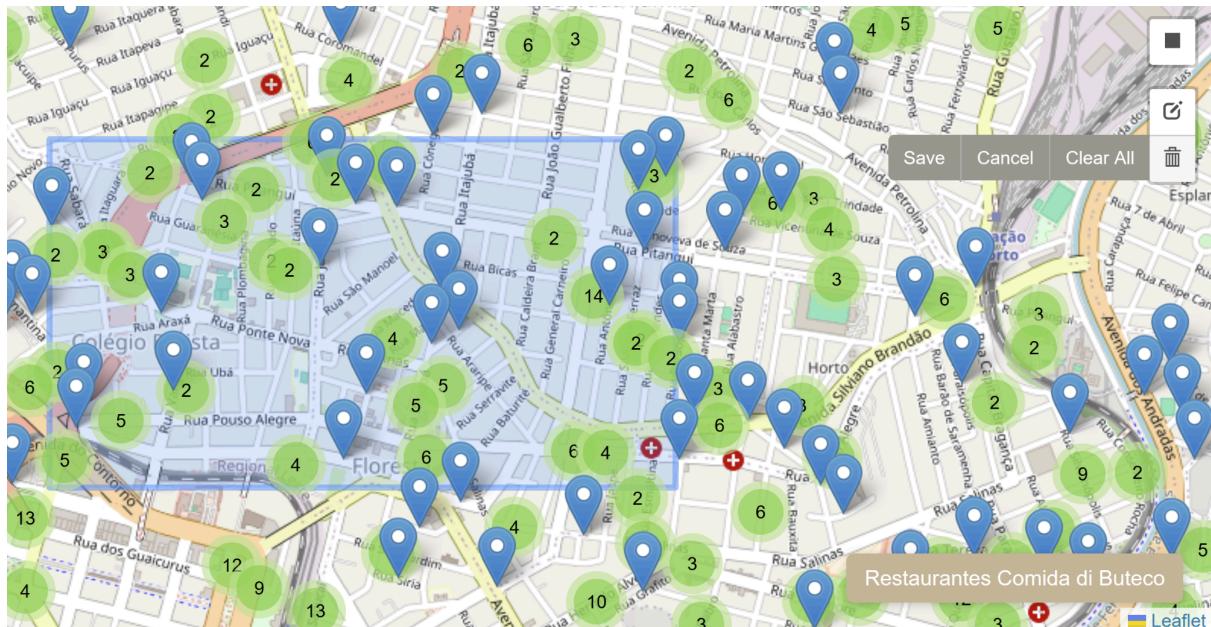


Figura 20: aba de opções da opção de deletar seleções.

Uma vez apertado o botão, clicando-se em qualquer retângulo desenhado ele desaparece da tela, mostrando que foi pré deletado. Clicando-se em seguida em “Save”, a deleção é confirmada; clicando-se em “Cancel”, a deleção não é concluída e o(s) retângulo(s) volta(m) a aparecer; clicando-se em “Clear All”, todos os retângulos são apagados. A ferramenta aceita que mais de um retângulo seja desenhado, mas apenas a seleção do primeiro retângulo tem influência sobre a tabela na porção inferior da tela, i.e. para consultar uma nova área de interesse, é necessário fazer a deleção de todas as áreas anteriormente selecionadas e desenhar uma nova, ou editar a primeira seleção feita para que compreenda a nova área desejada.

- por fim, o arquivo `server.py` integra todos os módulos, instancia a árvore e inicializa o servidor, mostrando o mapa pronto para uso e exploração.

```

tree = KDTree([
    Point(
        point=(row["LONGITUDE"], row["LATITUDE"]),
        name=row["NOME"],
        alvara=row["IND_POSSE_ALVARA"],
        date=row["DATA_INICIO_ATIVIDADE"],
        address=row["ENDERECO"]
    ) for _, row in df.iterrows()
])

```

Figura 21: instanciação da árvore (itera pelo *DataFrame* construído a partir do *.csv* resultado da seção 1.a., de pré-processamento).

d. Funcionalidade extra (Comida di Buteco)

```

scripts
| __init__.py
| scrap_buteco.py
data
| comida_di_buteco_2025.csv
| comida_di_buteco_corrigido.csv

```

O arquivo *scrape_buteco.py* da pasta *scripts* é responsável por coletar dados dos bares participantes do festival Comida di Buteco 2025 na cidade de Belo Horizonte. Trata-se de um *script* que acessa diretamente o site oficial do evento (parte mostrada na Figura 22), percorre todas as páginas da listagem de bares participantes e extrai os links individuais de cada estabelecimento. Em seguida, acessa cada uma dessas páginas e coleta informações relevantes como o nome do bar, o prato concorrente e sua descrição, endereço, telefone de contato e a URL da imagem do prato.

```

while True:
    url = f"{BASE_URL}/butecos/belo-horizonte/" if page == 1 else f"{BASE_URL}/butecos/belo-horizonte/page/{page}/"
    print(f"Página {page}: {url}")
    res = requests.get(url, headers=headers)
    soup = BeautifulSoup(res.text, "html.parser")
    items = soup.select("div.item")

```

Figura 22: coleta de todos os sites dos bares participantes.

Essas informações são organizadas em um dicionário e, ao final do processo, são reunidas em um *DataFrame* e exportadas como um arquivo *.csv* (*comida_di_buteco_2025.csv*). Tais dados são reunidos na tabela específica de bares participantes do festival, que é acionada por um botão disponível na interface, logo acima da tabela fixa da parte inferior:



Figura 23: botão que exibe a lista de bares do festival.

Selecionando qualquer bar da lista, um ícone com o logo do Comida di Buteco indica onde no mapa o bar se localiza e, clicando em cima do ponto, informações sobre o prato concorrente por esse estabelecimento são exibidas.

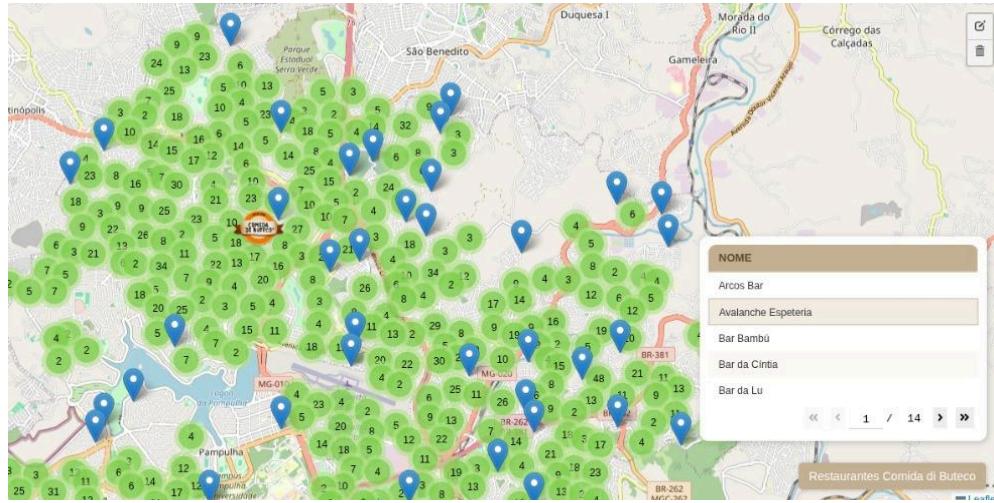


Figura 24: seleção de bar da lista e destaque dele no mapa.



Figura 5: exemplo de exibição de prato concorrente.

O código que estrutura o retorno dos dados da linha selecionada e o pop-up com as informações do prato concorrente estão junto da lógica de retorno da árvore bidimensional, em *callbacks.py*.

2. Experimentos

```

tests
| __init__.py
| test_kdtree.py

```

Para avaliar o correto funcionamento do sistema foram feitos testes automatizados e testes funcionais manuais.

a. Testes automatizados da *KD-Tree*

O arquivo *test_kdtree.py* implementa testes com a estrutura criada a fim de verificar seu correto funcionamento. Por meio da definição de seis pontos fictícios e da seleção de um, alguns ou nenhum deles, observou-se o retorno da busca, que, em todos os casos, resultaram em *assertions* verdadeiras. Primeiro, testamos se a árvore retornaria um único ponto, depois, pontos de borda, todos os pontos e, por fim, testamos se, definida um área sem nenhum ponto, a resposta seria vazia.

```
pontos = [
    Point(point=(-43.9, -19.9), name="Restaurante Dijkstra Gourmet", alvara="NÃO", date="2018-01-01", address="Rua A"),
    Point(point=(-43.91, -19.91), name="Buteco do Kruskal", alvara="SIM", date="2019-02-01", address="Rua B"),
    Point(point=(-43.92, -19.92), name="Café P vs NP", alvara="NÃO", date="2020-03-01", address="Rua C"),
    Point(point=(-43.93, -19.93), name="Petiscos do Backtrack", alvara="SIM", date="2021-04-01", address="Rua D"),
    Point(point=(-43.91, -19.89), name="Bubble Beer Bar", alvara="NÃO", date="2022-05-01", address="Rua E"),
    Point(point=(-43.89, -19.91), name="Lanchonete QuickByte", alvara="SIM", date="2023-06-01", address="Rua F")]
]
```

Figura 26: definição de pontos fictícios (com licença poética relacionada com a disciplina).

```
def test_only_one_point(tree):
    region = [(-43.915, -19.915), (-43.91, -19.905)]
    assert names(tree.search(region)) == ["Buteco do Kruskal"]

def test_border_points(tree):
    region = [(-43.91, -19.91), (-43.89, -19.88)]
    assert names(tree.search(region)) == sorted(["Lanchonete QuickByte", "Buteco do Kruskal", "Bubble Beer Bar", "Restaurante Dijkstra Gourmet"])

def test_all_points(tree):
    region = [(-43.94, -19.95), (-43.88, -19.88)]
    assert names(tree.search(region)) == sorted([p.name for p in pontos])

def test_none_point(tree):
    region = [(-44.0, -20.0), (-43.95, -19.96)]
    assert names(tree.search(region)) == []
```

Figura 27: definição dos testes.

b. Testes funcionais

Testes manuais com diferentes regiões do mapa verificaram a correta atualização da tabela com os estabelecimentos da área, bem como o correto funcionamento da exibição dos pratos do Comida di Buteco. As Figuras 18, 24 e 25 ilustram esses testes.

3. Conclusão

Ao final do desenvolvimento da tarefa, foi possível observar que a geometria computacional é bastante relevante para aplicações usadas cotidianamente. com respeito à estrutura de dados pedida, a árvore *KD* demonstrou ser eficaz para resolver o problema de forma escalável, garantindo respostas rápidas mesmo com grandes volumes de dados. Além disso, o uso de bibliotecas modernas como *Dash*, *Dash Leaflet* e ferramentas de geocodificação tornou possível a integração entre visualização interativa e estruturas

clássicas de dados, evidenciando a relevância prática desses algoritmos no contexto atual. Desenvolver todas as funções disponibilizadas nessas bibliotecas por nós mesmas seria uma tarefa muito mais custosa.

A parte mais desafiadora do trabalho foi a de ajustar os dados disponíveis em diversos formatos e fontes para estruturas mais amigáveis ao trabalho computacional. Por exemplo, a extração dos links das páginas dos bares participantes do Comida de Buteco, assim como a maioria das funções para conseguir metadados, exigiu pesquisa e adaptação aos objetivos da tarefa. Em paralelo, o tempo de processamento da geolocalização dos estabelecimentos foi de aproximadamente 4 horas, o que também exigiu espera.

Por fim, pode-se concluir que o trabalho demandou quantidade razoável de tempo para ser desenvolvido, sobretudo pela necessidade de integração das várias partes que o compunham. Foi satisfatório o resultado a que chegamos.

4. Referências

1. OpenStreetMaps. Link: <https://www.openstreetmap.org/#map=4/-15.13/-53.19>
2. Portal de Dados Abertos. Link: <https://dados.pbh.gov.br/dataset/atividades-economicas1>
3. Site oficial do Comida di Buteco. Link: <https://comidadibuteco.com.br/>
4. Beautiful Soup: Build a Web Scraper With Python. Link:
<https://realpython.com/beautiful-soup-web-scraping-with-python/>
5. Dash Leaflet Documentation. Link: <https://www.dash-leaflet.com/>
6. Dash Python User Guide. Link: <https://dash.plotly.com/>
7. Advanced Callbacks. Link: <https://dash.plotly.com/advanced-callbacks>
8. Search and Insertion in K Dimensional tree. Link:
<https://www.geeksforgeeks.org/search-and-insertion-in-k-dimensional-tree/>