Relatório Tarefa 1 Teoria de Grafos

Alunos: Gustavo de Souza e José Augusto Laube;

INTRODUÇÃO

Para compilar o código no Linux:
gcc teg_final.c -lm && ./a.out

Para plotar o grafo em 3D usando python é necessário escolher o arquivo txt que o nosso arquivo **teg_final.c** gerar e copiar os dados dele para o arquivo **grafo.txt** da pasta grafo-monitoria-main.

Ao compilar o código **teg_final.c** gera os seguintes arquivos:

grafo_adjascencia.txt: armazena todas as relações dos vértices gerada pelo limiar escolhido.

grafo_final.csv: arquivo csv que exibe no cabeçalho o que foi solicitado na tarefa 1a assim como a matriz de adjacência.

grafo_NaoSetosa_media.txt: contém os vértices do grupo das não sedosas utilizando a media como método de divisão em dois clusters.

grafo_setosa_media.txt: contém os vértices do grupo das setas utilizando a média como método de divisão em dois clusters.

grafo_Nsetosa_k_means.txt: contém os vértices do grupo das não setosas utilizando o k-means como método de divisão em dois clusters.

grafo_setosa_k_means.txt: contém os vértices do grupo das setosas utilizando o k-means como método de divisão em dois clusters.

O programa começa criando uma matriz de float (**matrix[150][4]**) que vai ser onde vamos salvar os valores do arquivo .csv, após isso o programa abre o arquivo IrisDataset.csv e

exportar os valores na matriz matrix[ ][ ], sendo que cada coluna da matrix[ ][ ], representa os valores de “Sepal.length”, “Sepal.width”, “Petal.length” e “Petal.width”. Com os valores do arquivo csv salvos podemos fechar o arquivo e começar a análise dos dados, para fazer a distância euclidiana normalizada, primeiro calculamos a distância euclidiana, para isso

```
float euclidiana(float mati1, float mati2, float mati3, float mati4, float matj1, float matj2, float matj3, float matj4)
{
    float petL = (mati1 - matj1);
    float petA = (mati2 - matj2);
    float sepL = (mati3 - matj3);
    float sepA = (mati4 - matj4);
    petL = pow(petL, 2);
    petA = pow(petA, 2);
    sepL = pow(sepL, 2);
    sepA = pow(sepA, 2);
    float resultado = sqrt(petL + petA + sepL + sepA);
    return resultado;
}
```

criamos a matriz **matrixEuc[150][150]**, e começamos, o cálculo deixando a diagonal principal zerada, criamos uma função chamada **euclidiana,** que irá fazer o cálculo para nós:

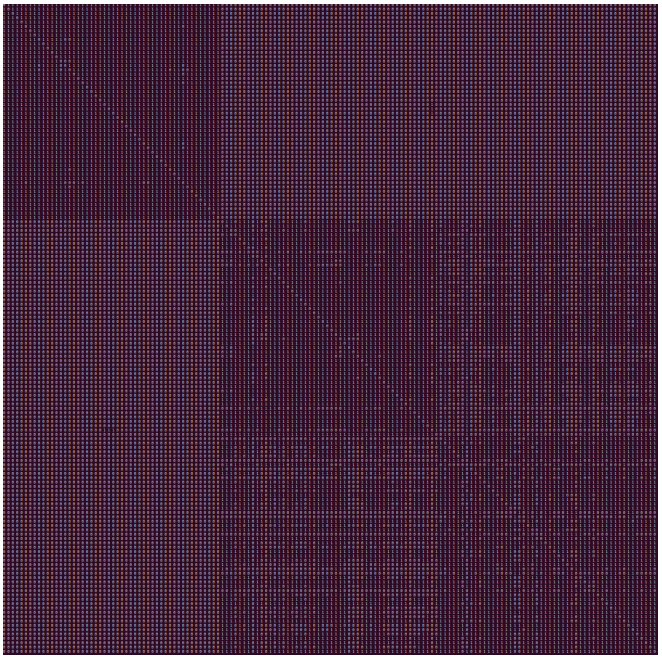
Ela recebe as dimensões que correspondem aos valores das colunas da matrix[ ][ ], da linha i e j da matriz, vai calcular o quadrado da diferença de dois pontos e depois calcular a raiz quadrada das somas das diferenças de quadrados, e vai salvar a distância na matrixEuc[i][j], enquanto os dados estão sendo salvos na matriz já se é verificados quem é o menor e o maior valor da matriz de distâncias euclidiana, a partir dos valores da matrixEuc podemos fazer a normalização das distâncias, para isso criamos uma matriz de float (**matrixNorm[150][150]**) e fazemos o cálculo da normalização das distâncias dentro da main:

```
// resetar as variaveis de max e min para a matriz normalizada
float maxDEN = 0, minDEN = 1;
int maxDEN_i = 0, maxDEN_j = 0, minDEN_i = 0, minDEN_j = 0;

// MATRIZ EUCLIDIANA NORMALIZADA
for (int i = 0; i < row; i++)
{
    for (int j = 0; j < row; j++)
    {
        // apenas considere valores fora da diagonal para min e max
        if (i != j)
        {
            if (matrixNorm[i][j] < minDEN)
            {
                minDEN = matrixNorm[i][j];
                minDEN_i = i;
                minDEN_j = j;
            }
            if (matrixNorm[i][j] > maxDEN)
            {
                maxDEN = matrixNorm[i][j];
                maxDEN_i = i;
                maxDEN_j = j;
            }
        }
        // printf("%.2f ", matrixNorm[i][j]);
    }
    // printf("\n");
}
```

Após a normalização, efetuamos a busca dos vértices que correspondem ao mínimo e ao máximo da normalização, para assim gerar a matriz de adjacência (**matAd[150][150]**), que vai ter um nos valores que são menores ou iguais ao limiar, limiar este que é uma variável global definida no começo do programa, inicialmente testamos com um limiar de 0.3 que foi sugerido pelo professor, e depois fizemos outros testes para ver como o programa se comporta.

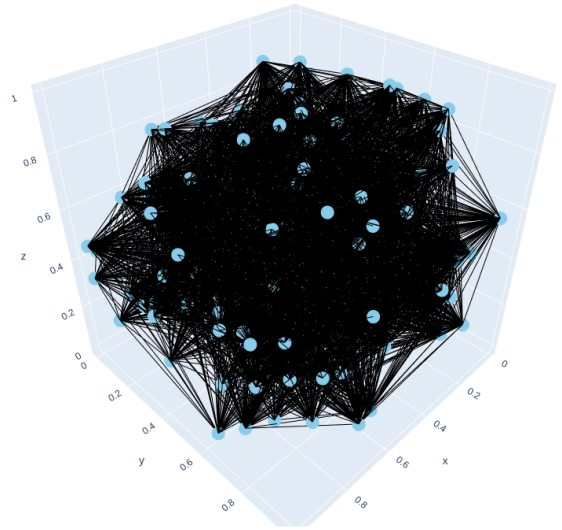
```
// Matriz de adjascencia
// int matAd[row][row];

for (int i = 0; i < row; i++)
{
    for (int j = 0; j < row; j++)
    {
        if ((i != j) && (matrixNorm[i][j] <= limiar))
        {
            matAd[i][j] = 1;
        }
        else
        {
            matAd[i][j] = 0;
        }
    }
}
```

Abaixo há um zoom-out da matriz de adjacência com limiar de 0.3, onde é possível visualmente distinguir o grupo das setosas (os primeiros 50 elementos) dos grupos das nao setosas (os 100 elementos restantes):



Ao compilar o nosso main irá gerar uma matriz de adjacência salvará a mesma em um arquivo **grafo_final.csv** conforme solicitado com os valores de maior DE, menor DE, maior DEN e menor DEN, também geramos um arquivo .txt, denominado de **grafo_adjascencia.txt**, este é usado para plotar o grafo usando o arquivo python caso desejado (a poluição visual é enorme com o limiar 0.3, mas plotamos o txt para fins de

estudos). A seguir há a plotagem 3D do grafo:



SEGMENTAÇÃO (MÉTODO MÉDIA)

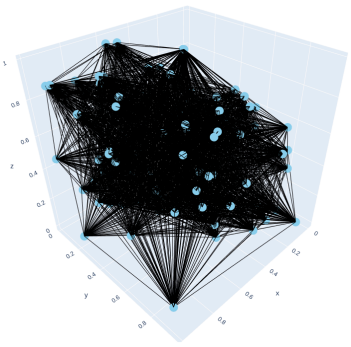
A partir dos dados da matriz de adjacência conseguimos fazer a separação em dois grandes grafos utilizando dois métodos diferentes, o primeiro é através da busca em em largura, onde ela vai percorrer a matriz por todas as conexões de cada grupo, para isso chutamos um centro para cada grande grafo, e o algoritmo vai percorrer por todas as conexões, e depois vai fazer o mesmo para o outro grupo, depois que separar os grupos principais, podemos calcular o centro real de cada grupo, o centro definido pelas médias das características das flores “Sepal.length”, “Sepal.width”, “Petal.length” e “Petal.width”, com essa média que é o elemento central de cada subgrafo, buscamos o elemento que mais se aproxima do desse elemento central, então definimos ele como o central, percorremos a matriz em busca dos vértices que ficam desconexos dos grandes grupos, direcionamos ele para o centro mais próximo, fizemos o uso da função fabs da biblioteca math.h para ter o valor em módulo dessa diferença de centros. Segue abaixo um trecho do código que efetua o cálculo do primeiro centro e busca o elemento que mais se aproxima dele:

```
double medias[4] = {0, 0, 0, 0};
int contador = 0;
for (int i = 0; i < NUM_VERTICES; i++)
{
    if (num1[i] == 1)
    {
        for (int j = 0; j < 4; j++)
        {
            medias[j] += matrix[i][j];
        }
        contador++;
    }
}
for (int i = 0; i < 4; i++)
{
    medias[i] /= contador;
}
double menorSet[4] = {100, 100, 100, 100};

for (int i = 0; i < contador; i++)
{
    if (fabs(medias[0] - matrix[i][0]) <= menorSet[0] && fabs(medias[1] - matrix[i][1]) <= menorSet[1] && fabs(medias[2]
    matrix[i][2]) <= menorSet[2] && fabs(medias[3] - matrix[i][3]) <= menorSet[3])
    {
        for (int j = 0; j < 4; j++)
        {
            menorSet[j] = fabs(medias[j] - matrix[i][j]);
        }
        centro1 = i;
    }
}
```

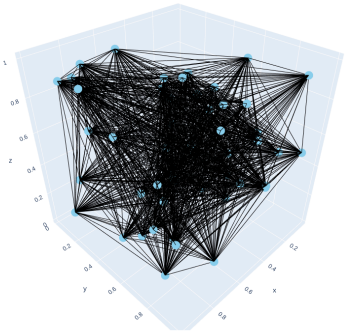
A busca pelos elementos que estão em casa grupo é analisado abaixo, o resultado de quais vértices estarão no subgrafo das setosas ou não setosas é armazenado nos vetores **matGp1[150]** (SETOSAS) e **matGp[150]** (NÃO SETOSAS) e pode ser vista abaixo:

```
int matGp1[150];
int matGp2[150];

for (int i = 0; i < NUM_VERTICES; i++)
{
    if (fabs(matrix[centro1] - matrix[i]) < fabs(matrix[centro2] - matrix[i]))
    {
        matGp1[i] = 1;
        matGp2[i] = 0;
    }
    else
    {
        matGp1[i] = 0;
        matGp2[i] = 1;
    }
}
```

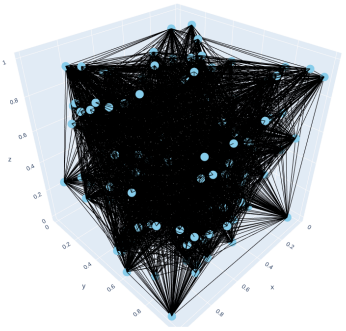
Ao segmentar os dois cluster nosso código gera um arquivo txt que armazena as relações entre os vértices para plotar em 3D usando python assim como explicado na introdução do relatório, a seguir há a representação gráfica dos dois subgrafos utilizando limiar 0.25:



Grafo das não setosas



Grafo das setosas



Grafo não dividido

~

SEGMENTAÇÃO (MÉTODO K-MEANS)

Também fizemos outra abordagem para a separação dos dois grupos, usando o método **k-means** que faz a separação dos clusterings por meio do grau de similaridade (quantidade de relações com outros vértices em comum) de dois vértices a partir da matriz de adjacência, vale ressaltar que os centros desse método devem começar aleatoriamente e serão recalculados constantemente para que mantenha a integridade dos dados analisados. Existem funções naturais do C que fazem a melhor escolha do centro mas como achamos fascinante o método então fizemos isso manualmente, é importante deixar claro que o valor do centro foi escolhido manualmente por nós no código pois se for definido por um método como srand pode acontecer que os centros sejam muito próximos e com os nossos testes os resultados não se mostram precisos pois a função que implementamos que faz o recálculo do centro não está preparada para tal abordagem (recomendamos caso queira trocar os valores iniciais do centro valores que tenham uma distância entre si de aproximadamente 100 unidades).
No main, o método de separação k-means consiste neste código abaixo:

```
centroids[0] = chute_centro1;
centroids[1] = chute_centro2;

int mudou = 1;
while (mudou)
{
    mudou = 0;

    // atribuir cada vertice ao centro mais prox
    for (int v = 0; v < NUM_VERTICES; v++)
    {
        int novo_cluster = centro_mais_proximo(v);
        if (novo_cluster != clusters[v])
        {
            clusters[v] = novo_cluster;
            mudou = 1;
        }
    }

    // Recalcular os centros
    recalcular_centroides();
}
```

A função **centro_mais_proximo()** busca qual centro é o mais próximo (do cluster 1 ou do cluster 2) a partir do retorno da função **similaridade(),** segue abaixo a implementação do cálculo do centro mais próximo:

```
int centro_mais_proximo(int vertice)
{
    int max_similaridade = -1;
    int cluster = 0;
    for (int i = 0; i < K; i++)
    {
        int sim = similaridade(vertice, centroids[i]);
        if (sim > max_similaridade)
        {
            max_similaridade = sim;
            cluster = i;
        }
    }
    return cluster;
}
```

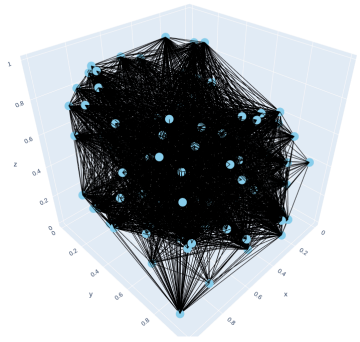
A função similaridade() simplesmente faz a análise do quão similar dois vértices são baseados nas relações que possuem com outros vértices da matriz de adjacência, vale ressaltar que setamos sempre a comparação da similaridade com um dado vértice e o centro para assim direciona-lo para o correto:

```
int similaridade(int vertice1, int vertice2)
{
    int similar = 0;
    for (int i = 0; i < NUM_VERTICES; i++)
    {
        if (matAd[vertice1][i] == 1 && matAd[vertice2][i] == 1)
        {
            similar++;
        }
    }
    return similar;
}
```

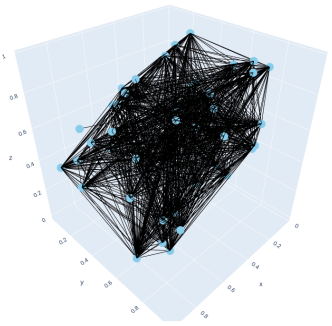
E por fim, a função **recalcular_centroids()** recalcula o centróide de cada cluster escolhendo o ponto que maximiza a soma das similaridades com os outros pontos do mesmo cluster. Isso busca garantir que o centróide seja o ponto mais representativo do cluster, ou seja, o mais próximo dos demais pontos em termos de similaridade.

```c
void recalcular_centroides()
{
    for (int i = 0; i < K; i++)
    {
        int max_similaridade = -1;
        int novo_centro = centroids[i];

        for (int v = 0; v < NUM_VERTICES; v++)
        {
            if (clusters[v] == i)
            {
                int total_similaridade = 0;
                for (int w = 0; w < NUM_VERTICES; w++)
                {
                    if (clusters[w] == i)
                    {
                        total_similaridade += similaridade(v, w);
                    }
                }
                if (total_similaridade > max_similaridade)
                {
                    max_similaridade = total_similaridade;
                    novo_centro = v;
                }
            }
        }
        centroids[i] = novo_centro;
    }
}
```

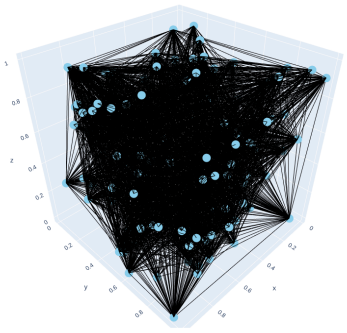
Ao segmentar os dois cluster usando o k-means nosso código gera um arquivo txt que armazena as relações entre os vértices para plotar em 3D usando python assim como explicado na introdução do relatório, a seguir há a representação gráfica dos dois subgrafos utilizando limiar 0.25:



Grafo não setosas



Grafo setosas



Grafo não dividido

ACURÁCIA

Como fizemos a divisão dos sub grafos utilizando métodos diferentes foi necessário implementar dois métodos diferentes para o cálculo da acurácia visto que os valores dos vértices de cada cluster foi salvo de formas diferentes em cada método. O método K-means foi relativamente mais fácil o cálculo da acurácia visto que os vértices foram salvos em um vetor denominado **clusters[150]** em que cada posição conta com o valor 0 (o vértice k da posição clusters[k] significa que ele está no primeiro cluster) ou o valor 1 (o vértice está no

segundo cluster). Abaixo há um print que sintetiza o cálculo:

```
// acuracia para k-means
int casos_TP = 0, casos_TN = 0, casos_FP = 0, casos_FN = 0;

// levar em conta que cluster[] == 0 eh setosa
for (int i = 0; i < NUM_VERTICES; i++)
{
    if (clusters[i] == 0 && i >= 0 && i <= 49)
    { // dito setosa e eh setosa (true positive)
        casos_TP++;
    }
    else if (clusters[i] == 0 && (i < 0 || i > 49))
    { // dito setosa mas n eh setosa (false positive)
        casos_FP++;
    }
    else if (clusters[i] == 1 && i >= 0 && i <= 49)
    { // dito n setosa mas eh setosa (false negative)
        casos_FN++;
    }
    else if (clusters[i] == 1 && (i < 0 || i > 49))
    { // dito n setosa e eh n setosa (true negative)
        casos_TN++;
    }
}

// acuracia utilizando k-means
float acuracia = (float)(casos_TP + casos_TN) / (casos_TP + casos_FP + casos_TN + casos_FN);
```

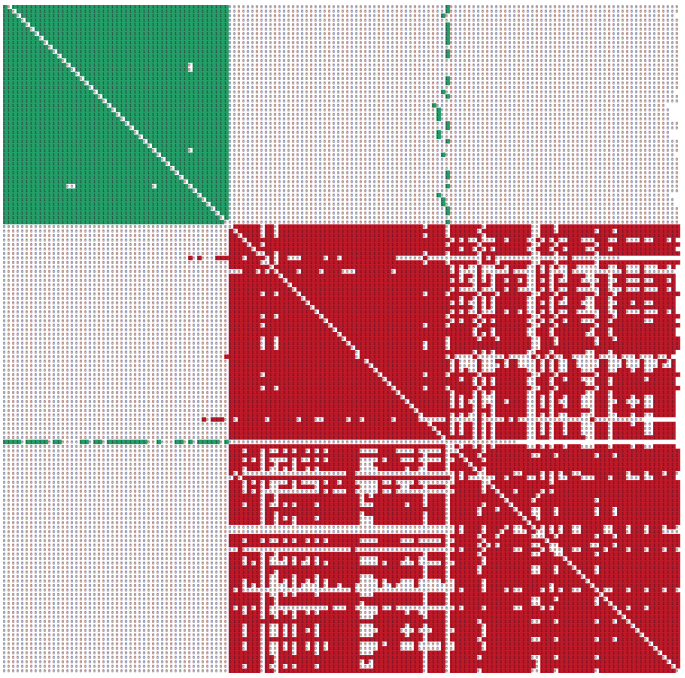
Já o método das médias exigiu uma implementação mais complexa visto que os vértices de cada cluster estão separados em dois vetores diferentes, entretanto o raciocínio lógico implementado é o mesmo:

```
// acuracia para metodo da media
int casos_TP_md = 0, casos_TN_md = 0, casos_FP_md = 0, casos_FN_md = 0;
// considerar que matGp1[] eh o vetor com os vertices setosas
for (int i = 0; i < NUM_VERTICES; i++)
{
    // checar se o indice esta entre 0 e 49 (setosas)
    if (i >= 0 && i <= 49)
    {
        // caso setosa e foi classificada corretamente (true positive)
        if (matGp1[i] == 1)
        {
            casos_TP_md++;
        }
        // caso nao setosa, mas foi classificada como setosa (false positive)
        else if (matGp1[i] == 0)
        {
            casos_FP_md++;
        }
    }
    // checar se o indice esta fora do intervalo (n setosas)
    else
    {
        // caso nao setosa e foi classificada corretamente (true negative)
        if (matGp1[i] == 0)
        {
            casos_TN_md++;
        }
        // caso setosa, mas foi classificada como nao setosa (false negative)
        else if (matGp1[i] == 1)
        {
            casos_FN_md++;
        }
    }
}

// acuracia utilizando metodo da media
float acuracia_md = (float)(casos_TP_md + casos_TN_md) / (casos_TP_md + casos_FP_md + casos_TN_md + casos_FN_md);
```

TESTES DE MESA

O nosso código será testado com 5 valores de limiar diferentes e mostraremos as alterações da acurácia de caso a caso, vale ressaltar que como saída printamos tanto a matriz de adjacências como os vértices que pertencem a cada cluster usando cores no

terminal para facilitar a interpretação conforme o print abaixo:



Aqui no relatório vamos apresentar somente os valores de acurácia em um dado limiar para fins interpretativos, mas para análises profundas é possível compilar o código e efetuar testes manualmente alterando a variável limiar definida logo no início.

Caso Limiar = 0.1

```
--- ANALISE DOS DADOS K-MEANS ---

Casos TP: 50
Casos TN: 66
Casos FP: 34
Casos FN: 0
Acuracia: 0.773333

--- ANALISE DOS DADOS MEDIA ---

Casos TP: 50
Casos TN: 74
Casos FP: 0
Casos FN: 26
Acuracia: 0.826667
```

```
--- CENTROS ---

Centro1_k final: 40  Centro2_k final: 78
Centro1_media final: 7  Centro2_media final: 145
```

Caso Limiar = 0.15

Obs: Esse é um dos casos citados na explicação do método k-means em que o valor do limiar só consegue definir o centro do cluster das setosas como o primeiro elemento.

```
--- ANALISE DOS DADOS K-MEANS ---

Casos TP: 50
Casos TN: 90
Casos FP: 10
Casos FN: 0
Acuracia: 0.933333

--- ANALISE DOS DADOS MEDIA ---

Casos TP: 50
Casos TN: 85
Casos FP: 0
Casos FN: 15
Acuracia: 0.900000
```

```
--- CENTROS ---

Centro1_k final: 0  Centro2_k final: 126
Centro1_media final: 7  Centro2_media final: 123
```

Caso Limiar = 0.2

Obs: Esse é um dos casos citados na explicação do método k-means em que o valor do limiar só consegue definir o centro do cluster das setosas como o primeiro elemento.

```
--- ANALISE DOS DADOS K-MEANS ---

Casos TP: 50
Casos TN: 98
Casos FP: 2
Casos FN: 0
Acuracia: 0.986667

--- ANALISE DOS DADOS MEDIA ---

Casos TP: 50
Casos TN: 85
Casos FP: 0
Casos FN: 15
Acuracia: 0.900000
```

```
--- CENTROS ---

Centro1_k final: 0  Centro2_k final: 138
Centro1_media final: 7  Centro2_media final: 123
```

Caso Limiar = 0.25

```
--- ANALISE DOS DADOS K-MEANS ---

Casos TP: 50
Casos TN: 100
Casos FP: 0
Casos FN: 0
Acuracia: 1.000000

--- ANALISE DOS DADOS MEDIA ---

Casos TP: 50
Casos TN: 85
Casos FP: 0
Casos FN: 15
Acuracia: 0.900000
```

```
--- CENTROS ---

Centro1_k final: 23  Centro2_k final: 123
Centro1_media final: 7  Centro2_media final: 123
```

Caso Limiar = 0.3

```
--- ANALISE DOS DADOS K-MEANS ---

Casos TP: 50
Casos TN: 99
Casos FP: 1
Casos FN: 0
Acuracia: 0.993333

--- ANALISE DOS DADOS MEDIA ---

Casos TP: 50
Casos TN: 85
Casos FP: 0
Casos FN: 15
Acuracia: 0.900000
```

```
--- CENTROS ---

Centro1_k final: 23  Centro2_k final: 77
Centro1_media final: 7  Centro2_media final: 123
```

CONCLUSÃO

Com base nos nossos testes podemos ver que quanto mais próximo de 0,25, que é o valor que melhor dividiu os grupos nos nossos testes, a acurácia fica mais próxima de 1.00, e na maioria dos casos os centróides são os mesmo vértice 7 para as setosas e 123 para as não setosas, por outro lado usando o k-means temos uma acurácia de 0.9 que é igual em quase todos os casos, porém ele encontrou centróides diferentes para cada teste de limiar, isso acontece pela forma com que este algoritmo funciona. Portanto podemos concluir que quando o limiar for muito distante do que melhor divide os sub grafos, os algoritmos vão ter mais dificuldade para separar os dois grupos o que não tira a validade dos algoritmos, podemos perceber também que quando o limiar for mais próximo do ideal a busca em largura é mais funcional do que o k-means já quando esse limiar for mais distante o k-means se da melhor, então cabe ao programador escolher qual vai funcionar melhor para cada problema.