UDESC UNIVERSIDADE DO ESTADO DE SANTA CATARINA

UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC

CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT

RELATÓRIO

Joinville, 2025

UDESC UNIVERSIDADE DO ESTADO DE SANTA CATARINA

SUMÁRIO

1. IDENTIFICAÇÃO DA EQUIPE

Sofia Petrykowski Soares e Elane Souza de Oliveira

2. IDENTIFICAÇÃO DA TAREFA

A Teoria dos Grafos é uma área essencial da Ciência da Computação voltada à representação e análise de relações entre entidades, tendo aplicações diretas em redes de comunicação, análise de dados, transporte, biologia computacional e sistemas complexos. Diante deste contexto, este trabalho teve como objetivo aplicar os princípios da Teoria dos Grafos ao implementar, analisar e visualizar grafos em linguagem C e Python, a partir de um conjunto de dados contendo 150 vértices, cada um descrito por quatro atributos numéricos. A conexão entre os vértices foi definida pela Distância Euclidiana Normalizada (DEN), sendo testados diferentes limiares (L = 0.0, 0.3, 0.5 e 0.9) para observar o comportamento estrutural do grafo e de seus componentes conexos. Ademais, para a representação computacional do grafo, optou-se pela utilização da matriz de adjacência, por sua clareza e eficiência na manipulação das conexões. Além disso, o programa identifica os componentes conexos do grafo gerado e permite a persistência dos dados em novos arquivos CSV, contendo os *clusters*, as coordenadas e os grafos para cada limiar, além de serem gerados histogramas e grafos 3D para cada limiar em formato PNG. A implementação foi dividida em duas partes principais: o núcleo computacional em C, responsável pela leitura, processamento e construção dos grafos; e o módulo de visualização em Python, voltado à geração de representações tridimensionais e histogramas dos resultados.

3. EXPLICAÇÃO CONCEITUAL SOBRE A SOLUÇÃO FORNECIDA

3.1. Implementação do grafo

3.1.1. Definição das Estruturas de Dados

O primeiro passo consistiu na definição das estruturas que representam os vértices e o grafo, sendo fundamental para representar o grafo e os vértices e manipular os dados do arquivo CSV.

Foi criada a estrutura *DadosVertice* Figura 1, que armazena os quatro atributos numéricos de cada vértice (equivalentes a coordenadas em um espaço 4D) e o identificador do cluster ao qual pertence, determinado após a análise de conectividade.

Figura 1 - Definição da estrutura utilizada.

```c
// Estrutura para os dados do vértice
typedef struct {
    double atributos[NUM_ATRIBUTOS]; // Os 4 atributos (X, Y, Z, W)
    int cluster_id; // Para armazenar o ID do componente conexo
    int especie_original;
} DadosVertice;
```

Fonte: Arquivo pessoal.

Cada vértice é tratado como um ponto em um espaço de quatro dimensões, e as relações entre eles são avaliadas com base em distâncias.

Em seguida, para representar o grafo de forma eficiente e simétrica, foi utilizada uma **matriz de adjacência** de tamanho 150×150, sendo uma escolha adequada para conjuntos de dados pequenos e densos, em que o custo de memória é compensado pela simplicidade de acesso.

3.1.2. Leitura e Normalização dos dados

Primeiramente ciou-se a função carregar_dados(), que é responsável pela leitura do arquivo CSV da base de dados “my_dataset.csv” contendo os 150 vértices. O processo inclui o tratamento de linhas vazias e verificação de formato. A função faz a abertura do arquivo, a leitura (linha por linha) e extrai os dados.

Após a leitura, as coordenadas são normalizadas entre 0 e 1 pela função normalizar_coordenadas(), garantindo que todos os atributos possuam a mesma escala. Essa etapa é fundamental para evitar que atributos de maior magnitude dominem o cálculo da distância euclidiana.

A normalização segue a transformação min–max, conforme a Figura 2:

Figura 2 - Equação utiliza.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Fonte: Arquivo pessoal.

3.1.3. Cálculo das Distâncias Euclidianas e Normalização (DEN)

Foi implementado o cálculo da distância euclidiana (DE) entre todos os pares de vértices obtidos. A distância entre dois vértices é obtida pela distância euclidiana no espaço 4D, utilizando a função calcular_distancia_euclidiana(), por meio da equação apresenta na Figura 3:

Figura 3 - Equação utilizada.

$$DE(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2 + (a_4 - b_4)^2}$$

Fonte: Arquivo pessoal.

Essas distâncias são calculadas para todos os pares de vértices, resultando em uma matriz de distâncias simétrica. A função na Figura 4 recebe dois vértices "v1" e "v2" e retorna a distância entre eles como um valor do tipo *double.* Para isso fez-se o uso de funções da biblioteca math.h.

Figura 4 - Função para calcular distância euclidiana.

```
double calcular_distancia_euclidiana(DadosVertice v1, DadosVertice v2) {
    double soma_quadrados = 0.0;
    for (int i = 0; i < NUM_ATRIBUTOS; i++) {
        soma_quadrados += pow(v1.atributos[i] - v2.atributos[i], 2);
    }
    return sqrt(soma_quadrados);
}
```

Fonte: Arquivo pessoal.

Em seguida, após o cálculo das distâncias euclidianas (DE), foi implementada a função responsável por calcular a distância euclidiana normalizada, onde as distâncias são normalizadas pela função normalizar_distancias(), produzindo a matriz de distâncias normalizadas (DEN), definida pela equação da Figura 5:

Figura 5 - Equação utilizada.

$$DEN_{ij} = \frac{DE_{ij} - DE_{min}}{DE_{max} - DE_{min}}$$

Fonte: Arquivo pessoal.

Onde DE_min e DE_max representam as menores e maiores distâncias entre quaisquer pares de vértices. Essa transformação permite comparar valores em uma escala padronizada entre 0 e 1.

3.1.4. Construção do grafo a partir dos Limiares

Nesta etapa, houve a criação das arestas entre os vértices com base na DEN. Assim, é possível conectar dois vértices se a DEN entre eles for menor ou igual ao limiar, o que indica uma proximidade espacial significativa entre os pontos, isto é, apenas os pares de vértices com proximidade espacial são conectados. A criação das arestas é feita pela função construir_grafo(). Para cada par de vértices, uma aresta é adicionada caso a distância

normalizada DENij ≤ L , onde L é o limiar de conexão. O objetivo é preencher a matriz_adj (0/1, simétrica) onde matriz_adj[i][j] = 1 se DEN[i][j] <= L, caso contrário 0, ademias o grafo é não-direcionado. A função inicializa matriz_adj com zeros (todas as posições) conforme o código da Figura 6:

Figura 6 - Função para construir o gráfico.

```c
// Função para construir o grafo (recebe DEN e L)
void construir_grafo(double matriz_den[NUM_VERTICES][NUM_VERTICES],
    int matriz_adj[NUM_VERTICES][NUM_VERTICES], double L) {
    // Inicializa a matriz de adjacências com 0
    for (int i = 0; i < NUM_VERTICES; i++) {
        for (int j = 0; j < NUM_VERTICES; j++) {
            matriz_adj[i][j] = 0;
        }
    }

    // Apenas na parte triangular superior (grafo não-direcionado)
    for (int i = 0; i < NUM_VERTICES; i++) {
        for (int j = i + 1; j < NUM_VERTICES; j++) {

            if (matriz_den[i][j] <= L) {
                matriz_adj[i][j] = 1;
                matriz_adj[j][i] = 1; // Grafo não-direcionado
            }
        }
    }
}
```

Fonte: Arquivo pessoal.

Esse método preenche o grafo na matriz, garantindo que a estrutura resultante represente as relações espaciais entre os vértices do grafo, ou seja, resulta em uma matriz binária. Isso permite as próximas análises, como a identificação de componentes conexos e a visualização tridimensional da estrutura grafo associada ao conjunto de dados. Além disso, a matriz de adjacência resultante é armazenada em arquivos separados no formato grafo_L_X_adj.csv para cada limiar.

3.2.Identificação dos componentes conexos

3.2.1. Análise Busca em Largura

A **Busca em Largura (Breadth-First Search - BFS)** é ideal para este tipo de análise por dois motivos principais:

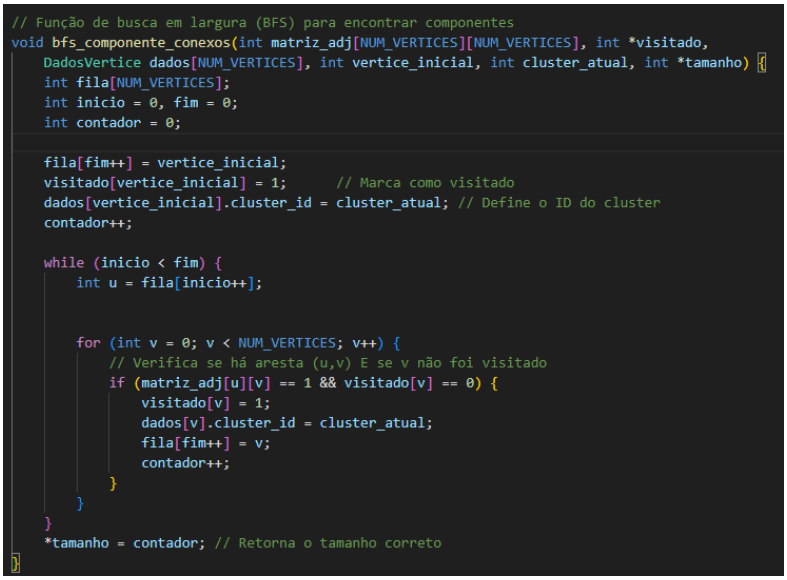
1) **Garantia de Descoberta:** O BFS garante que todos os nós acessíveis a partir de um vértice inicial sejam explorados **antes** de se mover para um novo componente não explorado.

2) **Identificação e Tamanho:** Adaptamos o BFS para, a partir de um nó ainda não visitado, marcá-lo e todos os seus vizinhos conectados com um novo **ID de Cluster**. Ao mesmo tempo, ele conta o número exato de vértices encontrados durante a busca, fornecendo o **tamanho do componente conexo**.

A identificação dos componentes conexos foi implementada nas funções bfs_componente_conexos() e analisar_componentes_conexos(). Elas servem para identificar e contar os componentes conexos (clusters) no grafo e depois armazenar o ID do cluster no vetor de dados e o tamanho de cada cluster. O algoritmo percorre a matriz de adjacência marcando vértices visitados e atribuindo um identificador de cluster para cada grupo conexo. O resultado é armazenado no campo cluster_id de cada vértice e no vetor tamanhos_componentes, que contém o tamanho de cada cluster identificado.

Conforme indicado no código presente na Figura 7, é feita uma bfs (busca em largura) a partir de vertice_inicial, marcando vértices visitados no array visitado, atribuindo o cluster_atual a dados[v].cluster_id para todos v do componente e retornando o tamanho do componente via ponteiro.

Figura 7 - Função de Busca em Largura



```c
// Função de busca em largura (BFS) para encontrar componentes
void bfs_componente_conexos(int matriz_adj[NUM_VERTICES][NUM_VERTICES], int *visitado,
    DadosVertice dados[NUM_VERTICES], int vertice_inicial, int cluster_atual, int *tamanho) {
    int fila[NUM_VERTICES];
    int inicio = 0, fim = 0;
    int contador = 0;

    fila[fim++] = vertice_inicial;
    visitado[vertice_inicial] = 1;      // Marca como visitado
    dados[vertice_inicial].cluster_id = cluster_atual; // Define o ID do cluster
    contador++;

    while (inicio < fim) {
        int u = fila[inicio++];


        for (int v = 0; v < NUM_VERTICES; v++) {
            // Verifica se há aresta (u,v) E se v não foi visitado
            if (matriz_adj[u][v] == 1 && visitado[v] == 0) {
                visitado[v] = 1;
                dados[v].cluster_id = cluster_atual;
                fila[fim++] = v;
                contador++;
            }
        }
    }
    *tamanho = contador; // Retorna o tamanho correto
}
```

Fonte: Arquivo pessoal.

Depois disso, passa pela função analisar_componentes_conexos percorrendo todos os vértices e, para cada vértice não visitado, chama bfs_componente_conexos e armazena o tamanho do componente no vetor tamanhos_componentes, depois retorna o número total de clusters (componentes conexos). Essa etapa é fundamental para compreender a estrutura do grafo, indicando o número de agrupamentos e sua distribuição de tamanhos para cada limiar L.

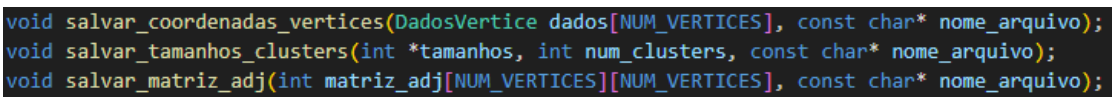
3.2.2. Persistência dos Dados em Arquivos CSV

Após a construção do grafo, os resultados do processamento são persistidos em arquivos CSV, permitindo análises externas e integração com o módulo Python. Os arquivos gerados incluem:

- grafo/grafo_L_X_adj.csv – matriz de adjacência do grafo;
- coordenadas/coord_L_X.csv – atributos normalizados e identificador de cluster;
- clusters/clusters_L_X.csv – tamanhos dos componentes conexos.

Sendo dessa forma separados em diferentes funções conforme a Figura 8. Essa separação modular facilita a importação dos dados em Python e outras ferramentas de análise.

Figura 8 - Cabeçalho das funções para salvar dados.



```c
void salvar_coordenadas_vertices(DadosVertice dados[NUM_VERTICES], const char* nome_arquivo);
void salvar_tamanhos_clusters(int *tamanhos, int num_clusters, const char* nome_arquivo);
void salvar_matriz_adj(int matriz_adj[NUM_VERTICES][NUM_VERTICES], const char* nome_arquivo);
```

Fonte: Arquivo pessoal.

3.2.3. Tridimensional e Geração de Histogramas

O código em Python foi desenvolvido com o objetivo de realizar análises e visualizações tridimensionais de grafos gerados a partir de matrizes de adjacência, bem como a representação gráfica da distribuição dos componentes conexos (clusters). O programa utiliza bibliotecas científicas e gráficas da linguagem Python, como NumPy, Matplotlib e NetworkX, para manipulação de dados, geração de grafos e criação de representações visuais que auxiliam na compreensão da estrutura de agrupamentos formados em diferentes limiares (thresholds) de conexão.

1) **O código faz uso das seguintes bibliotecas:**
    a) **matplotlib**: utilizada para geração de gráficos e visualizações tridimensionais.
    b) **numpy**: empregada para manipulação de matrizes e vetores numéricos.
    c) **networkx**: responsável pela criação e manipulação de grafos a partir de matrizes de adjacência.
    d) **random**: usada para gerar cores dinâmicas e aleatórias para os clusters.
    e) **mpl_toolkits.mplot3d**: módulo específico para permitir a criação de gráficos em três dimensões.

Figura 9 – Bibliotecas utilizadas em Python.

```python
import matplotlib as mpl
# Use 'Agg' para ambientes sem display ou 'TkAgg' se for local
mpl.use('TkAgg')

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import networkx as nx
import pandas as pd
import random # Importar para cores dinâmicas
```

Fonte: Arquivo pessoal.

2) **O programa está dividido em três partes principais:**
    a) **Definição de Funções Auxiliares** – responsáveis por gerar cores, plotar os grafos 3D e construir histogramas;

b) **Processamento Principal (Bloco if __name__ == "__main__":)** – responsável por carregar os arquivos, executar as funções e salvar as figuras;

c) **Saída de Resultados** – geração e salvamento automático de gráficos e histogramas em diretórios específicos.

3) **Função get_cores_dinamicas(num_cores)**

a) Esta função gera um conjunto de cores aleatórias no formato hexadecimal.

b) É utilizada para diferenciar visualmente os clusters no gráfico 3D.

c) A quantidade de cores geradas corresponde ao número de grupos (clusters) detectados no grafo.

Figura 10 - Função get_cores_dinamicas

```
15  def get_cores_dinamicas(num_cores):
16      """Gera um dicionário de cores aleatórias para os clusters."""
17      return [
18          '#%06X' % random.randint(0, 0xFFFFFF)
19          for _ in range(num_cores)
20      ]
21
```

Fonte: Arquivo pessoal.

4) **Função plot_grafo_3d_cluster(...)**

a) Responsável por gerar o gráfico tridimensional do grafo, onde cada vértice é colorido conforme o cluster ao qual pertence.

b) Etapas da função:

i) Converte a matriz de adjacência (matriz_adjacencias) em um grafo utilizando a função nx.from_numpy_array();

ii) Cria uma figura 3D com matplotlib;

iii) Atribui cores dinâmicas aos clusters e plota os vértices (pontos) conforme suas coordenadas tridimensionais;

iv) Desenha as arestas do grafo com baixa opacidade, para manter a clareza visual;

v) Adiciona rótulos, título, legenda e perspectiva;

vi) Salva o grafo em arquivo .png no diretório ../img/grafos3D/.

c) O resultado é uma visualização tridimensional que permite analisar a distribuição espacial dos clusters conforme o limiar de conexão utilizado.

5) **Função gerar_histograma(tamanhos_componentes, L)**

a) Esta função produz o histograma da distribuição dos tamanhos dos componentes conexos.
b) Cada barra do gráfico representa o número de componentes (clusters) que possuem um determinado tamanho (número de vértices).
c) Etapas principais:
    i) Define intervalos (bins) inteiros para o histograma;
    ii) Plota os dados com rótulos e grades para melhor legibilidade;
    iii) Salva o histograma no diretório ../img/histogramas/ com o nome que inclui o limiar L.
d) O gráfico permite identificar a variação no tamanho e na quantidade de componentes à medida que o limiar de adjacência muda.

6) Bloco Principal de Execução

a) O código principal define uma lista de limiares (L) — valores que controlam a formação de arestas entre vértices.
b) Para cada limiar, as seguintes etapas são executadas:
    i) Carregamento da matriz de adjacência:
        (1) Arquivos nomeados no formato grafo_L_X_adj.csv são lidos com numpy.genfromtxt.
        (2) Caso não sejam encontrados, o programa exibe uma mensagem de erro e passa para o próximo limiar.
    ii) Leitura das coordenadas e IDs de cluster:
        (1) Os arquivos coord_L_X.csv contêm as coordenadas normalizadas dos vértices (X, Y, Z) e o identificador do cluster.
        (2) Apenas as três primeiras colunas são utilizadas para o gráfico 3D.
    iii) Geração do grafo 3D:
        (1) Chama-se a função plot_grafo_3d_cluster(), que salva a imagem no diretório definido.
    iv) Geração do histograma de clusters:
        (1) Os tamanhos de cada cluster são lidos de clusters_L_X.csv.
        (2) Se o arquivo existir, é chamado gerar_histograma() para criar a figura correspondente.

c) Essas etapas se repetem para cada valor de L, permitindo observar como a estrutura do grafo e a quantidade de clusters mudam conforme o grau de conectividade aumenta.

7) **Saídas Geradas**
   a) O programa gera duas categorias principais de resultados gráficos:
      i) Gráficos 3D dos Grafos (/img/grafos3D/): Representações tridimensionais com cores distintas para cada cluster.
      ii) Histogramas dos Componentes Conexos (/img/histogramas/): Diagramas de barras que mostram a distribuição de tamanhos dos clusters para cada limiar de conexão.

4. INSTRUÇÕES SOBRE A COMPILAÇÃO E EXECUÇÃO

O projeto foi desenvolvido utilizando a IDE Visual Studio Code, com suporte à linguagem C configurado para ambos os sistemas operacionais Windows e Linux (Ubuntu) . O trabalho possui duas etapas de execução: a compilação e execução do código C, seguida pela execução do script Python para visualização.

4.1.Execução no Linux/Ubuntu

Pré-requisitos: Certifique-se de que o **GCC** (GNU Compiler Collection) e o **Python 3** estejam instalados. É de suma importância que você esteja com o Prompt de Comando ou o Terminal aberto na pasta "**src"**.

1) **Organização dos Arquivos**
   a) Certifique-se de que main.c e my_dataset.csv estão na pasta "src".
2) **Compilação do Código C**
   a) Use o GCC para compilar os arquivos C, incluindo a biblioteca matemática (-lm):
      i) gcc main.c -o programa_grafo -lm
3) **Execução do Programa C**
   a) Execute o arquivo binário gerado. Ele processará os 4 valores de L e salvará todos os arquivos CSV necessários.
      i) ./programa_grafo
4) **Execução do Script Python**
   a) Baixe as bibliotecas necessárias:

i) python3 install numpy matplotlib networkx pandas

b) Execute o script Python para gerar as visualizações e histogramas a partir dos arquivos CSV criados:

i) python3 visualizar_grafo.py

4.2.Execução no Windows

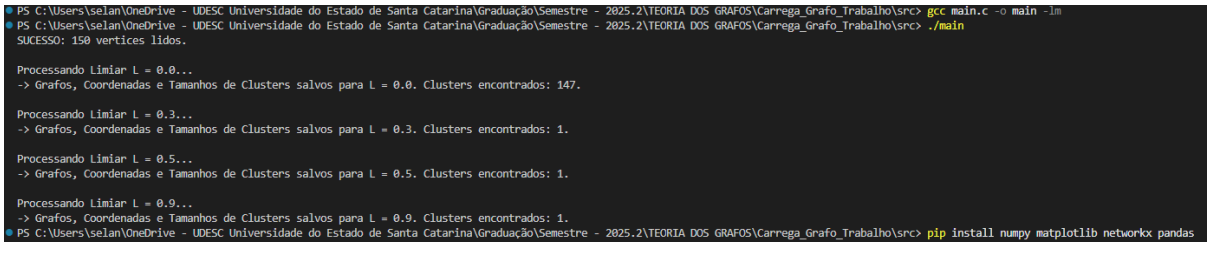
Pré-requisitos: Certifique-se de que o **GCC** (GNU Compiler Collection) e o **Python 3** estejam instalados. É de suma importância que você esteja com o Prompt de Comando ou o Terminal aberto na pasta "**src".**

1) **Compilação do Código C**
   a) Abra o Prompt de Comando ou o Terminal na pasta "**src"** e use o mesmo comando de compilação:
      i) gcc main.c -o programa_grafo -lm
2) **Execução do Programa C**
   a) Execute o binário gerado:
      i) ./programa_grafo
3) **Execução do Script Python**
   a) Baixe as bibliotecas necessárias:
      i) pip install numpy matplotlib networkx pandas
   b) Execute o script Python.
      i) python visualizar_grafo.py

5. RESULTADOS E ANÁLISE

A Figura 11 mostra o processo de compilação e execução do programa principal em C. Bem como, na Figura 12 temos a compilação do programa visualizar_grafo.py.

Figura 11 - Compilação do arquivo main.c.



Fonte: Arquivo pessoal.

Figura 12 - Compilação do visualizar_grafo.py.

```
PS C:\Users\selan\OneDrive - UDESC Universidade do Estado de Santa Catarina\Graduação\Semestre - 2025.2\TEORIA DOS GRAFOS\Carrega_Grafo_Trabalho\src> python visualizar_grafo.py
  Grafo 3D salvo em ../img/grafos3D/grafo_L_0_0.png
  Histograma salvo em ../img/histogramas/histograma_clusters_L_0_0.png
  Grafo 3D salvo em ../img/grafos3D/grafo_L_0_3.png
  Histograma salvo em ../img/histogramas/histograma_clusters_L_0_3.png
  Grafo 3D salvo em ../img/grafos3D/grafo_L_0_5.png
  Histograma salvo em ../img/histogramas/histograma_clusters_L_0_5.png
  Grafo 3D salvo em ../img/grafos3D/grafo_L_0_9.png
  Histograma salvo em ../img/histogramas/histograma_clusters_L_0_9.png
PS C:\Users\selan\OneDrive - UDESC Universidade do Estado de Santa Catarina\Graduação\Semestre - 2025.2\TEORIA DOS GRAFOS\Carrega_Grafo_Trabalho\src>
```

Fonte: Arquivo pessoal.

Por meio da Figura 11 e Figura 12, é possível acompanhar as etapas realizadas, desde o carregamento dos vértices, o cálculo das distâncias euclidianas e normalizadas (e os respectivos pares de vértices), até a construção do grafo com base no limiar estabelecido e a geração dos arquivos.

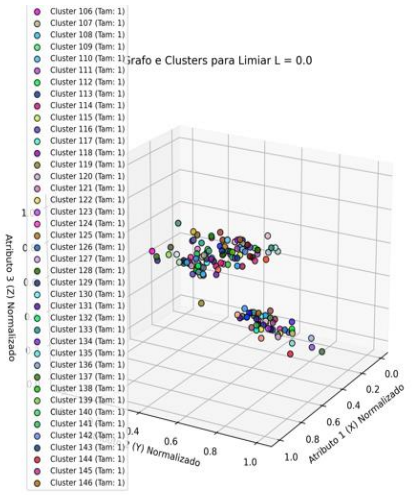
A geração do grafo com base no limiar de 0.0 resultou em 147 componentes conexos, enquanto a partir do Limiar 0.3 obteve-se um grande componente conexo de tamanho 150 e quanto maior o limiar maior o número de arestas entre os vértices. A menor distância euclidiana normalizada (DEN = 0,000000) identifica pares de vértices altamente semelhantes, enquanto a maior (DEN = 1,000000) reflete a grande variação espacial presente nos dados. Além disso, a utilização da matriz de adjacência mostrou-se eficiente para representar e manipular as conexões entre vértices.

A análise comparativa entre os limiares evidencia a influência direta de **L** na conectividade do grafo:

1) **L = 0.0:** ausência quase total de arestas; o grafo contém 147 componentes isolados.
2) **L >= 0.3:** ocorre a unificação de todos os vértices em um único componente conexo dominante.

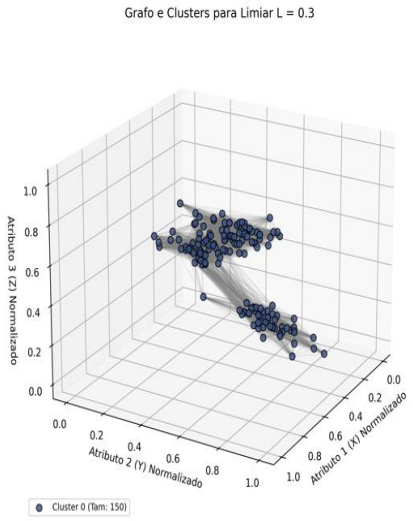
A seguir, conforme a Figura 13 e Figura 14, temos a visualização tridimensional dos grafos gerados a partir dos dados do arquivo CSV. Ela permitem ter uma ideia mais clara de como os vértices estão conectados no espaço, facilitando a compreensão da estrutura do grafo de forma visual e intuitiva.

Figura 13 – Grafo e Clusters para L = 0.0.



Fonte: Arquivo pessoal.

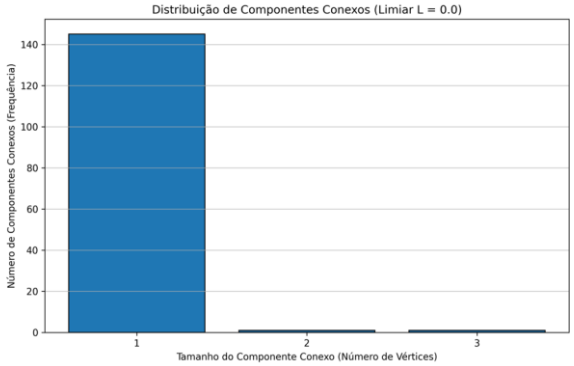
Figura 14 - Grafo e Clusters para L = 0.3.



Fonte: Arquivo pessoal.

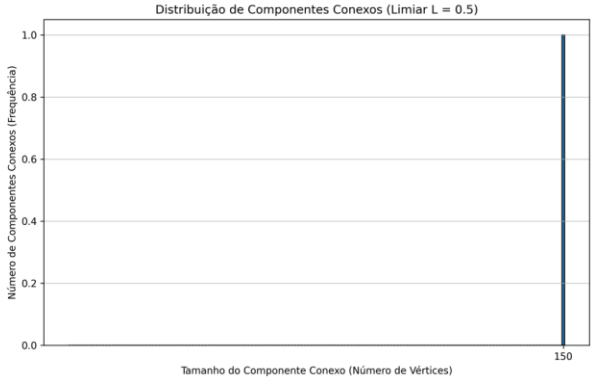
Os histogramas abaixo, conforme Figura 15 e Figura 16, mostram graficamente o crescimento do tamanho dos clusters à medida que o limiar aumenta, enquanto os grafos 3D destacam a formação visual dos grupos.

Figura 15 – Histograma com L = 0.0.



Fonte: Arquivo pessoal.

Figura 16 – Histograma com L = 0.3.



Fonte: Arquivo pessoal.

Esse comportamento confirma a **consistência da normalização e da BFS**, e evidencia como a densidade do grafo é controlada pelo valor de L.

6. BIBLIOTECAS UTILIZADAS

6.1.Linguagem C

1) <stdio.h>
2) <stdlib.h>
3) <string.h>
4) <math.h>

6.2.Python

1) matplotlib
2) networkx
3) numpy
4) pandas

5) mpl_toolkits.mplot3d