ANDER CONSELVAN DE OLIVEIRA

UM GERADOR DE NÚMEROS PSEUDO-ALEATÓRIOS BASEADO NO PRODUTO ZIG-ZAG DE GRAFOS EXPANSORES

Trabalho de graduação apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Federal do Paraná, como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Jair Donadelli Júnior

Curitiba

2006

Resumo

Grafos expansores são grafos com boa conexidade, em que cortes pequenos desconectam apenas pequenas partes do grafo. Suas aplicações na ciência da computação são bastante conhecidas. Uma delas, a pseudo-aleatoriedade, pode ser usada na solução de problemas de complexidade computacional. Neste trabalho, apresentamos uma implementação de um gerador de números pseudo-aleatórios baseado nessa característica de grafos expansores. Para implementá-lo, foi necessário superar o problema da construção desses grafos, que apesar de abundantes, são computacionalmente difíceis de identificar. O método de construção adotado é baseado em um produto entre grafos, o produto zig-zag. Testes realizados com o gerador implementado confirmam os resultados obtidos analiticamente, mostrando a aplicabilidade de grafos expansores e da construção baseada no produto zig-zag.

Sumário

1 Introdução p. 3

1.1 Notação . . . . . p. 4

2 Grafos Expansores p. 5

2.1 Expansores e Teoria Espectral de Grafos . . . . . p. 5

2.2 Passeios aleatórios em grafos . . . . . p. 7

3 Construção de grafos expansores através do produto zig-zag p. 9

3.1 O produto zig-zag . . . . . p. 9

3.2 Quadrado de um grafo . . . . . p. 11

3.3 Produto Tensor . . . . . p. 12

3.4 Família de expansores . . . . . p. 12

3.4.1 Escolha do grafo base . . . . . p. 13

4 Gerador de números pseudo-aleatórios p. 14

4.1 Implementação . . . . . p. 14

4.2 Análise experimental . . . . . p. 16

4.2.1 Resultados observados . . . . . p. 17

5 Conclusão p. 20

Referências p. 21

Apêndice A – Código fonte p. 22

1 Introdução

Grafos expansores são grafos esparsos altamente conexos. A aplicação desses grafos ajudou a solução de muitos problemas da ciência da computação, entre eles, problemas de complexidade computacional, construção de códigos de correção de erros, modelagem de redes de comunicação e desaleatorização. Esses grafos também têm aplicações em matemática pura, como em topologia, teoria dos grupos, teoria dos números e teoria da medida. Veja [6] e as referências de [6] para mais detalhes sobre aplicações de grafos expansores.

A existência de grafos expansores pode parecer improvável, mas um simples agumento probabilístico mostra que quase todo grafo regular é um expansor [6]. Porém, construí-los explicitamente pode ser complicado e a maioria das contruções conhecidas baseiam-se em conceitos de matemática pura, como teoria dos números. No entanto, a construção baseada no produto zig-zag, que veremos no capítulo 3 é uma construção preponderantemente combinatória.

Neste trabalho, estamos interessados nas propriedades de pseudo-aleatoriedade dos grafos expansores. Em especial, um passeio aleatótio num grafo expansor converge rapidamente para a distribuição estacionária. Essa propriedade é importante, por exemplo, no seguinte resultado em complexidade computacional.

Teorema 1 *Dado um algoritmo* **BPP** *que utiliza m bits aleatórios por execução, sua probabilidade de fracasso pode ser reduzida para $\frac{1}{2^k}$ usando não mais que $O(m+k)$ bits aleatórios.*

A classe de problemas **BPP** inclui todas as linguagens L que podem ser reconhecidas por um algoritmo A tal que

- Se a string x pertence a L, então A aceita x com probabilidade maior que $\frac{2}{3}$.
- Se a string x não pertence a L, então A rejeita x com probabilidade maior que $\frac{2}{3}$.

No capítulo 2, veremos a definição de grafo expansor e algumas de suas propriedades relevantes.

Além de apresentar resultados conhecidos sobre expansores, o propósito deste trabalho é testar a pseudo-aleatoriedade desses grafos. No capítulo 4, discutiremos a implementação de um gerador de números pseudo-aleatórios que utiliza as ferramentas citadas acima e a análise de testes estatísticos realizados nas seqüências geradas por ele.

1.1 Notação

Para um grafo G, as seguintes notações são utilizadas nesse documento:

1. V(G) é conjunto de vértices do grafo G.
2. E(G) é conjunto de arestas do grafo G.
3. Para $v, w \in V(G)$, denotamos por vw o conjunto $\{v, w\}$.
4. O grau do vértice v, denotado como $d(v)$ é igual a $\left|\{w \in V(G) : wv \in E(G)\}\right|$, ou seja, a quantidade de vértices adjacentes a v;
5. Se G é um grafo d-regular então $d(v) = d$, para todo $v \in V(G)$, e dizemos que o grau de G é d.
6. Seja $S \subseteq V(G)$, então denotamos por complemento de S o conjunto $\bar{S} = V(G) - S$.
7. Seja $S \subseteq V(G)$, então $\partial S = \{\{v, w\} : v \in S, w \in \bar{S}\}$, ou seja, o conjunto de arestas com um vétice em S e outro em $\bar{S}$.
8. Um passeio em um grafo é uma seqüência de vértices $v_1, v_2, \ldots, v_k$, em que v_i é adjacente a v_{i+1} para todo $i < k$. O tamanho de um passeio é o número de vértices na seqüência.
9. Um multigrafo é um grafo que pode ter arestas paralelas e laços, Ou seja, num multigrafo, dois vértices podem ser conectados por mais de uma arestas e podem existir arestas que conectam um vértice a ele mesmo.
10. Para $n \in \mathbb{N}$, o conjunto $[n]$ é igual a $\{1, 2, \ldots, n\}$.

2 Grafos Expansores

Informalmente, dizemos que um grafo é expansor se todo subconjunto de vértices S tem muitos vizinhos em seu complemento $\bar{S}$. Para formalizar esse conceito, utilizaremos as definições a seguir:

Definição 1 *A constante isoperimétrica de um grafo G é definida como*

$$h(G) = \min_{\substack{S \in V(G) \\ |S| \leq \frac{|V|}{2}}} \frac{|\partial(S)|}{|S|}.$$

Dizemos que um grafo G é ϵ-aresta-expansor se $h(G) \geq \epsilon$. Note que para todo grafo conexo, existe um ϵ para o qual ele é ϵ-aresta-expansor.

Definição 2 *Uma família de grafos $\mathcal{G} = \{G_1, G_2, \dots\}$ de grafos d-regulares, para d fixo, é uma família de expansores se $h(G_i) \geq c > 0$ para todo i, ou seja, G_i é c-aresta-expansor, e $|V(G_i)| = n_i$, onde n_i é uma função crescente em i.*

O problema que tem como entrada um grafo G e um $\epsilon > 0$ e decide se G é um grafo ϵ-aresta-expansor é *co-NP*-difícil. Isso acontece porque se um grafo é um ϵ-aresta-expansor então sua constante isoperimétrica é maior que ϵ e determiná-la é um problema *co-NP*-difícil [4]. Porém, é possível aproximar a constante isoperimétrica de um grafo através das técnicas da Teoria Espectral de Grafos.

2.1 Expansores e Teoria Espectral de Grafos

A Teoria Espectral de Grafos é o estudo de grafos em relação a sua matriz de adjacências, definida abaixo.

Definição 3 *A matriz de adjacências de um grafo G com n vértices é uma matriz $(a_{i,j})_{n\times n}$, onde*

$$a_{i,j} = \begin{cases} 1, & \text{se} \quad ij \in E(G) \\ 0, & \text{se} \quad ij \notin E(G) \end{cases}$$

Os autovalores da matriz de adjacências de um grafo, definidos a seguir, serão de particular interesse.

Definição 4 *Seja A uma matriz real quadradada $n \times n$. Dizemos que $\lambda \in \mathbb{C}$ é um autovalor de A se existe um vetor $\mathbf{v} \neq \mathbf{0}$ tal que*

$$A\mathbf{v} = \lambda\mathbf{v}$$

Dizemos que $\mathbf{v}$ é um autovetor associado a λ. Para matrizes reais simétricas, todos os autovalores são reais [2].

Sabemos que o maior autovalor λ_1 da matriz de adjacências de um grafo d-regular é igual a d. A diferença entre o primeiro e o segundo maior autovalor em valor absoluto, chamada de diferença espectral (*spectral gap*), é um indicicador do parâmetro de expansão do grafo, como mostra o teorema a seguir.

Teorema 2 ([2]) *Se um grafo G é conexo e d-regular, então*

$$\frac{d-\lambda}{2} \leq h(G) \leq \sqrt{2d(d-1)},$$

onde $\lambda = \max\{|\lambda_i| : i = 2, 3, \ldots n\}$.

Portanto, para um d fixo, um menor valor de λ garante um maior valor para $h(G)$. Tendo em vista essa relação, consideraremos que grafos expansores são grafos cujo segundo maior autovalor é pequeno. Para nos referirmos a esses grafos utilizaremos a definição abaixo.

Definição 5 *Um grafo (n, d, λ)-expansor é um grafo d-regular com n vértices cujo segundo maior autovalor (em valor absoluto) de sua matriz de adjacências normalizada é $\lambda \in [0, 1] \subseteq \mathbb{R}$.*

2.2 Passeios aleatórios em grafos

Um passeio aleatório num grafo é um percurso $v_0, v_1, \ldots, v_i, \ldots$ iniciado em algum vértice v_0 do grafo, em que o $i+1$-ésimo vértice do percurso v_{i+1} é escolhido uniformemente de maneira aleatória entre os vizinho do i-ésimo vértice v_i. Um passeio aleatório num grafo é um processo estocástico que pode ser representado por uma Cadeia de Markov, como veremos a seguir.

Definição 6 (Cadeia de Markov) *Uma Cadeia de Markov $\mathcal{M}$ é um processo estocástico discreto definido sobre um conjunto de estados S em termos de uma matriz de transição P. A matriz P tem uma linha e uma coluna para cada estado de S. Em qualquer tempo, a cadeia está em algum estado. Por simplicidade, assumiremos que o conjunto de estados de $\mathcal{M}$ é igual a $[n]$, para algum inteiro n. O elemento P_{ij} da matriz P é a probabilidade de que o próximo estado seja j dado que o estado atual é i. Portanto, $0 \leq P_{ij} \leq 1$ e $\sum_{j \in S} P_{ij} = 1$ para todo $i \in S$.*

A Cadeia de Markov do passeio aleatório em um grafo tem como conjunto de estados o conjunto de vértices do grafo. A matriz de probabilidade de transição é definida como $P = (P_{ij})_{|V(G)| \times |V(G)|}$ onde

$$P_{ij} = \begin{cases} \frac{1}{d(i)}, & \text{se} \quad ij \in E(G) \\ 0, & \text{se} \quad ij \notin E(G) \end{cases}$$

No caso em que o grafo é d-regular, a matriz de probabilidade de transição é igual a $\frac{1}{d}A$, uma matriz simétrica, e a chamamos de matriz de adjacências normalizada.

Considere o vetor linha $\mathbf{q^{(t)}} = (q_1^{(t)}, q_2^{(t)}, \ldots, q_n^{(t)})$, onde $q_i^{(t)}$ é a probabilidade do estado atual da cadeia ser i no instante t. Temos, então, que $q_j^{(t+1)} = \sum_{i \in S} q_i^{(t)} P_{ij}$. Logo, $\mathbf{q^{(t+1)}} = \mathbf{q^{(t)}} P$. Quando $\mathbf{q^{(t+1)}} = \mathbf{q^{(t)}}$, dizemos que $\mathbf{q^{(t)}}$ é uma distribuição estacionária da cadeia, denotada apenas como π.

Como a matriz de transição do passeio aleatório num grafo d-regular é simétrica, a soma dos elementos de uma de suas colunas totaliza 1. Portanto, se tomarmos $\pi = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$, temos que $\pi P = (\pi_j)_{1 \times n}$, onde

$$\pi_j = \sum_{i \in S} \frac{1}{n} P_{ij} = \frac{1}{n} \sum_{i \in S} P_{ij} = \frac{1}{n},$$

ou seja,

$$\pi P = \pi.$$

Em outras palavras, π é uma distribuição estacionária do passeio aleatório, e através do Teorema Fundamental das Cadeias de Markov, prova-se que é única [5].

Outra característica importante de passeios aleatórios em grafos expansores é a velocidade com que um passeio converge para distribuição estacionária. Se $\mathbf{q^{(0)}}$ é a distribuição inicial do passeio, $\mathbf{q^{(t)}}$ a distribuição no instante t e π é a distribuição estacionária, então

$$\max_i \frac{|q_i^{(t)} - \pi_i|}{\pi_i} \leq n^{1,5}\lambda^t.$$

Como o segundo autovalor $\lambda < 1$, a distância entre a distribuição no instante t e a distribuição estacionária é decrescente, e para grafos expansores, λ é pequeno, o que garante uma convergência rápida, como mostra o teorema a seguir [5].

Teorema 3 *Seja $\{G_n\}_n$ uma família de expansores. Então, em $t = O(\log n)$ passos, $\mathbf{q^{(t)}}$ difere da distribuição uniforme por um fator constante $\delta < 1$.*

Prova Tome t tal que

$$n^{1,5}\lambda^t < \delta.$$

Aplicando o logarítmo nos dois lados, obtemos

$$\log \delta > \log(n^{1,5}\lambda^t) = \log(n^{1,5}) + \log \lambda^t$$

$$\log \lambda^t < \log \delta - \log n^{1,5}$$

$$t < \frac{\log \frac{\delta}{n^{1,5}}}{\log \lambda}$$

Como $\lambda, \delta < 1$, seus logarítmos são negativos. Portanto

$$t < -\frac{\log \frac{\delta}{n^{1,5}}}{\log \frac{1}{\lambda}} = \frac{\log \frac{n^{1,5}}{\delta}}{\log \frac{1}{\lambda}},$$

ou seja, $t = O(\log n)$.

3 Construção de grafos expansores através do produto zig-zag

Como vimos no capítulo 2, o problema de determinar se um grafo é expansor é *co-*NP-difícil. Por isso, uma parte importante do estudo de grafos expansores envolve a construção de famílias $\{G_n\}_n$ desses grafos com parâmetro de expansão controlado. Em geral, estamos interessados em construções explícitas, em que a vizinhaça de um vértice pode ser determinada eficienteme nte sem conhecer todo o grafo. Por exemplo, num caminho em que os vértices são numerados sequencialmente, sabemos que os vizinho de um vértice i são $i-1$ e $i+1$, se i não é um vértice dos extremos.

Este capítulo apresenta uma família de grafos expansores com construção explícita, construída através do produto zig-zag. Para essa construção usaremos duas operações entre grafos descritas posteriormente: o produto tensor e o quadrado de um grafo.

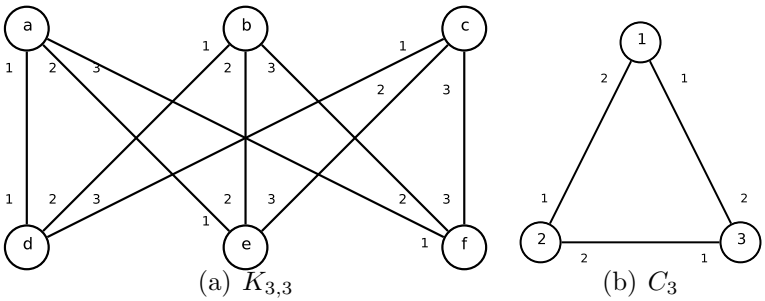
3.1 O produto zig-zag

O produto zig-zag tem como parâmetros um grafo grande e um grafo pequeno. O resultado do produto é um grafo que "herda" o tamanho do grafo grande e o grau do grafo pequeno. Intuitivamente, o produto zig-zag expande cada vértice no grafo grande para uma nuvem de vértices com o tamanho do grafo pequeno.

A definição do produto zig-zag assume que as arestas dos grafos são numeradas em uma ordem fixa arbitrária em relação a cada vértice. Ou seja, podemos nos referir a primeira aresta de cada vértice, a segunda aresta de cada vértice, e assim por diante.

A partir dessa numeração, definimos uma função chamada mapa de rotação, que relaciona a numeração das arestas entre vértices vizinhos.

Definição 7 (Mapa de rotação) *Seja G um multigrafo d-regular com n vértices. O*



(a) $K_{3,3}$ (b) C_3

Figura 1: Grafos com arestas numeradas

mapa de rotação de G é definido como

$$Rot_G : V(G) \times [d] \mapsto V(G) \times [d],$$

tal que $Rot_G(v, i) = (u, j)$ se a i-ésima aresta do vértice v é a j-ésima aresta do vértice u e tal aresta torna v e u adjacentes.

A figura 1 mostra dois grafos com suas arestas numeradas. Note que uma mesma aresta pode ter numerações diferentes em relação a cada vértice, como o caso da aresta ae do $K_{3,3}$, o que evidencia a necessidade do mapa de rotação.

Definição 8 (Produto zig-zag) *Seja G_1 um multigrafo m-regular com n vértices sobre o conjunto $[n]$ e G_2 um multigrafo d-regular com m vértices sobre o conjunto $[m]$. O produto zig-zag de G_1 e G_2, denotado por $G_1 ⓩ G_2$, é um multigrafo d^2-regular com nm vértices sobre o conjunto $[n] \times [m]$, cujo mapa de rotação $Rot_{G_1ⓩG_2}$ é definido a seguir*

$$Rot_{G_1ⓩG_2}((v, k), (i, j)) = ((w, l), (i', j')), \quad \text{onde}$$

1. $(k', i') = Rot_{G_2}(k, i)$,
2. $(w, l') = Rot_{G_1}(v, k')$,
3. $(l, j') = Rot_{G_2}(l', j)$.

A produto zig-zag pode ser interpretado como um passeio nos grafos G_1 e $G2$. Como o grau de G_1 é igual ao número de vértices de G_2, podemos utilizar um vértice de G_1 para determinar uma aresta de G_2. Para cada aresta partindo do vértice (v, k) de $G_1 ⓩ G_2$, escolhemos um vizinho de k em G_2 para determinar uma aresta em G_1. Essa aresta é utilizada para determinar um vizinho w de v em G_1 que determina qual a nuvem vizinha do vértice (v, k) em $G_1 ⓩ G_2$. O rótulo da aresta wv então é utilizado para determinar l',

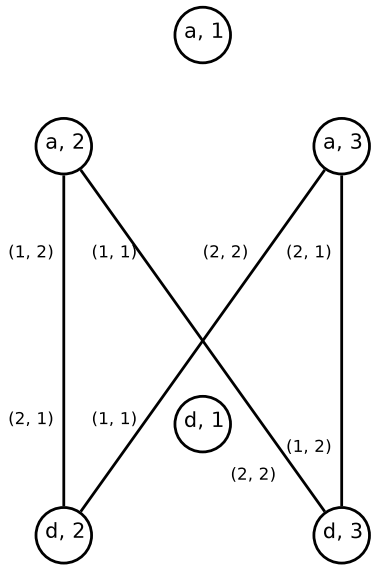


Figura 2: Parte do produto zig-zag entre o $K_{3,3}$ e o C_3

outro vértice em G_2. Um vizinho do vértice l' é escolhido para determinar a qual vértice da nuvem o vértice (v, k) deve estar ligado.

A figura 2 mostra as arestas entre as nuvens geradas pelos vértices a e d do $K_{3,3}$ no produto $K_{3,3} ⓩ C_3$. O mapa de rotação é o mesmo mostrado pela figura 1.

Teorema 4 ([6]) *Seja G_1 um (n, m, λ_1)-expansor e G_2 um (m, d, λ_2)-expansor. Então, o produto zig-zag de G_1 e G_2 é um $(nm, d^2, f(\lambda_1, \lambda_2))$-expansor, onde $f(\lambda_1, \lambda_2) \leq \lambda_1 + \lambda_2 + \lambda_2^2$ e $f(\lambda_1, \lambda_2) < 1$ quando $\lambda_1, \lambda_2 < 1$.*

3.2 Quadrado de um grafo

Definição 9 *Definimos o quadrado de um multigrafo regular G com matriz de adjacências A, denotado como G^2, como o multigrafo cuja matrix de adjacências é igual a A^2.*

Alternativamente, podemos definir o quadrado de um grafo em função de seu mapa de rotação.

Definição 10 (Definição alternativa do quadrado de um grafo) *Seja G um multigrafo d-regular sobre o conjunto $[n]$ com mapa de rotação Rot_G. Então, G^2 é um multigrafo d^2-regular sobre o conjunto $[n]$ com mapa de rotação $Rot_{G^2}(v_0, (k_1, k_2)) = (v_2, (l_2, l_1))$, onde $(v_2, l_2) = Rot_G(v_1, k_2)$ e $(v_1, l_1) = Rot_G(v_0, k_1)$.*

Sabemos que o elemento a_{ij} do quadrado da matriz de adjacências de um grafo representa o número de passeios de tamanho 2 partindo do vértice i que terminam no vértice j [2]. Intuitivamente, podemos analizar o mapa de rotação descrito acima como o passeio iniciado no vértice v_0 e terminado no vértice v_2. O primeiro passo é dado utilizando a aresta k_1 do vértice v_0, que leva ao vértice v_1. Em seguida, um novo passo é dado, levando do vértice v_1 ao vértice v_2 através da aresta k_2. As arestas l_2 e l_1 são as arestas do caminho de volta.

Proposição 5 ([6]) *Se G é um grafo (n, d, λ)-expansor, então G^2 é um grafo (n, d^2, λ^2)-expansor.*

3.3 Produto Tensor

Definição 11 *Seja G_1 um grafo d_1-regular e G_2 um grafo d_2-regular, definimos o produto tensor $G_1 \otimes G_2$ como o grafo d_1d_2-regular sobre o conjunto $V(G_1) \times V(G_2)$ tal que os vértices (v, u) e (v', u') são adjacentes se e somente se $vv' \in E(G_1)$ e $uu' \in E(G_2)$.*

Assim como o quadrado de um grafo, podemos definir o produto tensor em função de seu mapa de rotação.

Definição 12 (Definição alternativa de produto tensor) *Seja G_1 um grafo d_1-regular e G_2 um grafo d_2-regular, definimos o produto tensor $G_1 \otimes G_2$ como o grafo d_1d_2-regular sobre o conjunto $V(G_1) \times V(G_2)$ com mapa de rotação $Rot_{G_1 \otimes G_2}((v, w), (i, j)) = ((v', w'), (i', j'))$, onde $(v', i') = Rot_{G_1}(v, i)$ e $(w', j') = Rot_{G_2}(w, j)$.*

Proposição 6 ([6]) *Se G_1 é um grafo (n_1, d_1, λ_1)-expansor e G_2 é um grafo (n_2, d_2, λ_2)-expansor, então $G_1 \otimes G_2$ é um grafo $(n_1n_2, d_1d_2, \max\{\lambda_1, \lambda_2\})$-expansor.*

3.4 Família de expansores

Nesta seção, aplicaremos as operações descritas anteriormente para a definição de uma família infinita de grafos expansores. A família é definida em função de uma recursão em que a escolha correta do grafo utilizado como base é fundamental para geração de bons expansores.

Definição 13 *A família infinita de expansores $\{G_1, G_2, \dots\}$ é definida recursivamente a partir de um grafo H d-regular com d^8 vértices. O t-ésimo elemento da família G_t é definido como*

$$G_t = \begin{cases} H^2, & \text{se} \quad t = 1, \\ H \otimes H, & \text{se} \quad t = 2, \\ \left(G_{\lceil \frac{t-1}{2} \rceil} \otimes G_{\lfloor \frac{t-1}{2} \rfloor}\right)^2 ⓩ H & \text{se} \quad t > 2. \end{cases}$$

Teorema 7 ([6]) *Para todo $t \geq 1$, G_t é um grafo d^2-regular com d^{8t} vértices e segundo maior autovalor da matriz de adjacências normalizada $\lambda_t \leq \lambda + O(\lambda^2)$, onde λ é o segundo maior autovalor da matriz de adjacências normalizada do grafo H.*

Para que a recursão acima defina uma família de expansores, o segundo maior autovalor de H precisa ser suficientemente pequeno.

3.4.1 Escolha do grafo base

Para uma potência de um primo $q = p^t$, seja $\mathbb{F}_q$ o corpo finito de tamanho q. Defina o grafo AP_q com conjunto de vértices F_q^2 e conjunto de arestas $\big\{\{(a,b),(c,d)\} : ac = b + d\big\}$. Então, AP_q é um grafo $\left(q^2, q, \frac{1}{\sqrt{q}}\right)$-expansor.

Definição 14 *Definimos recursivamente o grafo AP_q^i como*

$$AP_q^1 = AP_q \otimes AP_q$$
$$AP_q^{i+1} = AP_q^i \, ⓩ \, AP_q$$

Proposição 8 ([6]) *AP_q^i é um grafo q^2-regular com $q^{2(i+1)}$ vértices e o segundo maior autovalor de sua matriz de adjacências normalizada é $O\left(\frac{i}{\sqrt{q}}\right)$.*

Tomando $i = 7$ e q suficientemente grande, obtemos um grafo propício para a construção apresentada anteriormente.

4 Gerador de números pseudo-aleatórios

Para testar empiricamente as características do produto zig-zag e a família de expansores apresentadas na seção anterior, implementamos um passeio aleatório em grafos dessa família. O teste consiste na análise da seqüência de números pseudo-aleatórios correspondente aos rótulos dos vértices visitados.

4.1 Implementação

O passeio aleatório foi implementado usando a linguagem de programação Python. Ela foi escolhido por seu suporte a orientação objeto e suporte nativo a tuplas. O código fonte pode ser visto no Apêndice A.

A implementação está organizada em classes, de modo que todas as operações em grafos são subclasses de uma classe genérica Grafo. A classe grafo possui os seguintes métodos:

$\boldsymbol{RotMap(u,e)}$ Retorna uma tupla (v, f), onde v é o e-ésimo vizinho de u e v é o f-ésimo vizinho de u.

Os métodos a seguir utilizam uma ordenação arbitrária dos vértices do grafo.

$\boldsymbol{FirstV()}$ Retorna o primeiro vértice do grafo.

$\boldsymbol{NextV(v)}$ Retorna o vértice posterior a v do grafo. Caso v seja o último vértice do grafo, o valor de retorno da função é o primeiro vértice.

Os métodos seguintes retornam rótulos que uma aresta pode ter e não arestas do grafo. Elas podem ser utilizadas como segundo parâmetro para o método $RotMap$.

***FirstE*()** Retorna o primeiro rótulo possível de uma arestas.

***NextE*(*e*)** Retorna o seguinte rótulo possível de uma aresta a partir do rótulo anterior e.

***splitV*(*v*)** Dada um tupla longa v, extrai os primeiros elementos necessários para formar um rótulo de vértice do grafo. O valor de retorno é uma tupla cujo primeiro elemento é o rótulo extraído e o segundo elemento é outra tupla com os elementos restantes de v.

***splitE*(*e*)** Dada um tupla longa e, extrai os primeiros elementos necessários para formar um rótulo de aresta do grafo. O valor de retorno é uma tupla cujo primeiro elemento é o rótulo extraído e o segundo elemento é outra tupla com os elementos restantes de e.

***randomV*()** Retorna um vértice aleatório do grafo.

***randomE*()** Retorna um rótulo de aresta aleatório do grafo.

A organização citada acima foi escolhida para tornar desnecessário o armazenamento de todas as adjacências do grafo. No caso de operações com grafos, o método $RotMap$ pode ser implementado em função dos métodos $RotMap$ dos grafos subjacentes. A implementação das classes ZigZagProduct, TensorProduct e Square utilizam essa técnica.

A classe AdjMatrixGraph implementa a construção de um grafo a partir de sua matriz de adjacências. A ordem dos vértices é a ordem em que aparecem na matriz de ajacências. A sua implementação das funções $randomV$ e $randomE$ utiliza um objeto da classe RandomOrgBits para obtenção de bits aleatórios. Esses bits são lidos a partir do site random.org, que os gera a partir de ruído atmosférico [1].

A função Constroi_H utiliza o método descrito na seção 3.4.1 para construir o grafo base da recursão a partir do corpo finito com 16 elementos $\mathbb{F}_{16}$.

A função main verifica os parâmetros passados pela linha de comando. O primeiro parâmetro define qual elemento da família de expansores será construído e o segundo indica qual o comprimento do passeio aleatório. Então, um grafo expansor com as características especificadas é gerado através da recursão apresentada na seção 3.4. Finalmente, o passeio aleatório é realizado e o rótulo de cada vértice é convertido para um número binário que é escrito codificado na saída padrão do programa.

O vértice inicial do passeio aleatório é escolhido aleatoriamente de maneira uniforme, utilizando bits aleatórios obtidos através do site random.org, de modo que a distribuição inicial de probabilidade da cadeia de Markov associada é igual a $\left(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right)$, ou seja a distribuição estacionária uniforme. Portanto, todos os vértices do grafo são equiprováveis em qualquer instante do passeio.

A cada passo do passeio, um rótulo de aresta é escolhido aleatoriamente, também usando bits extraídos do site random.org, e o próximo vértice é determinado como o vértice que a aresta escolhida torna adjacente ao vértice atual.

4.2 Análise experimental

Para determinar a qualidade do gerador de números pseudo-aleatórios implementado, várias seqüências foram geradas e sofreram uma série de testes empíricos. Os testes realizados foram aqueles implementados pela ferramenta ent [7]. Segue uma breve descrição de cada um deles.

Alguns testes referenciam a seqüência de números pseudo-aleatórios gerados $\langle U_n \rangle = U_0, U_1, \ldots, U_{n-1}$, onde $0 \leq U_i \leq 255$, para todo $i \leq n$.

Média artimética Esse teste consiste na soma de todos os bytes do arquivo. Como todos os números são equiprováveis, seu valor experado é

$$\sum_{i=0}^{255} i \frac{1}{256} = \frac{1}{256} \frac{256(255+0)}{2} = \frac{255}{2} = 127,5$$

Aproximação de Monte Carlo para π Nesse teste, cada par de números gerados é considerado um ponto dentro de um quadrado no plano cartesiano. A proporção entre os pontos cuja distância da origem é menor que a metade do lado do quadrado, ou seja, os pontos dentro do círculo inscrito no quadrado é computada. Se a seqüência é aleatória, esse valor deve ficar próximo a $\frac{4}{\pi}$, e através dele, é possível estimar o valor de π.

Teste de correlação serial O teste de correlação serial baseia-se no coeficiente de correlação serial, definido como

$$C = \frac{n(U_0U_1 + U_1U_2 + \ldots U_{n-2}U_{n-1} + U_{n-1}U_0) - (U_0 + U_1 + \cdots + U_{n-1})^2}{n(U_0^2 + U_1^2 + \ldots U_{n-1}^2) - (U_0 + U_1 + \cdots + U_{n-1})^2}.$$

Esse coeficiente de correlação tem seu valor no intervalo $[-1, 1]$, sendo que se seu valor é igual ou próximo a zero, então a seqüencia é independente [3].

Entropia A entropia da sequência $\langle U_n \rangle$ é definida como

$$\sum_{i=0}^{255} \frac{f(i)}{n} \log_2 \left(\frac{n}{f(i)} \right),$$

onde $f(i)$ é a freqüência do número i, ou seja, a quantidade de ocorrências de i na seqüencia $\langle U_n \rangle$.

Teste "Chi-quadrado" O teste Chi-quadrado serve para observações em que o resultado pode cair em uma de várias categorias. Suponha que o resultado de uma observação pode cair em k diferentes categorias. Fazemos n observações independentes, ou seja, o resultado de uma observação não pode influenciar nas outras. Seja p_s a probabilidade do resultado cair na categoria s e Y_s o número de observações que realmente caem na categoria s, ou seja, a freqüência de s. Então, definimos

$$V = \sum_{s=1}^{k} \frac{(Y_s - np_s)^2}{np_s}.$$

Para determinar um bom valor para V, observamos valores da distribuição chi-quadrado com $k - 1$ graus de liberdade. Em nossos experimentos, estamos interessados na distribuição chi-quadrado com 255 graus de liberdade. Um bom valor para V deve estar entre 284.3359 e 226.5204. Esse intervalo é determinado de modo que a probabilidade de V ser maior que o resultado obtido está entre 0.1 e 0.9. Se a probabilidade de V ser maior que o valor obtido estiver entre 0.05 e 0.1 ou entre 0.9 e 0.95, dizemos que os números são "quase suspeitos". Se essa probabilidade estiver entre 0.01 e 0.05 ou entre 0.95 e 0.99, dizemos que os números são suspeitos. Se a mesma probabilidade for menor que 0.01 ou maior que 0.99, rejeitamos a amostra como não suficientemente aleatória. Veja [3] para mais detalhes sobre o teste chi-quadrado.

4.2.1 Resultados observados

Foram executados testes para 30 seqüências distintas de 24000 bytes. A tabela 1 mostra os resultados obtidos. Os parâmetros ficaram dentro dos limites esperados, apesar do teste chi-quadrado identificou apenas uma amostra não suficientemente aleatória, duas

amostras suspeitas e uma amostra quase suspeita. Isso significa que o gerador tem um comportamento aleatório como era esperado.

	Média	Aproximação de π	Coeficiente de correlação serial	Entropia	Teste chi-quadrado
1	128.0567	3.136000000	-0.005616	7.991659	276.44 (25.00)
2	127.3606	3.148000000	0.000106	7.990382	319.68 (0.50)
3	127.2329	3.152000000	0.007044	7.992754	239.79 (50.00)
4	126.0127	3.167000000	0.007346	7.991824	272.53 (25.00)
5	127.7978	3.100000000	0.004871	7.992046	266.67 (50.00)
6	126.7201	3.177000000	-0.008487	7.990884	303.89 (2.50)
7	126.4874	3.148000000	0.003408	7.991664	279.23 (25.00)
8	128.1132	3.130000000	-0.001803	7.991404	281.96 (25.00)
9	127.1863	3.144000000	0.000121	7.992395	251.73 (50.00)
10	127.5608	3.091000000	-0.003739	7.991939	265.54 (50.00)
11	128.0020	3.114000000	-0.006605	7.990973	298.18 (5.00)
12	127.7555	3.098000000	0.009508	7.991650	280.15 (25.00)
13	126.6617	3.171000000	0.000681	7.992316	252.93 (50.00)
14	127.5705	3.162000000	-0.002918	7.991768	274.45 (25.00)
15	127.7519	3.167000000	-0.006611	7.992499	249.45 (50.00)
16	127.9099	3.137000000	0.003850	7.991589	278.91 (25.00)
17	126.9145	3.148000000	-0.003517	7.991647	277.72 (25.00)
18	127.3822	3.127000000	-0.002983	7.992853	234.73 (75.00)
19	128.0455	3.145000000	0.002493	7.992736	240.83 (50.00)
20	127.0523	3.154000000	-0.000241	7.992554	249.58 (50.00)
21	127.3127	3.118000000	0.006750	7.991995	264.85 (50.00)
22	127.6305	3.133000000	0.005671	7.992621	244.93 (50.00)
23	127.6302	3.085000000	-0.003619	7.991812	273.86 (25.00)
24	127.3841	3.136000000	-0.010010	7.990896	301.99 (2.50)
25	127.4475	3.162000000	-0.001483	7.992408	252.33 (50.00)
26	126.8902	3.159000000	-0.005251	7.991321	285.87 (10.00)
27	127.2491	3.126000000	-0.001231	7.992919	236.27 (75.00)
28	128.0328	3.131000000	0.001768	7.991547	278.68 (25.00)
29	127.1812	3.198000000	0.006989	7.992555	245.21 (50.00)
30	126.7630	3.166000000	0.006684	7.992118	259.18 (50.00)

Tabela 1: Resultado da execução dos testes estatísticos nas seqüências geradas

5 Conclusão

A implementação e execução de testes do gerador de números pseudo-aleatórios conseguiu mostrar, de maneira empírica, o mesmo resultado demonstrado analiticamente. Com isso, fica clara a qualidade dos grafos expansores gerados através do produto zig-zag.

Apesar da implementação simples do produto zig-zag, a construção de expansores através da recursão definida na seção 3.4 baseia-se em conceitos de teoria dos números, o que dificulta uma implementação mais flexível da construção de grafos expansores. Por outro lado, a implementação de uma versão mais completa do produto fica facilitada, uma vez que essa é a primeira implementação que conhecemos do produto zig-zag e seu código está disponível sob os termos de uma licença de software livre.

Referências

[1] Mads Haarhr. Introduction to randomness and random numbers. http://www.random.org/essay.html, 1999.

[2] Jair Donadelli Júnior. Notas de aulas de tópicos em grafos: Uma indrodução à teoria espectral de grafos, 2005.

[3] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[4] B. Mohar. Isoperimetric numbers of graphs. *J. Comb. Theory Ser. B*, 47(3):274–291, 1989.

[5] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[6] Omer Reingold, Salil Vadhan, , and Avi Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expander. Ann. of Math. (2), Vol. 155 (2002), no. 1, 157–187.

[7] John Walker. Ent: A pseudorandom number sequence test program. http://www.fourmilab.ch/random/, 1998.

APÊNDICE A – Código fonte

```
#!/usr/bin/python

"""
zigzag - Takes a random walk in an expander generated with the zigzag product.
Copyright (C) 2006  Ander Conselvan de Oliveira

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
"""

# Todos os grafos sao regulares

import random_org_http

import sys
import struct

from math import floor, ceil, log
from random import seed, random

# Tabela de soma e multiplicacao para o F_16
F16_soma = [ \
    [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15], \
    [ 1,  0,  5,  9, 15,  2, 11, 14, 10,  3,  8,  6, 13, 12,  7,  4], \
    [ 2,  5,  0,  6, 10,  1,  3, 12, 15, 11,  4,  9,  7, 14, 13,  8], \
    [ 3,  9,  6,  0,  7, 11,  2,  4, 13,  1, 12,  5, 10,  8, 15, 14], \
    [ 4, 15, 10,  7,  0,  8, 12,  3,  5, 14,  2, 13,  6, 11,  9,  1], \
    [ 5,  2,  1, 11,  8,  0,  9, 13,  4,  6, 15,  3, 14,  7, 12, 10], \
    [ 6, 11,  3,  2, 12,  9,  0, 10, 14,  5,  7,  1,  4, 15,  8, 13], \
    [ 7, 14, 12,  4,  3, 13, 10,  0, 11, 15,  6,  8,  2,  5,  1,  9], \
    [ 8, 10, 15, 13,  5,  4, 14, 11,  0, 12,  1,  7,  9,  3,  6,  2], \
    [ 9,  3, 11,  1, 14,  6,  5, 15, 12,  0, 13,  2,  8, 10,  4,  7], \
    [10,  8,  4, 12,  2, 15,  7,  6,  1, 13,  0, 14,  3,  9, 11,  6], \
    [11,  6,  9,  5, 13,  3,  1,  8,  7,  2, 14,  0, 15,  4, 10, 12], \
    [12, 13,  7, 10,  6, 14,  4,  2,  9,  8,  3, 15,  0,  1,  5, 11], \
    [13, 12, 14,  8, 11,  7, 15,  5,  3, 10,  9,  4,  1,  0,  2,  6], \
    [14,  7, 13, 15,  9, 12,  8,  1,  6,  4, 11, 10,  5,  2,  0,  3], \
    [15,  4,  8, 14,  1, 10, 13,  9,  2,  7,  5, 12, 11,  6,  3,  0], \
    ]
F16_mul = [ \
    [0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], \
    [0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15], \
    [0,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,  1], \
    [0,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,  1,  2], \
```

```
    [0,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,  1,  2,  3], \
    [0,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,  1,  2,  3,  4], \
    [0,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,  1,  2,  3,  4,  5], \
    [0,  7,  8,  9, 10, 11, 12, 13, 14, 15,  1,  2,  3,  4,  5,  6], \
    [0,  8,  9, 10, 11, 12, 13, 14, 15,  1,  2,  3,  4,  5,  6,  7], \
    [0,  9, 10, 11, 12, 13, 14, 15,  1,  2,  3,  4,  5,  6,  7,  8], \
    [0, 10, 11, 12, 13, 14, 15,  1,  2,  3,  4,  5,  6,  7,  8,  9], \
    [0, 11, 12, 13, 14, 15,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10], \
    [0, 12, 13, 14, 15,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11], \
    [0, 13, 14, 15,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12], \
    [0, 14, 15,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13], \
    [0, 15,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14], \
    ]

class RandomBits:
    def __init__ (self):
        self.random_bits = []
        self.nbits = 0

        seed ()

    def fill_random_bits (self):
        i = 0
        while i < 128:
            if random () < 0.5:
                self.random_bits.append (0)
            else:
                self.random_bits.append (1)

            i = i + 1

        self.nbits = self.nbits + 128

    def random_number (self, bits):
        while self.nbits < bits:
            self.fill_random_bits ()

        i = 0
        r = 0
        while i < bits:
            r = r * 2 + self.random_bits[0]
            self.random_bits = self.random_bits[1:]
            i = i + 1

        self.nbits = self.nbits - bits

        return r

class RandomOrgBits (RandomBits):
    def __init__ (self):
        self.random_bits = []
        self.nbits = 0

    def fill_random_bits (self):
        a = random_org_http.randbyte (4096, "f")

        while a != "":
            b = a[0]
            a = a[1:]

            (b,) = struct.unpack ("b", b)

            i = 128
            while i > 0:
                self.random_bits.append ((b & i) / i)
                i = i / 2

            self.nbits = self.nbits + 8

class Graph:
    def RotMap (self, u, e):
        return NULL
```

```
    def nVertices (self):
        return NULL

    def Degree (self):
        return NULL

    def FirstV (self):
        return NULL

    def NextV (self, v):
        return NULL

    def FirstE (self):
        return NULL

    def NextE (self, e):
        return NULL

    def splitV (self, v):
        return NULL

    def splitE (self, e):
        return NULL

    def randomV (self):
        return NULL

    def randomE (self):
        return NULL

# TODO: cria uma classe para gerar diretamente o APq
class AdjMatrixGraph (Graph):
    def __init__ (self, n, d, adj):
        self.adj = adj
        self.n = n
        self.d = d

        self.rg = None

    def FirstV (self):
        return (0, 0)

    def NextV (self, v):
        (a, b) = v
        u = (a * self.d + b + 1) % self.n
        return (u / self.d, u % self.d)

    def FirstE (self):
        r = 0
        return r,

    def NextE (self, e):
        (e,) = e
        r = (e + 1) % self.d
        return r,

    def splitV (self, v):
        u = v[0:2]

        return (u, v[2:])

    def splitE (self, e):
        if type(e) == type (()):
            f = e[0]
            e = e[1:]
        else:
            f = e
            e = ()

        f = f,
        return (f, e)
```

```
    def randomV (self):
        if self.rg == None:
            self.rg = RandomOrgBits ()

        v = self.rg.random_number (log (self.n, 2))
        return (v / self.d, v % self.d)

    def randomE (self):
        if self.rg == None:
            self.rg = RandomOrgBits ()

        e = self.rg.random_number (log (self.d, 2))
        return e,

    def RotMap (self, w, e):
        (a, b) = w
        u = a * self.d + b
        soma = 0
        i = 0

        (e,) = e
        e = e + 1
        while soma < e:
            soma = soma + self.adj[u][i]
            i = i + 1

        v = i - 1

        soma = 0
        i = u

        while i >= 0:
            soma = soma + self.adj[i][v]
            i = i - 1

        r = soma - 1,
        return ((v / self.d, v % self.d), r)

    def nVertices (self):
        return self.n

    def Degree (self):
        return self.d

class TensorProduct (Graph):
    def __init__ (self, G1, G2):
        self.G1 = G1
        self.G2 = G2

        self.n = G1.n * G2.n
        self.d = G1.d * G2.d

    def FirstV (self):
        return self.G1.FirstV () + self.G2.FirstV ()

    def NextV (self, w):
        (v, w) = self.G1.splitV (w)
        (u, w) = self.G2.splitV (w)

        if (self.G2.NextV (u) == self.G2.FirstV ()):
            return self.G1.NextV (v) + self.G2.FirstV ()
        else:
            return v + self.G2.NextV (u)

    def FirstE (self):
        return self.G1.FirstE () + self.G2.FirstE ()

    def NextE (self, d):
        (e, d) = self.G1.splitE (d)
        (f, d) = self.G2.splitE (d)
```

```
        if (self.G2.NextE (f) == self.G2.FirstE ()):
            return self.G1.NextE (e) + self.G2.FirstE ()
        else:
            return e + self.G2.NextE (f)

    def splitV (self, v):
        (u, v) = self.G1.splitV (v)
        (w, v) = self.G2.splitV (v)

        return (u + w, v)

    def splitE (self, e):
        (f, e) = self.G1.splitE (e)
        (g, e) = self.G2.splitE (e)

        return (f + g, e)

    def randomV (self):
        return self.G1.randomV () + self.G2.randomV ()

    def randomE (self):
        return self.G1.randomE () + self.G2.randomE ()

    def RotMap (self, u, e):
        (v, u) = self.G1.splitV (u)
        (w, u) = self.G2.splitV (u)

        (i, e) = self.G1.splitE (e)
        (j, e) = self.G2.splitE (e)

        (vl, il) = self.G1.RotMap (v, i)
        (wl, jl) = self.G2.RotMap (w, j)

        return vl + wl, il + jl

    def nVertices(self):
        return self.n

    def Degree(self):
        return self.d

class ZigZagProduct (Graph):
    def __init__ (self, G1, G2):
        self.G1 = G1
        self.G2 = G2

        self.n = self.G1.n * self.G2.n
        self.d = self.G2.d ** 2

        if G1.Degree () != G2.nVertices ():
            print "Produto ZigZag invalido"
            print "d(G1) = " + str (G1.Degree ())
            print "|V(G2)| = " + str (G2.nVertices ())

    def FirstV (self):
        return self.G1.FirstV () + self.G2.FirstV ()

    def NextV (self, w):
        (v, w) = self.G1.splitV (w)
        (u, w) = self.G2.splitV (w)

        if (self.G2.NextV (u) == self.G2.FirstV ()):
            return self.G1.NextV (v) + self.G2.FirstV ()
        else:
            return v + self.G2.NextV (u)

    def FirstE (self):
        return self.G2.FirstE () + self.G2.FirstE ()

    def NextE (self, d):
        (e, d) = self.G2.splitE (d)
        (f, d) = self.G2.splitE (d)
```

```
        if (self.G2.NextE (f) == self.G2.FirstE ()):
            return (self.G2.NextE (e) + self.G2.FirstE ())
        else:
            return (e + self.G2.NextE (f))

    def splitV (self, v):
        (u, v) = self.G1.splitV (v)
        (w, v) = self.G2.splitV (v)

        return (u + w, v)

    def splitE (self, e):
        (f, e) = self.G2.splitE (e)
        (g, e) = self.G2.splitE (e)

        return (f + g, e)

    def randomV (self):
        return self.G1.randomV () + self.G2.randomV ()

    def randomE (self):
        return self.G2.randomE () + self.G2.randomE ()

    def RotMap (self, u, e):
        (v, u) = self.G1.splitV (u)
        (k, u) = self.G2.splitV (u)

        (i, e) = self.G2.splitE (e)
        (j, e) = self.G2.splitE (e)

        (kl, il) = self.G2.RotMap (k, i)
        (w, ll) = self.G1.RotMap (v, kl)
        (l, jl) = self.G2.RotMap (ll, j)

        return (w + l, jl + il)

    def nVertices(self):
        return self.n

    def Degree(self):
        return self.d

class Square (Graph):
    def __init__ (self, G):
        self.G = G

        self.n = self.G.n
        self.d = self.G.d ** 2

    def FirstV (self):
        return self.G.FirstV ()

    def NextV (self, v):
        return self.G.NextV (v)

    def FirstE (self):
        e = self.G.FirstE ()
        f = self.G.FirstE ()

        return e + f

    def NextE (self, d):
        (e, d) = self.G.splitE (d)
        (f, d) = self.G.splitE (d)

        if (self.G.NextE (f) == self.G.FirstE ()):
            return self.G.NextE (e) + self.G.FirstE ()
        else:
            return e + self.G.NextE (f)

    def splitV (self, v):
```

```
        return self.G.splitV (v)

    def splitE (self, e):
        (f, e) = self.G.splitE (e)
        (g, e) = self.G.splitE (e)

        return (f + g, e)

    def randomV (self):
        return self.G.randomV ()

    def randomE (self):
        return self.G.randomE () + self.G.randomE ()

    def RotMap (self, u, e):
        v0 = u

        (k1, e) = self.G.splitE (e)
        (k2, e) = self.G.splitE (e)

        (v1, l1) = self.G.RotMap (v0, k1)
        (v2, l2) = self.G.RotMap (v1, k2)

        return (v2, l2 + l1)

    def nVertices (self):
        return self.n

    def Degree (self):
        return self.d

def imprime_grafo (G):
    v = G.FirstV ()
    v0 = v
    while 1:
        print v

        e = G.FirstE ()
        e0 = e
        while 1:
            print " " + "Rot_t (" + str (v) + ", " + str (e) + ") = " \
                  + str (G.RotMap (v, e))
            e = G.NextE (e)
            if (e == e0): break

        v = G.NextV (v)
        if (v == v0): break


def Constroi_H (q):
    if q == 4:
        # Tabela de soma e multiplicacao do F_4 (corpo finito com 4
elementos)
        Fq_soma = [ [0, 1, 2, 3], [1, 0, 3, 2], [2, 3, 0, 1], [3, 2, 1, 0]]
        Fq_mul = [ [0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 3, 1], [0, 3, 1, 2]]
    elif q == 16:
        Fq_soma = F16_soma
        Fq_mul = F16_mul

    adj = []

    a = 0
    while a < q:
        b = 0
        while b < q:
            linha = []
            c = 0
            while c < q:
                d = 0
                while d < q:
                    if Fq_mul[a][c] == Fq_soma[b][d]:
                        linha.append (1)
```

```
                    else:
                        linha.append (0)


                    d = d + 1
                c = c + 1
            adj.append (linha)
            b = b + 1
        a = a + 1

    APq = AdjMatrixGraph (q ** 2, q, adj)
    APqi = TensorProduct (APq, APq)

    i = 2
    while i <= 7:
        APqi = ZigZagProduct (APqi, APq)
        i = i + 1

    return APqi

if __name__ == "__main__":
    if len (sys.argv) == 3:
        t = int (sys.argv[1])
        passos = int (sys.argv[2])
    else:
        sys.stderr.write ('Modo de uso: zigzag <t> <passos>\n')
        sys.exit (1)

    # H precisa ser um (d^8, d, lambda)-expansor
    q = 16
    H = Constroi_H (q)

    G = ['', '', '']
    G[1] = Square (H)
    G[2] = TensorProduct (H, H)

    # Define que G_t deve ser gerado
    i = 3
    while i <= t:
        teto = int (ceil ((i - 1) / 2.0))
        piso = int (floor ((i - 1) / 2.0))

        tensor = TensorProduct (G[teto], G[piso])
        G.append (ZigZagProduct (Square (tensor), H))

        i = i + 1

    # Inicia o passeio aleatorio. O vertice inicial e' escolhido aleatoriamente

    i = 0;
    v = G[t].randomV ()

    while i < passos:
        u = v
        r = 0

        # Transforma o rotulo do vertice em uma string binaria
        # j conta quantos bits tem r
        j = 0
        while u != ():
            # u[0] varia entre 0 e q - 1  devido a escolha de F_q para
            # construcao do grafo base
            r = r * q + u[0]
            u = u[1:]

            j = j + log (q, 2)

            if j == 32:
                # Escreve o numero em binario
                sys.stdout.write (struct.pack ("L", r))
                r = 0
                j = 0
```

```
e = G[t].randomE ()
(v, e) = G[t].RotMap (v, e)

i = i + 1
```