Teoria algébrica de grafos, um trabalho computacional

Artur Teles

FGV-EMAP/Mat. Aplicada

1 Introdução

Este projeto tem como objetivo verificar alguns pontos teóricos de teoria algébrica de grafos já vistos durante o curso de Matemática Discreta, e alguns que não foram, computacionalmente. Portanto, irei omitir a prova da maioria dos teoremas presentes para poder trabalhar com mais resultados dado que o número de páginas deste documento foi limitado previamente.
Além de verificar o número de caminhos de tamanho k entre dois vértices de um grafo analisando a k-ésima potência da sua matriz de adjacência, analiseremos também o espectro da matriz. Adicionalmente, definiremos a matriz laplaciana de um grafo e nos perguntaremos o que seu espectro nos diz sobre o grafo, além de discutir alguns casos específicos como grafos regulares, bipartidos, completos etc.
Toda a parte computacional deste projeto foi feita em C usando bibliotecas matemáticas como Lapacke e Openblas. Caso seja necessário, os arquivos de código possuem comentários que dizem o que cada função faz de forma geral (em arquivos .h) e alguns pontos de implementação (em arquivos .c).

1.1 Como rodar os programas (caso necessário)

Não é necessário rodar nenhum código desse projeto, por mais que seja recomendado. Primeiro, a estrutura do projeto se encontra da seguinte forma:

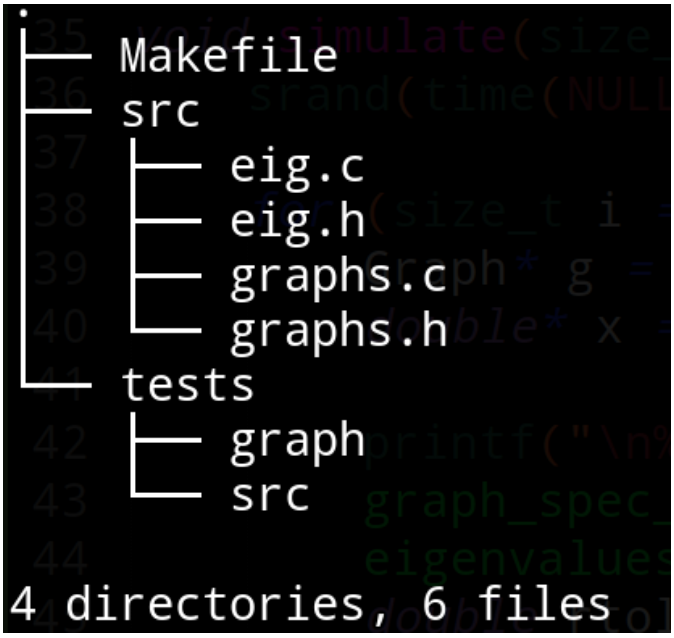

Figura 1: Estrutura do projeto. em /src estão os arquivos principais do projeto, enquanto em tests/src estão os arquivos onde estão as funções de simulação.

Esse projeto depende de bibliotecas como LAPACKE e OPENBLAS. Outra coisa importante a se dizer é que o projeto foi feito em um computador linux. Para compilar todos os arquivos, estando na raiz do projeto:

```
make
```

Para rodar todos os tests ao mesmo tempo:

```
make tests
```

Para rodar um test em específico:

```
make tests-one NAME=nomedotest
```

Para limpar os arquivos compilados e executáveis:

```
make clean
```

2 A matriz de adjacência e espectro de um grafo simples

Seja $G = (V, E)$ um grafo simples, então definimos $A = A(G)$ sua matriz de adjacência onde as esntradas a_{ij} são dadas por:

$$a_{ij} = \begin{cases} 1, & \text{se } v_i \text{ e } v_j \text{ são adjacentes} \\ 0, & \text{c.c.} \end{cases}$$

Segue diretamente da definição que A é simétrica e $Tr(A) = 0$.
Em C, um grafo pode ser definido da seguinte forma:

```c
typedef struct {
    size_t n;           /* N° de vértices*/
    bool directed;      /* Grafo direcionado?*/
    double* A;          /* Matriz de adjacência*/
} Graph;
```

Figura 2: Definição de um grafo em C. (graphs.h)

Suponha que λ é um autovalor de A. Portanto, como A é real e simétrica, λ é real com multiplicidade, como uma raiz de $det(\lambda I - A) = 0$, igual à dimensão do autoespaço correspondente a λ.
Define-se então o espectro de um grafo G como o conjunto de números que são autovalores de $A(G)$ junto com suas multiplicidades:

$$Spec(G) = \begin{pmatrix} \lambda_0 & \lambda_1 & ... & \lambda_{s-1} \\ m(\lambda_0) & m(\lambda_1) & ... & m(\lambda_{s-1}) \end{pmatrix}$$

2.1 Polinômio característico

A primeira propriedade que iremos verificar computacionalmente envolve os coeficientes de $\chi(G; \lambda) = \lambda^n + c_1\lambda^{n-1} + ... + c_n$, o polinômio característico de A:
(1) $c_1 = 0$
(2) $-c_2 = |E|$
(3) $-c_3$ é igual a duas vezes o número de triângulos em G

Para isso, vamos criar grafos aleatórios de tamanhos quaisquer, achar os coeficientes do polinômio característico da adj. matriz e verificar se os valores obtidos

são iguais aos esperados, onde esses valores esperados são calculados separadamente usando outras funções. Podemos usar uma função igual ou parecida com a seguinte para verificar o resultado:

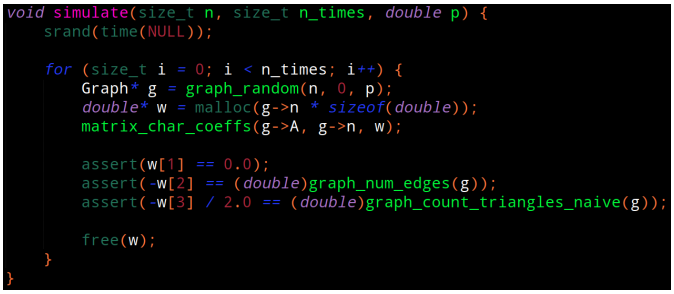

Figura 3: Verificando coeficientes do polinômio característico. (coef_pol_car.c)

E ao rodar o programa com parâmetros, por exemplo, $n = 10$, $n\_times = 100$ e $p = 0.5$, percebemos que nenhum assert gerou algum erro, indicando que todas as expressões eram verdadeiras.

2.2 Número de passos entre dois vértices

O número de passos (pode repetir vértices) de tamanho k entre dois vértices v_i e v_j de um grafo é igual a $(A^k)_{ij}$.
Prova: se $k = 0$, então o resultado é imediato. Agora suponha que o resultado é verdadeiro para $k = K$. Ir de v_i até v_j em $K + 1$ é equivalente a andar K passos de v_i até um vizinho v_h de v_j e depois ir de v_h até v_j. Portanto:

$$\text{(passos de tamanho } K+1) = \sum_{\{v_h,v_j\}\in E(G)} (A^K)_{ij} = \sum_{h=1}^{n} (A^K)_{ij} a_{hj} = (A^{K+1})_{ij}.$$

Portanto o número de passos de tamanho $K + 1$ entre v_i e v_j é igual a $(A^{K+1})_{ij}$.
O resultado geral se segue por indução.
Para verificarmos isso computacionalmente, vamos, por exemplo, usar um K_4 com vértices numerados da seguinte forma:

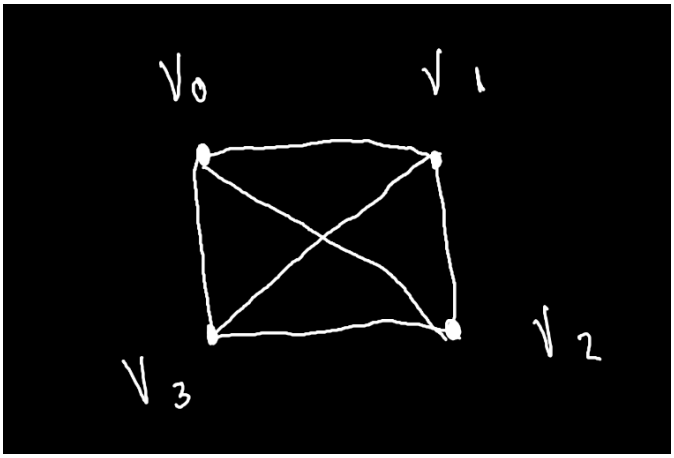

Figura 4: Grafo completo de 4 vértices.

Vamos achar o número de passos de até tamanho 3 entre v_0 ev_2:

```c
int main() {
    Graph* g = graph_kn(4);

    for (size_t i = 1; i < g->n; i++) {
        int paths = (int) graph_paths_length(g, 0, 2, i);
        printf("número de passos de tamanho %ld = %d \n", i, paths);
    }
```

Figura 5: Simulação de passos. (k4_paths.c)

O resultado é o seguinte:

```
=== tests/bin/k4_paths ===
número de passos de tamanho 1 = 1
número de passos de tamanho 2 = 2
número de passos de tamanho 3 = 7
```

Figura 6: Número de passos.

Onde todos os passos de tamanho 3 são:
$0 \to 2 \to 1 \to 2$
$0 \to 2 \to 3 \to 2$
$0 \to 3 \to 1 \to 2$
$0 \to 1 \to 3 \to 2$
$0 \to 3 \to 0 \to 2$
$0 \to 1 \to 0 \to 2$
$0 \to 2 \to 0 \to 2$

2.3 Grafos completos

A matriz de adjacência de um grafo completo G de n vértices é da seguinte forma:

$$a_{ij} = \begin{cases} 1, \text{ se } i \neq j \\ 0, \text{ se } i = j \end{cases}$$

Ou seja, $A = J - I$ onde J é a matriz formada por 1's e I é a identidade.
J tem autovalor n com multiplicidade 1 e autovalor 0 com multiplicidade $n-1$, portanto o espectro de G tem a seguinte forma:

$$Spec(G) = \begin{pmatrix} n-1 & -1 \\ 1 & n-1 \end{pmatrix}$$

Para verificarmos isso computacionalmente, vamos simular criar grafos conexos de diversos tamanhos e verificar quais são seus autovalores e multiplicidades associadas. A função é esta:

```c
void simulate(size_t maxit) {
    assert(maxit >= 3);
    double rtol = 1e-9;
    double atol = 1e-12;

    for (size_t i = 3; i <= maxit; i++) {
        Graph* g = graph_kn(i);
        double* w = malloc(i * sizeof(double));

        graph_spec_adj(g, w);
```

Figura 7: Implementação da simulação. Primeiro criamos um grafo completo e depois achamos os autovalores de seu espectro e suas respectivas multiplicidades. (eig_kn.c)

Verificando resultados para $maxit = 20$:

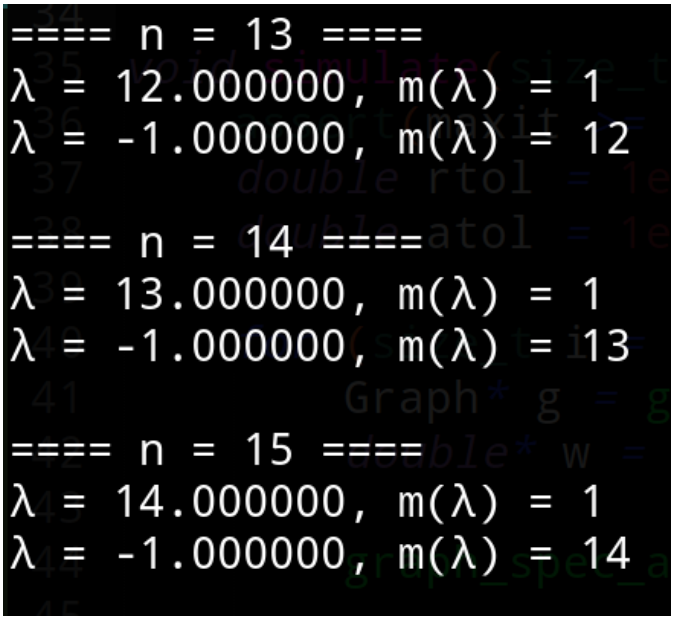

Figura 8: Resultado da implementação para maxit = 20.

2.4 Grafos regulares

Se G é um grafo regular de grau k, então:

1) k é autovalor de G
2) se G é conexo, então $m(k) = 1$
3) para qualquer autovalor λ de G, tem-se que $|\lambda| \leq k$

Aqui está a função que usaremos:

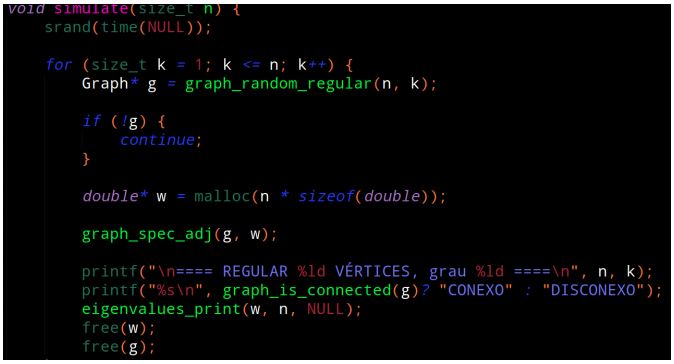

Figura 9: Função de simulação para verificar propriedades do espectro de grafos regulares. Primeiro criamos um grafo regular aleatório de n vértices e de grau k para todo $k \leq n$ e que é possivel gerar um grafo regular, depois verificamos se ele é conexo e depois printamos todos seus autovalores. (regular_graphs.c)

E aqui uma instância do resultado para $n = 10$:

```
==== REGULAR 10 VÉRTICES, grau 4 ====
CONEXO
λ[0] = -2.966448
λ[1] = -1.714715
λ[2] = -1.618034
λ[3] = -1.000000
λ[4] = -1.000000
λ[5] = -0.000000
λ[6] = 0.618034
λ[7] = 1.348414
λ[8] = 2.332749
λ[9] = 4.000000
```

Figura 10: Resultado da implementação para $n = 10$.

2.5 Grafos bipartidos

Seja G um grafo bipartido e X e Y a partição de vértices correspondente, então se você enumerar os vértices de forma que os de X venham primeiro e depois os de Y, a matriz de adjacência de G terá a seguinte cara:

$$A = \begin{pmatrix} 0 & B \\ B^T & 0 \end{pmatrix}$$

Onde o bloco B indica as arestas entre X e Y. A recíproca também é verdadeira.

Outro fato interessante é que o espectro de G é simétrico em torno de 0, ou seja, se λ é autovalor, então $-\lambda$ também é. O mais surpreendente é que a recíproca também é verdadeira.
Vamos provar essa última afirmação (a primeira implicação):

Suponha que G é bipartido, então sua matriz de adjâcência é como o primeiro resultado diz, então:

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} Bx \\ B^T y \end{pmatrix}$$

Portanto, se $\begin{pmatrix} x \\ y \end{pmatrix}$ é autovetor com autovalor λ então $\begin{pmatrix} x \\ -y \end{pmatrix}$ é auutovetor com autovalor $-\lambda$.
A simulação deste subcapítulo se resume em gerar grafos bipartidos a partir do primeiro teorema (matriz em blocos) e verificar o segundo (espectro simétrico). Veja o arquivo graphs.c para detalhes de implementação.
Aqui está a função de simulação:

```c
void simulate(size_t n1, size_t n2, double p, size_t maxit) {
    srand(time(NULL));

    for (size_t i = 0; i < maxit; i++) {
        Graph* g = graph_random_bipartite(n1, n2, p);

        double* w = malloc(g->n * sizeof(double));

        graph_spec_adj(g, w);

        printf("\n==== n1 = %ld n2 = %ld ====\n", n1, n2);
        graph_print(g, "Matriz de adjacência");
        printf("\n");
        eigenvalues_print(w, g->n, "Autovalores");

        free(w);
        free(g);
    }
}
```

Figura 11: Implementação da função de simulação. Criamos grafos bipartidos aleatórios, depois calculamos seus autovalores. No fim printamos tanto sua matriz de adjacência quanto seus autovalores. (bipartite_graphs.c)

Instância do resultado de simulate com $n_1 = 5$ e $n_2 = 3$:

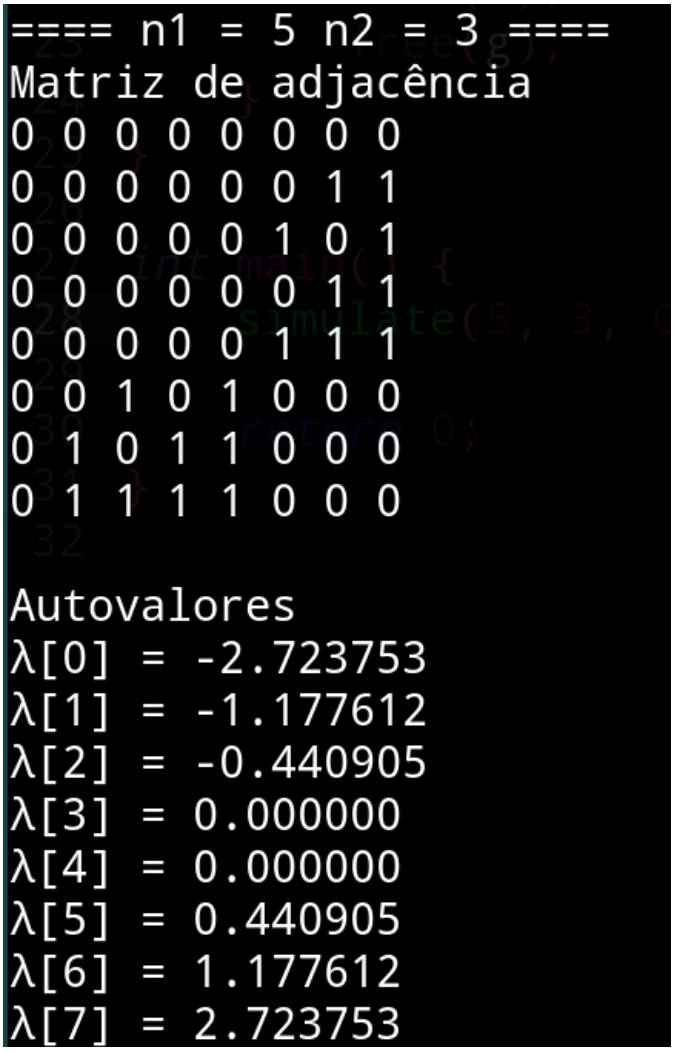

Figura 12: Matriz de adjacência e autovalores do grafo gerado pela simulação.

3 A laplaciana e a matriz de incidência de um grafo

Vamos terminar este projeto falando sobre a laplaciana e a matriz de incidência de um grafo. Esses dois conceitos nos permitem descobrir quantas componentes conexas um grafo tem e algumas outras coisas sobre os ciclos presentes nele.

3.1 Matriz de incidência

A matriz de incidência de um grafo $G = (V, E)$ com uma determinada orientação (a orientação é apenas uma conveniência para não termos que falar de soma módulo 2) é a matriz ∂ definida da seguinte forma:

$$\partial_{ij} = \begin{cases} 1, \text{ se } v_i \text{ é a origem de } e_j \\ -1, \text{ se } v_i \text{ é o destino de } e_j \\ 0, \text{ caso contrário} \end{cases}$$

Portanto, as linhas de ∂ correspondem aos vértices de G, enquanto as colunas às arestas.
Um fato importante sobre ∂ é o seguinte: se G tem n vértices e c componentes conexas, então $rank(\partial) = n - c$. A prova vem ao entendermos que a soma dos elementos de uma coluna é sempre zero e, portanto, a soma das linhas de ∂ somam zero, isto é, elas são linearmente dependentes (dado os espaços vetoriais corretos, os quais preferi omitir desta discussão) e, portanto, $rank(\partial) <= n-1$. Além disso, se ordenarmos os vértices de G de uma forma "inteligente", podemos escrever ∂ como uma matriz diagonal em blocos onde os blocos na diagonal correspondem à matriz de incidência de uma determinada componente conexa. Depois disso, basta mostrar que o rank desse bloco é igual $n_i - 1$ onde n_i é o número de vértices na componente conexa correspondente.
Antes de continuarmos, vamos fazer uma simulação com o seguinte grafo de 3 componentes conexas:

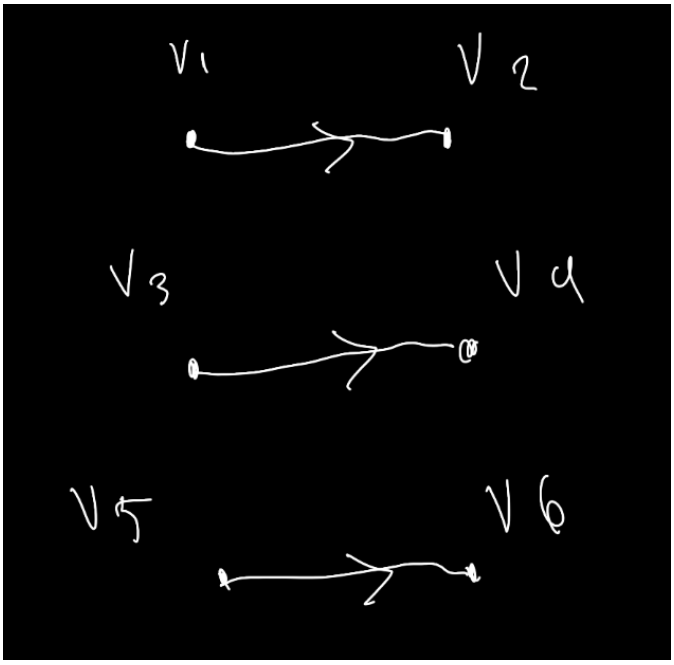

Figura 13: Grafo orientado de 3 componentes conexas.

Nossa função de simulação é a seguinte:

```c
void simulate() {
    double A[] = {
        0, 1, 0, 0, 0, 0,
        1, 0, 0, 0, 0, 0,
        0, 0, 0, 1, 0, 0,
        0, 0, 1, 0, 0, 0,
        0, 0, 0, 0, 0, 1,
        0, 0, 0, 0, 1, 0
    };

    Graph g = {6, true, A};

    double* B = graph_incidence_matrix(&g);
    print_matrix(B, g.n, graph_num_edges(&g));

    unsigned int rank = matrix_rank(B, g.n, graph_num_edges(&g), 10e-9);
    printf("rank de B: %d\n", rank);
    printf("componentes conexas: %d\n", (int) g.n - rank);

    free(B);
}
```

Figura 14: Função de simulação para calcular número de componentes conexas do grafo definido pela matriz de adjacência A. Primeiro criamos a matriz de incidência e achamos seu rank. Logo depois achamos o número de componentes conexas e o printamos junto com a matriz de incidência. (incidence_matrix.c)

Com resultado:

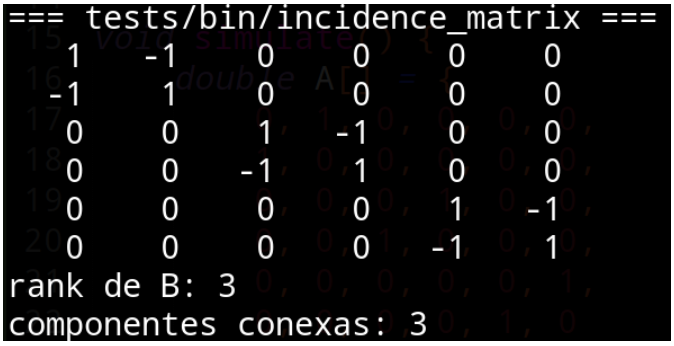

Figura 15: Resultado da simulação.

Outras propriedades que podemos tirar sobre G a partir de ∂ (as quais não farei simulações por fugirem do escopo) são as seguintes:

1) O núcleo/kernel de ∂ é o espaço vetorial formado por ciclos de G.
2) O complemento ortogonal de $kernel(\partial)$ é o espaço vetorial formado pelos cortes (cuts) de G.

3.2 Laplaciana

A matriz laplaciana de um grafo G de n vértices é a matriz L definida da seguinte forma:

$$L = \partial\partial^t = D - A,$$

onde D é uma matriz nxn diagonal onde a entrada na diagonal é o grau do vértice v_i $(1 \leq i \leq n)$ e A é a matriz de adjacência.

Vem da definição que:
1) L é simétrica
2) L é positiva semidefinida.

Também temos as seguintes propriedades:
1) 0 é sempre um autovalor de L e (número de componentes conexas de G) = $m(0) = dim(kernel(L))$.
2) λ_1 (segundo menor autovalor de L) é positivo quando G é conexo.

Vamos verificar as propriedades anteriores criando grafos aleatórios, verificar se são conexos ou não e ver o espectro de sua laplaciana. Para isso, vamos usar a seguinte função:

```c
void simulate(size_t n, size_t maxit, double p) {
    srand(time(NULL));

    for (size_t i = 0; i < maxit; i++) {
        Graph* g = graph_random(n, p);
        double* x = malloc(g->n * sizeof(double));

        printf("\n%s\n", (graph_is_connected(g))? "CONEXO" : "DISCONEXO");
        graph_spec_lap(g, x);
        eigenvalues_print(x, g->n, "Autovalores da laplaciana");
        double rtol = 1e-9;
        double atol = 1e-12;
        unsigned int count_zero = count_mult(x, 0.0, g->n, rtol, atol);
        printf("número de componentes conexas: %d\n", count_zero);

        free(g);
        free(x);
    }
}
```

Figura 16: Simulação da laplaciana. Começamos criando um grafo aleatório e depois achando os autovalores de sua matriz laplaciana. Devemos ver que 0 sempre é um autovalor de L e que $\lambda_1 > 0$ se G for conexo. (laplacian.c)

Rodando a simulação com $n = 10$, $maxit = 10$ e $p = 0.5$, achamos os seguintes resultados para grafos conexos e disconexos:

```
CONEXO
Autovalores da laplaciana
λ[0] = 0.000000
λ[1] = 1.356873
λ[2] = 1.730407
λ[3] = 2.320301
λ[4] = 2.665422
λ[5] = 3.427809
λ[6] = 4.162102
λ[7] = 5.989318
λ[8] = 6.540565
λ[9] = 7.807204
número de componentes conexas: 1
```

Figura 17: Grafo conexo.

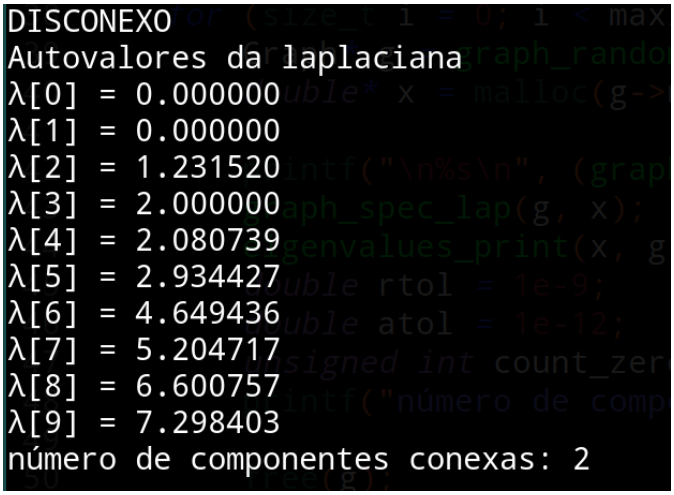

Figura 18: Grafo disconexo.

Um outro teorema interessante, mas que não vou verificar computacionalmente, é o teorema da matriz de Kirchhoff (teorema das árvores), que diz que:

$\tau(G) = \frac{1}{n} \prod_{i=2}^{n} \lambda_i,$

onde $\tau(G)$ é o número de spanning trees (árvores geradoras) de G e λ_i são os autovalores não-nulos de L. Ou seja, o teorema diz que o determinante de qualquer submatriz de L ao removermos uma linha e uma coluna é o número de spanning trees de G.

4 Bibliografia

- Algebraic Graph Theory (Norman Biggs)
- Graph Theory (Reinhard Diestel)
- Numerical Linear Algebra (Trefethen-Bau)