Relatorio EP 1
Limiar de Conexidade para certo Grafos Geométricos

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 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.

1.1 int main

Parâmetros de entrada : -Nxxx, -sxxx, -dx.xx, -Mxxx, -v, -V, -D, -C.
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, s e d.
- Cálculo da densidade normalizada crítica.
- 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.

1.2 getArgv

Parâmetros de entrada : argc, argv[ ].
Saída: N, 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.

2 Instance.c

Módulo que concentra as principais funções de instanciação do programa.

2.1 searchDensity

Parâmetros de entrada : M, N, out.
Saída: Informa na tela ou em arquivo de saída (dependendo do valor de out) o valor do limiar de conexividade da instância M $_i$, com $0 \geq i \geq M$, a densidade normalizada critica de M $_i$ e a média das densidades para (N,M).

Descrição: A função recebi suas diretivas de funcionamento e chama a função lessConnectivity para localizar o d_i*. C_i* é cálculado e seu valor é salvo em um vetor. Após o fim da execução de todas instâncias seus resultados são somados e divididos por M.

2.2 lessConnectivity

Parâmetros de entrada : N.
Saída: O menor d tal que o grafo é conexo.

Descrição: A função cria uma nova instância com N pontos e começa a verificação de conexidade com um valor *edge default* de 0.5. O algoritmo adotado é similar a uma busca binária, caso o grafo seja conexo com um dado d este é multiplicado por 0.9, estreitando o valor de busca. Caso seja conexo ainda o menor valor é considerado como teto, e caso o grafo deixe de ser conexo d é multiplicado por 0.05 e somado com si mesmo. O maior valor de não conexidade é salvo e atua como chão, estreitando assim a busca do limiar de conexidade entre um teto e um chão.

O algoritmo é repetido diversas vezes até que a diferentença entre teto e chão seja no máximo de 0.0021, e com isso o valor retornado é o menor d encontrado desde que ele seja conexo.

2.3 newGraph

Parâmetros de entrada : N, v.
Saída: Retorna TRUE ou FALSE, dependendo da conexidade do grafo.

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

3 Creation.c

Este módulo agrupa as principais funções de criação dos elementros que compõem uma instância Grafo, como seus pontos e suas flags de inicialização. Os pontos são criados aleatóriamente com o uso da funcão rand() e uma flag de inicialização é setada para cada par ordenado. Esta flag é utilizada para informar se um dado ponto p_i ja foi verificado pelo algoritmo de conexidade. Ainda neste módulo existe a função reset flag, que trabalha em conjunto com a função lessConnectivity, resetando as flags de uma instância a cada nova execução do algoritmo.

3.1 createList

Parâmetros de entrada : N.
Saída: Retorna um ponteiro para vetor de structs do tipo point de tamanho N.

Descrição: Dado uma configuração fornecida pelo usuário o programa cria um vetor de struct point de tamanho N e atribui N pares ordenados 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.

3.2 r_point

Parâmetros de entrada : Nenhum.
Saída: Retorna uma struct "a"com um par ordenado e uma flag setada em 0.

Descrição: Gera dois pontos aleatóriamente distribuidos dentro do intervalo [0,1] e uma flag de inicialização com valor zero.

3.3 resetFlag

Parâmetros de entrada : Um vetor de structs do tipo point e N.
Saída: Nenhuma.

Descrição: Executa um laço para resetar o valor das flags anteriormente utilizadas.

4 Operation.c

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

4.1 pointOutput

Parâmetros de entrada : Um vetor de structs do tipo point e N.
Saída: Nenhuma.

Descrição: A função passei pelo vetor de struct imprimindo as coordenadas em um arquivo de saida.

4.2 Algoritmo de conexidade

Funções

- searchEdge
- initVector
- distance.
- binRelat.
- arranjeList.

Entrada: Um vetor de struct point 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 um vetor de inteiros de tamanho N para salvar um dada raiz e a correspondência de suas ramificações.
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	...	$i_n - 3$	$i_n - 2$	$i_n - 1$

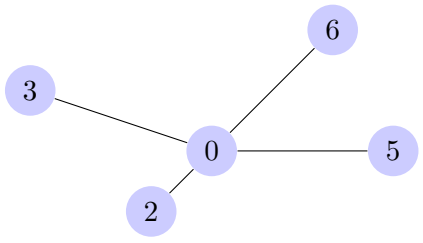
Ele possui tamanho N, com indice indo de 0 até n-1. O valor de cada casela é setado em -1. O algoritmo começa sua análise no primeiro ponto da lista coordenada de gList[i]. O primeiro laço seta a coordenada i e passei através desta lista com o Índice k.

```
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:

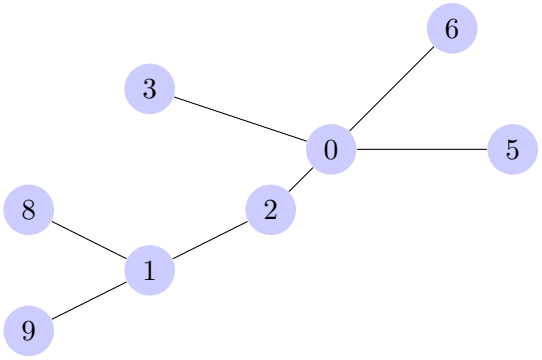
R	0	-1	0	0	-1	0	0	-1	-1	-1	...	-1	-1	-1
i	0	1	2	3	4	5	6	7	8	9	...	$i_n - 3$	$i_n - 2$	$i_n - 1$

O grafo pareceria com algo assim:



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

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	...	$i_n - 3$	$i_n - 2$	$i_n - 1$



Inicialmente a raíz do grafo é o vértice 0. Como visto pelo teste, 0 não é conexo com 8 por vértice, mas sim conexo por caminhos. Como 1 ja era conexo inicialmente com 0 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	...	$i_n - 3$	$i_n - 2$	$i_n - 1$

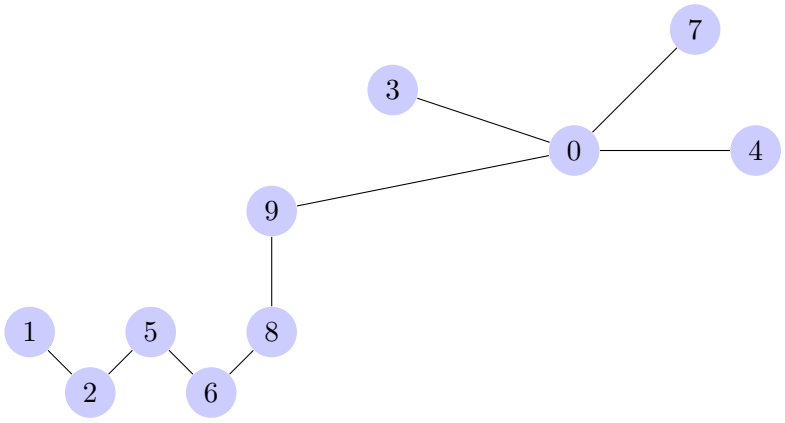
Se i e k possuem uma mesma raíz eles são tidos como vizinhos, e para que isso aconteça estes devem ter o mesmo valor na tabela de conexidades nos índices i e k. Esta verificação é feita no seguinte trecho:

```
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(distance(gList[i],gList[k]))
    binRelat(adjVector,&gList[i],&gList[k],i,k, N);
.
.
.
```

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



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

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

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(!A->initFlag && !B->initFlag){
        A->initFlag = TRUE;
        B->initFlag = TRUE;
        adjVector[i] = i;
        adjVector[k] = i;
    }
    else{
        if(!B->initFlag){
            B->initFlag = TRUE;
            adjVector[k] = adjVector[i];
        }
        else{
            if(!A->initFlag){
                A->initFlag = TRUE;
                adjVector[i] = adjVector[k];
            }
            else{
```

```
                arrangeList(adjVector, adjVector[i]
                ,adjVector[k], N);
                }
        }
    }
```

entramos no terceiro else, na chamada da função arrangeList:

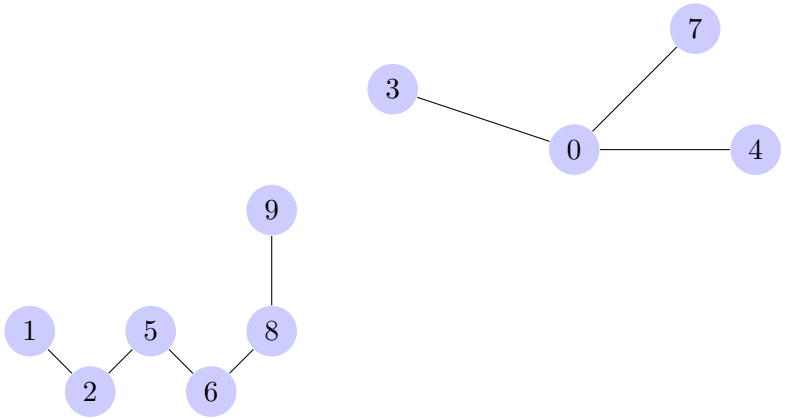
```
    int j;

  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

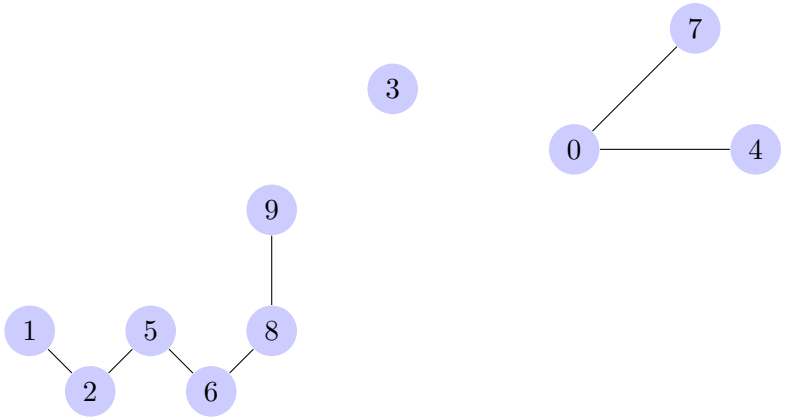
Para um grafo ser conexo dados vértices i e k devem ter uma K raíz comum. Para verificar isto basta que o valor cada casela de índice i do vetor de raíz tenha um mesmo valor K. Caso duas caselas tenham valores diferentes o grafo não é conexo, exemplo:

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 todos valores contidos na tabela e a igualdade com a 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.

5 Graph.h

Este header contem as duas estruturas de dados utilizadas ao longo do EP. A estrutura Grafo contem um ponteiro para struct point, que representa a lista de coordenadas de um Grafo G. Na estrutura point temos as coordenadas (x,y) de um vértice i definidas como variáveis flutuantes e uma flag de inicialização que ajuda durante o algoritmo de conexidade.