Alunos:
Lucas Gabriel
Santiago Cardoso

Arquivo: LucasG_SantiagoC

Documentação da parte 2 do Trabalho de TEG

PARTE 1:

- **Importar todos os dados:**
  Criamos uma struct chamada Iris, a qual, contém os atributos: float sep_len, float sep_width, float pet_len, float pet_width, char *variety. Depois realizamos a leitura dos dados do arquivo CSV “IrisDataset.csv” e criamos diversos nós flores.

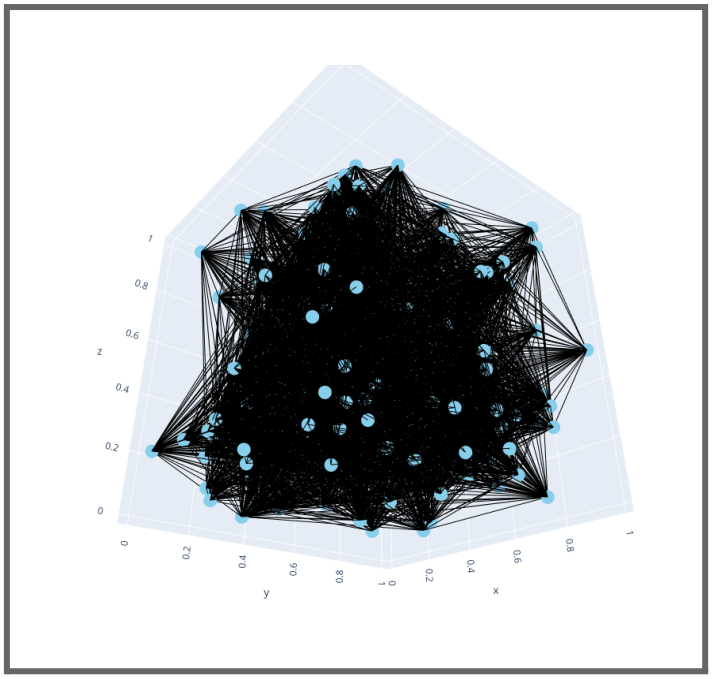
- **Montar a tabela euclidiana**
  Realizamos o cálculo da distância euclidiana para cada nó flor em relação aos demais nós, e alocamos isso em uma matriz, na qual, cada índice representa a ordem de leitura do arquivo CSV.

- **Construção da tabela euclidiana normalizada**
  Efetuamos o algoritmo de normalização euclidiana fornecido no trabalho, e aplicamos ele na tabela, assim, criando uma tabela normalizada.

- **Aplicar o limiar de 0.3**
  Após montarmos e criarmos a tabela euclidiana normalizada, aplicamos o limiar de 0.3 em cada linha para formar uma lista de adjacências para cada um dos vértices.

- **Salvar o grafo no arquivo Grafo.csv**
  Após todas as operações, salvamos a tabela euclidiana normalizada em um arquivo CSV “Grafo.csv”, no qual, cada linha possui um par de nós que são conectados.

- **Visualização do grafo**
  Com todo o código já finalizado, utilizamos o algoritmo de visualização do grafo em python fornecido para gerar um grafo 3D visualizável.



PARTE 2:

- **Execução:**
  Para executar o programa primeiro compilamos “gcc iris.c -o iris -lm” e depois executamos “./iris 0.3”, esse valor “0.3” pode ser qualquer valor que você queira estabelecer de limiar inicial.

- **Menu interativo:**
  Apresentamos um menu interativo para o usuário, que possibilita-o fazer diversas operações dentro do código.

- **void estudo_componentes_conexos();**
  Essa função foi utilizada para estudar e visualizar a mudança dos limiares e a formação dos componentes conexos no grafo, no qual, ela executava o arquivo “displayGrafo.py” com os novos limiares estabelecidos.

- **Novos elementos:**
  Foram criadas diversas novas funções durante o código, entre elas:

  int criar_matriz_adjacencias( );**
  Cria uma nova matriz de adjacências, e executa todas as funções utilizadas anteriormente na primeira etapa do trabalho.

  int DFS(int vertice);
  Optamos por utilizar o DFS como algoritmo de varredura, por ele visitar todos os vértices do grafo e atribuir um marcador em uma matriz de vértices visitados, sua fácil implementação e características computacionais eram de agrado da execução da tarefa, e assim, não interferia de maneira grave na resolução do problema.

  int contarComponentesConexos(int v);

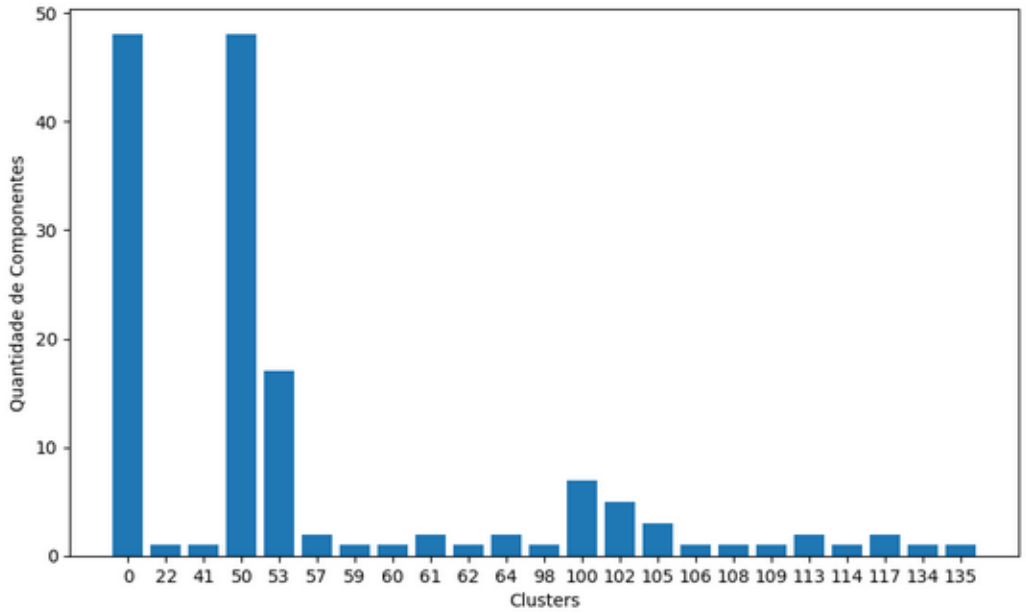
Essa função atravessa nossa matriz de visitados gerada anteriormente, e caso encontre o marcador, ela incrementa o valor de componentes conexos, por fim, retornando-o.

```
if(visitados[0][i] == 1){
            num_comps++
```

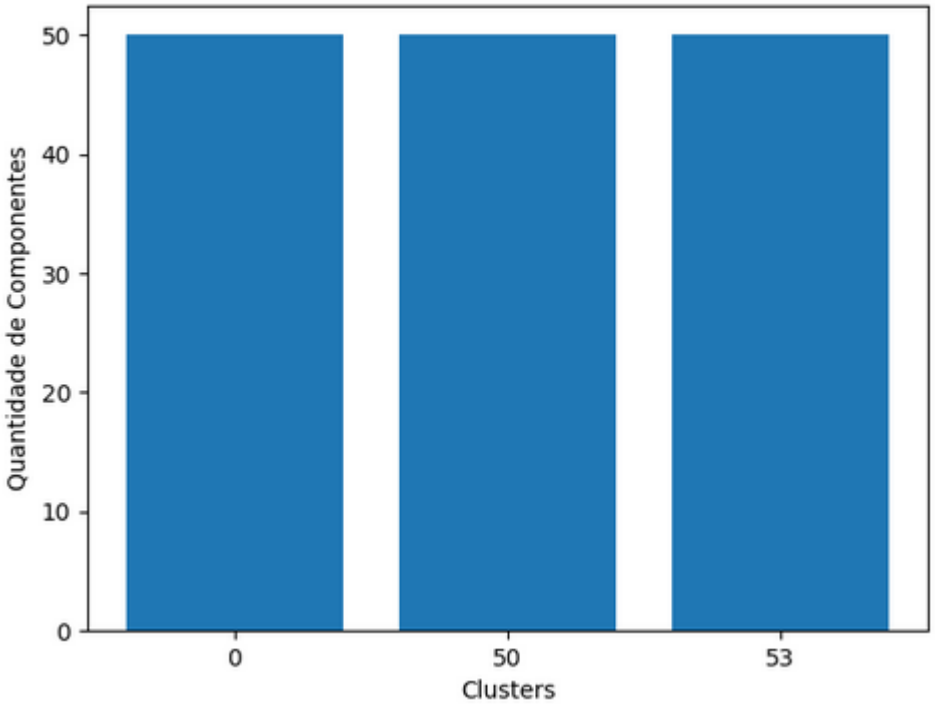
void histograma();

Função definida para a criação do histograma final, a qual, apresenta os clusters e a quantidade de componente conexos em cada cluster.

Primeira geração do histograma:



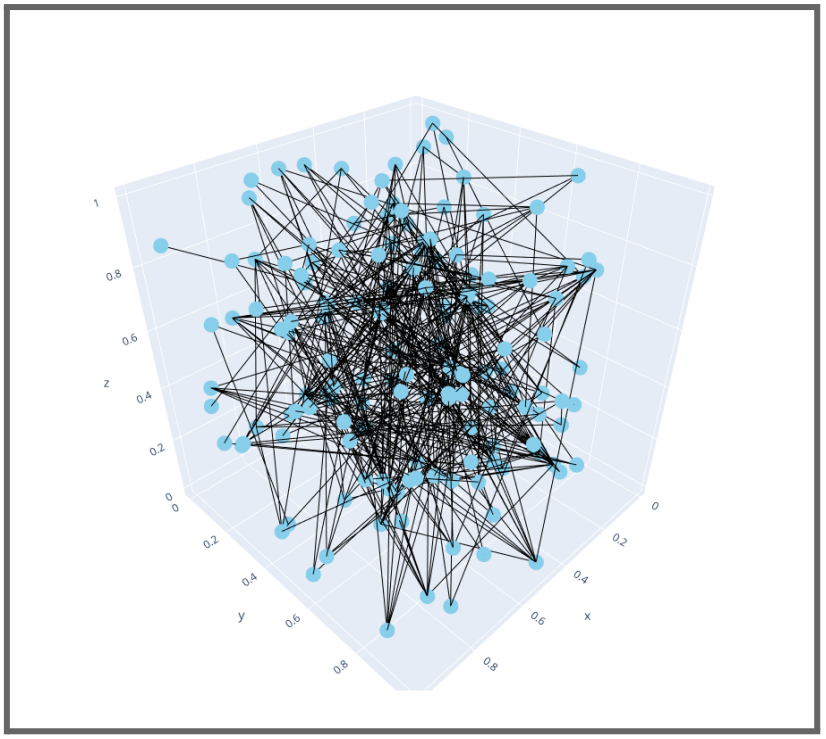
Após otimizações:



Entre algumas funções auxiliares também criadas:

```
void zerarVisitados();
void pop(int *pilha);
void push(int *pilha, int valor);
int *criar_pilha();
int len(int v[]);
```

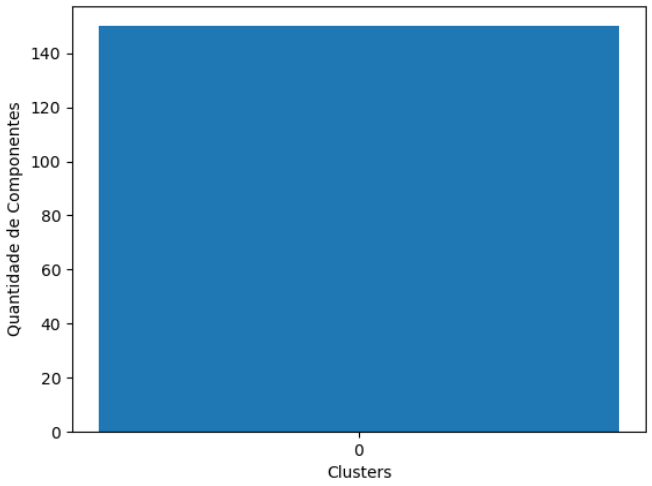
- **Novo grafo gerado**



RESPOSTAS DAS PERGUNTAS:

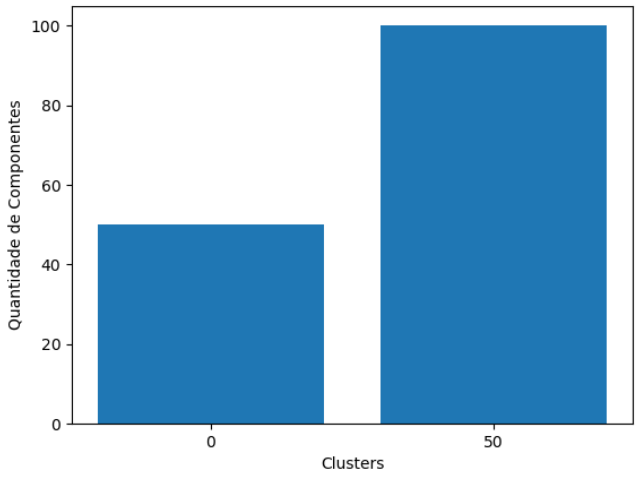
- **1:**
  Optamos por utilizar o DFS como algoritmo de varredura, por ele visitar todos os vértices do grafo e atribuir um marcador em uma matriz de vértices visitados, sua fácil implementação e características computacionais eram de agrado da execução da tarefa, e assim, não interferia de maneira grave na resolução do problema.

- **2:**
  Limiar 0.3:



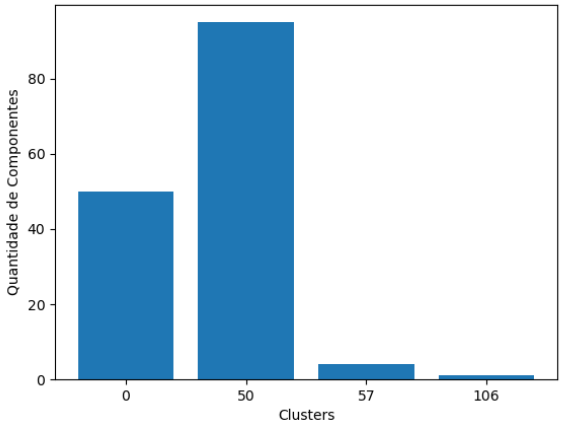
Soma: 150
Clusters: 1

Limiar 0.2



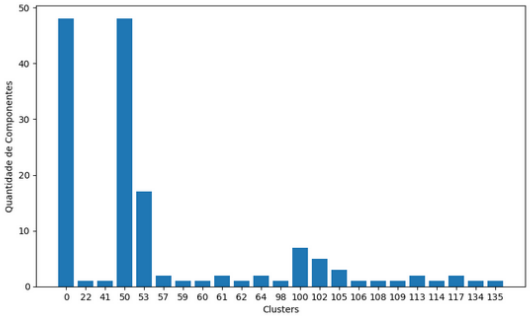
Soma: 150
Clusters: 2

Limiar 0.1



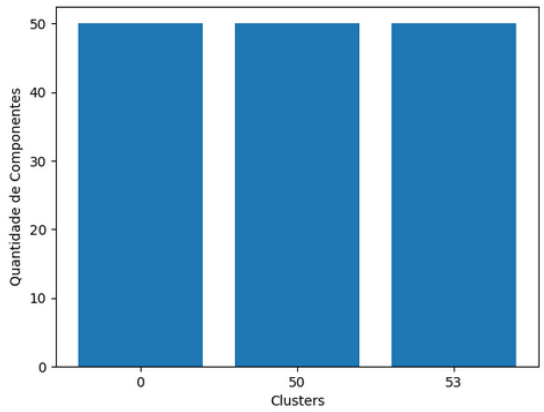
Soma: 150
Clusters: 4

Limiar 0.03



Soma: 150
Clusters: 23

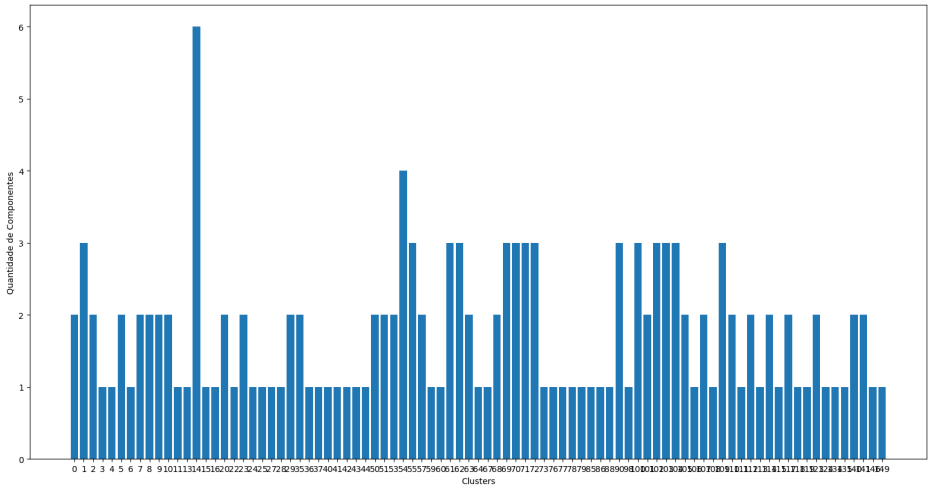
Otimizado:



Soma: 150
Clusters: 3

O valor esperado para as somas é 150 pois temos 150 vértices.
Caso queira saber mais, execute a função estudo_componentes_conexos();

- **3:**



Arestas: 155
Vértices: 150
Não, pois tendo um limiar de 0.0 está exigindo igualdade entre os vértices, entretanto, nenhum dos vértices são completamente iguais, assim, o resultado está equivocado segundo nossa previsão.

- **4:**
Limiar: 0.03
Encontramos dois grandes grupos, e vários vértices dissimilares, após aplicarmos uma otimização, na qual, juntamos os vértices isolados com a raiz do cluster com maior similaridade a este vértice, dessa maneira, foi possível formar 3 grandes grupos.