Relatorio EP 2
Limiar de Conexidade para certo Grafos Geométricos II

Fellipe Souto Sampaio[1]

MAC 0323 Estrutura de Dados
Prof. Dr. Yoshiharu Kohayakawa

Instituto de Matemática e Estatística - IME USP
Rua do Matão 1010
05311-970 Cidade Universitária, São Paulo - SP

[1]e-mail: fellipe.sampaio@usp.com

1 Introdução

Este relatório serve como descrição para o exercício programa 2 e a forma como sua solução foi implementada. O algorítmo utilizado no EP2 é similar ao utilizado no EP1, com pequenas mudanças que serão descritas a seguir.

2 Client.c

Este é o arquivo principal do programa, o qual contêm as funções *int main* e *getArgv*. Explicarei sucintamente seu funcionamento, assim como dos próximos módulos e funções que compoem o programa.

2.1 int main

Parâmetros de entrada : -Nxxx,-Dxxx, -sxxx, -dx.xx, -Mxxx, -v, -V, -D, -C, -L.
Saída: 0 se a execução ocorrer sem falha.

Descrição: O programa funciona em três modos de execução basicamente:

- Teste de conectividade para um dado N, D, s e d.
- Cálculo da média de conexidade crítica para instâncias D, N e M.
- Busca do menor d, tal que o grafo seja conexo.

Dado uma entrada na chamada do programa através da linha de comando o programa verifica e executa a rotina requisitada pelo usuário.

2.2 getArgv

Parâmetros de entrada : argc, argv[ ].
Saída: N, D, d, s, v, C, m.

Descrição: Le as strings contidas em argv[argc] e procura os padrões de funcionamento do programa, atribuindo os valores fornecidos as suas respectivas variáveis.

3 Instance.c

Módulo que concentra as principais funções de instanciação do programa.

3.1 searchDensity

Parâmetros de entrada : Variáveis globais M e out.
Saída: Informa na tela ou em arquivo de saída (dependendo do valor de out) o valor do limiar de conexidade típico d*(p_i), com $0 \geq i \geq M$ e a média do limiar de conexidade para (N,D,M) instâncias aleatórias. O usuário pode optar por imprimir os valores de d*(p_i) e também os próprios p_i, se desejar.

Descrição: A função recebi suas diretivas de funcionamento e chama a função lessConnectivity para localizar o d_i*. O valor encontrado é somado em avgConex e depois dividido por M para cálcular a média da conexidade típica.

3.2 lessConnectivity

Parâmetros de entrada : Variável global edge.
Saída: O menor d tal que o grafo é conexo.

Descrição: A função cria uma nova instância com N pontos, D dimensões e começa a verificar conexidade com um valor *startBin*. O algoritmo para encontrar o menor *edge* é uma busca binária dentro do intervalo [0,1]. É utilizado um valor inicial de 0.5 para a busca, até que a diferença entre o máximo conexidade e o mínimo não conexidade seja igual a 0.000001.

3.3 newGraph

Parâmetros de entrada : Variável global out.
Saída: Retorna TRUE ou FALSE, dependendo da conexidade do grafo.

Descrição: A função cria uma nova instância com N pontos aleatórios em dimensão D, imprime em arquivo os pontos gerados se for requisitado pelo usuário, e utiliza da função searchEdge para testar conexidade na instância criada, retornando o valor lógico do teste em conex.

4 Point.h

Este header concentra todas funções para operação do tipo abstrato de dado point. A unica diferença entre Point.S.c e Point.C.s está na função randPoint e createList. Ademais as outras funções são iguais nos dois casos, explicarei as funćões iguais e separarei em dois casos as funções diferentes.

4.1 Randomize

Parâmetros de entrada : newseed.
Saída: Nenhuma.

Descrição: Faz um reset no gerador de números aleatórios utilizando a semente *newseed*.

4.2 InteiroRandomico

Parâmetros de entrada : min e max.
Saída: Um número inteiro aleatório no intervalo [min,max].

Descrição: Sortei um inteiro randômico no intervalo fechado [min,max] utilizando como base o gerador de números aleatórios.

4.3 RealRandomico

Parâmetros de entrada : min e max.
Saída: Um número real aleatório no intervalo [min,max].

Descrição: Sortei um real randômico no intervalo fechado [min,max] utilizando como base o gerador de números aleatórios.

4.4 Distance

Parâmetros de entrada : ponto a e ponto b.
Saída: A norma euclidiana entre os dois ponto.

Descrição: A função cálcula a distância euclidiana entre dois pontos em dimensão D utilizando a seguinte formula:

$$\sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_n - b_n)^2} \qquad (1)$$

4.5 pointOutput

Parâmetros de entrada : Vetor de Point e op.
Saída: Nenhuma.

Descrição: A função imprime na saída padrão ou em arquivo, de acordo com o valor de op, todos os pontos e suas respectivas coordenadas.

4.6 Cubo

4.6.1 createList

Parâmetros de entrada : Variáveis globais N e seed.
Saída: Retorna um vetor de Point de tamanho N.

Descrição: Dado uma configuração fornecida pelo usuário o programa cria um vetor de Point de tamanho N e atribui N enuplas ordenadas de tamanho D para posições de $K_0 \ldots K_{n-1}$. Se o usuário preferir digitar os pontos é perguntado qual a quantidade desejada e suas respectivas coordenadas. As coordenadas inseridas pelo usuário devem estar dentro do intervalo fechado $[0, \frac{1}{\sqrt{D}}]$.

4.6.2 randPoint

Parâmetros de entrada : Variável global dimension.
Saída: Retorna um vetor point com D números gerados aleatóriamente dentro do intervalo $[0, \frac{1}{\sqrt{D}}]$.

Descrição: A função aloca um vetor point de comprimento dimension e para cada casela gera um número real randômicamente distribuído no intervalo $[0, \frac{1}{\sqrt{D}}]$.

4.7 Esfera

4.7.1 createList

Parâmetros de entrada : Variáveis globais N e seed.
Saída: Retorna um vetor de Point de tamanho N.

Descrição: Dado uma configuração fornecida pelo usuário o programa cria um vetor de Point de tamanho N e atribui N enuplas ordenadas de tamanho D para posições de $K_0 \ldots K_{n-1}$. Se o usuário preferir digitar os pontos é perguntado qual a quantidade desejada e suas respectivas coordenadas. As coordenadas inseridas pelo usuário devem estar dentro do intervalo fechado [-1, -1]. Para que as coordenadas fornecidas pelo usuário estejam na superfície é necessário que sejam compensadas através da seguinte conta:

$$R_{sqrt} = \sum_{i=1}^{n} (a_i)^2 \tag{2}$$

$$\lambda = \sqrt{\frac{0.25}{R_{sqrt}}} \tag{3}$$

$$a_i = a_i \cdot \lambda \ , \forall a_i \text{ em } 0 \ldots n \tag{4}$$

com isso a coordenada antes no espaço agora estará na superfície da hiperesfera.

4.7.2 randPoint

Parâmetros de entrada : Variável global dimension.
Saída: Retorna um vetor point com D números gerados aleatoriamente, tal que point está na superfície da hiperesfera.

Descrição: A função aloca um vetor point de comprimento dimension e para cada casela gera um número real randômicamente distribuído no intervalo [-1,1]. Após isso as coordenadas são compensadas através de (2), (3) e (4) de modo que o ponto esteja na superfície da hiperesfera.

5 Operation.c

Este módulo agrupa as funções de operação dos pontos e o algoritmo de conexidade.

5.1 Algoritmo de conexidade

Funções

- searchEdge
- initVector
- nodeConex
- binRelat
- arranjeList

Entrada: Um vetor de points e o tamanho N do vetor.
Saída: TRUE ou FALSE, dependendo da conexidade do grafo.

O algoritmo para teste de conexidade se baseia em um método nomeado por mim como "conexidade por raíz comum". Este método utiliza dois vetores de inteiros, flagList que funciona como um vetor de flags para cada ponto, guardando o estatus de inicialização de p_i e adjVector que guarda a raíz de um dado ponto.
Considere o seguinte vetor:

R	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...	-1	-1	-1
i	0	1	2	3	4	5	6	7	8	9	...	$i_n - 3$	$i_n - 2$	$i_n - 1$

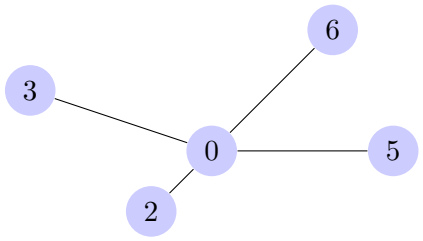
Ele possui tamanho N, com indice indo de 0 até n-1. O valor de cada casela é setado em -1. O algoritmo começa sua análise no primeiro ponto da lista coordenada de gList[i]. O primeiro laço seta a coordenada i e passei através desta lista com o índice k.

```
for(i = 0; i < N; i++){
        for (k = i+1; k < N; k++){
.
.
.
```

Se i for conexo com k é atribuido inicialmente um d para i e este mesmo d é atribuido também para k. Por exemplo, se o d de i for igual a 0 e for conexo com k = (2; 3; 5; 6) teremos:

R	0	-1	0	0	-1	0	0	-1	-1	-1	...	-1	-1	-1
i	0	1	2	3	4	5	6	7	8	9	...	$i_n - 3$	$i_n - 2$	$i_n - 1$

O grafo pareceria com algo assim:

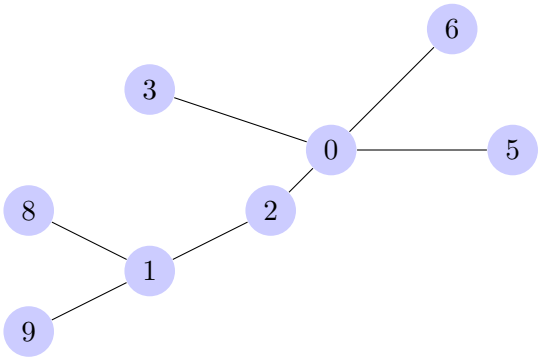


A conexidade é testada na função nodeConex, que compara o valor edge da instância com o valor retornado por distance(), se a distancia for menor ou igual a edge temos vértices conexo, caso contrário são desconexos.

Considere agora que temos i como 1 e com ele são conexos k = (2; 8; 9), teremos a seguinte tabela:

R	0	1	0,1	0	-1	0	0	-1	1	1	...	-1	-1	-1
i	0	1	2	3	4	5	6	7	8	9	...	$i_n - 3$	$i_n - 2$	$i_n - 1$

e o grafo:



Inicialmente a raíz do grafo é o vértice 0. Como visto pelo teste, 0 não é conexo com 8 por arestas, mas sim conexo por caminhos. Como 1 ja era conexo inicialmente com 2 extendemos a conexidade com a raíz 0 para os novos vértices, que são 8 e 9. Atualizando a tabela teremos então:

R	0	1	0	0	-1	0	0	-1	0	0	...	-1	-1	-1
i	0	1	2	3	4	5	6	7	8	9	...	$i_n - 3$	$i_n - 2$	$i_n - 1$

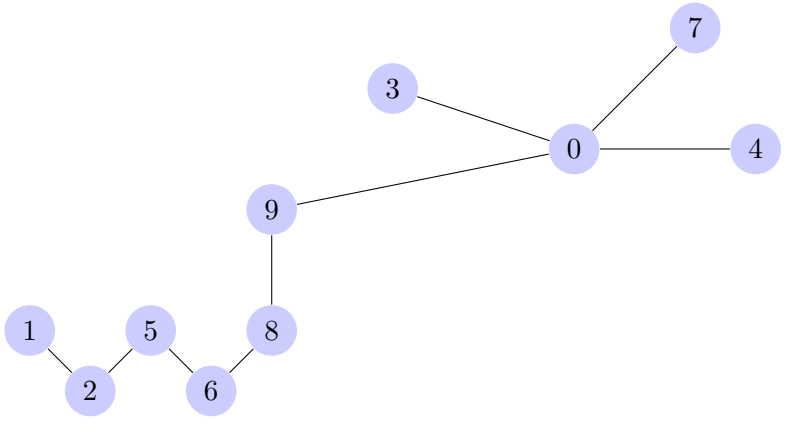
Se i e k possuem uma mesma raíz eles são tidos como vizinhos, e para que isso aconteça estes devem ter o mesmo valor na tabela de conexidades nos índices i e k. Esta verificação é feita no seguinte trecho:

```
if(adjVector[i] == adjVector[k] && adjVector[i]!= -1)
  k++;
.
.
.
```

,caso sejam vizinhos por conexidade direta ou por caminhos k é incrementado, passando para o próximo ponto da lista. Se não forem é testado se a distância entre i e k é menor ou igual a Edge:

```
else if(nodeConex(gList[i],gList[k])){
  binRelat(adjVector,&flagList[i],&flagList[k],i,k);
.
.
.
```

se forem conexos a função binRelat testa se o ponto ja foi inicializado, através da flagList de cada ponto. Dependendo do caso, atribuições são feitas podendo a própria raíz mudar de posição, não ficando estática em 0. Considere o seguinte exemplo:



R	0	-1	-1	0	0	-1	-1	0	-1	0
i	0	1	2	3	4	5	6	7	8	9

Inicialmente nossa raíz era o vértice 0, que teve k = (3; 4; 7; 9) e tabela:

Com i igual a 1 houve apenas conexidade com k = (2), e sucessivamente até i igual e 6 e k = (8).

R	0	1	1	0	0	1	1	0	1	0
i	0	1	2	3	4	5	6	7	8	9

Temos duas raízes e o grafo não está fechado. Na última comparação 8 é conexo com 9, ambas flags estão incializadas e eles não são vizinhos, analisando a função binRelat :

```
    if(!*flagA && !*flagB){
        *flagA = TRUE;
        *flagB = TRUE;
        adjVector[i] = i;
        adjVector[k] = i;
    }
    else{
        if(!*flagB){
            *flagB = TRUE;
            adjVector[k] = adjVector[i];
        }
        else{
            if(!*flagA){
               *flagA = TRUE;
               adjVector[i] = adjVector[k];
            }
            else{
                arrangeList(adjVector, adjVector[i]
                ,adjVector[k]);
            }
        }
    }

}
```

entramos no terceiro else, na chamada da função arrangeList:

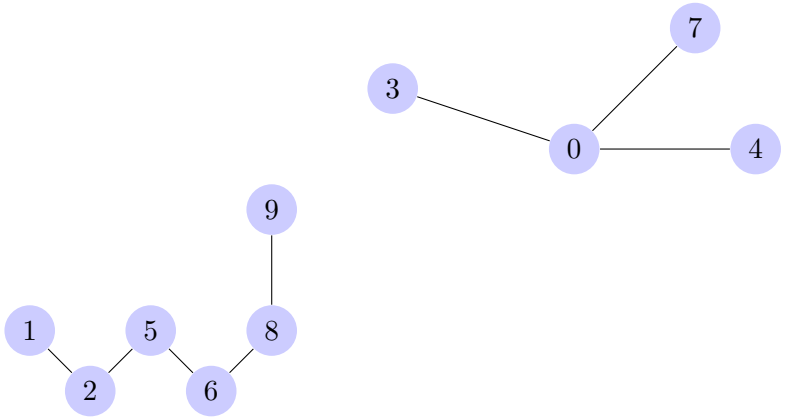
```
    int j;
    for (j = 0; j < N; j++)
        if (adjVector[j] == k)
            adjVector[j] = i;
```

Na função o valor de i, no caso 1, e atribui a todos os k's presente na lista, que são 0. O valor da nova raíz do grafo é 1 e a nova tabela é a seguinte:

R	1	1	1	1	1	1	1	1	1	1
i	0	1	2	3	4	5	6	7	8	9

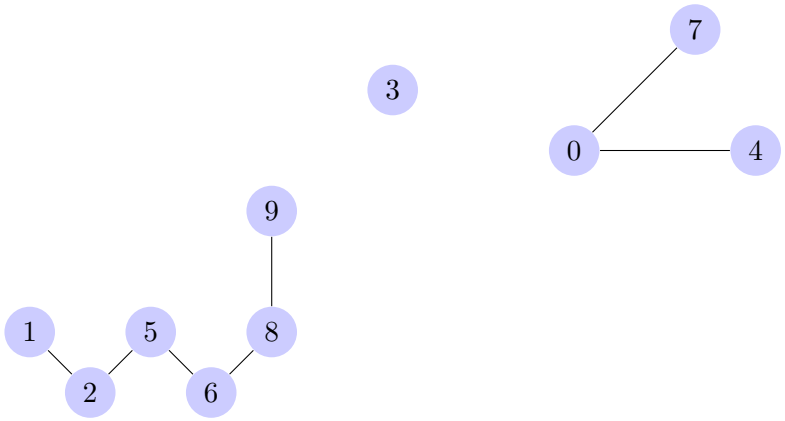
Para um grafo ser conexo dados vértices i e k devem ter uma K raíz comum. Para verificar isto basta que o valor contido em cada casela de índice i de adjVector tenha um mesmo valor K. Caso duas caselas tenham valores diferentes o grafo não é conexo, exemplo:

R	0	1	1	0	0	1	1	0	1	1
i	0	1	2	3	4	5	6	7	8	9



Se caso um dado vértice i ficar excluído de qualquer conexão o grafo também é considerado não conexo. Exemplo:

R	0	1	1	-1	0	1	1	0	1	1
i	0	1	2	3	4	5	6	7	8	9



Após análisar a conexidade de todos as coordenadas o trecho:

```
connectIndx = adjVector[0];
if (connectIndx != -1){
    for (i = 0; i < N; i++){
        if(connectIndx != adjVector[i]){
            conex = FALSE;
            break;
        }
    }
}
```

verifica se todas caselas possuem uma mesma raíz K. Se possuírem o mesmo K, o grafo é conexo e é retornado o valor lógico TRUE, caso um dos valores da lista seja diferente da raíz ou igual a -1, retorna-se FALSE.