UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
PROGRAMA DE PÓS-GRADUAÇÃO EM MATEMÁTICA APLICADA E ESTATÍSTICA

ANÁLISE E COMPARAÇÃO ENTRE ALGORITMOS DE PERCOLAÇÃO

ISAAC DAYAN BASTOS DA SILVA

Dissertação apresentada como requisito parcial
para a obtenção do título de
Mestre em Matemática

Orientador: Marcelo Gomes Pereira
Co-orientador: Joaquim Elias de Freitas

NATAL
Julho/2008

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

CENTRO DE CIÊNCIAS EXATAS E DA TERRA

PROGRAMA DE PÓS-GRADUAÇÃO EM MATEMÁTICA APLICADA E ESTATÍSTICA

BANCA EXAMINADORA

Orientador:

Marcelo Gomes Pereira

Examinador externo:

Ary Vasconcelos Medino

Examinador interno:

Joaquim Elias de Freitas

Agradecimentos

A Deus por todas as oportunidades que tem me dado.

Aos meus pais e às minhas irmãs, a quem devo a vida e tudo o que sou.

Ao meu orientador e co-orientador, Prof. Marcelo Gomes Pereira e Prof. Joaquim Elias de Freitas sou muito grato pelas orientações.

A minha esposa Ana Clara, pelo apoio e pela paciência durante esta jornada.

Aos colegas de pós-graduação, pelo companheirismo e ajuda.

Aos amigos do 3º/1º GCC que muito me ajudaram nessa trajetória.

Ao meu amigo Zolin, pela ajuda em um dos momentos mais difíceis da realização deste trabalho.

Aos meus pais

Sumário

Lista de Figuras

Resumo

Nesta dissertação estudamos e comparamos dois algoritmos de percolação, um elaborado por Elias e o outro por Newman e Ziff, utilizando ferramentas teóricas da complexidade de algoritmos e um algoritmo que efetuou uma comparação experimental. Dividimos este trabalho em três capítulos. O primeiro aborda algumas definições e teoremas necessários a um estudo matemático mais formal da percolação. O segundo apresenta técnicas utilizadas para o cálculo estimativo de complexidade de algoritmos, sejam elas: pior caso, melhor caso e caso médio. Utilizamos a técnica do pior caso para estimar a complexidade de ambos algoritmos e assim podermos compará-los.

O último capítulo mostra diversas características de cada um dos algoritmos e através da estimativa teórica da complexidade e da comparação entre os tempos de execução da parte mais importante de cada um, conseguimos comparar esses importantes algoritmos que simulam a percolação.

Abstract

In this work, we study and compare two percolation algorithms, one of then elaborated by Elias, and the other one by Newman and Ziff, using theorical tools of algorithms complexity and another algorithm that makes an experimental comparation. This work is divided in three chapters. The first one approaches some necessary definitions and theorems to a more formal mathematical study of percolation. The second presents technics that were used for the estimative calculation of the algorithms complexity, are they: worse case, better case e average case. We use the technique of the worse case to estimate the complexity of both algorithms and thus we can compare them.

The last chapter shows several characteristics of each one of the algorithms and through the theoretical estimate of the complexity and the comparison between the execution time of the most important part of each one, we can compare these important algorithms that simulate the percolation.

Capítulo 1

INTRODUÇÃO

Na atualidade, existem muitos trabalhos científicos que utilizam a percolação para modelar diversos problemas, muitas vezes, de difícil solução e que são resolvidos apenas de forma aproximada com a ajuda de computadores. Este trabalho tem como principal finalidade a comparação, utilizando métodos computacionais e teóricos, entre dois algoritmos que simulam o processo de percolação. Um deles foi elaborado por Joaquim Elias de Freitas e o outro por Mark E. J. Newman e Robert M. Ziff, um independente do outro, ambos com muitas diferenças porém com o mesmo objetivo: modelagem computacional da percolação. Podemos destacar como objetivo secundário o estudo desses algoritmos para utilização dos conhecimentos adquiridos em futuras aplicações práticas, possivelmente em outras áreas do conhecimento.

Esta dissertação é constituída de cinco capítulos, o segundo e o terceiro são requisitos básicos para o entendimento do quarto que contém o objetivo principal descrito acima.

O segundo capítulo traz vários conceitos, definições e teoremas necessários a uma melhor compreensão da percolação. Começamos abordando a percolação por ligações e depois fazemos algumas definições interligando esta à percolação por sítios. Um conceito importante estudado é o da probabilidade crítica de percolação e o cálculo aproximado do seu valor quando temos o surgimento do aglomerado percolante.

O capítulo 3 trata de conceitos de complexidade de algoritmos que irão contribuir para a análise teórica dos algoritmos estudados nesse trabalho e para efetuarmos a comparação entre eles. Mostraremos as abordagens de complexidades existentes, quais rotinas são usadas para o cálculo estimado da com-

plexidade, como se faz a comparação entre dois algoritmos que resolvem o mesmo problema e as definições utilizadas para se estimar a complexidade.

No quarto capítulo, citamos as principais características tais como: dados de entrada e saída dos algoritmos, como é feita a união dos aglomerados, modo de "armazenamento" dos vizinhos etc. Fazemos a descrição de alguns pontos relevantes do funcionamento dos dois algoritmos e a comparação entre eles utilizando dois métodos: cálculo da complexidade teórica e mesclagem dos algoritmos para medição dos tempos gastos por cada um deles.

Capítulo 2

PERCOLAÇÃO

2.1 O QUE É PERCOLAÇÃO?

Percolação é um modelo matemático criado para estudar problemas baseados em meios desordenados, como por exemplo:

- Propagação de epidemias;
- Modelagem da propagação de fogo em florestas;
- Redes Elétricas desordenadas;
- Distribuição de óleo ou do gás dentro das rochas porosas em reservatórios de óleo, etc.

Para mais detalhes sobre aplicações de percolação, veja [1], [2], [3] e [4].

O termo "percolação" vem do latim *percolatio* que significa filtragem. Existem vários tipos de percolação, dentre eles podemos destacar a percolação por sítio, por ligação e por sítios mais ligações.

Na percolação por sítios, representamos o meio pelo conjunto $\mathbb{Z}^2$ onde cada ponto será chamado de sítio, o qual estará aberto com probabilidade p. Na percolação por ligações, é a ligação entre dois sítios vizinhos que estará aberta dependendo da probabilidade p. Unimos os sítios e as ligações como elementos que estarão abertos ou fechados na percolação por sítios mais ligações.

À medida que sítios vizinhos estiverem abertos, estes formarão aglomerados. Da mesma forma, ao conjunto de ligações abertas que unem sítios vizinhos denominamos aglomerados de ligações.

Sítios e ligações abertos formam os aglomerados na percolação por sítios mais ligações.

Mostramos nas Figuras 2.1 e 2.2 exemplos de processos de percolação por sítio, ligações e por sítios mais ligações.

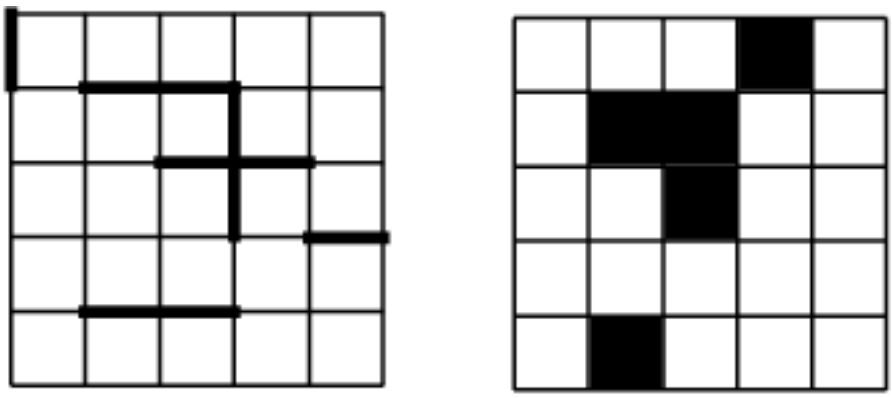

Figura 2.1: Percolação por ligação e por sítios representadas por uma malha 5x5.

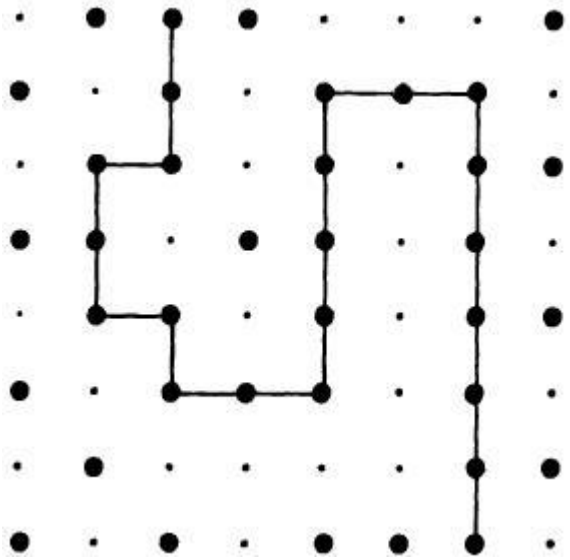

Figura 2.2: Percolação por por sítios mais ligações representada por uma malha 5x5.

As definições desses tipos de percolação podem ser estendidas para um conjunto $\mathbb{Z}^d$, onde cada sítio vai ter $2d$ vizinhos.

No processo de percolação, o tamanho dos aglomerados depende da probabilidade p (probabilidade de cada sítio estar aberto), a medida que aumenta-se o valor de p temos o aumento do número de

sítios abertos o que provoca o aumento dos aglomerados. Para $p = 1$ a malha estará toda preenchida por sítios abertos. Um dos interesses da percolação é justamente o menor valor de p para o qual existe um aglomerado de tamanho infinito. Esse valor é indicado por p_c; onde p_c é a probabilidade crítica ou limiar de percolação.

Nesse mesmo estudo existe o objetivo de calcular ou estimar do valor de p_c. Utilizando a malha quadrada, para a percolação por ligações temos $p_c^{bond} = 1/2$, já na percolação por sítios o valor estimado computacionalmente é $p_c^{site} = 0,5927....$

Listamos valores de p_c para alguns tipos de malha quando utilizamos a percolação por sítios ou por ligações na tabela 2.1 retirada de [1].

MALHA	SÍTIO	LIGAÇÃO
HEXAGONAL	0,6962	0,65271
QUADRADA	0,592746	0,50000
TRIANGULAR	0,50000	0,34729
CÚBICA SIMPLES	0,3116	0,2488
HIPERCÚBICA D=4	0,197	0,1601
HIPERCÚBICA D=5	0,141	0,1182
HIPERCÚBICA D=6	0,107	0,0942
HIPERCÚBICA D=7	0,089	0,0787

Tabela 2.1: Valores de p_c para vários tipos de malhas usando a percolação por sítios e por ligações.

O modo como são dispostos no espaço $\mathbb{Z}^d$ os sítios ou as ligações dá origem aos tipos de malhas. As malhas hipercúbicas de dimensão d são formadas a partir de $\mathbb{Z}^d$ então, por exemplo, uma malha hipercúbica de dimensão 4 é formada a partir de $\mathbb{Z}^4$. O algoritmo de Elias trabalha com as malhas quadrada e triangular. Outro tipo de malha importante para nosso estudo é a hexagonal que também é conhecida com honeycomb. Temos um exemplo de malha honeycomb na Figura 2.3.

O espaço $\mathbb{Z}^2$ pode ser visto como exemplo de uma malha quadrada onde cada sítio (x_1, x_2) possui quatro vizinhos $(x_1, x_2 + 1)$; $(x_1 + 1, x_2)$; $(x_1, x_2 - 1)$ e $(x_1 - 1, x_2)$. Para obtermos uma malha triangular basta que cada sítio (x_1, x_2) tenha como vizinhos adicionais os sítios $(x_1 + 1, x_2 - 1)$ e $(x_1 - 1, x_2 + 1)$. Podemos estender esse raciocínio para $\mathbb{Z}^d$ de forma análoga. Temos exemplos de malhas na Figura 2.4.

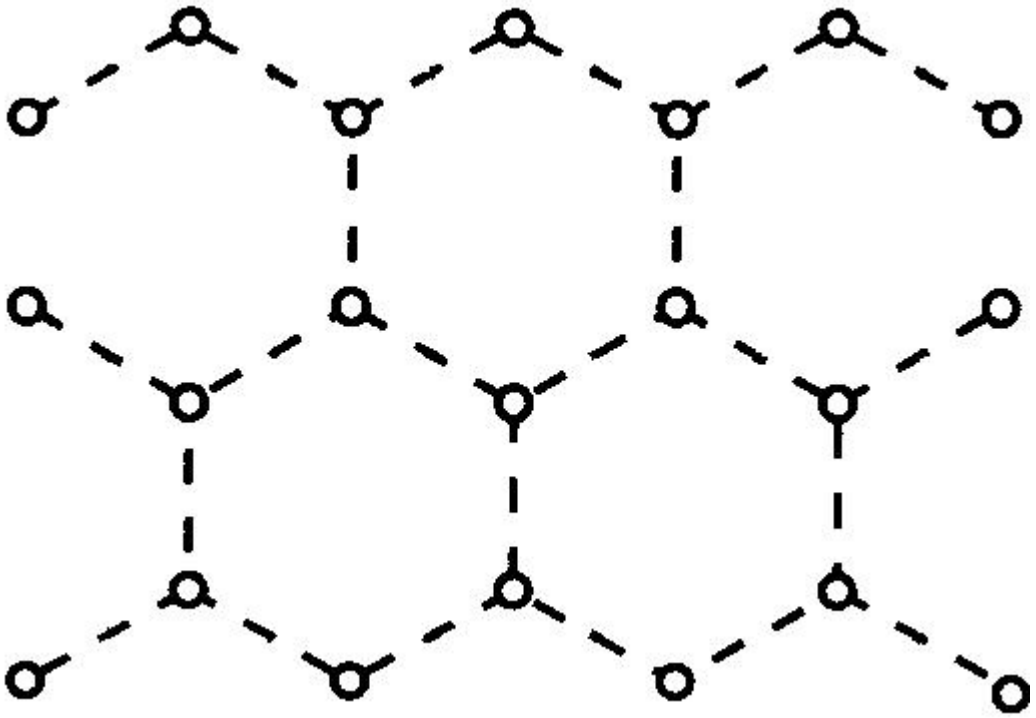

Figura 2.3: Malha Hexagonal ou Honeycomb.

O algoritmo que simula esse processo trabalha com três tipos de percolação e dois tamanhos de malhas sempre finitas e tem como um dos objetivos estimar p_c quando houver o surgimento do aglomerado percolante que é aquele que une lados opostos da malha.

2.2 DEFINIÇÕES

Vamos nos limitar inicialmente a percolação por ligações em $\mathbb{Z}^d$ e começaremos por algumas definições.

Definição 1 *Dados $x, y \in \mathbb{Z}^d$, $d \geq 1$, definimos a distância*

$$\delta(x, y) = \sum_{i=1}^{d} |x_i - y_i| \tag{2.1}$$

no espaço $\mathbb{Z}^d$. Seja também $\mathbb{E}^d$ o conjunto dos segmentos unindo x e y tais que $\delta(x, y) = 1$; chamamos de Malha o conjunto $\mathbb{L}^d$ que é a união entre os conjuntos $\mathbb{Z}^d$, que é o conjunto dos

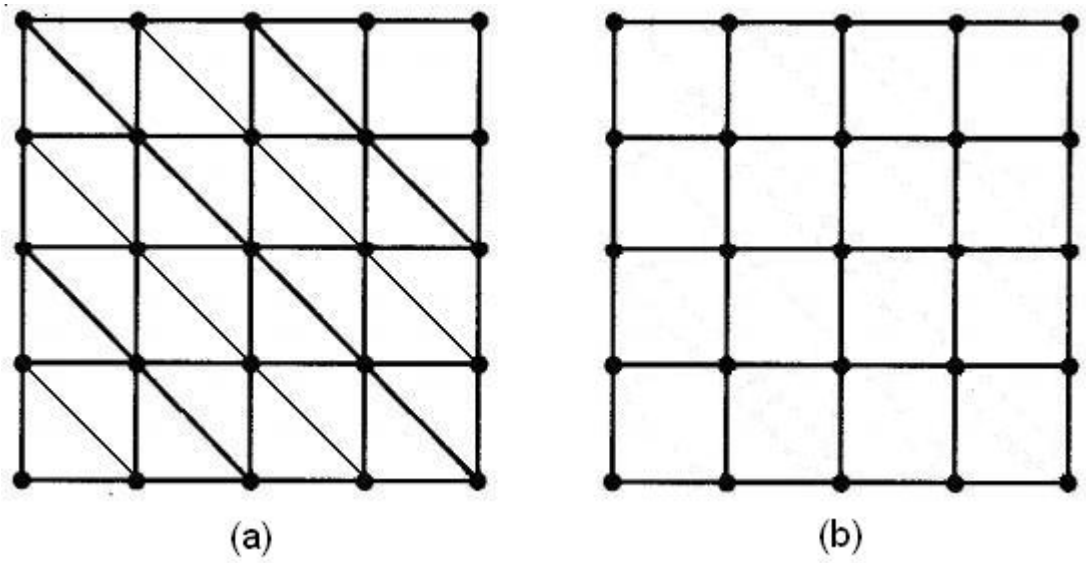

Figura 2.4: Malha triangular na fig. (a) e malha quadrada na fig. (b), ambas com dimensões 5x5. Verifique que cada sítio possui quatro vizinhos na quadrada e seis vizinhos na triangular.

vértices, e $\mathbb{E}^d$ *o conjunto das ligações, ou seja,*

$$\mathbb{L}^d = \mathbb{Z}^d \bigcup \mathbb{E}^d \tag{2.2}$$

Veja na Figura 2.5 exemplo de vértices e ligações em uma malha $\mathbb{L}^2$

Definição 2 *Dizemos que* x *e* y *são vizinhos quando* $\delta(x, y) = 1$ *e denotaremos* $< x, y >$ *o segmento que liga* x *a* y*.*

Veja na Figura 2.6 os quatro vizinhos do ponto $(2, 2)$.

Definição 3 *Uma* σ *- álgebra* F *sobre um conjunto* Ω *é uma coleção de subconjuntos de* Ω *que possui as seguintes propriedades:*

- $\Omega \in F$
- *Se* $A \in F$ *então* $A^c \in F$ *onde* A^c *é o complementar de A.*
- *Se* A_n *é uma seqüência de elementos de F então* $\bigcup_{i=1}^{\infty} A_i \in F$

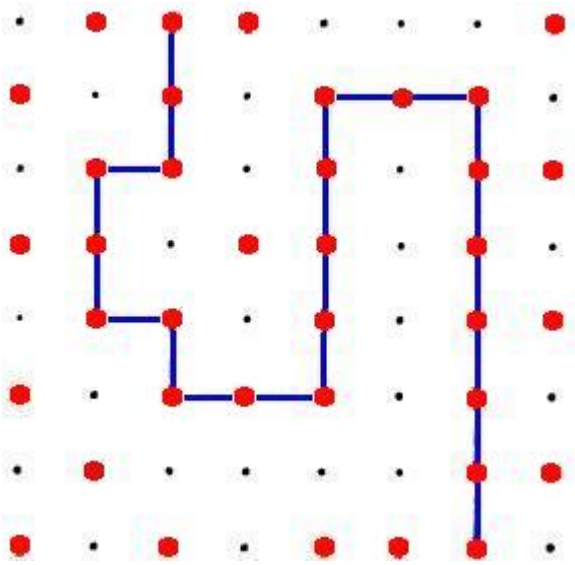

Figura 2.5: Malha $\mathbb{L}^2$ com tamanho 8x8 cujos vétices de $\mathbb{Z}^2$ estão em vermelho e ligações de $\mathbb{E}^2$ estão em azul.

Podemos exemplificar essa definição da seguinte forma: dado um conjunto $A = \{4, 6, 8\}$, o conjunto das partes de A, $\wp(A) = \{\oslash, A, \{4\}, \{6\}, \{8\}, \{4, 6\}, \{4, 8\}, \{6, 8\}\}$, é uma σ-álgebra. Para comprovarmos isso basta ver se o conjunto das partes satisfaz as três condições expostas acima.

Definiremos agora um espaço amostral $\Omega = \prod_{e \in \mathbb{E}^d} \{0, 1\}$ e seus respectivos pontos representados por $\omega = (\omega(e) : e \in E^d)$ onde ω é chamada de configuração. $\omega(e) = 0$ indica que e é fechado, logo quando $\omega(e) = 1$ temos e aberto. Seja F a σ-álgebra dos subconjuntos de Ω gerados por cilindros de dimensão finita. Dessa forma podemos definir uma medida de probabilidade com densidade p em (Ω, F).

Definição 4 *Denotamos P_p a probabilidade de ocorrer determinada configuração e definimos*

$$P_p = \prod_{e \in \mathbb{E}^d} \mu_e \tag{2.3}$$

onde μ_e é a medida de Bernoulli em $\{0, 1\}$, dada por $\mu_e(\omega(e) = 0) = q$, $\mu_e(\omega(e) = 1) = p$ e p, q tais que $0 \leq p \leq 1$ e $p + q = 1$. Segundo [3] temos $\omega_1 \leq \omega_2$ quando $\omega_1(e) \leq \omega_2(e)$ para todo $e \in \mathbb{E}^d$.

Definição 5 *Definimos a variável aleatória η tal que*

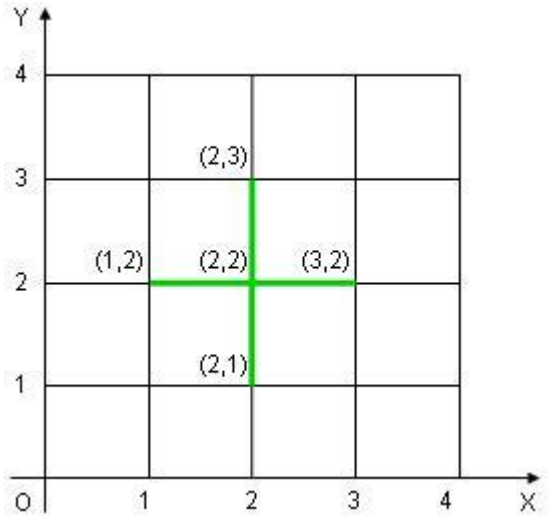

Figura 2.6: Malha $\mathbb{L}^2$ com tamanho 4x4 mostrando todos os vizinhos do ponto $(2, 2)$.

$$\eta_p(e) = \begin{cases} 1 & \textit{se } X(e) < p \\ 0 & \textit{se } X(e) \geq p \end{cases} \tag{2.4}$$

onde X é uma variável aleatória indexada pelos segmentos(ligações) de $\mathbb{E}^d$ e cada $X(e)$ é uniformemente distribuída no intervalo $[0, 1]$. Temos também $P(\eta_p(e) = 0) = 1-p$ e $P(\eta_p(e) = 1) = p$. Segundo [3],

$$\eta_{p_1} \leq \eta_{p_2} \quad \textit{quando} \quad p_1 \leq p_2 \tag{2.5}$$

Nós podemos pensar η_p como sendo o resultado randômico do processo de percolação por ligações em $\mathbb{L}^d$ com probabilidade p.

Considere o subgrafo de $\mathbb{L}^d$ contendo os vértices e as ligações abertas. As componentes que estão conectadas desse grafo são chamadas aglomerados. Escrevemos $C(x)$ para o aglomerado contendo x. Se A e B são conjuntos de vértices em $\mathbb{L}^d$, escrevemos $A \longleftrightarrow B$ quando existe algum caminho aberto ligando algum vértice de A a algum vértice de B. Podemos escrever matematicamente o aglomerado $C(x)$ pela expressão

$$C(x) = \left\{ y \in \mathbb{Z}^d | x \leftrightarrow y \right\} \tag{2.6}$$

Mostramos na Figura 2.7 um aglomerado de sítios.

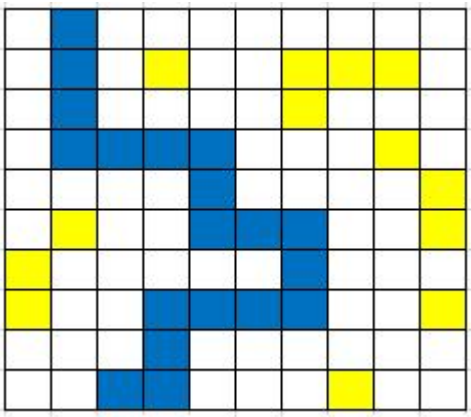

Figura 2.7: Malha $\mathbb{L}^2$ com tamanho $10x10$. Temos exemplo de um aglomerado percolante em azul.

Uma variável importante a ser estudada é a probabilidade de determinado vértice pertencer a um aglomerado infinito $\theta(p)$. Sem perda de generalidade podemos admitir que esse vértice é a origem. Denotaremos o tamanho desse aglomerado que possui a origem por $|C(0)|$ ou apenas $|C|$. Assim definimos

$$\theta(p) = P_p(|C| = \infty) \tag{2.7}$$

esta última expressão é equivalente a

$$\theta(p) = 1 - \sum_{n=1}^{\infty} P_p(|C| = n) \tag{2.8}$$

Sabemos que $\theta(0) = 0$, $\theta(1) = 1$ e que a função θ é monótona não-decrescente (veja [3]), assim é razoável acreditar que existe um valor crítico $p_c = p_c(d)$ de p tal que

$$\theta_p \begin{cases} = 0 & \text{se } p < p_c \\ > 0 & \text{se } p > p_c \end{cases} \tag{2.9}$$

Definição 6 *$p_c(d)$ é chamada de probabilidade crítica ou limiar de percolação e é formalmente definida como*

$$p_c := sup\,\{p|\theta(p) = 0\} \tag{2.10}$$

É importante notarmos que $\theta(p)$ permanece constante e igual a zero enquanto $p < p_c$. Para $p \geq p_c$, o valor dessa função vai crescer até assumir o valor 1. Isso é plausível pois, sendo ela a probabilidade de um determinado sítio pertencer ao aglomerado infinito, à medida que p aumenta, o número de aglomerados aumenta e o tamanho destes também, crescendo assim a possibilidade desse sítio ser anexado ao aglomerado de tamanho infinito.

Podemos ver na Figura 2.8 o possível gráfico de $\theta(p)$.

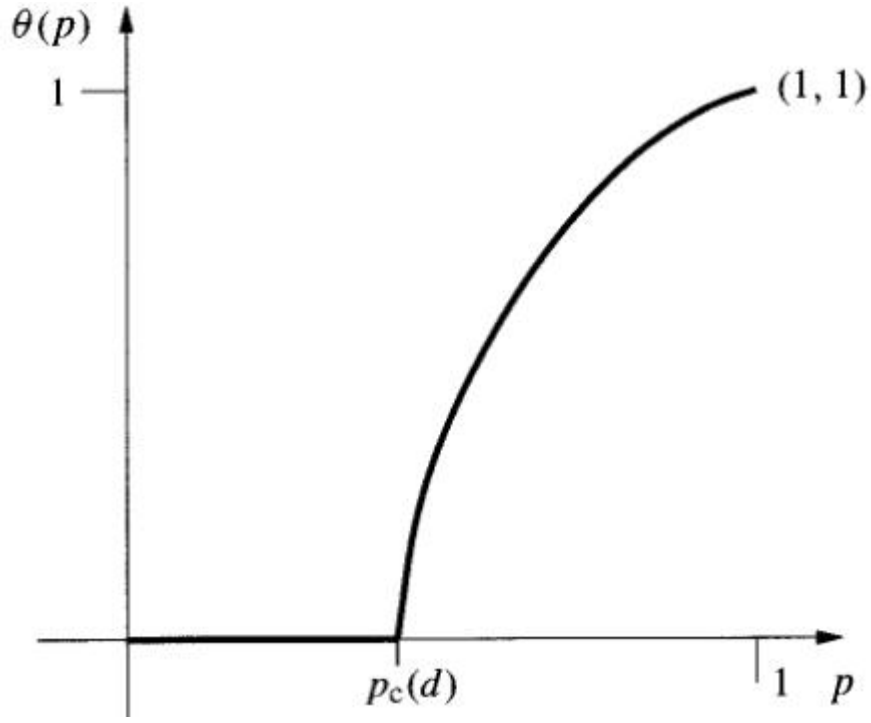

Figura 2.8: Gráfico do possível comportamento de $\theta(p)$.

2.3 TEOREMAS

Nesta seção demonstraremos alguns teoremas importantes para obtermos mais informações sobre a probabilidade crítica de percolação p_c. Para isso, começaremos com uma definição.

Definição 7 *De maneira geral, dado o grafo G, o grafo dual de G, G_d, é obtido colocando-se um vétice em cada face de G e unindo tais vértices sempre que as faces correspondentes de G tiverem uma aresta comum.*

Na Figura 2.9 mostramos exemplos de grafos duais das malhas quadrada e triangular.

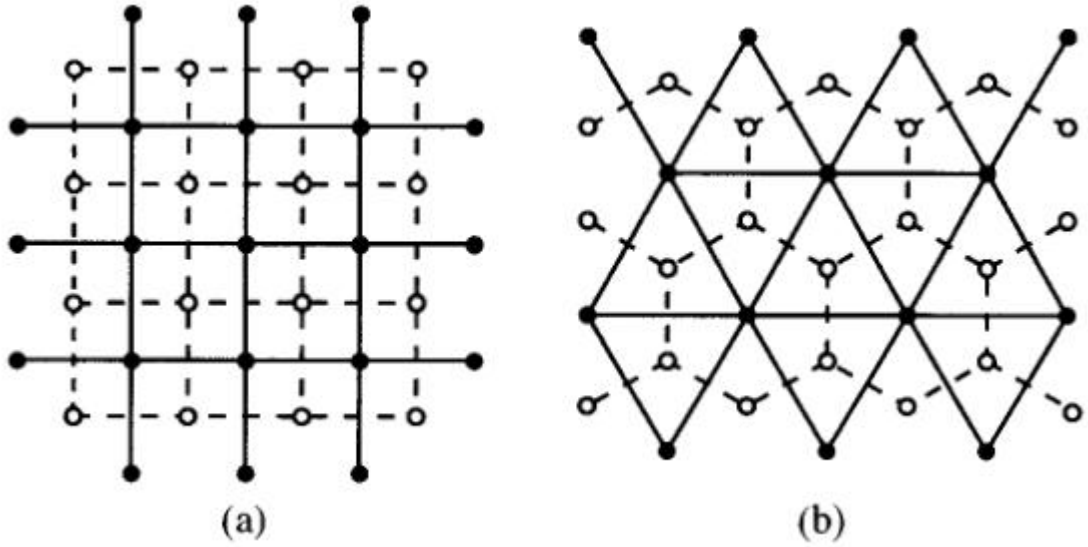

Figura 2.9: Na letra (a) temos a malha quadrada que é dual de si própria e na letra (b) a malha triangular que é dual da malha hexagonal.

Teorema 1 *Se* $d \geq 2$*, então* $0 < p_c(d) < 1$.

Podemos fazer o espaço $\mathbb{L}^d$ ser incluso em $\mathbb{L}^{d+1}$ de forma natural como a projeção de $\mathbb{L}^{d+1}$ no subespaço gerado pelas d primeiras coordenadas. Com essa inclusão, a origem de $\mathbb{L}^{d+1}$ pertence a um aglomerado infinito para um valor particular de p sempre que ela pertencer a um algomerado infinito da submalha $\mathbb{L}^d$.

Se $\mathbb{L}^d \subset \mathbb{L}^{d+1}$ então $\theta_d(p) \leq \theta_{d+1}(p)$, e portanto, $p_c(d) \geq p_c(d+1)$.

Assim, $p_c(d)$ é não decrescente. É necessário mostrar que $p_c(d) > 0$ e $p_c(2) < 1$ pois $p_c(d) \leq p_c(2)$ para $d \geq 2$.

1. Mostremos que $p_c(d) > 0$. Para isso precisamos fazer algumas definições.

   Seja $\sigma(n)$ o número total de caminhos de tamanho n que passam pela origem e não possuem auto-interseção e $N(n)$ o número de tais caminhos que estão abertos. Quaisquer desses caminhos estão abertos com probabilidade p^n então

$$E_p(N(n)) = p^n \sigma(n) \tag{2.11}$$

onde $E_p(N(n))$ é a esperança da variável aleatória $N(n)$.

Se a origem está em um aglomerado de tamanho infinito então temos caminhos de todos os tamanhos passando pela origem, ou seja,

$$\begin{aligned} \theta(p) &\leq P_p(N(n) \geq 1) \\ &\leq E_p(N(n)) = p^n \sigma(n) \end{aligned} \tag{2.12}$$

a segunda desigualdade decorre pela definição de esperança.

Definimos

$$\lambda(d) = \lim_{n \to \infty} (\sigma(n))^{1/n} \tag{2.13}$$

como sendo a constante de conectividade de $\mathbb{L}^d$.

Precisamos limitar $\sigma(n)$. Para isso criaremos um caminho de ligações de tamanho n numa malha de dimensão d colocando ligação por ligação e para a primeira temos $2d$ possibilidades (número de vizinhos de cada sítio). Agora para todas as outras $n-1$ ligações vamos ter $(2d-1)$ possibilidades pois não podemos ligar o sítio em questão ao anterior (já conectado) pois esse caminho não possui auto-interseção. Pelo princípio multiplicativo, obtemos:

$$\sigma(n) \leq 2d(2d-1)^{n-1} \tag{2.14}$$

Logo,

$$(\sigma(n))^{1/n} \leq (2d(2d-1)^{n-1})^{1/n} = (2d/2d-1)^{1/n}.(2d-1) \tag{2.15}$$

Então pela definição da constante de conectividade temos

$$\lambda(d) \leq \lim_{n \to \infty} (2d/2d-1)^{1/n}.(2d-1) = 2d-1 \tag{2.16}$$

Podemos rescrever a Equação 2.13 da seguinte maneira,

$$\sigma(n) = (\lambda(d) + o(1))^n, \quad \text{onde} \quad o(1) \to 0 \tag{2.17}$$

Substituindo a Equação 2.17 na Equação 2.12 obtemos,

$$\theta(p) \leq p^n(\lambda(d) + o(1))^n \tag{2.18}$$

e teremos

$$\theta(p) \leq (p\lambda(d) + o(1))^n \to 0 \quad \text{se} \quad p\lambda(d) < 1 \quad \text{ou} \quad p < 1/\lambda(d) \tag{2.19}$$

Pela definição de p_c e pela Equação 2.16 temos $p_c \geq 1/\lambda(d) \geq 1/2d - 1 > 0$.

2. Mostraremos agora que $p_c(2) < 1$.

   Sabemos que $\mathbb{L}^d = \mathbb{Z}^d \cup \mathbb{E}^d$. Definamos circuito sendo um aglomerado fechado.

   Seja

$$A = \left\{x + (1/2, 1/2) | x \in \mathbb{Z}^2\right\} \cup \{\text{segmentos que unem os pontos de A e que têm distância 1}\} \tag{2.20}$$

   Seja $\rho(n)$ o número de circuitos da malha dual de tamanho n contendo a origem de $\mathbb{L}^2$ em seu interior. Vamos estimar o valor de $\rho(n)$. Cada um desses $\rho(n)$ circuitos passam por algum ponto da forma $(k + 1/2, 1/2)$, onde $0 \leq k < n$ pois como eles circundam a origem eles passam por algum vértice $(k + 1/2, 1/2)$, para algum $k \geq 0$ e eles não podem passar por vétices $(k + 1/2, 1/2)$, para $k \geq n$, pois então teriam pelo menos tamanho $2n$.

   Cada um desses circuitos possui um caminho sem auto-interseção de tamanho $n-1$ começando de algum vétice da forma $(k + 1/2, 1/2)$, onde $0 \leq k < n$. O número desses caminhos sem auto-interseção é no máximo $n\sigma(n-1)$ e assim obtemos

$$\rho(n) \leq n\sigma(n-1) \tag{2.21}$$

Um aglomerado é finito se, e só se, existe um circuito da malha dual que o envolve. Seja γ um circuito fechado malha dual contendo a origem de $\mathbb{L}^2$ em seu interior e seja $M(n)$ o número desses circuitos de tamanho n.

O processo de percolação em $\mathbb{L}^2$ induz um processo igual na malha dual. Basta considerar uma ligação na malha dual aberta sempre que esta for cortada por uma ligação aberta de $\mathbb{L}^2$.

Usando a Equação 2.21 temos,

$$\begin{aligned}\sum_{\lambda} P_p(\gamma \quad fechado) &\leq \sum_{n} q^n n\sigma(n-1) \\ &= \sum_{n} qn\left\{q\lambda(2)+o(1)\right\}^{n-1} < \infty \quad \text{se} \quad q\lambda(2)<1\end{aligned} \tag{2.22}$$

Mais ainda,

$$\sum_{\gamma} P_p(\gamma \quad \text{fechado}) \to 0 \tag{2.23}$$

se $q \to 0$. Então, deve haver $q_0 = 1 - p_0$ tal que

$$\sum_{\gamma} P_{p_0} \leq 1/2 \quad \text{se} \quad p > p_0 \tag{2.24}$$

Agora,

$$\begin{aligned}P_p(|C(0)| = \infty) = P_p(M(n)=0) &= 1 - P_p(M(n) \geq 1) \\ &\geq 1 - \sum_{\gamma} P_p(\gamma \quad \text{fechado}) \geq 1/2\end{aligned} \tag{2.25}$$

sempre que $p > p_0$. Assim $p_c(2) \leq p_0 < 1$ e como $p_c(d) \leq p_c(2)$ para $d \geq 2$ temos $p_c(d) \leq p_0 < 1$, pois a probabilidade de termos um aglomerado de tamanho infinito deixou de ser zero.

Teorema 2 *A probabilidade $\psi(p)$ de existência de um aglomerado aberto infinito satisfaz:*

$$\psi(p) = \begin{cases} 0 & \textit{se } \theta(p) = 0 \\ 1 & \textit{se } \theta(p) > 0 \end{cases}$$

Demonstração.

Se $\theta(p) = 0$ então $\psi(p) \leq \sum_{x \in \mathbb{Z}^d} P_p(|C(x)| = \infty) = 0$, ou seja, $\psi = 0$ pois ψ é uma probabilidade.

Por outro lado, se $\theta(p) > 0$ então $\psi(p) \geq \sum_{x \in \mathbb{Z}^d} P_p(|C(x)| = \infty) = 1$, pois $displaystyle \sum_{x \in \mathbb{Z}^d} P_p(|C(x)| = \infty) = 1$ é a soma das probabilidades de todos os pontos de $\mathbb{Z}^d$.

Assim, $\psi(p) = 1$.

Definição 8 *O evento A é chamado crescente quando*

$$I_A(\omega) \leq I_A(\omega') \quad \textit{sempre que} \quad \omega \leq \omega' \tag{2.26}$$

onde I_A é a função indicadora de A, dada por $I_A(\omega) = 1$, se $\omega \in A$ e $I_A(\omega) = 0$, se $\omega \notin A$.

O evento $\{|C| = \infty\}$ é um evento crescente pois se $\omega \in \{|C| = \infty\}$ então $\omega' \in \{|C| = \infty\}$ se $\omega \leq \omega'$.

Definição 9 *Uma variável aleatória N é chamada crescente quando $\omega \leq \omega' \Rightarrow N(\omega) \leq N(\omega')$. N é dita decrescente quando -N é crescente.*

Seja a variável aleatória $N(x, y)$ o número de diferentes caminhos unindo os vértices x e y. Se temos $\omega \leq \omega'$ é intuitivo que tenhamos $N(\omega) \leq N(\omega')$.

Teorema 3 *Se N é uma variável aleatória crescente, então*

$$E_{p_1}(N) \leq E_{p_2}(N) \quad \textit{quando} \quad p_1 \leq p_2 \tag{2.27}$$

desde que esses valores existam. Se A é um evento crescente, então

$$P_{p_1}(A) \leq P_{p_2}(A) \quad \textit{quando} \quad p_1 \leq p_2 \tag{2.28}$$

Demonstração.

Seja $(X(e) : e \in E^d)$, $X(e) \in [0, 1]$ e

$$\eta_p(e) = \begin{cases} 1 & \text{, se } X(e) < p \\ 0 & \text{, se } X(e) \geq p \end{cases} \tag{2.29}$$

Se $p_1 \leq p_2$, então $\eta_{p_1}(e) \leq \eta_{p_2}(e)$, assim $N(\eta_{p_1}) \leq N(\eta_{p_2})$, e portanto, $E_{p_1}(N) \leq E_{p_2}(N)$

Temos ainda, $I_A(\omega) = 1$, se $\omega \in A$ e $I_A(\omega) = 0$, se $\omega \notin A$.

Se A é crescente então I_A é uma variável aleatória crescente. Logo $E_{p_1}(I_A) \leq E_{p_2}(I_A)$ e podemos afirmar $P_{p_1}(A) \leq P_{p_2}(A)$.

Teorema 4 *(Desilgualdade FKG - Fortuin, Kasteleyn, Ginibre -1971). a) Se X e Y são variáveis aleatórias crescentes tais que $E_p(X^2) < \infty$ e $E_p(Y^2) < \infty$, então*

$$E_p(XY) \geq E_p(X)E_p(Y) \tag{2.30}$$

b) Se A e B são eventos crescentes então,

$$P_p(A \cap B) \geq P_p(A)P_p(B) \tag{2.31}$$

Demonstração.

a) Sejam X e Y variáveis aleatórias crescentes que dependem somente de uma quantidade finita de ligações $\omega(e_1)$, $\omega(e_2)$,..., $\omega(e_n)$.

Usaremos Indução sobre n.

Se $n = 1$ então X e Y são funções do estado $\omega(e_1)$ de e_1 que assume os valores 0 ou 1. Assim temos,

$$0 \leq (X(\omega_1) - X(\omega_2))(Y(\omega_1) - Y(\omega_2)) \tag{2.32}$$

para todos os pares ω_1 e ω_2. A desigualdade é naturalmente válida se $\omega_1 = \omega_2$ e de outra forma pela monotonicidade de X e Y como mostramos abaixo:

Como X é uma variável aleatória crescente então $\omega(e_1) = 1$ e $\omega(e_2) = 0 \Rightarrow X(\omega_1) > X(\omega_2) \Rightarrow X(\omega_1) - X(\omega_2) > 0$ ou $\omega(e_1) = 0$ e $\omega(e_2) = 1 \Rightarrow X(\omega_1) < X(\omega_2) \Rightarrow X(\omega_1) - X(\omega_2) < 0$ podemos mostrar de forma análoga que vale para a variável aleatória Y. Assim nos dois casos vale a desigualdade acima.

Então

$$\begin{aligned} 0 &\leq \sum_{\omega_1,\omega_2} \{X(\omega_1) - X(\omega_2)\} \{Y(\omega_1) - Y(\omega_2)\} P_p(\omega(e_1) = \omega_1) P_p(\omega(e_1) = \omega_2) \\ &= 2(E_p(XY) - E_p(X)E_p(Y)) \end{aligned} \tag{2.33}$$

Chegamos a

$$E_p(XY) \geq E_p(X)E_p(Y) \tag{2.34}$$

Suponha o resultado válido para valores de n satisfazendo $n < k$ e que X e Y dependem dos estados $\omega(e_1)$, $\omega(e_2)$,..., $\omega(e_k)$. Então

$$\begin{aligned} E(XY) &= E_p(E_p(XY|\omega(e_1), .., \omega(e_{k-1}))) \\ &\geq E_p(E_p(X|\omega(e_1), .., \omega(e_{k-1}))E_p(Y|\omega(e_1), .., \omega(e_{k-1}))), \end{aligned} \tag{2.35}$$

fixados os valores de $\omega(e_1), .., \omega(e_{k-1})$, pois neste caso X e Y funcionam como variáveis crescentes somente em função de $\omega(e_k)$.

Agora, $E_p(X|\omega(e_1), .., \omega(e_{k-1})$ é uma função crescente dos estados $\omega(e_1)$,..., $\omega(e_{k-1})$ igualmente à função correspondente a Y. Pela hipótese a esperança do produto de duas variáveis aleatórias é maior ou igual ao produto das esperanças de cada uma das variáveis em questão e assim obtemos:

$$\begin{aligned} E(XY) &\geq E_p(E_p(X|\omega(e_1), .., \omega(e_{k-1})))E_p(E_p(Y|\omega(e_1), .., \omega(e_{k-1}))) \\ &= E_p(X)E_p(Y) \end{aligned} \tag{2.36}$$

Vamos retirar a condição de que X e Y dependem de apenas uma quantidade finita de ligações. Seja X e Y variáveis aleatórias crescentes que possuem o segundo momento finito e $e_1, e_2, ...$ uma ordem fixa de ligações em $\mathbb{L}^d$ e definimos

$$X_n = E_p(X|\omega(e_1), ..., \omega(e_n)) \quad \text{e} \quad Y_n = E_p(Y|\omega(e_1), ..., \omega(e_n)) \tag{2.37}$$

Agora X_n e Y_n são funções crescentes das ligações $e_1, e_2, ..., e_n$ então

$$E_p(X_nY_n) \geq E_p(X_n)E_p(Y_n) \tag{2.38}$$

pelo que expomos acima. Como conseqüência do teorema da convergência de martingale (veja [5]) temos, quando $n \to \infty$,

$$(X_n \to X \quad e \quad Y_n \to Y) \quad q.c. \tag{2.39}$$

Assim,

$$E_p(X_n) \to E_p(X) \quad e \quad E_p(Y_n) \to E_p(Y) \tag{2.40}$$

Pelas inequações do triângulo e de Cauchy-Schwarz temos

$$\begin{aligned} E_p(|X_nY_n - XY|) &= E_p(|X_nY_n - XY_n - XY + XY_n|) \\ &= E_p(|(X_n - X)Y_n + X(Y_n - Y)|) \\ &\leq E_p(|(X_n - X)Y_n| + |X(Y_n - Y)|) \\ &\leq \sqrt{E_p((X_n - X)^2)E_p(Y_n^2)} + \sqrt{E_p(X^2)E_p((Y_n - Y)^2)} \\ &\to 0 \quad quando \quad n \to \infty \end{aligned} \tag{2.41}$$

Dessa forma, $E_p(X_nY_n) \to E_p(XY)$.

Aplicamos o limite com $n \to \infty$ na Equação 2.38, obtemos

$$E_p(XY) \geq E_p(X)E_p(Y) \tag{2.42}$$

b) Para provarmos a segunda parte deste teorema basta aplicarmos a parte (a) nas indicadoras I_A e I_B, onde

$$I_A(\omega) = \begin{cases} 1 & \text{,se } w \in A \\ 0 & \text{,se } w \notin A \end{cases}$$

Assim,

$$E_p(I_AI_B) \geq E_p(I_A)E_p(I_B) \tag{2.43}$$

Segundo [6] se $C = A \cap B$ então $I_C = I_{A \cap B}$. Logo,

$$E_p(I_{A \cap B}) \geq E_p(I_A)E_p(I_B) \tag{2.44}$$

e como temos a igualdade $E_p(I_A) = P_p(A)$ concluimos

$$P_p(A \cap B) \geq P_p(A)P_p(B) \tag{2.45}$$

2.4 PERCOLAÇÃO POR SÍTIOS

Indicaremos a probabilidade crítica da percolação por ligações por p_c^{bond} e a probabilidade crítica da percolação por sítios por p_c^{site}.

Cada ponto em L^d está aberto com uma probabilidade p e fechado com probabilidade $1 - p$. Lembremos que na malha quadrada, que é o tipo de malha a ser explorada em nosso trabalho de comparação dos algoritmos, temos: $p_c^{bond} = 1/2$ e $p_c^{site} \cong 0,592$.

Omitiremos a demonstração do próximo teorema. A prova encontra-se em [3].

Teorema 5 *Seja $G = (V, E)$ um grafo infinito com ligações enumeráveis, origem O e número máximo de ligações por vértices igual a Δ. As probabilidades críticas de G satisfazem:*

$$1/(\Delta - 1) \leq p_c^{bond} \leq p_c^{site} \leq 1 - (1 - p_c^{bond})^{\Delta} \tag{2.46}$$

Teorema 6 *Seja G um grafo infinito com ligações enumeráveis. Os valores da probabilidade crítica de ligações $p_c^{bond}(x)$ e da probabilidade crítica de sítios $p_c^{site}(x)$ independem da escolha de x.*

Demonstração.

Seja $\theta(p, x)$ a probabilidade de x pertencer a um aglomerado aberto infinito.

Observe que $\{x \leftrightarrow y\} \cap \{y \leftrightarrow \infty\} \subset \{x \leftrightarrow \infty\}$ então $\theta(p, x) \geq P_p(\{x \leftrightarrow y\} \cap \{y \leftrightarrow \infty\}) \geq P_p(\{x \leftrightarrow y\})P_p(\{y \leftrightarrow \infty\})$. Obtemos a última desigualdade aplicando FKG.

Assim, temos $\theta(p, x) \geq \theta(p, y)$

De forma análoga, $\theta(p, y) \geq \theta(p, x)$ Portanto, $\theta(p, y) = \theta(p, x)$.

Se tomarmos como sítios os centros dos triângulos da malha triangular, ao unirmos esses sítios, obtemos a malha hexagonal ou "honeycomb". Isso pode ser visto na Figura 2.10.

Seja $\mathbb{L}$ a submalha de $\mathbb{L}^2$ e $\mathbb{L}_d$ o seu dual. Com $p_c(\mathbb{L})$ e $p_c(\mathbb{L}_d)$ as respectivas probabilidades críticas.

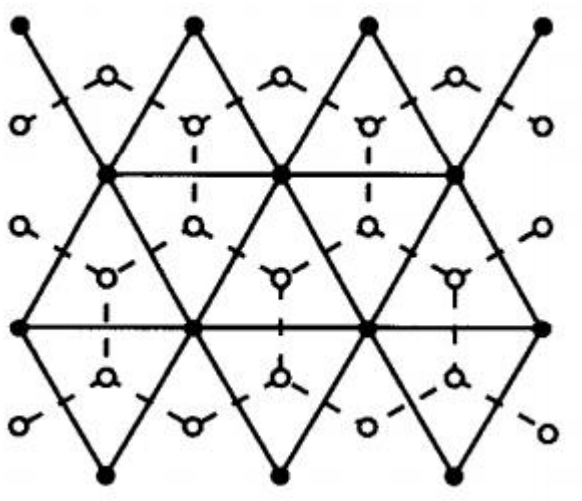

Figura 2.10: As linhas tracejadas mostram a malha hexagonal ou "honeycomb" e as linhas cheias mostram a malha triangular.

Se $p > p_c(\mathbb{L})$ então existe um aglomerado aberto infinito, logo todos os aglomerados fechados em $\mathbb{L}_d$ são finitos, portanto $1 - p < p_c(\mathbb{L}_d)$.

Por outro lado, $p < p_c(\mathbb{L}) \Rightarrow$ todos os aglomerados abertos em $\mathbb{L}$ são finitos $\Rightarrow$ existe um aglomerado fechado infinito em $\mathbb{L}_d \Rightarrow 1 - p > p_c(\mathbb{L}_d)$.

Então, $1 - p_c(\mathbb{L}) = p_c(\mathbb{L}_d) \Rightarrow p_c(\mathbb{L}) + p_c(\mathbb{L}_d) = 1$.

Sendo $\mathbb{L}$ submalha de $\mathbb{L}^2$, temos $p_c(\mathbb{L}) = p_c(\mathbb{L}_d) \Rightarrow p_c(\mathbb{L}^2) = 1/2$.

Citamos dois exemplos de probabilidades críticas abaixo:

$p_c(\Pi) = 2.\sin(\pi/18)$ e p_c(honeycomb) $= 1 - 2.\sin(\pi/18)$, onde Π é a malha triangular.

Podemos perceber empiricamente que $\mathbb{L}^2 \subset \Pi$ pois no conjunto Π acrescentam-se as diagonais. Vejamos a Figura 2.11.

Vamos provar que realmente que $\mathbb{L}^2 \subset \Pi$. Temos $P_p(0 \longleftrightarrow \infty(\mathbb{L}^2)) \leq P_p(0 \longleftrightarrow \infty(\Pi))$ então $\theta(\mathbb{L}^2) \leq \theta(\Pi)$, portanto $p_c(\mathbb{L}^2) \geq p_c(\Pi)$. Agora podemos perguntar se vale $p_c(\Pi) < p_c(\mathbb{L}^2)$. Para a resposta mostraremos um lema e demonstraremos um teorema mas antes vamos a algumas definições.

Seja $p, s \in [0,1]^2$ onde p é a probabilidade de que uma ligação da rede quadrada esteja aberta e s a probabilidade de que uma diagonal esteja aberta em Π.

Escrevemos $P_{p,s}$ e definimos

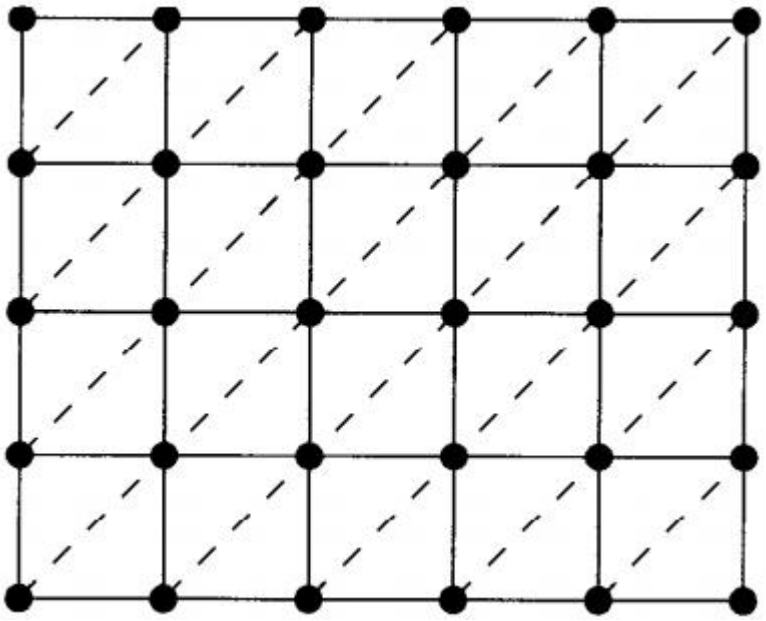

Figura 2.11: A malha triangular pode ser obtida acrescentando algumas diagonais à malha quadrada

$$\theta(p,s) = P_{p,s}(0 \leftrightarrow \infty) \tag{2.47}$$

Nosso objetivo é estabelecer desigualdades entre as derivadas parciais, comparando assim $\frac{\partial\theta}{\partial p}$ com $\frac{\partial\theta}{\partial s}$ em qualquer subconjunto fechado de $[0,1]^2$. Como não temos informações suficientes da diferenciabilidade de θ, vamos aproximá-la pela quantidade finita de volume θ_n e trabalharemos com as derivadas parciais de θ_n.

Seja $B[n] = [-n,n]^2$ sendo caixa fechada. Definimos a seqüência

$$\theta_n(p,s) = P_{p,s}(0 \longleftrightarrow \partial B[n]) \tag{2.48}$$

Notemos que θ_n é um polinômio em p e s e que

$$\theta_n \to \theta \qquad \text{quando} \qquad n \to \infty \tag{2.49}$$

Omitiremos a demonstração do próximo lema por ser longa. Pode-se encontrar essa demonstração em [3].

Lema.

Existem um inteiro positivo L e uma função contínua

$$\alpha : (0,1)^2 \to (0,\infty) \tag{2.50}$$

tais que

$$(\alpha(p,s))^{-1}\frac{\partial}{\partial p}\theta_n(p,s) \geq \frac{\partial}{\partial s}\theta_n(p,s) \geq \alpha(p,s)\frac{\partial}{\partial p}\theta_n(p,s) \tag{2.51}$$

para $n \geq L$ e $0 < p, s < 1$.

Teorema 7 $p_c(\Pi) < p_c(\mathbb{L}^2)$

A curva crítica mostrada na Figura 2.12 pode ser escrita assim $h(p,s) = 0$. Onde h é uma função crescente e continuamente diferenciável, satisfazendo

$$h(p,s) = \theta(p,s) \quad \text{quando} \quad \theta(p,s) > 0 \tag{2.52}$$

É suficiente provar que a curva crítica não possui segmento vertical. Provaremos isso usando o vetor gradiente

$$\nabla h = (\frac{\partial h}{\partial p}, \frac{\partial h}{\partial s}) \tag{2.53}$$

Usando a desigualdade do lema acima e aplicando o limite quando $n \to \infty$ obtemos:

$$\nabla h \bullet (0,1) = \frac{\partial h}{\partial s} \geq \alpha\frac{\partial h}{\partial p} \tag{2.54}$$

onde $\bullet$ é o produto escalar usual.

Assim,

$$\begin{aligned} \frac{1}{|\nabla h|}\frac{\partial h}{\partial s} = ((\frac{\partial h/\partial p}{\partial h/\partial s})^2 + 1)^{-1/2} &= (\frac{h_s^2}{h_p^2 + h_s^2})^{1/2} \\ &= \frac{h_s}{\sqrt{h_p^2 + h_s^2}} \geq \frac{h_s}{\sqrt{(\frac{1}{\alpha}h_s)^2 + h_s^2}} \\ &= \frac{1}{\sqrt{\frac{1}{\alpha^2} + 1}} = \frac{\alpha}{\sqrt{\alpha^2 + 1}} \end{aligned} \tag{2.55}$$

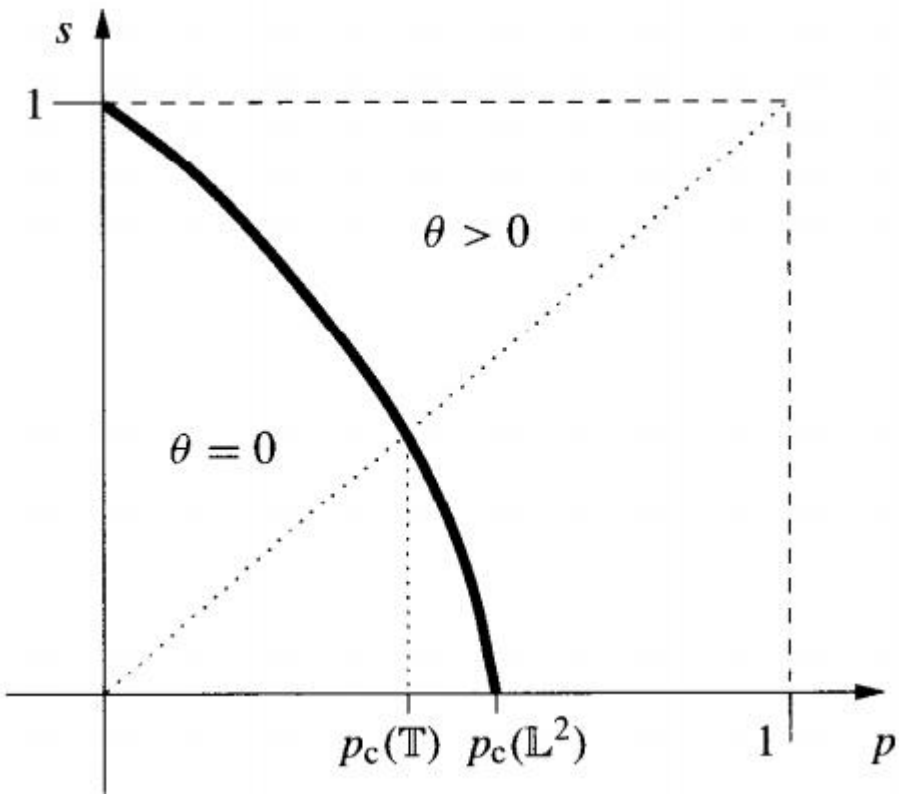

Figura 2.12: Gráfico da curva crítica. A área abaixo da curva é o conjunto de pares (p, s) tais que $\theta(p, s) = 0$.

como esta razão fica afastada do zero em qualquer subconjunto fechado de $[0, 1]^2$, isso mostra que a curva crítica não possui segmento vertical.

Capítulo 3

COMPLEXIDADE DE ALGORITMOS

A Complexidade Computacional é um ramo da Matemática Computacional que estuda a eficiência dos algoritmos. Para medir a eficiência de um algoritmo freqüentemente usamos uma estimativa teórica do tempo que o programa leva para encontrar uma resposta em função dos dados de entrada.

Existem duas abordagens da análise da complexidade de um algoritmo: Experimental e Teórica.

A Abordagem Experimental é a medição do tempo de execução sobre diferentes conjuntos de dados de entrada e necessita de implementação correta e cuidadosa do algoritmo, já na abordagem teórica não há a necessidade de implementação do algoritmo. Deve-se complementar análise teórica com um levantamento experimental.

A complexidade de um algoritmo é um indicativo de como suas implementações irão se comportar quando forem executados num ambiente computacional. Um algoritmo tem duas medidas de complexidade: a temporal que é aproximadamente o número de instruções que ele executa e a espacial que é a quantidade de memória que ele utiliza durante a sua execução.

A avaliação da complexidade teórica de um algoritmo é feita de forma menos precisa por causa do excesso de trabalho para contabilizar cada instrução e também pelo fato do custo de executar cada instrução variar muito.

Agora, estabeleceremos de que forma o tamanho da entrada influirá no comportamento do algoritmo. Se um algoritmo não é recursivo, não contém iterações, não utiliza algoritmos que têm essas características então o número de passos é independente da entrada, assim a complexidade temporal é constante. Dessa forma, o bloco de instruções que são executadas uma única vez possuem custo

unitário. Vamos exemplificar utilizando algumas linhas de comando retiradas do Algoritmo de Elias

cout<<"Media Busca Raiz Ziff= "<<CntRaizZiff/(QntRun*N); → tempo gasto: t_1

cout<<"Media Busca Raiz Elias= "<<CntRaiz/(QntRun*N); → tempo gasto: t_2

cout<<"Tempo Ziff = "<<TimeExec[1]; → tempo gasto: t_3

cout<<"Tempo Elias = "<<TimeExec[2]; → tempo gasto: t_4

Então a complexidade temporal dessas linhas de comando será constante e igual a $C_1 \leq t_1 + t_2 + t_3 + t_4$ pois não depende da quantidade de dados de entrada.

O custo de executar um laço é igual ao tempo gasto pelo bloco de instruções dentro do laço multiplicado pelo número de vezes que o mesmo é executado. Quando temos C iterações aninhadas e cada uma delas dependendo de n temos a complexidade temporal é aproximadamente n^c. Agora se o algoritmo é recursivo, a solução é verificar quantas vezes ele é chamado até chegar-se ao resultado esperado. Analisemos a seguinte função, retirada do algoritmo de Ziff e Newman, que possui laços dependentes de N (número total de sítios).

void permutation( )

{

int i,j;

int temp;

for (i=0; i<N; i++)

order$[i] = i$; → tempo gasto: k_1

for (i=0; i<N; i++){

j = i + (N-i)*RNGLONG; → tempo gasto: k_2

temp = order$[i]$; → tempo gasto: k_3

order$[i] = order[j]$; → tempo gasto: k_4

order$[j] = temp;$}} → tempo gasto: k_5

Assim podemos calcular a função custo C_2 dessa rotina dessa forma: $C_2 \leq [N.k_1 + N.(k_2 + k_3 + k_4 + k_5)] = N.(k_1 + k_2 + k_3 + k_4 + k_5)$ então $C \leq q.N$ onde $q = k_1 + k_2 + k_3 + k_4 + k_5$, ou seja, a complexidade da função permutation é linear.

Agora temos a seguir uma rotina recursiva, também retirada do algoritmo de Ziff e Newman.

int findroot(int i)

{

if ($ptr[i] < 0$) return i;

return ptr[i] = findroot($ptr[i]$);

}

Para calcularmos a função custo desta rotina precisamos saber quantas vezes ela é executada. Precisaremos no próximo capítulo realizar estes cálculos para atingir o objetivo de comparar os algoritmos em estudo.

Se a dependência do tempo com relação aos dados de entrada for polinomial, o programa é considerado rápido, pois dado um polinômio $p(x)$ sabemos que $\|p(x)\|$ cresce para $+\infty$ com $\|x\|$, onde $\|p(x)\|$ é o módulo do polinômio $p(x)$. Entretanto, se a dependência do tempo for exponencial o programa é considerado lento.

A classe de algoritmos P é formada pelos procedimentos para os quais existe um polinômio $p(n)$ que limita o número de passos do processamento se este for iniciado com uma entrada de tamanho n. NP é uma classe de problemas para os quais existe um algoritmo polinomial, embora não determinístico.

Para compararmos quaisquer dois algoritmos devemos estudar as suas complexidades. Já calculadas as funções de custo espaço e tempo dos algoritmos devemos compará-las usando, por exemplo, os seus respectivos gráficos. Existem algumas configurações possíveis. Sejam F e G duas funções custo que queremos comparar.

Se F é sempre inferior a G, ou seja, o gráfico de F fica sempre abaixo do gráfico de G então a escolha pelo algoritmo que corresponde a F é obvia.

Se F às vezes é inferior a G e vice-versa e os gráficos de F e G se intersectam em um número infinito de pontos então há um empate e a função custo não ajuda a escolher o algoritmo.

Se F às vezes é inferior a G e vice-versa e os gráficos de F e G se intersectam em um número finito de pontos então a partir de um valor de n o gráfico de F é superior ou inferior ao de G. Escolheremos o algoritmo cujo gráfico fique abaixo a partir desse valor de n.

Veremos algumas notações que nos auxiliarão a estimar a complexidade teórica dos algoritmos a serem comparados neste trabalho. Para mais detalhes veja [7].

Limite superior ou pior caso: A notação O.

Definição 10 *(Grande O): A função custo $C(n)$ é $O(F(n))$ se existe constantes positivas c e n_0 tais que $C(n) \leq c.F(n)$ quando $n \geq n_0$.*

Então podemos perceber que se a complexidade de um algoritmo é $O(n^2)$ e $C(n)$ seja a respectiva função custo então existe n_0 tal que para $n \geq n_0$ temos $C(n) \leq c.n^2$.

O algoritmo de ordenação Quicksort, que apresenta desempenho no pior caso é $O(n^2)$, mas sabe-se que na prática o algoritmo Quicksort raramente terá este desempenho pessimista.

Mais um exemplo, a função quadrática $g(n) = n^2$ cresce mais rapidamente que uma linear, $f(n) = 7n + 13$. Diz-se que $f(n)$ é $O(g(n))$. A Figura 3.1 ilustra esse exemplo.

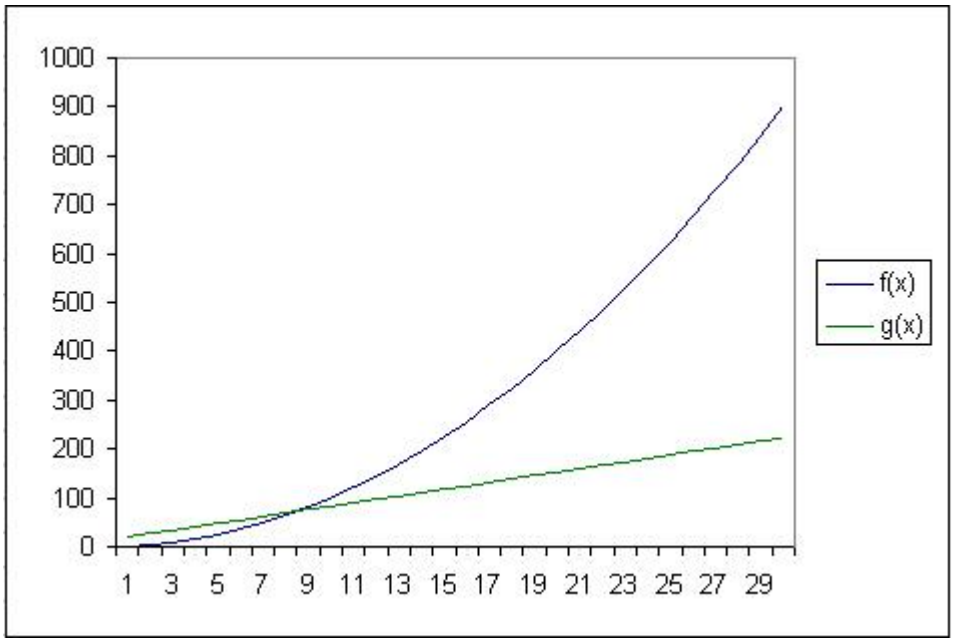

Figura 3.1: Gráfico das funções f e g onde $f(n) = O(g(n))$.

Limite inferior ou melhor caso: A notação Ω.

Definição 11 *(Grande Omega): $C(n) = \Omega(F(n))$ se existem constantes positivas c e n_0 tais que $C(n) \geq c.F(n)$ quando $n \geq n_0$.*

Um algoritmo de ordenação possui função custo C igual a $\Omega(n.log(n))$, ou seja, existem c e n_0 tais que $C(n) \geq c.n.log(n)$ quando $n \geq n_0$.

Por exemplo, uma função cúbica, $g(n) = 7.n^3 + 5$, cresce mais lentamente que uma função exponencial, $f(n) = 2^n$. Diz-se que a função $f(n)$ é $\Omega(g(n))$. Podemos ver isso na Figura 3.2.

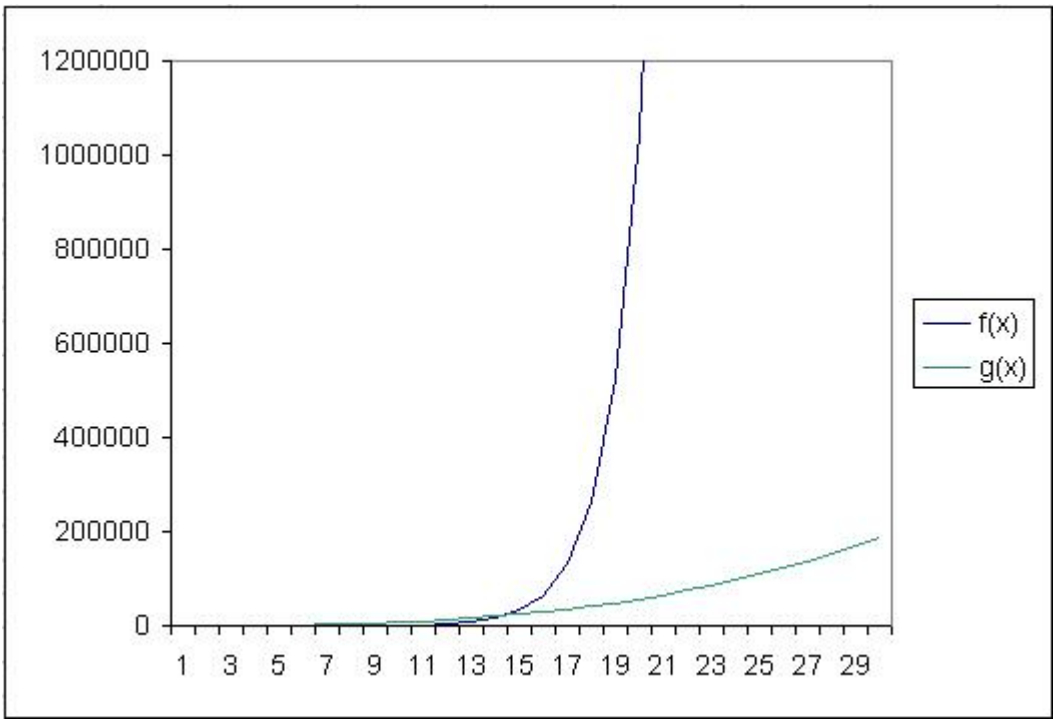

Figura 3.2: Curvas das funções f e g onde $f(n) = \Omega(g(n))$.

Caso médio ou complexidade exata: a notação Θ

Definição 12 *(Grande Theta):* $C(n) = \Theta(F(n))$ *quando* $C(n) = O(F(n))$ *e* $C(n) = \Omega(F(n))$*, ou seja, a complexidade do algoritmo cresce tão rápido quanto a função F.*

Por exemplo, as funções quadráticas $g(n) = n^2 + 3$ e $f(n) = 7.n^2+$ crescem com a mesma rapidez. Diz-se que a função $f(n)$ é $\Theta(g(n))$. Podemos ver isso na Figura 3.3.

Outro exemplo é o algoritmo de ordenação Quicksort que apresenta, segundo [8], desempenho médio $\Theta(n.logn)$.

Neste trabalho, utilizaremos a notação de pior caso para estimarmos a complexidade temporal de forma teórica dos algoritmos de Newman e Ziff e de Elias.

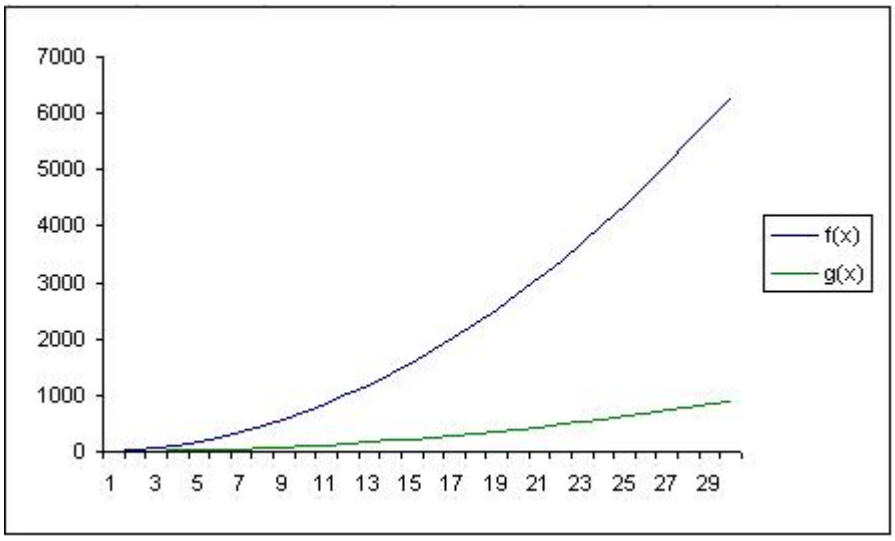

Figura 3.3: Curvas das funções f e g onde $f(n) = \Theta(g(n))$.

Capítulo 4

COMPARAÇÃO ENTRE OS ALGORITMOS

4.1 CONSIDERAÇÕES INICIAIS

O objetivo central deste capítulo é efetuar uma comparação entre os algoritmos de Elias (chamaremos de algoritmo A) e de Ziff e Newman(chamaremos de algoritmo B).

Para isso utilizaremos duas técnicas: Cálculo da complexidade teórica (utilizando a notação de pior caso) da parte mais importante de cada um deles e utilização de um algoritmo mesclando os códigos dos dois para fazer a comparação entre as complexidades temporais desses algoritmos de forma experimental.

4.2 ALGORITMO DE ELIAS

Este algoritmo simula a percolação com as seguintes características:

- Três tipos de percolação: sítios, ligações e sítios mais ligações;
- Dois tipos de malha: triangular e quadrada;
- Há a possibilidade de simular a percolação em N dimensões; e

- Escolha do Tipo de lado (fixo ou variável) da malha.

A dimensão, tipo de malha, tipo de percolação, tamanho dos lados são definidas pelo usuário.

Depois de ser feita a identificação numérica de cada sítio, o algoritmo realiza uma correspondência entre a rede d-dimensional e uma rede linear. Essa correspondência permite que o algoritmo seja mais rápido pois todos as operações posteriores serão feitas com o vetor (rede linear) e não com uma matriz (rede n-dimensional).

É gerado um vetor de incrementos com tamanho igual ao número de vizinhos por sítio que é utilizado para a obtenção dos vizinhos de um sítio dado e isso ocorre da seguinte forma: o valor do vizinho a ser pesquisado é igual a soma do sítio sorteado com o respectivo elemento desse vetor.

É gerada uma estrutura de classe (aglomerado) que vai representar os aglomerados que irão se formando durante o processo. A classe aglomerado possui três atributos, sejam eles:

- Quantidade (Q) de sítios pertencentes a este aglomerado se ele for sítio raiz, zero caso contrário;
- Valor booleano de Fronteira (Fr); e
- Identificação do aglomerado (No).

Após ser feito o sorteio do sítio, é feita a pesquisa dos respectivos vizinhos para saber se algum deles já foi sorteado e em caso positivo ocorre a incorporação do sítio sorteado ao aglomerado já existente.

O processo de formação dos aglomerados é feito através de uma pilha $(p[i])$ que recebe o valor dos sítios (sorteado e vizinhos) e ela é, a todo momento, ordenada para termos o sítio raiz (sítio com maior número de Q) na posição $p[0]$. Esse artifício permite que a "junção" de mais de dois aglomerados seja otimizada, deixando o algoritmo mais eficiente.

A cada rodada é realizada a pesquisa do valor booleano do sítio raiz para saber se houve a percolação.

Para gerar o vetor de incrementos que funcione para todos os sítios, seja ele pertecente ao centro ou a fronteira da malha, este algoritmo gera uma malha estendida que é a malha original acrescida de duas linhas e duas colunas que farão o papel de nova fronteira e assim os sítios da nova fronteira serão os vizinhos dos sítios da fronteira original.

As saídas do programa são os valores das probabilidades críticas de percolação vertical e horizontal e da dimensão fractal do aglomerado percolante.

Segundo [9], essa dimensão fractal é um valor que está associado à forma geométrica de um objeto de maneira única. Em uma rede infinita, no limiar de percolação, o aglomerado que atravessa a rede tem buracos de todos os tamanhos, em todas as escalas, o que permite que sua estrutura seja descrita por conceitos fractais. Podemos, assim, calcular a dimensão fractal deste aglomerado.

Apesar de p_c depender do tipo de rede que for utilizada e se estivermos considerando sítios ocupados ou conectados, a dimensão fractal do aglomerado percolante de uma rede infinita nessa concentração não varia. Em [1] é encontrado o valor $d_f = 91/48$.

As fómulas para seqüenciamento dos sítios e a explicação sobre os valores booleanos utilizados nesse algoritmo estão explicadas em [10].

4.3 ALGORITMO DE NEWMAN E ZIFF

É importante frisar que a versão do algoritmo de Ziff e Newman usada nessa dissertação foi obtida em [11]. Esse algoritmo caracteriza-se por simular a percolação:

- Em redes de sítios bi-dimensionais;
- Utilizando malha quadrada;
- Com condições de contorno circulares;
- Por sítios apenas;

Como no algoritmo A os sítios são representados por números. Para cada sítio são armazenados os vizinhos em uma matriz.

Este algoritmo realiza a formação dos aglomerados da seguinte forma: Todos os sítios recebem inicialmente o valor $-N$ onde N é o número total de sítios da malha. Quando o sítio é sorteado o ponteiro dele recebe o valor -1; Se após isso algum vizinho dele for sorteado, ele passa a receber o valor -2 (tamanho do aglomerado representado por esse sítio raiz) e o sítio sorteado por último passa a ter como valor o identificador do sítio raiz que é positivo. Assim sabemos se um sítio é raiz de um

aglomerado se o ponteiro é negativo e se não o for, ele apontará para o raiz do aglomerado ao qual pertence. Durante sorteio dos outros sítios que se unem ao aglomerado o algoritmo repete o processo.

Quando há uma união entre dois aglomerados, o maior aglomerado “absorve” o menor, ou seja, o sítio raiz do menor apontará para o raiz do maior.

Este algoritmo utiliza condições de contorno circulares que são aquelas em que admite-se que os sítios da fronteira da malha possam ser vizinhos pertencentes às extremidades opostas da própria malha. Exemplificamos esse fato na figura abaixo:

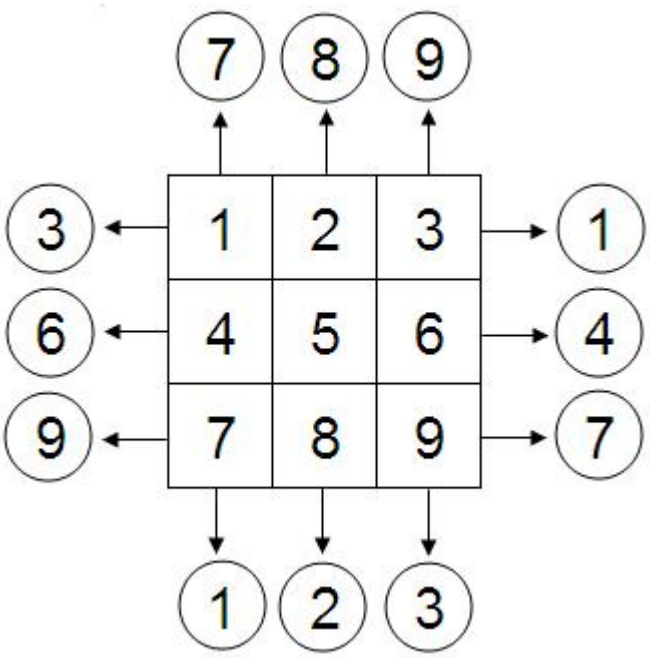

Figura 4.1: Esquema de uma Malha com nove sítios e indicação de vizinhos

Podemos ver no esquema de uma malha com nove sítios acima. Quando adotamos condições de contorno circulares, o sítio 3, além dos vizinhos dois e seis, passa a ser vizinho do sítio nove e do sítio um. Outro exemplo seria o sítio dois que, além dos sítios três, um e cinco, seria também vizinho do sítio oito.

Nenhuma das características relacionadas acima (tipo de percolação, malha, etc.) é escolhida pelo usuário e a saída deste programa é apenas a listagem do número da rodada e o valor do tamanho do maior aglomerado.

Neste algoritmo não é realizado o processo de percolação, existe somente a formação dos aglomerados até o momento em que todos os sítios são sorteados.

4.4 COMPARAÇÃO ENTRE OS DOIS ALGORITMOS

Houve a necessidade de efetuar alterações nos códigos originais dos dois algoritmos para que eles fossem executados de forma equivalente, o algoritmo A teve que ser reduzido e alterado para executar as mesmas funções nas mesmas condições que o algoritmo B, já este último não foi alterado. Para que pudéssemos estimar a complexidade teórica e medir os tempos de execução, fizemos as seguintes modificações:

Usou-se a matriz que armazena os valores dos vizinhos de cada sítio e a função que gera o valor do sítio aleatoriamente do algoritmo B no Algoritmo A. Retirou-se os critérios de percolação usados no algoritmo A. Implementou-se um laço que possibilitaria executar as partes principais dos dois algoritmos tantas vezes fossem necessárias ao estudo.

A contagem do tempo, realizada pela função **clock**, restringiu-se à função **percolate** do algoritmo de Ziff e Newman e de uma parte da algoritmo de Elias responsável pelo processo de escolha dos vizinhos e criação dos aglomerados.

Ao ser retirado o processo de simular a percolação do algoritmo de Elias, houve a possibilidade de estimar a complexidade teórica das partes dos algoritmos com alguns recursos computacionais e matemáticos.

Em relação à medida da complexidade teórica, percebeu-se que, em regra, todas as rotinas presentes nos dois algoritmos resultantes tinham complexidade linear (isto é mostrado no apêndice A), as exceções são a função **findroot** que é recursiva e a parte do algoritmo A que desempenha a mesma tarefa: pesquisa a raiz do aglomerado durante o processo de inclusão de um novo sítio. Isso acontece pelo fato de que todos os sítios pertencentes a um aglomerado tem que "apontar" para a raiz deste último. Então a cada sítio incorporado ao aglomerado também deve "apontar" para a raiz, assim a rotina "visita" sítio a sítio do aglomerado a procura da raiz.

O maior desafio desse cálculo teórico foi saber qual seria o número máximo de execuções dessa rotina (visitas a sítios vizinhos). Para a solução desse problema analisamos o número de vezes que

a rotina é executada para malhas de tamanhos diferentes. Percebeu-se que esse quantidade de visitas era diferente para cada tamanho de malha mas a razão entre número de visitas e o de sítios da malha era quase constante. O comportamento dessa razão versus o tamanho do aglomerado é apresentado no gráfico abaixo.

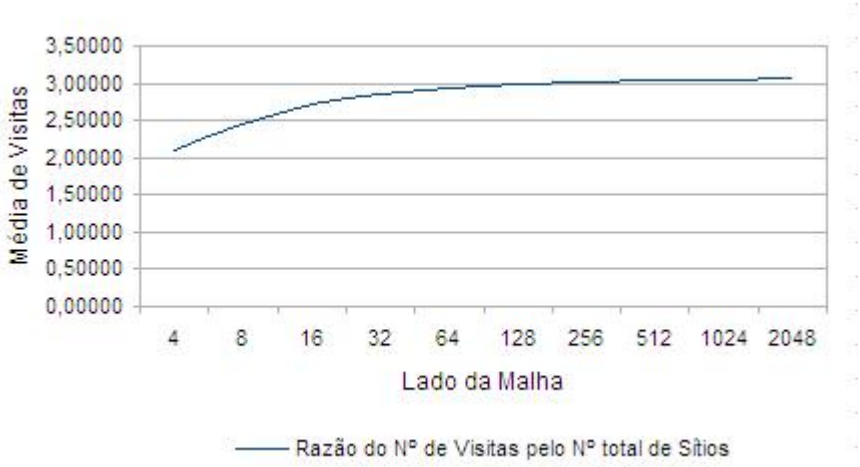

Figura 4.2: Gráfico da média de visitas versus lado da malha do algoritmo A

Percebemos no gráfico da Figura 4.2 que a média de visitas por sítio da malha possui um comportamento assintótico e à medida que aumentamos o tamanho da malha essa média tende a um valor próximo a 3,1.

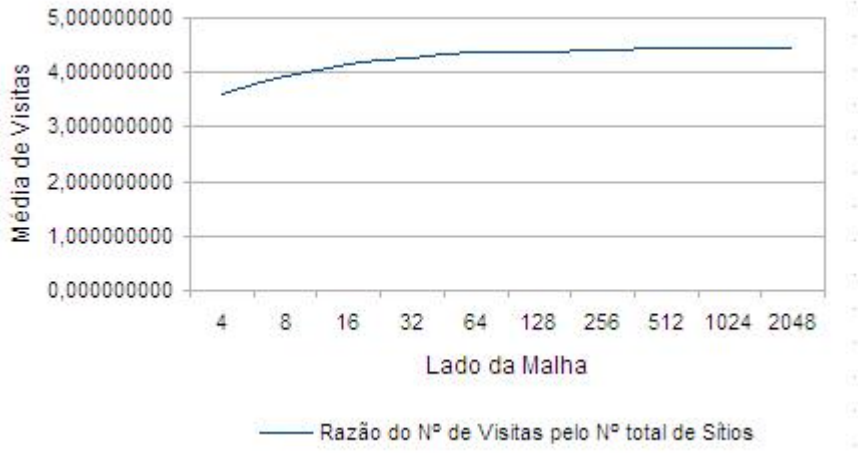

Figura 4.3: Gráfico da média de visitas versus lado da malha do algoritmo B

Da mesma forma, a Figura 4.3 mostra que para o algoritmo B a média de visitas por sítio tende a um valor próximo a 4,4. Podemos ver que apesar dos dois algoritmos possuírem comportamentos lineares o valor de visitas aos sítios da malha é menor no algoritmo de Elias.

Capítulo 5

CONCLUSÃO

Nesta dissertação foram estudados e comparados dois algoritmos que simulam o processo da percolação. Para isso, utilizamos não só conhecimentos de percolação, como também ferramentas da teoria de complexidade de algoritmos que possibilitaram a obtenção do resultado pretendido nesse estudo.

Iniciamos este trabalho abordando conhecimentos básicos sobre percolação por ligações. Necessitou-se fazer várias definições para que chegássemos aos teoremas que envolvem bastante a probabilidade crítica de percolação. Logo após, procuramos interligar aos conhecimentos precedentes a percolação por sítios que é o tipo utilizado pelos algoritmos estudados.

Houve a necessidade de abordar a teoria de complexidade de algoritmos para termos o conhecimento de como deve-se proceder para comparar dois algoritmos e quais ferramentas são utilizadas para estimar teoricamente a função custo e assim a complexidade de cada algoritmo.

Estudamos o código de cada algoritmo para conhecer de forma mais completa as rotinas e a forma como cada um deles simula a percolação. Após isso, modificamos os algoritmos para que eles utilizassem as mesmas ferramentas e fossem diferenciados apenas na parte essencial e crítica para o cálculo da complexidade que é a formação dos aglomerados.

Utilizamos duas técnicas para efetuarmos a comparação: construímos um algoritmo que possibilitasse a comparação do tempo de execução dos algoritmos em estudo que foram executados 100 vezes cada um e a partir do tempo total calculamos o tempo médio; a outra foi a utilização da técnica do pior caso para obtermos uma estimativa teórica do comportamento computacional de cada um deles.

A partir do algoritmo resultante da modificação dos algoritmos A e B calculamos o tempo gasto pelo algoritmo de Ziff, que foi de 201,422s enquanto que o de Elias foi de 287,347s (diferença de 85,925s) para 100 execuções com lado da malha $L = 2048$ (4.194.304 de sítios). Executamos o algoritmo que efetuou essas medições em um computador com a seguinte configuração: processador AMD Athlon(tm) 64 3800+ 2,4GHz e 512MB de memória RAM.

Na estimativa teórica da complexidade os dois algoritmos apresentaram complexidades iguais a $O(n)$. Isso nos diz que a função custo de cada algoritmo possui, no máximo, um comportamento linear e em um ambiente computacional ambos algoritmos se comportariam de maneira equivalente, o que privilegia o algoritmo A pois o mesmo possui uma estrutura que consegue efetuar a percolação em casos mais gerais que o algoritmo B.

Na comparação experimental do tempo, o algoritmo de Ziff é ligeiramente mais rápido como podemos perceber na pequena diferença de tempo (85,925s) ao executarmos os algoritmos como lado $L = 2048$. Temos que levar em conta que o algoritmo de Elias foi preparado para executar o processo de percolação em diversos tipos de malha e em diversas dimensões e por isso possui uma estrutura mais complexa. Além disso, o fato de termos usado a matriz de vizinhos do algoritmo de Ziff e Newman dentro do algoritmo de Elias tornou este último mais lento pois fez com que não acontecesse a otimização do tempo proporcionada pelo processo de linearização da rede (explicado no capítulo anterior). Assim é explicável o fato do algoritmo A possuir um tempo de execução maior.

Referências Bibliográficas

[1] D. STAUFFER and A. AHARONY. *Introduction to Percolation Theory*. Taylor & Francis, 1994.

[2] M. SAHIMI. *Application of Percolation Theory*. Taylor & Francis, 1993.

[3] G. R. GRIMMETT. *Percolation*. SPRINGER, 1999.

[4] A. G. HUNT. *Percolation Theory for flow in Porous Media*. Springer, 2005.

[5] G. R. GRIMMETT and D. STIRZAKER. *Probability and Random Processes*. Oxford University Press, 2001.

[6] Marcos Nascimento MAGALHÃES. *Probabilidade e Variáveis Aleatórias*. Editora da Universidade de São Paulo, 2006.

[7] T. H. CORMEN, C. E. LEISERSON, and R. L. RIVEST. *Introduction to Algorithms*. The MIT Press and McGraw Hill, 1990.

[8] M. A. Castro Barbosa, L. V. Toscani, and L. Ribeiro. Anac uma ferramenta para apoio ao ensino de complexidade de algoritmos. *REVISTA DO CCEI - Centro de Ciências da Economia e Informática da URCAMP- Universidade da Região da Campanha*, pages 89–97, 2001.

[9] Marcelo Gomes Pereira. Aplicações da teoria da percolação à modelagem e simulação de reservatórios de petróleo. Tese de doutorado, Universidade Federal do Rio Grande do Norte, UFRN, Setembro 2006.

[10] Joaquim Elias Freitas. Estudo de alguns sistemas complexos: percolação dependente do tempo, aplicação a problemas de petróleo, percolação de longo alcance e modelo de reação-difusão. Tese de doutorado, Universidade Federal do Rio Grande do Norte, UFRN, Fevereiro 2002.

[11] M. E. J. NEWMAN and R. M. ZIFF. A fast monte carlo algorithm for site or bond percolation. *Physical Review E*, 64(16), april 2001.

Apêndice A

Estimativa da complexidade da parte principal dos Algoritmos A e B

A.1 Algoritmo A

1. for (long iVert=0; iVert<N; iVert++) {
2. Ag[iVert].Q=0; tempo t_1
3. Ag[iVert].No=iVert;} tempo t_2
4. unsigned VertRest;
5. for (long iVert=0; iVert<N; iVert++) {
6. L=order[iVert]; tempo t_3
7. m=0; tempo t_4
8. n=0; tempo t_5
9. for(i = 1; i <= TVz; i++){
10. for(j=Ag[nn[L][i-1]].No;Ag[j].No != j; j=Ag[j].No){
11. n++; tempo t_6
12. NoRotuloAlt[n]=j; } tempo t_7
13. if ((Ag[j]).Q>0){
14. p[m]=j; tempo t_8
15. m++;}} tempo t_9

16. p[m]=L; tempo t_{10}

17. Ag[L].No=Ag[p[0]].No; tempo t_{11}

18. for(i=m-1;i>0;i–) {

19. if((Ag[p[i-1]]).Q<=(Ag[p[i]]).Q)

20. if(((Ag[p[i-1]]).Q<(Ag[p[i]]).Q||Ag[p[i-1]].No<Ag[p[i]].No)){

21. p[20]=p[i]; tempo t_{12}

22. p[i]=p[i-1]; tempo t_{13}

23. p[i-1]=p[20]; tempo t_{14}

24. i+=2;}} tempo t_{15}

25. (Ag[p[0]]).Q++; tempo t_{16}

26. for(i=1;i<m;i++) {

27. k=p[i]; tempo t_{17}

28. if (Ag[k].No!=Ag[p[i-1]].No){

29. Ag[p[0]].Q +=Ag[k].Q; tempo t_{18}

30. Ag[k].No=Ag[p[0]].No; tempo t_{19}

31. }

32. }

33. for (i=1; i < n;i++)

34. Ag[NoRotuloAlt[i]].No=p[0]; tempo t_{20}

35. }

Fazendo a estimativa do tempo t_A temos: $t_A \leq nx(t_1 + t_2) + nx((t_3 + t_4 + t_5) + 4x(t_8 + t_9 + t_{10} + t_{11}) + 3x(t_{12} + t_{13} + t_{14} + t_{15} + t_{16}) + 3x(t_{17} + t_{18} + t_{19}) + (n-1)xt_{20}) = nx((t_1 + t_2 + t_3 + t_4 + t_5) + 4x(t_8 + t_9 + t_{10} + t_{11}) + 3x(t_{12} + t_{13} + t_{14} + t_{15} + t_{16} + t_{17} + t_{18} + t_{19} + t_{20})) - t_{20} = k.n - q = k.n$

Desconsideramos as linhas 10 a 12 porque são justamente elas que devem ter um tratamento para o cálculo da complexidade, o qual foi realizado no capítulo 3 e chegou-se a conclusão de que ela também possui comportamento linear.

A.2 Algoritmo B

1. void percolate()
2. {
3. int i,j;
4. int s1,s2;
5. int r1,r2;
6. int big=0;
7. for (i=0; i<N; i++)
8. ptr[i] = EMPTY; tempo t_1
9. for (i=0; i<N; i++) {
10. r1 = s1 = order[i]; tempo t_2
11. ptr[s1] = -1; tempo t_3
12. for (j=0; j<4; j++) {
13. s2 = nn[s1][j]; tempo t_4
14. if (ptr[s2]!=EMPTY) {
15. r2 = findroot(s2); tempo t_5
16. if (r2!=r1) {
17. if (ptr[r1]>ptr[r2]) {
18. ptr[r2] += ptr[r1]; tempo t_6
19. ptr[r1] = r2; tempo t_7
20. r1 = r2; tempo t_8
21. } else {
22. ptr[r1] += ptr[r2]; tempo t_9
23. ptr[r2] = r1; tempo t_{10}
24. }
25. if (-ptr[r1]>big) big = -ptr[r1]; tempo t_{11}
26. }
27. }
28. }

29. }

Assim, temos que o tempo t_B gasto pelo algoritmo para ser executado é: $t \leq nxt_1 + nx((t_2 + t_3)+4x(t_4+t_6+t_7+t_8+t_9+t_{10}+t_{11})) = nx(t_1+t_2+t_3+4x(t_4+t_5+t_6+t_7+t_8+t_9+t_{10}+t_{11})) = k.n$

A linha 15 do algoritmo não entra nessa conta pois ela necessita do estudo à parte da função findroot, o qual foi realizado no último capítulo e verificou-se que a complexidade dessa função possui comportamento linear.

Apêndice B

Algoritmo de Elias

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
int main (void) {
long iii=time(NULL);
long Semente=iii;
srand(Semente);
long TempoExec[1002];
long B[32];
long d=0;
char Topologia;
long TipoRede;
cout << "Tipo de rede:Sitio(1),Ligacao(2),Sit.+Lig.(3)):";cin >> TipoRede;
cout << "Topologia:Hipercubo(H),hiperTriangular(T)):"; cin >> Topologia;
cout <<"Informe a dimensao do Hipercubo"; cin >> d;
long TipoLado;
cout << "Tipo de lado:todos iguais(0),variam(1)):";cin >> TipoLado;
if (TipoLado==0 ) {
cout << "Informe o lado da Rede"; cin >> B[1];
```

```
for (int i=2; i <= d; i++) B[i] = B[1];
} else {
for (int i=1; i <= d; i++) {
cout « "Informe a aresta Rede n* «< i « ": "; cin » B[i];
}
}
long TsAB[33];
long TsOBe[33];
long TLAB[33];
long TLtAB[33];
TsAB[0]=1;TsOBe[0] = 1;
for (int i=1; i <= d; i++) TsOBe[i] = TsOBe[i-1]*(B[i]+2);
for (int i=1; i <= d; i++) TsAB [i] = TsAB [i-1]* B[i];
TLAB [d] =0;
for (int i=1; i <= d; i++) TLAB [d] +=(TsAB [d]/B[i])*(B[i]-1);
TLtAB [d] =TLAB [d];
for (int j=2; j <= d; j++)
for (int i=1; i < j; i++)
TLtAB [d] +=(TsAB [d]/(B[i]*B[i]))*(B[i]-1)*(B[j]-1);
long dt=d, s[64][2];
if ( Topologia==’T’){
TLAB [d]=TLtAB [d];
dt=(d*d+d)/2;
long k;
for (long j=1; j<=d; j++)
for (long i=1; i<j; i++){
k=d+i+((j-1)*(j-1)-j+1)/2;
s[k][0]=i;
s[k][1]=j;
```

```
}
}
long TamRede, TamRedeEsp; if (TipoRede==1){TamRedeEsp=TsOBe[d]; TamRede=TsAB [d];}
if (TipoRede==2){TamRedeEsp=TsOBe[d]*dt; TamRede=TLAB [d];}
if (TipoRede==3){TamRedeEsp=TsOBe[d]*(dt+1);TamRede=TsAB [d]+TLAB [d];}
long Incr[32][64],TVz;
if (TipoRede==1){
TVz = 2*dt;
long SOBeX;
SOBeX=0;
for ( long t=1; t <= d; t++)
SOBeX += 2*TsOBe[t-1];
for ( long k=1; k <= dt; k++){
long i=0,j=0,eit,ejt,ekt;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeX.Fk=0,SOBeXFk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=0;
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
SOBeX.Fk += (2+eit-ejt-ekt )*TsOBe[t-1];
SOBeXFk += (2-eit+ejt+ekt )*TsOBe[t-1];
}
Incr[0][2*k-1]=SOBeX.Fk - SOBeX;
Incr[0][2*k] =SOBeXFk - SOBeX;
}
}
```

```
if (TipoRede==2 ) {
TVz = 4*dt - 2;
long i,j,m,n,p,eit,ejt,ekt,elt,emt,ent;
long SOBeX;
SOBeX=0;
for ( long t=1; t <= d; t++)
SOBeX += 2*TsOBe[t-1];
long LtOBeXl;
for ( long l=1; l <= dt; l++) {
m=n=0;
if ( l > d )
m=s[l][0]; n=s[l][1];
LtOBeXl= dt*SOBeX + l;
p=0;
for ( long k=1; k <= dt; k++){
long LtOBeXk=dt*SOBeX + k;
if ( k!= l ) {
p++;
Incr[l][p]=LtOBeXk - LtOBeXl;
}
}
for ( long k=1; k <= dt; k++){
i=j=0;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeX.Fk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=0;
if ( i==t ) eit=1;
```

```
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
SOBeX.Fk += (2+eit-ejt-ekt )*TsOBe[t-1];
}
long LtOBeX.Fkk=dt*SOBeX.Fk + k;
p++;
Incr[l][p]=LtOBeX.Fkk - LtOBeXl;
}
long SOBeXFl=0;
for ( long t=1; t <= d; t++) {
ent=emt=elt=0;
if ( n==t ) ent=1;
if ( m==t ) emt=1;
if ( l==t ) elt=1;
SOBeXFl += (2-emt+ent+elt )*TsOBe[t-1];
}
for ( long k=1; k <= dt; k++){
long LtOBeXFlk=dt*SOBeXFl + k;
p++;
Incr[l][p]=LtOBeXFlk - LtOBeXl;
}
for ( long k=1; k <= dt; k++){
if ( k != 1 ) {
i=j=0;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeXFl.Fk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=elt=emt=ent=0;
```

```
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
if ( l==t ) elt=1;
if ( m==t ) emt=1;
if ( n==t ) ent=1;
SOBeXFl.Fk += (2+eit-ejt-emt+ent+elt-ekt )*TsOBe[t-1];
}
long LtOBeXFl.Fkk=dt*SOBeXFl.Fk + k;//LtOBeXFl.Fkk=Lt(O,Be,X+Fl-Fk,k);
p++;
Incr[l][p]=LtOBeXFl.Fkk - LtOBeXl;
}
}
}
}
if (TipoRede==3 ) {
long SOBeX=0;
long i,j,p;
long eit,ejt,ekt;
for ( long t=1; t <= d; t++)
SOBeX += 2*TsOBe[t-1];
long SLtOBeX0= (dt+1)*SOBeX;
p=0;
for ( long k=1; k <= dt; k++){
p++;
Incr[0][p]=k;
}
for ( long k=1; k <= dt; k++){
p++;
```

```
i=j=0;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeX.Fk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=0;
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
SOBeX.Fk += (2+eit-ejt-ekt )*TsOBe[t-1];
}
long SLtOBeXFl.Fkk=(dt+1)*SOBeX.Fk + k;
Incr[0][p]=SLtOBeXFl.Fkk - SLtOBeX0;
}
for ( long k=1; k <= dt; k++){
Incr[k][1]=-k;
i=j=0;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeXFk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=0;
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
SOBeXFk += (2-eit+ejt+ekt )*TsOBe[t-1];
}
long SLtOBeXFk0=(dt+1)*SOBeXFk;
long SLtOBeXk =(dt+1)*SOBeX + k;
```

```
Incr[k][2]=SLtOBeXFk0 - SLtOBeXk;
}
}
struct aglomerado {
public:
long Q;
long No;
long Fr; };
aglomerado *Ag ; Ag = new aglomerado [TamRedeEsp+10];
long *SeqEl; SeqEl = new long [TamRede+1];
aglomerado Ag0; Ag0.Q = 0; Ag0.Fr = -1;
TempoExec[0]=time(NULL);
long X[32], SOBeX[32], FrX[32];
X[d]=-1;FrX[d+1]=0;SOBeX[d+1]=0;
long StRede[32], l=1; StRede[d+1]=1;
for ( long i=d;i <= d;i– ) {
X[i]++;
if ( X[i] == (B[i]+2)) i=i+2;
else {
if ( X[i] == 0 || X[i] == (B[i]+1) )
StRede[i]= 0;
else StRede[i]=StRede[i+1];
FrX[i]=FrX[i+1];
if ( X[i] == 1 )
FrX[i]|=long(powl(2,2*i-2)+.001);
if ( X[i] == B[i] )
FrX[i]|=long(powl(2,2*i-1)+.001);
SOBeX[i]=SOBeX[i+1]+X[i]*TsOBe[i-1];
X[i-1]=-1;
```

```
if ( i == 1 ) {
if ( TipoRede==1 ) {
Ag[SOBeX[1]] = Ag0;
Ag[SOBeX[1]].No = SOBeX[1];
if (StRede[1]==1) {
Ag[SOBeX[1]].Fr=FrX[1];
SeqEl[l]=SOBeX[1];
l++;
}
} else {
for (long k = (3-TipoRede); k <= dt; k++ ) {
long PLe;
PLe=SOBeX[1]*(dt+(TipoRede-2))+k;
Ag[PLe] = Ag0;
Ag[PLe].No = PLe;
long FlRede=StRede[1],i,j,AtFr=FrX[1];
if (k > 0) {
if ( k <= d ) {
if ( X[k] == (B[k]-1))
AtFr |=long(powl(2,2*k-1)+.001);
if (X[k] >= B[k] )
FlRede=0;
} else {
long i,j;
i = s[k][0]; j = s[k][1];
if ( X[j] == (B[j] - 1))
AtFr |=long(powl(2,2*j-1)+.001);
if ( X[i] == 2)
AtFr |=long(powl(2,2*i-2)+.001);
```

```
if (X[i] < 2||X[j] >= B[j])
FlRede=0;
}
}
if ( FlRede==1 ) {
if ( l>TamRede)
l++;
Ag[PLe].Fr = AtFr;
SeqEl[l]=PLe; l++;
if (PLe == 0)
l–;
}
}
}
i++;
}
}
}
long p[21], i, j, k, m, n, Di,L,NoRotuloAlt[1000],St;
p[0]=0;
long FlPerc=0,FlPercFace[32];
for (i=0; i<d; i++) {
FlPercFace[i]=long(powl(2,2*i)+.001)|long(powl(2,2*i+1)+.001);
FlPerc|=FlPercFace[i];
}
double Pc1=0,Pc2=0;
unsigned VertRest;
for (VertRest = TamRede; Ag[p[0]].Fr != FlPerc; VertRest–) {
St = (long(rand())«15)+long(rand());
```

```
St=(St%VertRest)+1;
L=SeqEl[St]; SeqEl[St]=SeqEl[VertRest];SeqEl[VertRest]=L;
m=0; n=0;
if ( TipoRede == 1 ) Di=0;
else if ( TipoRede == 2 ) Di = ((L-1) % dt)+1;
else {
Di = (L % (dt+1));
if (Di>0) TVz=2;
else TVz=2*dt;
}
for(i = 1; i <= TVz; i++){
for(j=Ag[L+Incr[Di][i]].No;Ag[j].No != j; j=Ag[j].No){
n++;
NoRotuloAlt[n]=j;}
if ((Ag[j]).Q>0){
p[m]=j;
m++;}}
p[m]=L;
for(i=m-1;i>0;i–) {
if((Ag[p[i-1]]).Q<=(Ag[p[i]]).Q)
if(((Ag[p[i-1]]).Q<(Ag[p[i]]).Q||Ag[p[i-1]].No<Ag[p[i]].No)){
p[20]=p[i];
p[i]=p[i-1];
p[i-1]=p[20];
i+=2;}}
(Ag[p[0]]).Q++;
for(i=1;i<=m;i++) {
k=p[i];
if (Ag[k].No!=Ag[p[i-1]].No){
```

```
Ag[k].No=Ag[p[0]].No;
Ag[p[0]].Q +=Ag[k].Q;
Ag[p[0]].Fr|=Ag[k].Fr;}
}
for (i=1; i < n;i++)
Ag[NoRotuloAlt[i]].No=p[0];
if ( Pc1==0) {
long QntFaceIso=0,TesteIso;
for (i=0; i < d &&QntFaceIso < 2;i++){
TesteIso=Ag[p[0]].Fr & FlPercFace[i];
if ( TesteIso != FlPercFace[i])
QntFaceIso++;
}
if ( QntFaceIso < 2 )
Pc1= double(TamRede-VertRest+1) / double(TamRede);
}
}
Pc2= double(TamRede-VertRest) / double(TamRede);
double DimFract = d*log10(double((Ag[p[0]]).Q)) / log10(double(TamRede));
TempoExec[1]=time(NULL)-TempoExec[0];
cout « "Pc1 = ";
cout « Pc1;
cout « "Pc2 = ";
cout « Pc2;
cout « ", DimFract = «< DimFract « "e Tempo = «< TempoExec[1];
cin>>Pc1;
}
```

Apêndice C

Algoritmo de Newman e Ziff

```
#include <iostream.h> #include <stdlib.h> #include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define RNGCONV 2.3283064365387e-10
#define RNGLONG (RNGCONV*(rngia[rngp=rngmod[rngp]] += rngia[rngpp=rngmod[rngpp]]))
#define RNGLONGINT (rngia[rngp=rngmod[rngp]] += rngia[rngpp=rngmod[rngpp]])
extern unsigned long int rngia[1279];
extern int rngp,rngpp;
extern int rngmod[1279];
void rngseed(unsigned long int s);
#include <time.h>
#define A 2416
#define C 374441
#define M 1771875
#define CONV 2423.96743336861
unsigned long int i;
unsigned long int rngia[1279];
int rngp,rngpp;
int rngmod[1279];
```

```
void rngseed(unsigned long int s)
{
int n;
if (s==0) s = time(NULL);
i = s;
for (n=0; n<1279; n++) {
rngia[n] = CONV*(i=(A*i+C)%M);
rngmod[n] = n-1;
}
rngmod[0] = 54;
rngp = 0;
rngpp = 418;
for (n=0; n<1000000; n++)
rngia[rngp=rngmod[rngp]] += rngia[rngpp=rngmod[rngpp]];
}
#include "rnglong.h"
#define L 128
#define N (L*L)
#define EMPTY (-N-1)
int ptr[N];
int nn[N][4];
int order[N];
void boundaries()
{
int i;
for (i=0; i<N; i++) {
nn[i][0] = (i+1)%N;
nn[i][1] = (i+N-1)%N;
nn[i][2] = (i+L)%N;
```

```
nn[i][3] = (i+N-L)%N;
if (i%L==0) nn[i][1] = i+L-1;
if ((i+1)%L==0) nn[i][0] = i-L+1;
}
}
void permutation()
{
int i,j;
int temp;
for (i=0; i<N; i++) order[i] = i;
for (i=0; i<N; i++) {
j = i + (N-i)*RNGLONG;
temp = order[i];
order[i] = order[j];
order[j] = temp;
}
}
int findroot(int i)
{
if (ptr[i]<0) return i;
return ptr[i] = findroot(ptr[i]);
}
void percolate()
{
int i,j;
int s1,s2;
int r1,r2;
int big=0;
for (i=0; i<N; i++) ptr[i] = EMPTY;
```

```
for (i=0; i<N; i++) {
r1 = s1 = order[i];
ptr[s1] = -1;
for (j=0; j<4; j++) {
s2 = nn[s1][j];
if (ptr[s2]!=EMPTY) {
r2 = findroot(s2);
if (r2!=r1) {
if (ptr[r1]>ptr[r2]) {
ptr[r2] += ptr[r1];
ptr[r1] = r2;
r1 = r2;
} else {
ptr[r1] += ptr[r2];
ptr[r2] = r1;
}
if (-ptr[r1]>big) big = -ptr[r1];
}
}
}
printf("%i %i",i+1,big);
}
}
main()
{
rngseed(0);
boundaries();
permutation();
percolate();
```

```
}
```

Apêndice D

Novo algoritmo: Mesclagem dos algoritmos anteriores

```
#include <iostream.h> #include <stdlib.h> #include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define RNGCONV 2.3283064365387e-10
#define RNGLONG (RNGCONV*(rngia[rngp=rngmod[rngp]] += rngia[rngpp=rngmod[rngpp]]))
#define RNGLONGINT (rngia[rngp=rngmod[rngp]] += rngia[rngpp=rngmod[rngpp]])
extern unsigned long int rngia[1279];
extern int rngp,rngpp;
extern int rngmod[1279];
void rngseed(unsigned long int s);
#include <time.h>
#define A 2416
#define C 374441
#define M 1771875
#define CONV 2423.96743336861
unsigned long int i;
unsigned long int rngia[1279];
```

```
int rngp,rngpp;
int rngmod[1279];
void rngseed(unsigned long int s)
{
int n;
if (s==0) s = time(NULL);
i = s;
for (n=0; n<1279; n++) {
rngia[n] = CONV*(i=(A*i+C)%M);
rngmod[n] = n-1;
}
rngmod[0] = 54;
rngp = 0;
rngpp = 418;
for (n=0; n<1000000; n++)
rngia[rngp=rngmod[rngp]] += rngia[rngpp=rngmod[rngpp]];
}
#define A 2200
#define N (A*A)
#define EMPTY (-N-1)
int ptr[N];
int nn[N][4];
int order[N];
double CntRaizZiff;
void boundaries()
{
int i;
for (i=0; i<N; i++) {
nn[i][0] = (i+1)%N;
```

```
nn[i][1] = (i+N-1)%N;
nn[i][2] = (i+A)%N;
nn[i][3] = (i+N-A)%N;
if (i%A==0) nn[i][1] = i+A-1;
if ((i+1)%A==0) nn[i][0] = i-A+1;
}
}
void permutation()
{
int i,j;
int temp;
for (i=0; i<N; i++) order[i] = i;
for (i=0; i<N; i++) {
j = i + (N-i)*RNGLONG;
temp = order[i];
order[i] = order[j];
order[j] = temp;
}
}
int findroot(int i)
{
CntRaizZiff++;
if (ptr[i]<0) return i;
return ptr[i] = findroot(ptr[i]);
}
long big;
void percolate()
{
int i,j;
```

```
int s1,s2;
int r1,r2;
big=0;
for (i=0; i<N; i++) ptr[i] = EMPTY;
for (i=0; i<N; i++) {
r1 = s1 = order[i];
ptr[s1] = -1;
for (j=0; j<4; j++) {
s2 = nn[s1][j];
if (ptr[s2]!=EMPTY) {
r2 = findroot(s2);
if (r2!=r1) {
if (ptr[r1]>ptr[r2]) {
ptr[r2] += ptr[r1];
ptr[r1] = r2;
r1 = r2;
} else {
ptr[r1] += ptr[r2];
ptr[r2] = r1;
}
if (-ptr[r1]>big) big = -ptr[r1];
}
}
}
}
}
int main (void) {
long iii=time(NULL);
long Semente=iii;
```

```
srand(Semente);
long TempoExec[1002];
long B[32];
long d=0;
char Topologia;
long TipoRede;
TipoRede=1;
Topologia='H';
d=2;
long TipoLado;
B[31]=N;B[30]=A;
TipoLado=0;
if (TipoLado==0 ) {
B[1]=A;
for (int i=2; i <= d; i++) B[i] = B[1];
} else {
for (int i=1; i <= d; i++) {
cout « "Informe a aresta Rede n* "<< i << ": "; cin » B[i];
}
}
long TsAB[33];
long TsOBe[33];
long TLAB[33];
long TLtAB[33];
TsAB[0]=1;TsOBe[0] = 1;
for (int i=1; i <= d; i++) TsOBe[i] = TsOBe[i-1]*(B[i]+2);
for (int i=1; i <= d; i++) TsAB [i] = TsAB [i-1]* B[i];
TLAB [d] =0;
for (int i=1; i <= d; i++) TLAB [d] +=(TsAB [d]/B[i])*(B[i]-1);
```

```
TLtAB [d] =TLAB [d];
for (int j=2; j <= d; j++)
for (int i=1; i < j; i++)
TLtAB [d] +=(TsAB [d]/(B[i]*B[i]))*(B[i]-1)*(B[j]-1);
long dt=d, s[64][2];
if ( Topologia==’T’){
TLAB [d]=TLtAB [d];
dt=(d*d+d)/2;
long k;
for (long j=1; j<=d; j++)
for (long i=1; i<j; i++){
k=d+i+((j-1)*(j-1)-j+1)/2;
s[k][0]=i;
s[k][1]=j;
}
}
long TamRede, TamRedeEsp;
if (TipoRede==1){TamRedeEsp=TsOBe[d]; TamRede=TsAB [d];}
if (TipoRede==2){TamRedeEsp=TsOBe[d]*dt; TamRede=TLAB [d];}
if (TipoRede==3){TamRedeEsp=TsOBe[d]*(dt+1);TamRede=TsAB [d]+TLAB [d];}
long Incr[32][64],TVz;
if (TipoRede==1){
TVz = 2*dt;
long SOBeX;
SOBeX=0;
for ( long t=1; t <= d; t++)
SOBeX += 2*TsOBe[t-1];
for ( long k=1; k <= dt; k++){
long i=0,j=0,eit,ejt,ekt;
```

```
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeX.Fk=0,SOBeXFk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=0;
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
SOBeX.Fk += (2+eit-ejt-ekt )*TsOBe[t-1];
SOBeXFk += (2-eit+ejt+ekt )*TsOBe[t-1];
}
Incr[0][2*k-1]=SOBeX.Fk - SOBeX;
Incr[0][2*k] =SOBeXFk - SOBeX;
}
}
if (TipoRede==2 ) {
TVz = 4*dt - 2;
long i,j,m,n,p,eit,ejt,ekt,elt,emt,ent;
long SOBeX;
SOBeX=0;
for ( long t=1; t <= d; t++)
SOBeX += 2*TsOBe[t-1];
long LtOBeXl;
for ( long l=1; l <= dt; l++) {
m=n=0;
if ( l > d )
m=s[l][0]; n=s[l][1];
LtOBeXl= dt*SOBeX + l;
p=0;
```

```
for ( long k=1; k <= dt; k++){
long LtOBeXk=dt*SOBeX + k;
if ( k!= l ) {
p++;
Incr[l][p]=LtOBeXk - LtOBeXl;
}
}
for ( long k=1; k <= dt; k++){
i=j=0;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeX.Fk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=0;
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
SOBeX.Fk += (2+eit-ejt-ekt )*TsOBe[t-1];
}
long LtOBeX.Fkk=dt*SOBeX.Fk + k;
p++;
Incr[l][p]=LtOBeX.Fkk - LtOBeXl;
}
long SOBeXFl=0;
for ( long t=1; t <= d; t++) {
ent=emt=elt=0;
if ( n==t ) ent=1;
if ( m==t ) emt=1;
if ( l==t ) elt=1;
```

```
SOBeXFl += (2-emt+ent+elt )*TsOBe[t-1];
}
for ( long k=1; k <= dt; k++){
long LtOBeXFlk=dt*SOBeXFl + k;
p++;
Incr[l][p]=LtOBeXFlk - LtOBeXl;
}
for ( long k=1; k <= dt; k++){
if ( k != l ) {
i=j=0;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeXFl.Fk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=elt=emt=ent=0;
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
if ( l==t ) elt=1;
if ( m==t ) emt=1;
if ( n==t ) ent=1;
SOBeXFl.Fk += (2+eit-ejt-emt+ent+elt-ekt )*TsOBe[t-1];
}
long LtOBeXFl.Fkk=dt*SOBeXFl.Fk + k;//LtOBeXFl.Fkk=Lt(O,Be,X+Fl-Fk,k);
p++;
Incr[l][p]=LtOBeXFl.Fkk - LtOBeXl;
}
}
}
```

```
}
if (TipoRede==3 ) {
long SOBeX=0;
long i,j,p;
long eit,ejt,ekt;
for ( long t=1; t <= d; t++)
SOBeX += 2*TsOBe[t-1];
long SLtOBeX0= (dt+1)*SOBeX;
p=0;
for ( long k=1; k <= dt; k++){
p++;
Incr[0][p]=k;
}
for ( long k=1; k <= dt; k++){
p++;
i=j=0;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeX.Fk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=0;
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
SOBeX.Fk += (2+eit-ejt-ekt )*TsOBe[t-1];
}
long SLtOBeXFl.Fkk=(dt+1)*SOBeX.Fk + k;
Incr[0][p]=SLtOBeXFl.Fkk - SLtOBeX0;
}
```

```
for ( long k=1; k <= dt; k++){
Incr[k][1]=-k;
i=j=0;
if ( k > d )
i=s[k][0]; j=s[k][1];
long SOBeXFk=0;
for ( long t=1; t <= d; t++) {
eit=ejt=ekt=0;
if ( i==t ) eit=1;
if ( j==t ) ejt=1;
if ( k==t ) ekt=1;
SOBeXFk += (2-eit+ejt+ekt )*TsOBe[t-1];
}
long SLtOBeXFk0=(dt+1)*SOBeXFk;
long SLtOBeXk =(dt+1)*SOBeX + k;
Incr[k][2]=SLtOBeXFk0 - SLtOBeXk;
}
}
struct aglomerado {
public:
long Q;
long No;
long Fr;
};
aglomerado *Ag ; Ag = new aglomerado [TamRedeEsp+10];
long *SeqEl; SeqEl = new long [TamRede+1];
aglomerado Ag0; Ag0.Q = 0; Ag0.Fr = -1;
long X[32], SOBeX[32], FrX[32];
X[d]=-1;FrX[d+1]=0;SOBeX[d+1]=0;
```

```
long StRede[32], l=1; StRede[d+1]=1;
for ( long i=d;i <= d;i– ) {
X[i]++;
if ( X[i] == (B[i]+2)) i=i+2;
else
if ( X[i] == 0 || X[i] == (B[i]+1) )
StRede[i]= 0;
else StRede[i]=StRede[i+1];
FrX[i]=FrX[i+1];
if ( X[i] == 1 )
FrX[i]|=long(powl(2,2*i-2)+.001);
if ( X[i] == B[i] )
FrX[i]|=long(powl(2,2*i-1)+.001);
SOBeX[i]=SOBeX[i+1]+X[i]*TsOBe[i-1];
X[i-1]=-1;
if ( i == 1 ) {
if ( TipoRede==1 ) {
Ag[SOBeX[1]] = Ag0;
Ag[SOBeX[1]].No = SOBeX[1];
if (StRede[1]==1) {
Ag[SOBeX[1]].Fr=FrX[1];
SeqEl[l]=SOBeX[1];
l++;
}
} else {
for (long k = (3-TipoRede); k <= dt; k++ ) {
long PLe;
PLe=SOBeX[1]*(dt+(TipoRede-2))+k;
Ag[PLe] = Ag0;
```

```
Ag[PLe].No = PLe;
long FlRede=StRede[1],i,j,AtFr=FrX[1];
if (k > 0) {
if ( k <= d ) {
if ( X[k] == (B[k]-1))
AtFr |=long(powl(2,2*k-1)+.001);
if (X[k] >= B[k] )
FlRede=0;
} else {
long i,j;
i=s[k][0]; j=s[k][1];
if ( X[j] == (B[j]-1))
AtFr|=long(powl(2,2*j-1)+.001);
if ( X[i] == 2)
AtFr|=long(powl(2,2*i-2)+.001);
if (X[i] < 2 || X[j] >= B[j])
FlRede=0;
}
}
if ( FlRede==1 ) {
if ( l>TamRede)
l++;
Ag[PLe].Fr = AtFr;
SeqEl[l]=PLe;
l++;
if (PLe == 0)
l–;
}
}
```

```
}
i++;
}
}
} long p[21], i, j, k, m, n, Di,L,NoRotuloAlt[1000],St;
p[0]=0;
long FlPerc=0,FlPercFace[32];
for (i=0; i<d; i++) {
FlPercFace[i]=long(powl(2,2*i)+.001)|long(powl(2,2*i+1)+.001);
FlPerc|=FlPercFace[i];
}
double Pc1=0,Pc2=0;
long QntRun=100, iVert;
double CntRaiz; CntRaizZiff=0;
unsigned long TempExecZiff=0, TempExecElias=0,BufTemp;
double TimeExec[100];
boundaries();
TempoExec[1]=TempoExec[2]=0;
for ( long CntRun=0; CntRun<QntRun; CntRun++) {
rngseed(CntRun);
permutation();
TimeExec[0] = clock();
percolate();
TimeExec[3]=clock();
TimeExec[1]+=TimeExec[3]-TimeExec[0];
for (iVert=0; iVert<N; iVert++) {Ag[iVert].Q=0;Ag[iVert].No=iVert;
TimeExec[0] = clock();
unsigned VertRest;
for (long iVert=0; iVert<N; iVert++) {
```

```
L=order[iVert];
m=0; n=0;
for(i = 1; i <= TVz; i++){
if ( Ag[nn[L][i-1]].Q>0){
CntRaiz++;
for(j=Ag[nn[L][i-1]].No;Ag[j].No != j; j=Ag[j].No){
n++;
NoRotuloAlt[n]=j; }
if ((Ag[j]).Q>0){
p[m]=j;
m++;}}
}
p[m]=L;
Ag[L].Q=1;
for(i=m-1;i>0;i–) {
j=p[i-1];
k=p[i];
if(Ag[j].Q <Ag[k].Q ||
(Ag[j].Q==Ag[k].Q && Ag[j].No<Ag[k].No) ){
p[i]=j;
p[i-1]=k;
i+=2;
}}
(Ag[p[0]]).Q++;
for(i=1;i<m;i++) {
k=p[i];
if (Ag[k].No!=Ag[p[i-1]].No){
Ag[p[0]].Q +=Ag[k].Q;
Ag[k].No=Ag[p[0]].No;
```

```
}
}
Ag[L].No=Ag[p[0]].No;
CntRaiz +=n;
for (i=1; i <= n;i++)
Ag[NoRotuloAlt[i]].No=p[0];
}
}
cout << "Media Busca Raiz Ziff= "<< CntRaizZiff/(QntRun*N);
cout << "Media Busca Raiz Elias= "<< CntRaiz/(QntRun*N);
cout <<"Tempo Ziff = "<< TimeExec[1];
cout << "Tempo Elias = "<< TimeExec[2];
cout << "; iVert = "<< iVert << "L = "<< L << "Q = "<< Ag[p[0]].Q << "; big = "<< big;
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo