UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

HENRIQUE LUÍS SCALON

PARALELIZAÇÃO DO CÁLCULO DA ENTROPIA CONFIGURACIONAL

CAXIAS DO SUL
2018

HENRIQUE LUÍS SCALON

PARALELIZAÇÃO DO CÁLCULO DA ENTROPIA CONFIGURACIONAL

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. André Luís Martinotto
Coorientador: Prof. Dr. Cláudio Antônio Perottoni

CAXIAS DO SUL
2018

HENRIQUE LUÍS SCALON

PARALELIZAÇÃO DO CÁLCULO DA ENTROPIA CONFIGURACIONAL

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 05/12/2018

Banca Examinadora

Prof. Dr. André Luis Martinotto
Universidade de Caxias do Sul - UCS

Prof. Dr. Cláudio Antônio Perottoni
Universidade de Caxias do Sul - UCS

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

RESUMO

Em sistemas amorfos a rede de ligações entre os átomos permite uma vasta quantidade de topologias, sendo denominada de entropia configuracional o número de diferentes topologias que um material pode ter. Neste trabalho, foi desenvolvida uma implementação para o cálculo da entropia configuracional baseada no método proposto por Vink e Barkema e publicado no periódico *Physical Review Letters*, volume 89 de julho de 2002. Esse método baseia-se na utilização da teoria da informação, onde somente as posições dos átomos e as ligações entre eles são utilizadas. Para o cálculo da entropia configuracional, são geradas posições aleatórias e são gerados os grafos formados com os vizinhos mais próximos de cada uma dessas posições. Além disso, é utilizada a técnica de isomorfismo de grafos para compará-los e a entropia de Shannon é calculada com as probabilidades de cada topologia ocorrer. A implementação foi desenvolvida utilizando as linguagens de programação *Python* e *C++*. Foram utilizados os pacotes na linguagem *Python*: *ASE*, para manipulação de estruturas atômicas; *Matplotlib*, para criação de gráficos e *NumPy*, para realização de cálculos matriciais e vetoriais. Já na linguagem *C++* foram utilizadas as bibliotecas: *PyBind11*, para realizar a comunicação entre as linguagens *Python* e *C++*; *Aboria*, para o tratamento de partículas; *BGL*, para o tratamento de grafos; e *Nauty*, para as verificações de isomorfismo. A paralelização foi feita utilizando a *API OpenMP*. Para a validação da implementação, foi utilizada a estrutura cristalina do Cobre e uma estrutura amorfa de Silício e Oxigênio disponibilizada por Vink. Para a estrutura do Cobre foi obtido o resultado de 0 k_b, o que é condizente com a entropia configuracional esperada em materiais cristalinos. Já para a estrutura amorfa, foi encontrado o resultado de 0,88 k_b, que é um valor idêntico ao apresentado por Vink e Barkema no artigo publicado no *Physical Review Letters*. Já a paralelização resultou em um *Speedup* de 3,1x utilizando 12 *cores*.

Palavras-chave: entropia configuracional. redes amorfas. grafos.

LISTA DE ILUSTAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

CRN	*Continuous Random Network*
BCC	*Body Centered Cubic*
FCC	*Face Centered Cubic*
HCP	*Hexagonal Close Packed*
ASE	*Atomistic Simulation Environment*
NP	*Nondeterministic Polynomial time*
API	*Application Programming Interface*

LISTA DE SÍMBOLOS

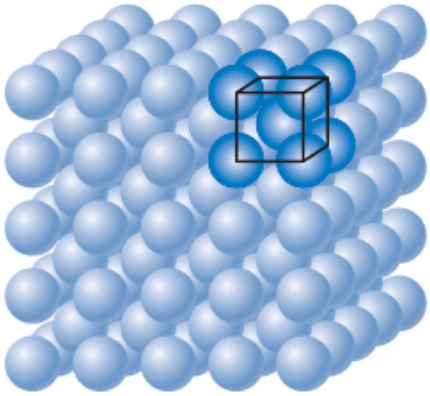
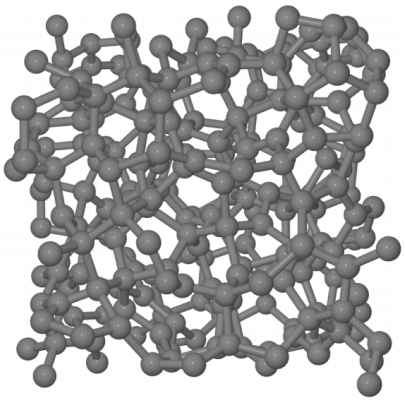
Å	Ångström
$\in$	Pertence
$\approx$	Valor aproximado
$\sum$	Somatório
$\equiv$	Valor equivalente
k_b	Constante de Boltzmann, aproximadamente igual a $1.38065 \times 10^{-23} J/K$

SUMÁRIO

1 INTRODUÇÃO

Os sólidos podem ser classificados de acordo com a regularidade pela qual os átomos ou íons estão arranjados uns em relação aos outros. Denomina-se material cristalino aquele no qual os átomos estão posicionados segundo um arranjo periódico ou repetitivo ao longo de grandes distâncias atômicas, isto é, existe uma ordem de longo alcance, tal que, quando ocorre a solidificação, os átomos se posicionam em um padrão tridimensional repetitivo. Os materiais nos quais esta ordem atômica de longo alcance não está presente são chamados de amorfos (JR; RETHWISCH, 2018). Na Figura 1a, tem-se um exemplo de um material cristalino e na Figura 1b tem-se um exemplo de um material amorfo.

Figura 1 – Representação de duas estruturas 3D, sendo a primeira, uma estrutura cristalina BCC (*Body Centered Cubic*) (JR; RETHWISCH, 2018) e a segunda, uma estrutura amorfa de carbono (JORNADA, 2010).

Figura 1a)	Figura 1b)
	

A sílica vítrea (*vitreous silica*), silício amorfo (*amorphous silicon*) ou gelo vítreo (*vitreous ice*) são alguns exemplos de materiais amorfos. Nesses materiais, enquanto o ambiente local de cada partícula é normalmente ordenado, a rede de ligações entre eles permite uma grande variedade de topologias. Dá-se o nome de entropia configuracional à medida do número de diferentes topologias que um material pode ter (VINK, 2002).

Segundo Graeser et. al (2010), a entropia configuracional é importante em sistemas amorfos, pois está envolvida em questões termodinâmicas, cumprindo um papel importante no cálculo da mobilidade molecular (aparece na equação de Adam-Gibbs (ADAM; GIBBS, 1965)) e fornecendo informações sobre o aumento da solubilidade da forma amorfa quando comparada com a sua contraparte cristalina.

Em 2002, Vink e Barkema propuseram um método para calcular a entropia configuracional que pode ser aplicado a qualquer estrutura, desde que as coordenadas dos átomos e uma lista de ligações sejam conhecidas (VINK; BARKEMA, 2002). Primeiramente, deve-se escolher um número elevado de posições aleatórias em uma célula de simulação, sendo que para cada uma dessas posições, procura-se os átomos mais próximos e identifica-se o grafo formado pelas ligações que conectam esses átomos. Então é determinado um rótulo para esse grafo, sendo que um rótulo diferente é atribuído para cada um dos grafos com diferentes topologias. A comparação dos grafos é baseada no isomorfismo de grafos (MCKAY; PIPERNO, 2014).

A Equação 1.1 é usada para estimar a probabilidade de cada grafo ocorrer, onde fi corresponde ao número de vezes que um grafo foi observado e m consiste no número de posições aleatórias geradas. O resultado dessa função é usado na Equação 1.2 para a entropia de Shannon *H(n)*, que é dada por um somatório envolvendo a probabilidade de um grafo i ocorrer, onde n equivale ao número de vizinhos mais próximos que foram utilizados para a criação do grafo (VINK; BARKEMA, 2002).

$$p(i) \approx \frac{fi}{m}, \tag{1.1}$$

$$H(n) = -\sum_{i} p(i) \log_2 p(i), \tag{1.2}$$

O objetivo deste trabalho consistiu em desenvolver uma implementação computacional do método proposto por Vink e Barkema. Para a validação desta implementação, foram utilizados materiais cristalinos, como por exemplo, o Cobre e o Titânio. Além desses, foram utilizadas estruturas amorfas, cujos arquivos foram disponibilizados, através de contato, por Vink. Após, foi utilizada uma abordagem de programação paralela objetivando diminuir o tempo de execução, sendo que a paralelização foi desenvolvida de forma a utilizar de maneira eficiente os recursos computacionais de arquiteturas multiprocessadas e *multi-cores*.

1.1 OBJETIVO GERAL

O objetivo deste trabalho consistiu na implementação do método proposto por Vink e Barkema (VINK; BARKEMA, 2002).

Com base no objetivo geral, foram elaborados os seguintes objetivos específicos:

1. Implementação de um *software* para o cálculo da entropia configuracional, utilizando o método de Vink e Barkema (VINK; BARKEMA, 2002);

2. Paralelização do *software* de forma a utilizar eficientemente os recursos de arquiteturas com memória compartilhada;

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está organizado da seguinte forma:

- No Capítulo 1 é feita uma breve introdução do trabalho, apresentando suas motivações e objetivos.
- No Capítulo 2 é apresentado o método proposto por Vink e Barkema para o cálculo da entropia configuracional. Além disso, são apresentados alguns conceitos relativos a Teoria dos Grafos, que serão utilizados no desenvolvimento deste trabalho.
- No Capítulo 3 são apresentados detalhes de como o *software* foi desenvolvido, tais como: linguagem, bibliotecas e algoritmos utilizados. Além disso, são apresentados os testes com estruturas cristalinas que foram realizados para uma validação inicial da implementação sequencial.
- No Capítulo 4 são apresentadas as otimizações realizadas no *software*, bem como a sua paralelização. Também são apresentados os resultados obtidos utilizando as estruturas disponibilizadas por Vink.
- No Capítulo 5 são apresentadas as conclusões do trabalho, bem como sugestões de trabalhos futuros.

2 CÁLCULO DA ENTROPIA CONFIGURACIONAL

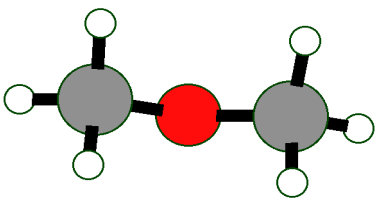
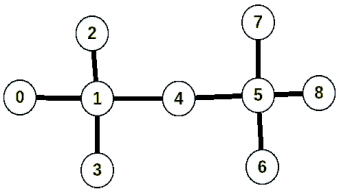
Para o cálculo da entropia configuracional será utilizado o método proposto por Vink e Barkema (VINK, 2002). Esse método é baseado na teoria da informação, que foi originalmente proposta por Claude Shannon em 1948 (PINEDA, 2006).

Segundo Shannon, a informação de uma mensagem pode ser mensurada por uma quantidade denominada "entropia", que está relacionada com a frequência de ocorrências dos símbolos, sendo 0 (zero) quando somente um símbolo é encontrado e máxima (1) quando a frequência é equiprovável (PINEDA, 2006). O uso da teoria da informação para o cálculo da entropia configuracional difere do método convencional de cálculo de entropia, onde a temperatura e a energia dos átomos são utilizados (BENGUIGUI, 2013). No caso da teoria da informação, somente a topologia dos átomos é considerada, ou seja, a entropia é determinada pela estrutura organizacional (ligações) e não pelas coordenadas atômicas (VINK, 2002).

2.1 MODELAGEM ATRAVÉS DE GRAFOS

Um sistema cristalino ou amorfo pode ser representado como um grafo. Na Figura 2a podemos ver a representação gráfica em três dimensões da molécula do dimetil éter (CH_3OCH_3). Já na Figura 2b tem-se a mesma molécula representada através de um grafo.

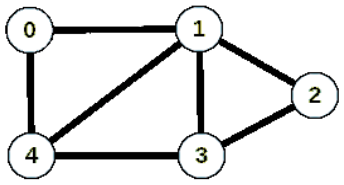
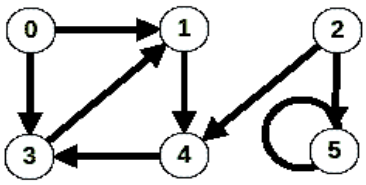
Figura 2 – Representações da molécula dimetil éter (CH_3OCH_3) (Própria, 2018).

Figura 2a)	Figura 2b)
	

Um grafo $G = (V, E)$ é formado por dois conjuntos, onde o conjunto V representa os vértices do grafo e o conjunto E, as arestas do grafo. Considerando materiais cristalinos e amorfos, os vértices representam os átomos e as arestas representam as ligações químicas.

Um grafo pode ser classificado como não dirigido ou dirigido. Nos grafos não dirigidos, para cada aresta que parte do vértice V para o vértice V' há uma aresta que parte do vértice V' para o vértice V, sendo que nos grafos dirigidos isso não ocorre. Na Figura 3a, podemos ver a representação de um grafo não dirigido com cinco vértices e sete arestas. Já na Figura 3b tem-se um exemplo de um grafo dirigido com seis vértices e oito arestas.

Figura 3 – Representação de um grafo não dirigido e de um grafo dirigido. Adaptado de (CORMEN. et al., 2012).

Figura 3a)	Figura 3b)
	

2.2 ESTRUTURAS DE DADOS PARA A REPRESENTAÇÃO DE GRAFOS

Frequentemente, são utilizados dois tipos de estrutura de dados para a representação de um grafo, que são: listas de adjacências e matrizes de adjacências.

Considerando um grafo $G = (V, E)$ formado pelo conjunto V de vértices e E de arestas, a representação por listas de adjacências do grafo consiste em um arranjo Adj de $|V|$ listas, uma para cada vértice u em V. Para cada $u \in V$, a lista de adjacências $Adj[u]$ contém todos os vértices v tais que existe uma aresta $(u, v) \in E$. Ou seja, $Adj[u]$ consiste em uma lista com todos os vértices que possuem uma aresta em comum com o vértice u. Essa representação é indicada para grafos esparsos, onde o número de arestas é menor que o número de vértices ao quadrado (CORMEN. et al., 2012). Na Figura 4, temos dois grafos representados por listas de adjacências, sendo que a Figura 4a apresenta o grafo não dirigido da Figura 3a e a Figura 4b apresenta o grafo dirigido da Figura 3b.

Figura 4 – Representações de dois grafos por listas de adjacências. Adaptado de (CORMEN. et al., 2012).

Figura 4a)

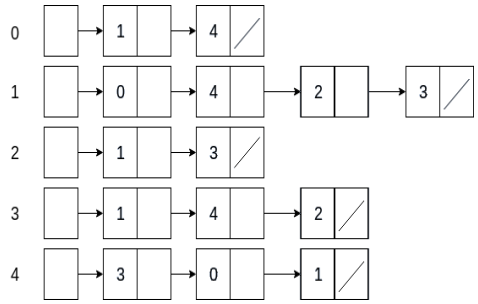
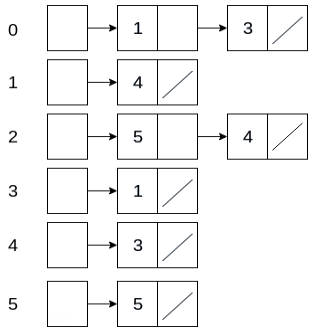


Figura 4b)



Já na representação por matrizes de adjacências, um grafo é representado através de uma matriz quadrada A de ordem $|V|$, tal que (a_{ij}) é dado por

$$a_{ij} = \begin{cases} 1 & se\ (i,j) \in E, \\ 0 & caso\ contrário, \end{cases} \tag{2.1}$$

ou seja, o elemento $a_{ij} = 1$ se existe uma aresta conectando os vértices i e j, e $a_{ij} = 0$, caso essa aresta não exista. Essa representação é preferível quando o grafo é denso (o número de arestas é próximo do número de vértices ao quadrado) ou quando se necessita acessar rapidamente uma aresta (CORMEN. et al., 2012). Na Figura 5, temos dois grafos ilustrados por matrizes de adjacências, sendo que a Figura 5a apresenta a matriz de adjacências do grafo da Figura 3a e a Figura 5b apresenta o grafo dirigido da Figura 3b.

Figura 5 – Representações de dois grafos por matrizes de adjacências. Adaptado de (CORMEN. et al., 2012).

Figura 5a)

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Figura 5b)

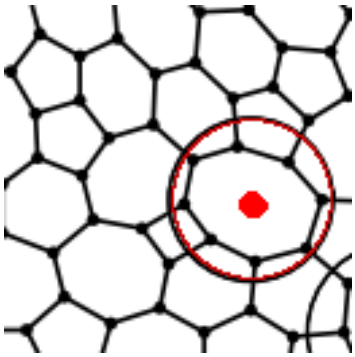
	0	1	2	3	4	5
0	0	1	0	1	0	0
1	0	0	0	0	1	0
2	0	0	0	0	1	1
3	0	1	0	0	0	0
4	0	0	0	1	0	0
5	0	0	0	0	0	1

2.3 MÉTODO DE VINK E BARKEMA

De acordo com o método de Vink e Barkema, deve-se inicialmente gerar um grafo que represente a estrutura do material. Esse grafo é não dirigido, uma vez que não existe uma ligação unilateral entre dois átomos. Neste caso, sugere-se ainda a utilização de uma representação por listas de adjacências, uma vez que esse grafo tende a ser esparso.

Posteriormente, escolhe-se um número m de posições aleatórias em uma célula de simulação, sendo que para cada uma dessas posições, escolhe-se um número n de átomos mais próximos (vizinhos). Então é identificado o grafo formado pelas arestas (ligações) que conectam esses átomos, e é determinado um rótulo para ele. Na Figura 6, pode-se ver um grafo formado pelos oito vértices mais próximos de uma posição aleatória m.

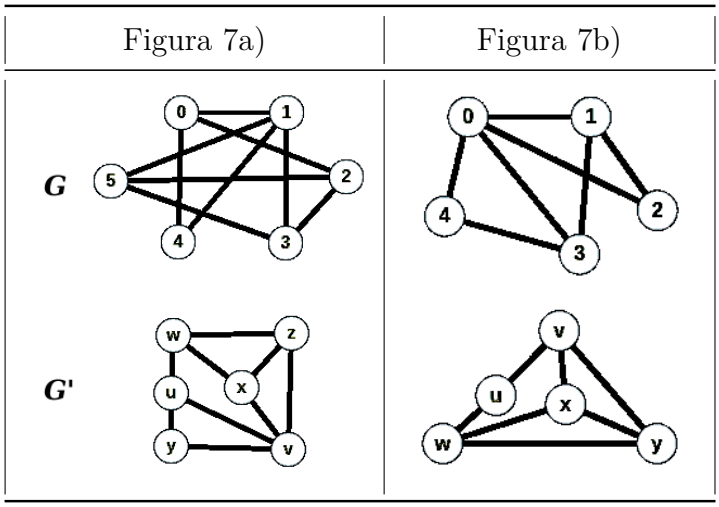
Figura 6 – Representação de um grafo formado por oito vértices, considerando $n = 8$. O ponto vermelho representa uma posição aleatória, e o círculo vermelho representa os oito vizinhos mais próximos. Adaptado de (VINK, 2002).



Os m grafos formados são então comparados, de modo a verificar se são isomorfos ou não (VINK, 2002). Dois grafos G e G' são isomorfos se é possível renomear os vértices de G como vértices de G', mantendo as arestas correspondentes em G e G' (CORMEN. et al., 2012). Na Figura 7a, tem-se um exemplo de dois grafos isomorfos. O mapeamento entre os vértices de G e G' é dado por $f(0) = u$, $f(1) = v$, $f(2) = w$, $f(3) = x$, $f(4) = y$ e $f(5) = z$. Já na Figura 7b, os grafos não são isomorfos, visto que G apresenta um vértice de grau 4 [1] (vértice 0) e G' não tem (CORMEN. et al., 2012).

[1] O grau de um vértice é definido como o número de vértices adjacentes a ele (MARTINEZ, 2005).

Figura 7 – Representação de grafos isomorfos e não isomorfos. Adaptado de (CORMEN. et al., 2012).



Posteriormente, é contabilizado o número de vezes que um grafo é observado. A Equação 2.2 é usada para estimar a probabilidade de cada grafo ocorrer, onde fi corresponde ao número de vezes que um grafo foi observado e m consiste no número de posições aleatórias geradas.

$$p(i) \approx \frac{fi}{m}, \tag{2.2}$$

O resultado da Equação 2.2 é incluído na Equação 2.3, para obter a entropia de Shannon $H(n)$, que é dada por um somatório envolvendo a probabilidade de um grafo i ocorrer, sendo n o número de átomos vizinhos utilizados para a criação do grafo após a escolha de uma posição aleatória (VINK; BARKEMA, 2002).

$$H(n) = -\sum_{i} p(i) \log_2 p(i), \tag{2.3}$$

Segundo demonstrado por Jaynes (1965), em sistemas em equilíbrio, a entropia de Shannon (Eq. 2.3) e a entropia termodinâmica são equivalentes, à parte de um fator $k_b \ln(2)$. Desse modo, a equação utilizada para o cálculo da entropia de Shannon pode ser escrita de acordo com a Equação 2.4 (VINK, 2002).

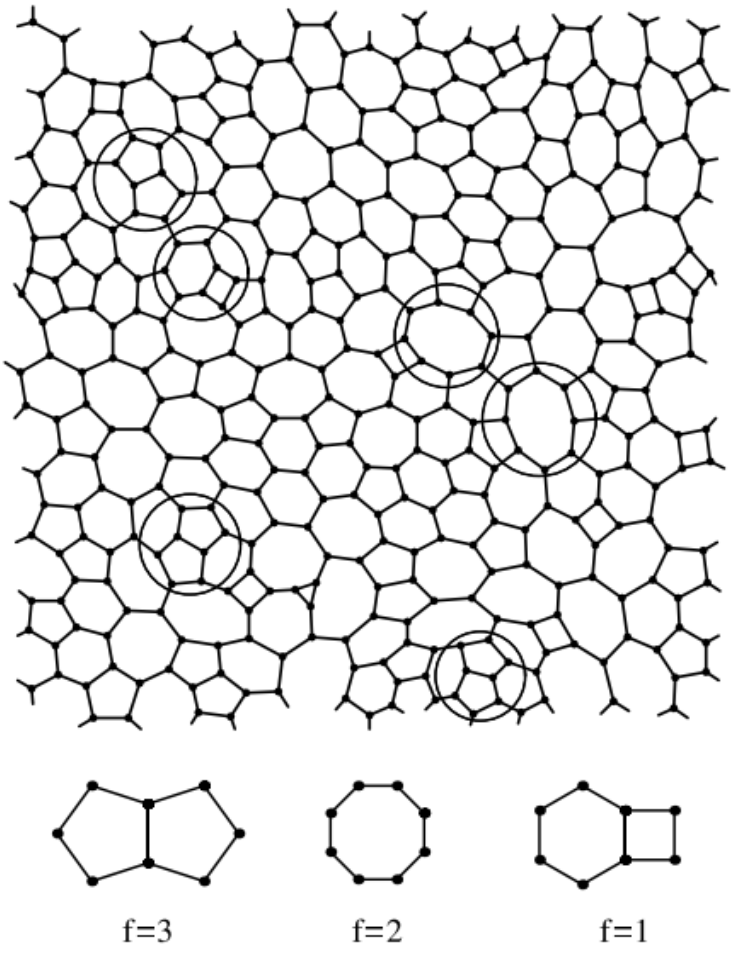
$$H(n) = -k_b \sum_{i} p(i) \ln p(i) \tag{2.4}$$

O procedimento descrito pode ser observado na Figura 8 para uma rede amorfa de duas dimensões. Nesse exemplo, é calculado o valor de $H(n)$ considerando 8 átomos vizinhos ($n = 8$), e sendo escolhidas $m = 6$ posições aleatórias. Os grafos gerados encontram-se

circulados na Figura 8. A frequência de ocorrência de cada grafo é mostrada na parte inferior da figura, sendo que o grafo da esquerda pode ser observado três vezes, o do meio duas vezes e o da direita uma. Baseado nessas posições aleatórias, as probabilidades de ocorrência são 3/6, 2/6 e 1/6, respectivamente (VINK, 2002). Essas probabilidades são inseridas na Equação 2.4, resultando em

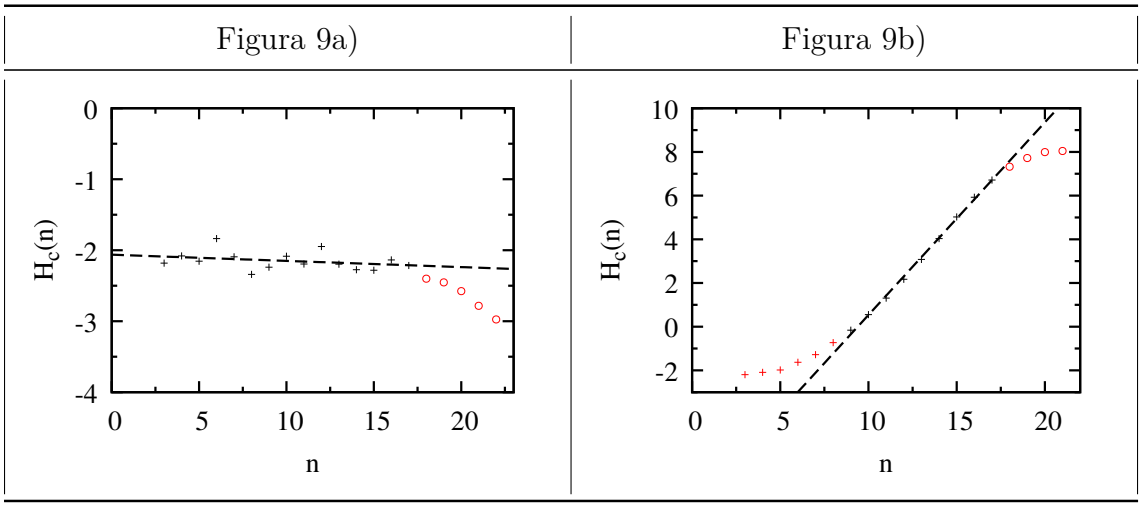
$$H(8) = -(3/6)\ln 3/6 - (2/6)\ln 2/6 - (1/6)\ln 1/6 \approx 1.0114. \tag{2.5}$$

Figura 8 – Exemplo da escolha de grafos, através de posições aleatórias, pelo método de Vink e Barkema (VINK, 2002).



Com base nos valores encontrados para a entropia de Shannon $H(n)$, em função do número de átomos vizinhos (n), são gerados os gráficos da Figura 9, onde a inclinação da reta ajustada $H(n)$ é o valor estimado para a entropia configuracional. Em materiais cristalinos, a entropia configuracional é igual a zero. Já em materiais amorfos a entropia configuracional apresenta um valor maior que zero. Na Figura 9a tem-se o exemplo de um gráfico de H(n) para uma estrutura cristalina, e na Figura 9b para uma estrutura amorfa. Nestes gráficos, estão sendo desconsiderados os valores de $g(n)$ e $H_c(n)$, que serão discutidos posteriormente.

Figura 9 – Representação de um gráfico formado para estimar a probabilidade de uma estrutura cristalina na esquerda (Próprio, 2018) e de uma estrutura amorfa na direita (Próprio, 2018). A linha tracejada é o ajuste de reta usado para estimar a entropia configuracional.

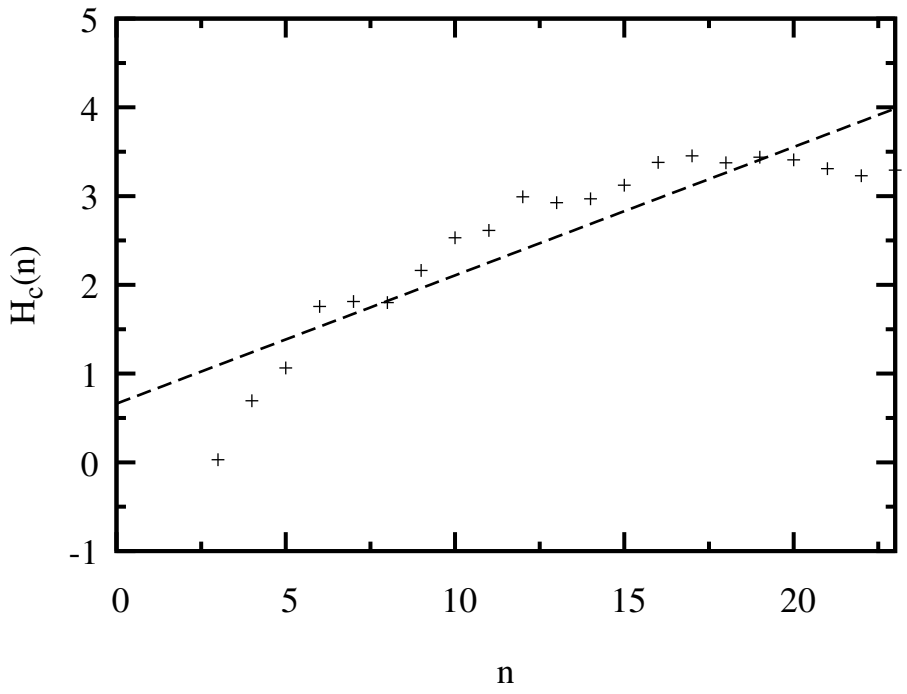


A escolha de posições aleatórias produz um efeito colateral, onde um pequeno deslocamento de uma dessas posições produz os mesmos vizinhos mais próximos e, consequentemente, o mesmo grafo. Logo há um limite máximo de posições aleatórias a serem escolhidas. Uma estimativa foi obtida por Vink e Barkema, utilizando a distância típica na qual uma posição aleatória pode ser modificada sem alterar o grafo selecionado. Nesse caso, foram obtidos como limites superiores os valores de $m = 1.6nN$ para redes de duas dimensões e $m = 3.4n^2N$ para redes de três dimensões, sendo que m é o número de posições aleatórias, n o número de vizinhos mais próximos e N o total de átomos do sistema (VINK; BARKEMA, 2002).

Segundo Vink e Barkema, outro efeito colateral da escolha de posições aleatórias é que o número de grafos diferentes observados é multiplicado por um fator proporcional a n^{d-1}, sendo d igual ao número de dimensões espaciais da rede (VINK; BARKEMA, 2002). Ou seja, usando-se como valor de $m = 1.6nN$ (2D) e $m = 3.4n^2N$ (3D), m cresce proporcionalmente a n^{d-1}. Da mesma forma, a quantidade total de grafos diferentes

observados também cresce proporcionalmente a n^{d-1}, superestimando o valor de $H(n)$. Esse efeito pode ser observado na Figura 10, onde tem-se uma representação de $H(n)$ para uma estrutura de Cobre, que apresenta uma rede cristalina *FCC*, e consequentemente, deveria possuir uma entropia igual a zero. Como pode ser observado, a entropia configuracional (inclinação da reta), neste caso, é diferente de zero.

Figura 10 – Exemplo de medição de $H(n)$ para uma estrutura de Cobre com 48 átomos. A linha tracejada representa o ajuste de reta usado para estimar a entropia configuracional (Próprio, 2018).



O efeito de superstimação do valor de $H(n)$ pode ser corrigido através da Equação $H'(n)$

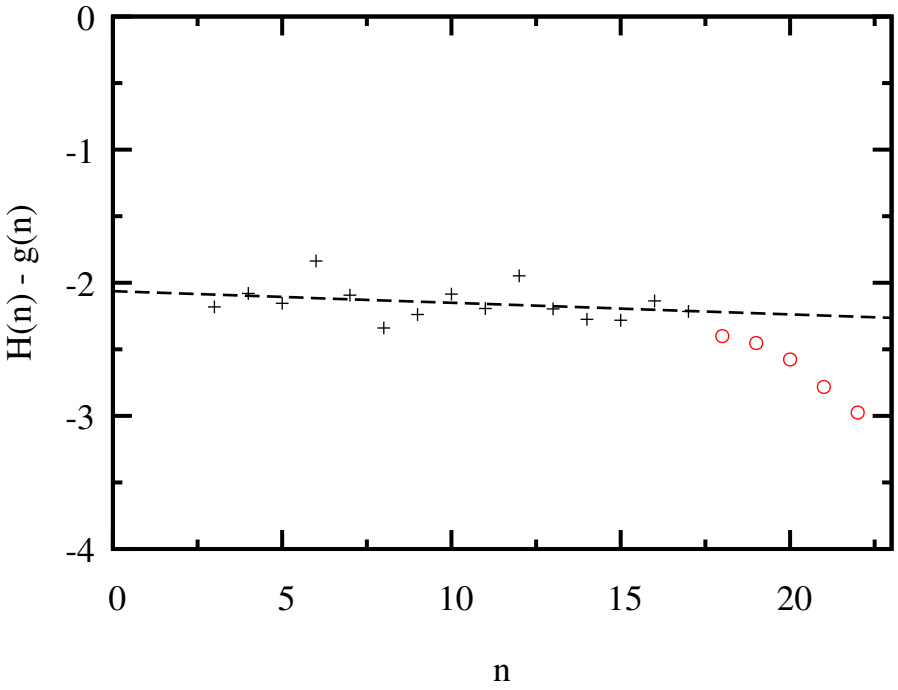
$$H'(n) = H(n) - g(n), \tag{2.6}$$

onde

$$g(n) = (d-1)\ln(n), \tag{2.7}$$

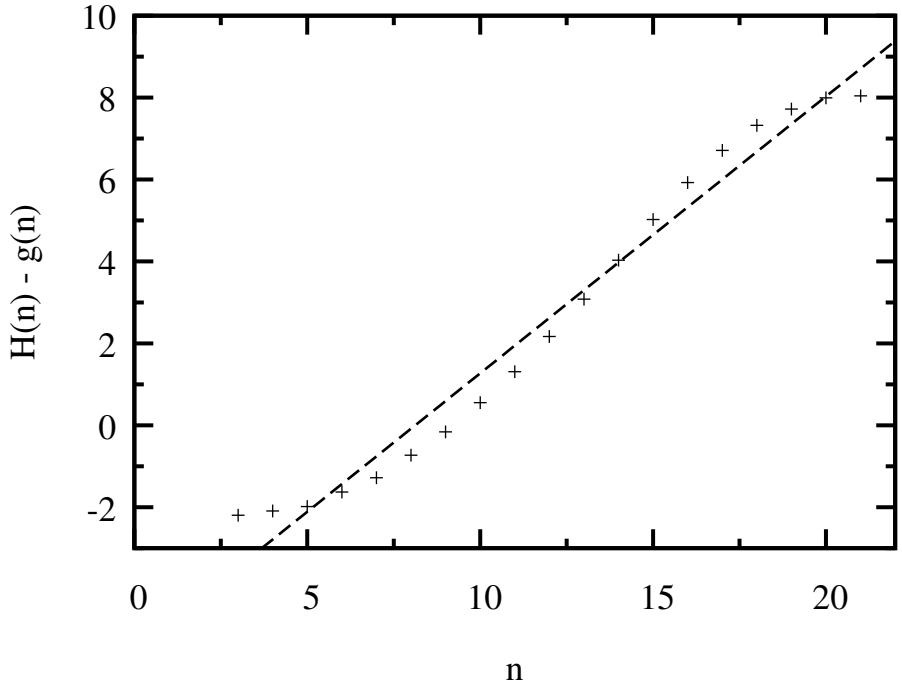
sendo d o número de dimensões espaciais da rede e n o número de vizinhos de uma posição aleatória. Na Figura 11, pode-se verificar o resultado dessa correção quando aplicada aos valores de $H(n)$ da estrutura de Cobre, que foram apresentados na Figura 10 (VINK; BARKEMA, 2002).

Figura 11 – Exemplo de medição considerando $H(n) - g(n)$ em uma estrutura de Cobre com 48 átomos, que apresenta uma rede cristalina *FCC*. A linha tracejada representa o ajuste de reta usado para estimar a entropia configuracional (Próprio, 2018).



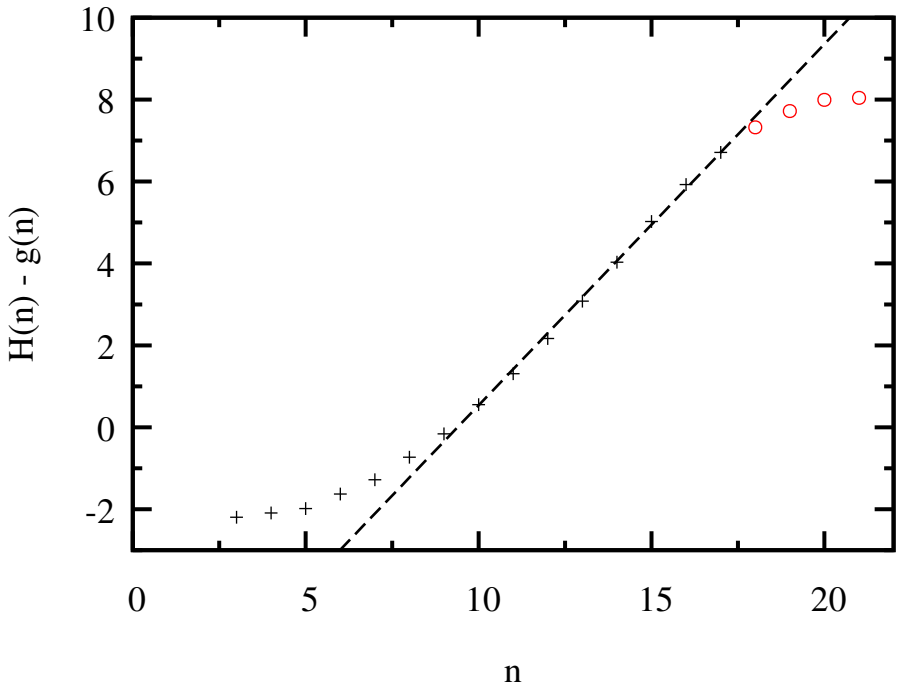
A medida que o valor de n aumenta (número de átomos vizinhos), os grafos gerados tendem a ser mais complexos, e a probabilidade de serem observados mais de uma vez diminui. Isso impacta no cálculo da entropia configuracional, uma vez que altera a convergência dos valores de $H(n)$ e, consequentemente, alterando a inclinação da reta, conforme pode ser visto na Figura 12 (VINK; BARKEMA, 2002). De fato, observa-se que, a partir dos valores de n maiores que 17, tem-se uma alteração no comportamento da reta.

Figura 12 – Exemplo do comportamento de $H(n) - g(n)$ com o aumento do valor de n para uma estrutura amorfa. A linha tracejada representa o ajuste de reta usado para estimar a entropia configuracional (Próprio, 2018).



De forma a eliminar esse efeito, tem-se o valor $H_1(n)$, que é definido como a contribuição para $H(n)$ das topologias observadas apenas uma vez. As medições de $H(n)$ em que o valor de $H_1(n)$ excede em 1% do total devem ser desconsideradas. Na Figura 13, tem-se os resultados de $H(n) - g(n)$ (Figura 12) excluindo-se as medições de $H(n)$ em que $H_1(n)$ excede 1% do total. Os círculos após $n = 17$ representam os pontos onde $H_1(n)$ excede um por cento de $H(n)$ e que devem ser desconsiderados (VINK; BARKEMA, 2002).

Figura 13 – Exemplo do comportamento de $H(n) - g(n)$ desconsiderando as medições em que o valor de $H_1(n)$ excede um por cento do total. A linha tracejada é somente um guia para os olhos (Próprio, 2018).



Além disso, Vink e Barkema adicionaram um fator de correção c de forma a acelerar a convergência de $H(n)$ com o aumento de pontos aleatórios m, conforme pode ser visto na Equação 2.8 (VINK; BARKEMA, 2002).

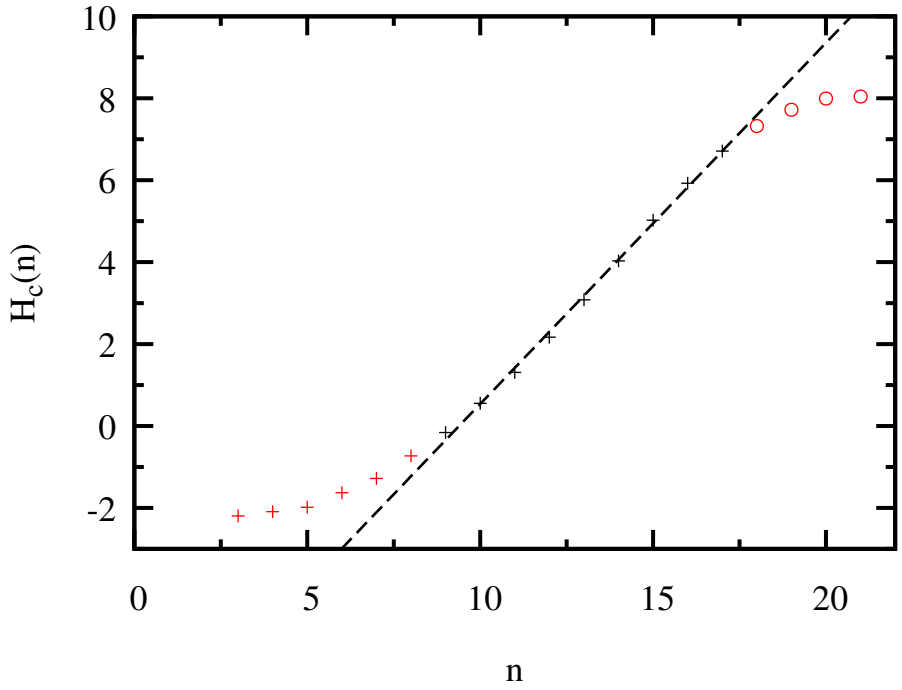
$$H_e(n) = H(n) + c \times H_1(n)/H(n), \tag{2.8}$$

Desse modo, $H_e(n)$ é usado como uma estimativa para $H(n)$. Assim, a entropia de Shannon corrigida dá-se pela Equação 2.9 (VINK; BARKEMA, 2002).

$$H_c(n) \equiv H_e(n) - g(n). \tag{2.9}$$

A Figura 14 apresenta uma curva com o comportamento típico de entropia de Shannon corrigida $H_c(n)$ em função do número de vizinhos, n. A convergência é demonstrada pelo comportamento linear de $H_c(n)$ (linha tracejada), sendo que a entropia configuracional por átomo é igual a inclinação da reta (VINK; BARKEMA, 2002).

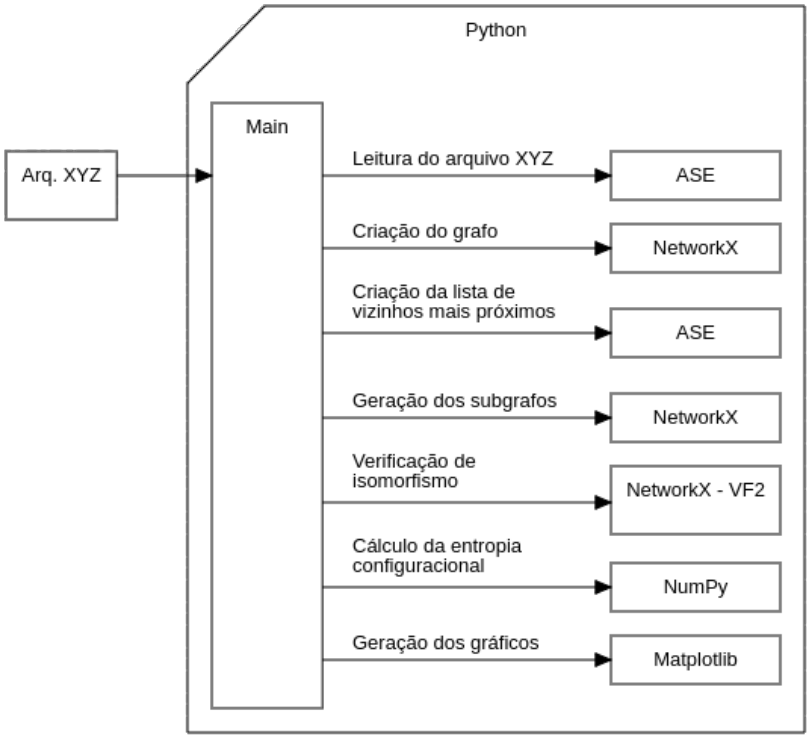
Figura 14 – Comportamento típico de $H_c(n)$ em função de n. A linha tracejada é um ajuste de reta, iniciando em $n = 9$ e segue as cruzes, até $n = 17$, sendo sua inclinação a estimativa para a entropia configuracional por átomo. Os valores circulados em vermelho são desconsiderados (Próprio, 2018).



3 IMPLEMENTAÇÃO DESENVOLVIDA

A implementação foi desenvolvida na linguagem de programação *Python*, que é uma linguagem *open-source*, com uma extensa comunidade e que possui um grande número de pacotes disponibilizados por terceiros (PYTHON, 2018a). Mais especificamente, para o desenvolvimento inicial foram utilizados os pacotes: *ASE (Atomistic Simulation Environment)* (ASE, 2018), para manipulação das estruturas atômicas; *NetworkX* (NETWORKX, 2018a), para o tratamento dos grafos; *Matplotlib* (MATPLOTLIB, 2018), para a criação de gráficos e *NumPy* (NUMPY, 2018), para realização de cálculos matriciais e vetoriais. Na Figura 15 tem-se o Fluxograma que apresenta a estrutura do programa e as bibliotecas utilizadas. O código fonte da implementação encontra-se no Apêndice A.

Figura 15 – Fluxograma da estrutura do programa e as bibliotecas utilizadas (Próprio, 2018).



3.1 ENTRADA DE DADOS

A implementação apresenta como entrada um arquivo no formato *Extended XYZ* com a estrutura do material. O formato *XYZ* é usado para representar estruturas químicas, sendo que: na primeira linha do arquivo encontra-se o número de átomos; na segunda linha, tem-se os comentários; e nas linhas restantes, encontram-se o símbolo atômico e as coordenadas x, y e z (em ångströms), de cada um dos átomos que formam a estrutura (OPENLABEL, 2007). No Quadro 1, tem-se um exemplo de um arquivo no formato *XYZ* com 48 átomos de cobre, sendo mostradas somente as cinco primeiras linhas do arquivo.

Quadro 1 – Exemplo de um arquivo resumido no formato *XYZ*. (Próprio, 2018).

```
48

Cu      0.00000000      0.00000000      0.00000000
Cu      0.00000000      1.47377633      2.08423447
Cu      0.00000000      2.94755266      4.16846894
...
```

Já o formato *Extended XYZ* adiciona melhorias ao formato *XYZ*, sendo que alguns parâmetros foram inseridos na linha de comentário, no formato chave e valor, tais como: *Lattice*, *Properties*, *pbc*, entre outros (KERMODE, 2016). O parâmetro *Lattice* recebe uma matriz 3x3 (no formato "1x 1y 1z 2x 2y 2z 3x 3y 3z") que representa a célula unitária da estrutura (KERMODE, 2016). O parâmetro *Properties* descreve a formatação do arquivo, como, por exemplo, que o átomo vai ser representado por um valor textual no início da linha, seguido por três valores reais (HIREL, 2010). Já o parâmetro *pbc* especifica as condições periódicas de fronteira[1] que serão utilizadas durante a simulação, tendo o formato de três caracteres, que podem ser verdadeiro (T) ou falso (F), para as coordenadas x, y, z, respectivamente (KERMODE, 2014). No Quadro 2, tem-se um exemplo de um arquivo no formato *Extended XYZ*, com 48 átomos de cobre. Neste exemplo, são apresentadas apenas as cinco primeiras linhas (a segunda linha foi quebrada em virtude da formatação).

[1] Condições de fronteira periódicas são usadas para simular um ambiente grande (infinito) em um pequeno (finito), através de uma célula unitária que é replicada no espaço da simulação (ZHIGILEI, 2018).

Quadro 2 – Exemplo de um arquivo resumido no formato *Extended XYZ*. (Próprio, 2018).

```
48
Lattice="5.10531 0.0 0.0 0.0 8.84265 0.0 0.0 0.0 12.50540"
Properties=species:S:1:pos:R:3:Z:I:1 pbc="T T F"
Cu       0.00000000      0.00000000      0.00000000      29
Cu       0.00000000      1.47377633      2.08423447      29
Cu       0.00000000      2.94755266      4.16846894      29
...
```

A implementação desenvolvida permite ainda a utilização de resultados gerados a partir de uma execução anterior. Para tanto é gerado um arquivo como o apresentado no Quadro 3, e que possui o seguinte formato: na primeira linha tem-se um cabeçalho com informações como, o nome do arquivo *Extended XYZ*, o valor da variável c (Equação 2.8) e o número de átomos vizinhos (n) inicial (n1) e final (n2). Após, são inseridas as linhas com os resultados obtidos para cada valor de n, contendo: o número de átomos vizinhos (n), o número de posições aleatórias (m), a entropia de Shannon ($H(n)$), a contribuição para a entropia de Shannon das topologias observadas apenas uma vez ($H_1(n)$) e se esse resultado deve ser considerado para o cálculo.

Quadro 3 – Exemplo de um arquivo de resultados gerado pelo programa. (Próprio, 2018).

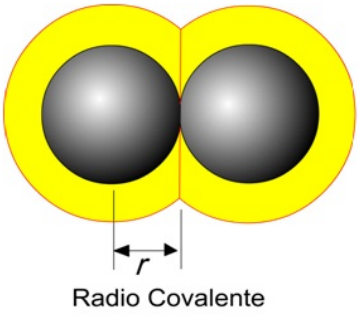
```
filepath: arquivos_xyz/3k.xyz; covalent: 1.12; c: 90.0; n1: 3; n2: 11;
n: 3; m: 135000; H_n: 0.000503576823688; H1n: 0.0; consider: N;
n: 4; m: 240000; H_n: 0.681242883508; H1n: 0.0; consider: N;
n: 5; m: 375000; H_n: 1.23530067251; H1n: 0.0; consider: N;
n: 6; m: 540000; H_n: 1.95555229655; H1n: 0.0; consider: N;
n: 7; m: 735000; H_n: 2.61075776749; H1n: 0.0; consider: N;
n: 8; m: 960000; H_n: 3.4298199016; H1n: 0.0; consider: N;
n: 9; m: 1215000; H_n: 4.2318146022; H1n: 6.9186442639e-05; consider: Y;
n: 10; m: 1500000; H_n: 5.15354753703; H1n: 0.000208574309774; consider: Y;
```

3.2 GERAÇÃO DO GRAFO INICIAL

O arquivo no formato *Extended XYZ* é lido e processado utilizando a biblioteca *ASE* (ASE, 2018), sendo armazenadas as posições e as distâncias entre os átomos, considerando condições periódicas de fronteira. Na identificação das arestas do grafo, dois átomos são considerados conectados se a distância euclidiana entre eles for menor do que a soma dos seus raios covalentes, multiplicada por um valor de *cutoff* [2], sendo que neste trabalho foi utilizado um valor de *cutoff* igual a 1.12 Å, porém esse parâmetro pode ser alterado pelo usuário (CHEMAXON, 2004). O valor do raio covalente de um átomo é considerado a metade da distância de ligação entre dois átomos idênticos (Figura 16) (UFMG, 2018).

[2] Valor usado para considerar pequenas variações nas posições dos átomos.

Figura 16 – Representação do raio covalente de um átomo (UFMG, 2018).



3.3 ISOMORFISMO DE GRAFOS

Após a criação do grafo, são geradas as m posições aleatórias e são criados os subgrafos correspondentes aos n vizinhos mais próximos. Os vizinhos de uma posição aleatória m são identificados como os átomos com a menor distância euclidiana em relação a posição m, levando em consideração as condições periódicas de fronteira. Após, é verificado se os subgrafos são isomorfos, sendo contabilizado o número de topologias diferentes encontradas, bem como o número de grafos incluídos em cada uma dessas topologias. Posteriormente, é calculada a entropia configuracional conforme o procedimento descrito no Capítulo 2.

O problema do isomorfismo de grafos pertence a uma parte de problemas em que a solução não é conhecida em tempo polinomial e nem faz parte dos problemas NP-Completos[3] (GAREY; JOHNSON, 1979). Assim, é candidato a fazer parte da classe de problemas NP-Intermediários[4] (LOZANO; RAGHAVAN, 1998). Entre os algoritmos usados para verificar se dois grafos são isomorfos, destacam-se os algoritmos de Babai e Luks (BABAI; LUKS, 1983), Babai (BABAI, 2015) e VF2 (CORDELLA et al., 2004). Para o desenvolvimento deste trabalho, inicialmente foi utilizado o método VF2.

3.4 MÉTODO VF2

O pacote *NetworkX* implementa o método VF2 (NETWORKX, 2018b), que possui complexidade $\theta(N^2)$ no melhor caso e $\theta(N!N)$ no pior, e complexidade de espaço $\theta(N)$

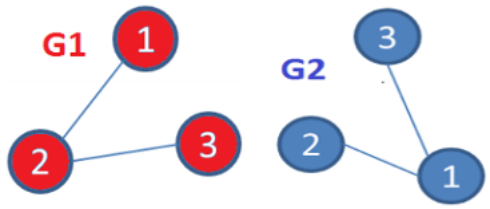
[3] Um problema está na classe NP-Completo se ele está em NP (sua solução é verificável em tempo polinomial) e é tão "difícil"quanto qualquer problema em NP (CORMEN. et al., 2012). Diz-se que um problema Y não é mais difícil que um problema X se qualquer algoritmo para X pode ser usado para resolver Y em três passos: 1) transforme a instância dada de Y, em tempo polinomial, numa instância de X. 2) submeta a instância de X ao algoritmo para X, que produz uma solução S. 3) transforme S, em tempo polinomial, numa solução da instância de Y (FEOFILOFF, 2018).

[4] Assumindo que $P \neq NP$, os problemas NP-Intermediários são os problemas P ou NP que não fazem parte dos problemas NP-Difíceis (JONSSONA; LAGERKVISTA; NORDHB, 2015).

em ambos os casos (CORDELLA et al., 2004). Esse utiliza uma técnica de *backtracking* [5], baseada em uma estrutura de árvore para mapeamentos (CORDELLA et al., 2004). O pseudocódigo do algoritmo pode ser visto no Anexo A.

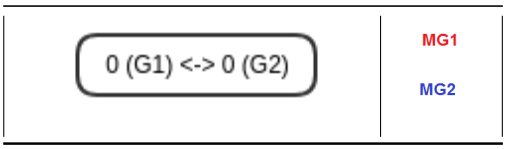
Na Figura 17, tem-se a representação de dois grafos (G1 e G2), os quais se deseja verificar se são isomorfos. Neste caso utiliza-se dois conjuntos, MG1 e MG2, para armazenar os mapeamentos de G1 e G2.

Figura 17 – Dois grafos não dirigidos com duas arestas cada. Adaptado de (MAKSOV; LI; BUTLER, 2015).



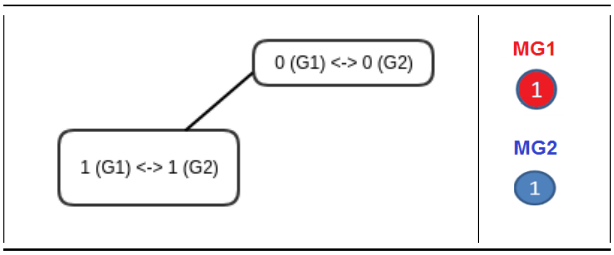
O método VF2 inicialmente compara os mapeamentos $MG1$ e $MG2$, que iniciam vazios, resultando em verdadeiro (Figura 18).

Figura 18 – Estado atual: $MG1 = \{\}$ e $MG2 = \{\}$ (Próprio, 2018).



Posteriormente, os mapeamentos $MG1$ e $MG2$ são comparados inserindo o nodo 1, sendo que o resultado neste caso também é verdadeiro (Figura 19).

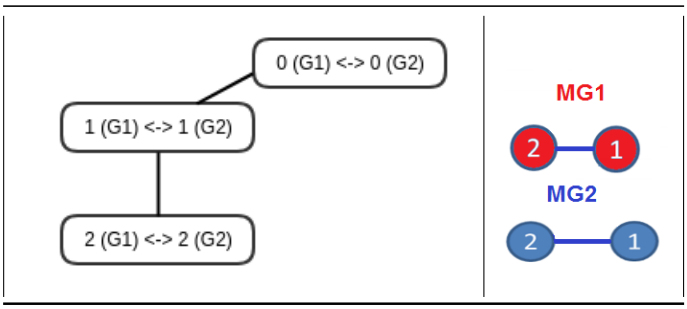
Figura 19 – Estado atual: $MG1 = \{1\}$ e $MG2 = \{1\}$ (Próprio, 2018).



[5] *Backtracking* é uma técnica de busca sistemática de todas as soluções, garantindo que uma solução será encontrada (COMBA, 2018).

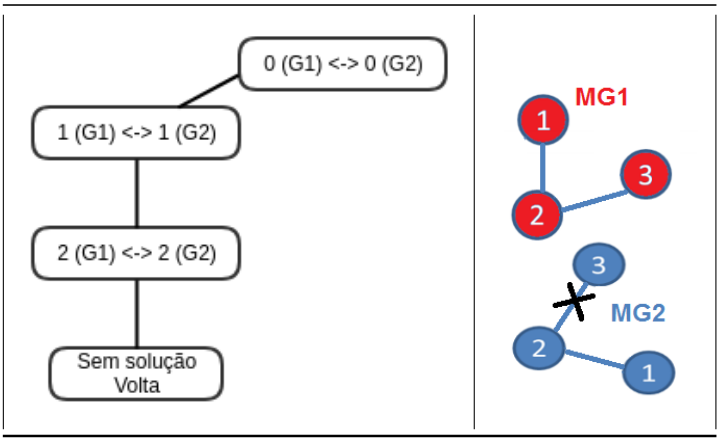
Após, é inserido o nodo 2 nos mapeamentos $MG1$ e $MG2$, e o algoritmo continua sua execução, uma vez que o subgrafo {1, 2} de G1 é isomorfo ao subgrafo {1, 2} de G2 (Figura 20).

Figura 20 – Estado atual: $MG1 = \{1, 2\}$ e $MG2 = \{1, 2\}$ (Próprio, 2018).



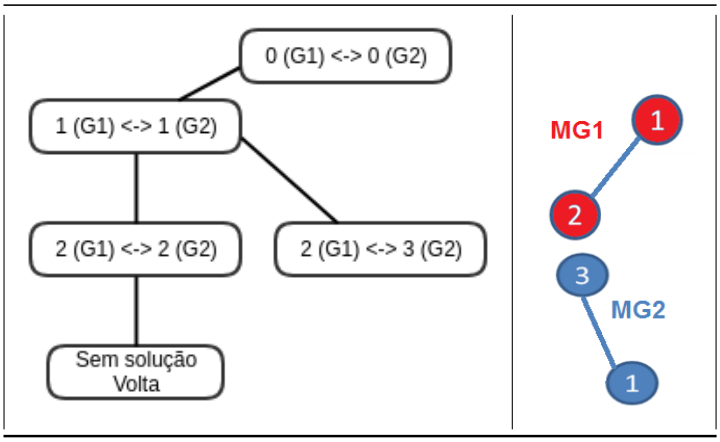
Após, os mapeamentos $MG1$ e $MG2$ são comparados inserindo, em ambos, o nodo 3. Essa inserção não é possível, uma vez que no grafo G1 o vértice 2 tem uma aresta com o vértice 3, mas no grafo G2 não existe essa ligação. Assim, o algoritmo efetua um retrocesso (*backtracking*) removendo o nodo 3 e o nodo 2 de ambos os mapeamentos (não há outras ligações nesse caminho) e volta a ter somente o vértice 1 em $MG1$ e $MG2$ (Figura 21).

Figura 21 – Estado atual: $MG1 = \{1\}$ e $MG2 = \{1\}$ (Próprio, 2018).



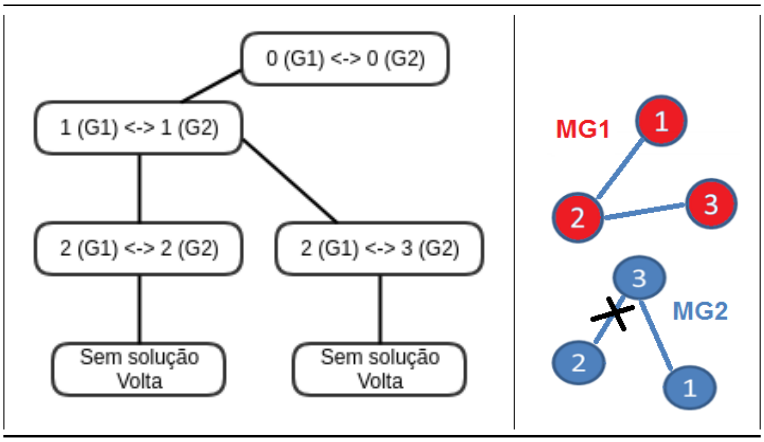
Então, os mapeamentos são comparados novamente, inserindo o nodo 2 no mapeamento $MG1$ e o nodo 3 no mapeamento $MG2$. O resultado é verdadeiro, pois o subgrafo {1, 2} do grafo G1 é isomorfo ao subgrafo {1, 3} do grafo G2 (Figura 22).

Figura 22 – Estado atual: $MG1 = \{1, 2\}$ e $MG2 = \{1, 3\}$ (Próprio, 2018).



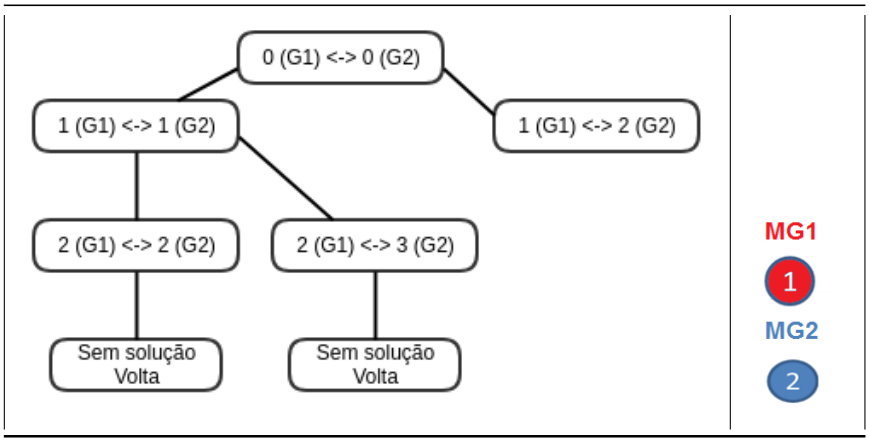
Posteriormente, são inseridos o nodo 3 no mapeamento $MG1$ e o nodo 2 no mapeamento $MG2$. Essa operação não é possível, uma vez que no grafo G1 o vértice 2 tem uma aresta com o vértice 3, porém no grafo G2 o vértice 3 não apresenta uma aresta com o vértice 2. Assim, é efetuado um *backtracking*, sendo removidos todos os nodos de $MG1$ e $MG2$, voltando para o estado inicial, onde ambos os mapeamentos estão vazios (Figura 23).

Figura 23 – Estado atual: $MG1 = \{\}$ e $MG2 = \{\}$ (Próprio, 2018).



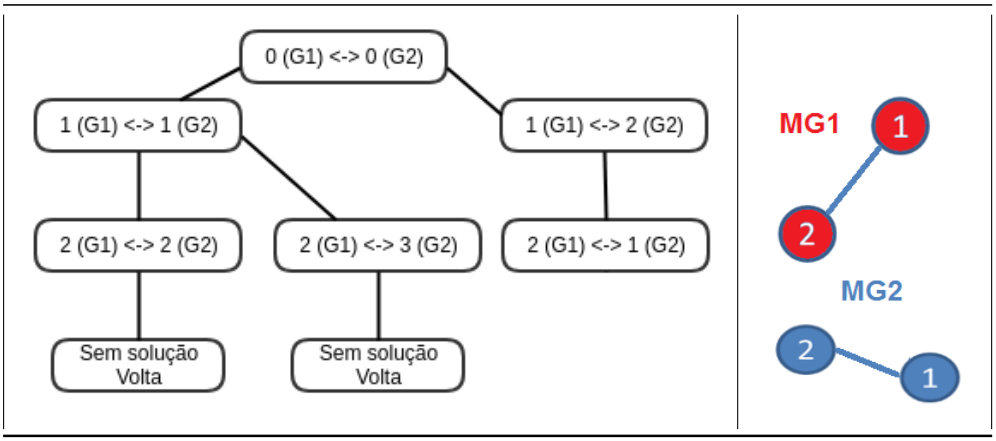
O algoritmo inicia novamente, agora inserindo o vértice 1 no mapeamento $MG1$ e o vértice 2 no mapeamento $MG2$ (Figura 24).

Figura 24 – Estado atual: $MG1 = \{1\}$ e $MG2 = \{2\}$ (Próprio, 2018).



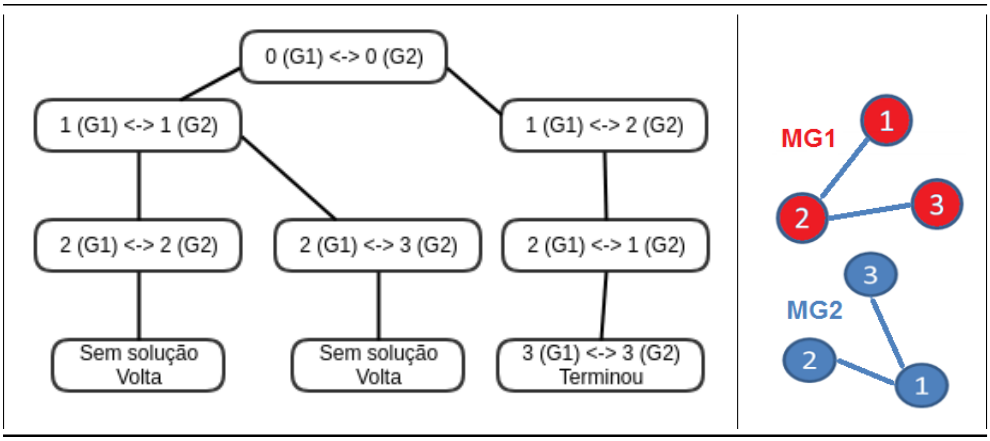
Após, os mapeamentos são comparados inserindo o nodo 2 no mapeamento $MG1$ e o nodo 1 no mapeamento $MG2$. O resultado é verdadeiro, uma vez que o subgrafo {1, 2} do grafo G1 é isomorfo ao subgrafo {2, 1} do grafo G2 (Figura 25).

Figura 25 – Estado atual: $MG1 = \{1, 2\}$ e $MG2 = \{2, 1\}$ (Próprio, 2018).



Por fim, é inserido o nodo 3 nos mapeamentos $MG1$ e $MG2$. O algoritmo termina com sucesso, pois o subgrafo {1, 2, 3} do grafo G1 é isomorfo ao subgrafo {2, 1, 3} do grafo G2, e não há mais nodos a serem percorridos. Desse modo, os mapeamentos encontrados entre os grafos G1 e G2 são $MG1 = \{1, 2, 3\}$ e $MG2 = \{2, 1, 3\}$, que podem ser representados pela função f, sendo $f(1) = 2$, $f(2) = 1$, $f(3) = 3$ (Figura 26).

Figura 26 – Estado atual: $MG1 = \{1, 2, 3\}$ e $MG2 = \{2, 1, 3\}$ (Próprio, 2018).



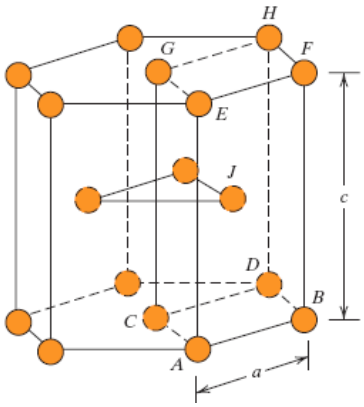
3.5 VALIDAÇÃO DA IMPLEMENTAÇÃO SEQUENCIAL

Para uma validação inicial da implementação foram utilizadas somente estruturas cristalinas, cuja entropia configuracional é igual a 0 (zero). Mais especificamente, foram utilizadas as estruturas cristalinas dos metais Titânio e Cobre. Para a validação com estruturas amorfas, foi realizado um contato com o pesquisador Richard Vink e esse disponibilizou as estruturas que foram utilizadas no artigo publicado em conjunto com o pesquisador Gerard Barkema (VINK; BARKEMA, 2002). Porém, a menor estrutura disponibilizada possui 3000 átomos, tornando inviável a sua execução de forma sequencial. Desta forma, a validação com essas estruturas foi feita posteriormente, quando foram realizadas otimizações no programa, bem como a sua paralelização. As otimizações realizadas e o processo de paralelização serão descritos no Capítulo 4.

3.5.1 Titânio

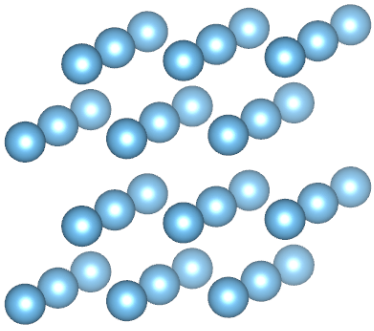
O Titânio é um metal que possui uma célula unitária hexagonal compacta *HCP* (*Hexagonal Close Packed*) (Figura 27).

Figura 27 – Representação da estrutura de um *HCP* (JR; RETHWISCH, 2018).



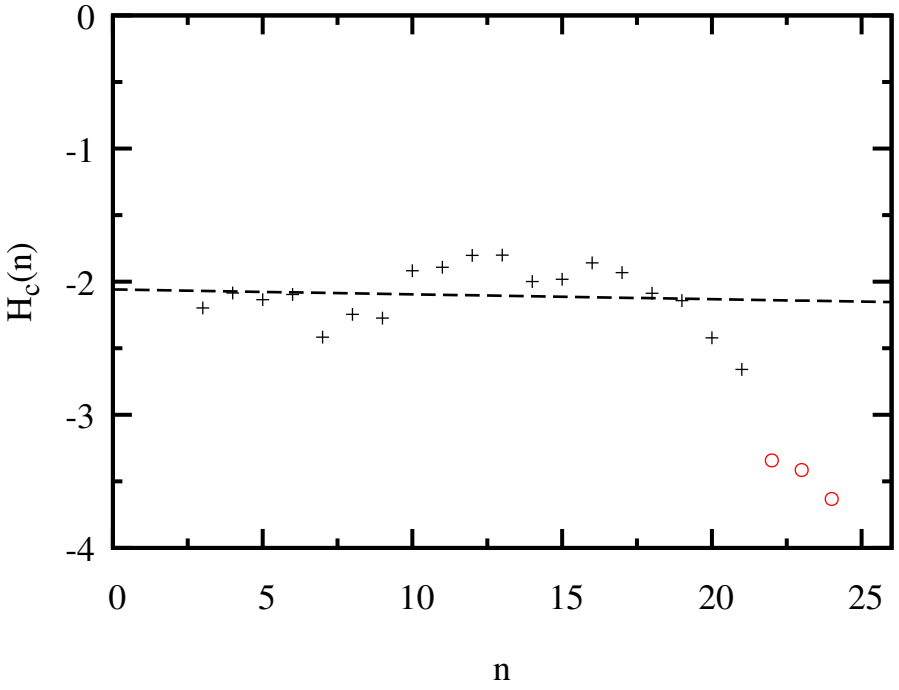
Para a validação, foi gerada uma estrutura cristalina de Titânio através do pacote *ASE*, com 36 átomos e condições periódicas de fronteira nos eixos x e y. Na Figura 28, tem-se uma representação dessa estrutura. Para validação na implementação foram utilizadas um número de posições aleatórias $m = n^2N$; *cutoff* do raio covalente = 1,12x e $c = 0$.

Figura 28 – Representação da estrutura hexagonal compacta de Titânio com 36 átomos (Próprio, 2018).



Neste caso, foi obtida uma entropia configuracional por átomo de aproximadamente 0 k_B, conforme pode ser observado na Figura 29, onde as cruzes correspondem ao valor de $H_c(n)$ e a inclinação da linha tracejada, a entropia configuracional (os valores circulados em vermelho foram desconsiderados no ajuste de reta). Esse resultado é condizente com a condição dos materiais cristalinos, onde a entropia configuracional é zero.

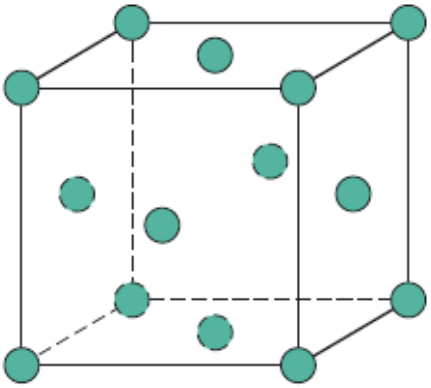
Figura 29 – Gráfico obtido para $H_c(n)$ na estrutura de Titânio utilizada. A entropia configuracional por átomo, em unidades k_B, é representada pela inclinação da linha tracejada, que inicia em $n = 3$ até $n = 21$. Os valores circulados em vermelho foram desconsiderados no cálculo do ajuste de reta (Próprio, 2018).



3.5.2 Cobre

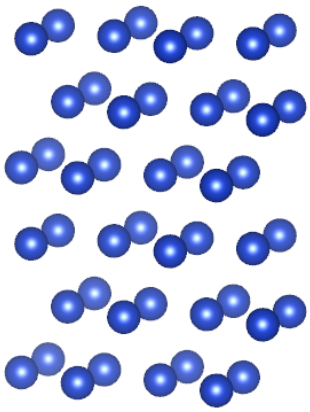
O Cobre possui uma célula unitária cúbica *FCC* (*Face Centered Cubic*), na qual os átomos estão localizados em cada um dos vértices e nos centros de todas as faces do cubo. Outros metais que possuem essa estrutura são o Alumínio, a Prata e o Ouro (JR; RETHWISCH, 2018). Um exemplo de uma estrutura *FCC* pode ser observado na Figura 30.

Figura 30 – Representação da estrutura de um *FCC* (JR; RETHWISCH, 2018).



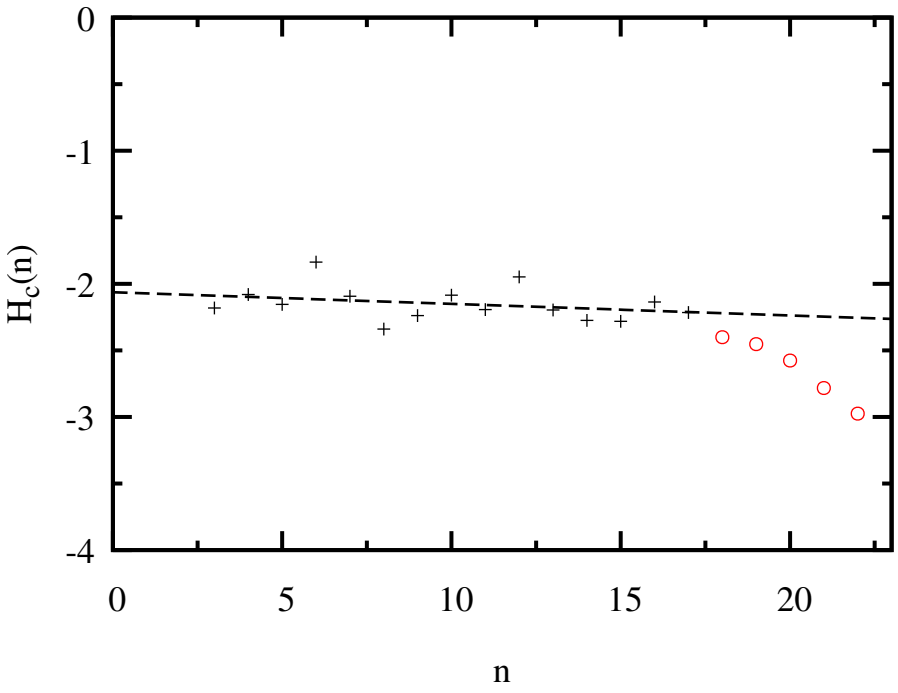
Neste caso, foi gerada uma estrutura com 48 átomos e condições periódicas de fronteira nos eixos x e y. Na Figura 31, pode-se ver uma representação dessa estrutura. Os parâmetros utilizados foram os mesmos que foram usados para o cálculo da entropia configuracional da estrutura de Titânio ($m = n^2N$; *cutoff* do raio covalente = 1,12x e $c = 0$).

Figura 31 – Representação de uma estrutura *FCC* com 48 átomos de Cobre (Próprio, 2018).



A entropia configuracional calculada foi aproximadamente 0 k_B. Na Figura 32, tem-se os valores de $H_c(n)$, sendo que a inclinação da linha tracejada corresponde à entropia configuracional, que nesse caso é igual a 0 (zero). Os valores circulados em vermelho foram desconsiderados no ajuste de reta.

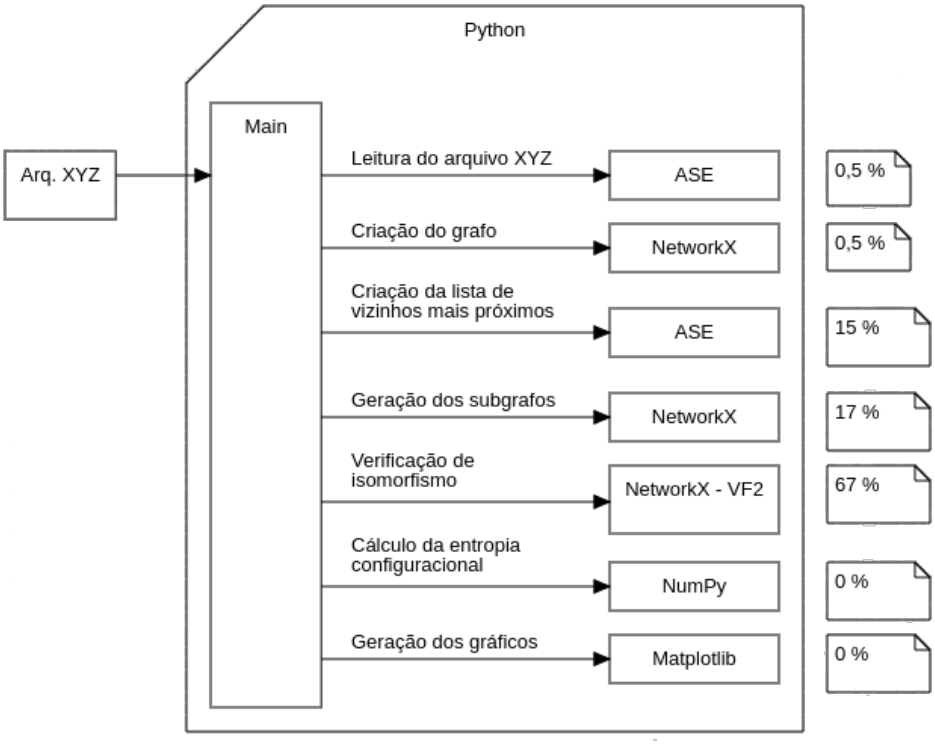
Figura 32 – Gráfico obtido para $H_c(n)$ na estrutura de Cobre utilizada. A entropia configuracional por átomo, em unidades k_B, é representada pela inclinação da linha tracejada, que inicia em $n = 3$ até $n = 17$. Os valores circulados em vermelho foram desconsiderados no cálculo do ajuste de reta (Próprio, 2018).



4 OTIMIZAÇÕES E PARALELIZAÇÃO

A primeira versão foi desenvolvida de forma sequencial na linguagem *Python*, sendo utilizadas as bibliotecas *ASE* (ASE, 2018), *Matplotlib* (MATPLOTLIB, 2018), *NumPy* (NUMPY, 2018) e *NetworkX* (NETWORKX, 2018a). O Fluxograma da Figura 33 apresenta a estrutura da implementação desenvolvida e as bibliotecas utilizadas, bem como a porcentagem de tempo que cada etapa apresentou. Para o perfilamento foi utilizado como entrada uma estrutura de Cobre *FCC* com 48 átomos e o perfilador *cProfile* (PYTHON, 2018b). Todos as execuções deste Capítulo, a menos onde for explicitado o contrário, foram realizadas utilizando um *notebook* com as seguintes especificações: processador Intel Core i5 8250U com 4 *cores*, 8 *threads*, 1.60 GHz de frequência e 6 MB de memória *cache* L3; memória *RAM* com 8 GB DDR4 e 2133 MHz de frequência; sistema operacional Fedora 28 e Kernel 4.18.12-200. Foi usado o *Python* 2.7.15 e o *g++* 8.2.1.

Figura 33 – Fluxograma que mostra a estrutura da implementação desenvolvida e os resultados do perfilamento da aplicação (Próprio, 2018).

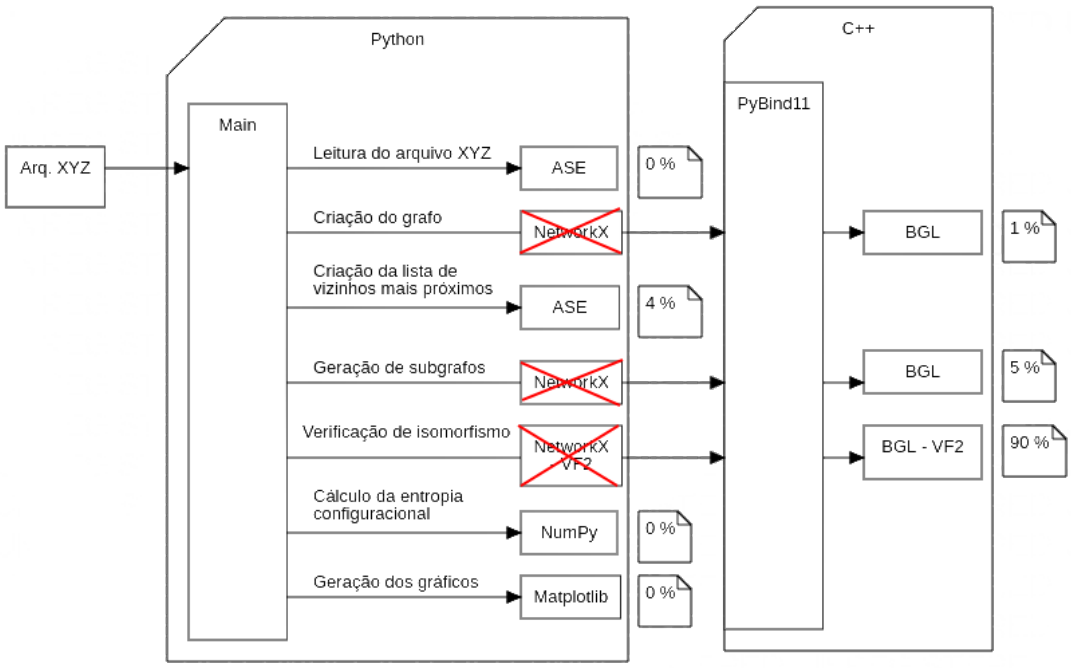


Observa-se na Figura 33 que o maior tempo encontra-se na verificação de isomorfismo. De fato, somente a função que verifica se dois grafos são isomorfos representa 67% do tempo total de execução. É importante destacar que o tempo de execução cresce conforme o aumento do número de átomos e do número de posições aleatórias (m), uma vez que esses fatores influenciam no número de subgrafos e, consequentemente, nas verificações de isomorfismo. O tipo de estrutura utilizada também influencia significativamente no tempo de execução, já que em estruturas amorfas tem-se um número maior de subgrafos diferentes, e assim, um número maior de verificações de isomorfismo.

O tempo de execução, utilizando como entrada uma estrutura de Cobre *FCC* com 48 átomos, com um número de vizinhos n variando de 3 até 25 e um número de posições aleatórias m igual a n^2N, sendo N o número total de átomos, foi de 71 minutos. Todas as execuções mostradas neste Capítulo, a não ser quando explicitadas ao contrário, foram realizadas utilizando esses valores para os parâmetros n e m. Com uma estrutura de Cobre *FCC* com 3000 átomos (tamanho da menor estrutura disponibilizada por Vink), estima-se que o tempo de execução seria de aproximadamente 89 horas.

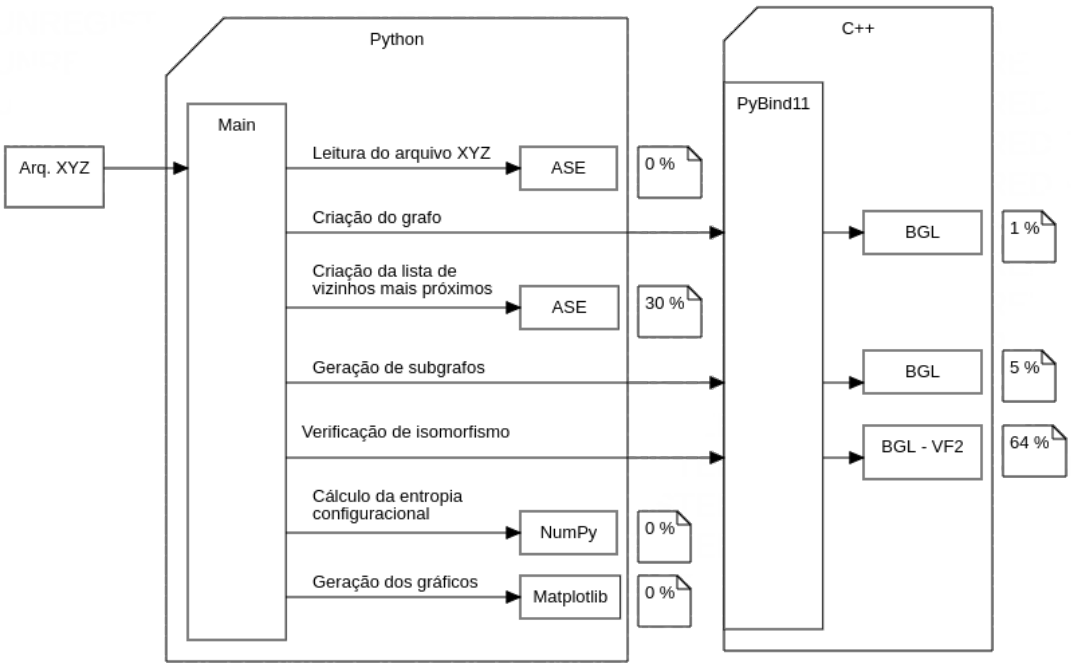
Desta forma, foi desenvolvida uma biblioteca na linguagem *C++* como uma tentativa de reduzir o tempo de execução da função de verificação de isomorfismo. A primeira versão da implementação *C++* foi realizada utilizando as bibliotecas: *PyBind11* (JAKOB, 2017), que é utilizada para realizar a comunicação entre as linguagens *Python* e *C++*; e a biblioteca *Boost* (DAWES; ABRAHAMS; RIVERA, 2007), mais especificamente a *BGL* (*Boost Graph Library*) (SIEK; LEE; LUMSDAINE, 2001), para a representação dos grafos e a verificação de isomorfismo. A biblioteca *BGL* também utiliza o método VF2, sendo assim, essa otimização foi uma simples substituição da biblioteca *NetworkX* do *Python* para a biblioteca *BGL* em *C++*. O Fluxograma da Figura 34 apresenta as mudanças realizadas na estrutura do programa, além da porcentagem de tempo que cada etapa consumiu utilizando-se como entrada uma estrutura de Cobre *FCC* com 48 átomos.

Figura 34 – Fluxograma que mostra a estrutura da implementação e os resultados do perfilamento da aplicação após a substituição do pacote *NetworkX* (*Python*) pela biblioteca *BGL* (*C++*) (Próprio, 2018).



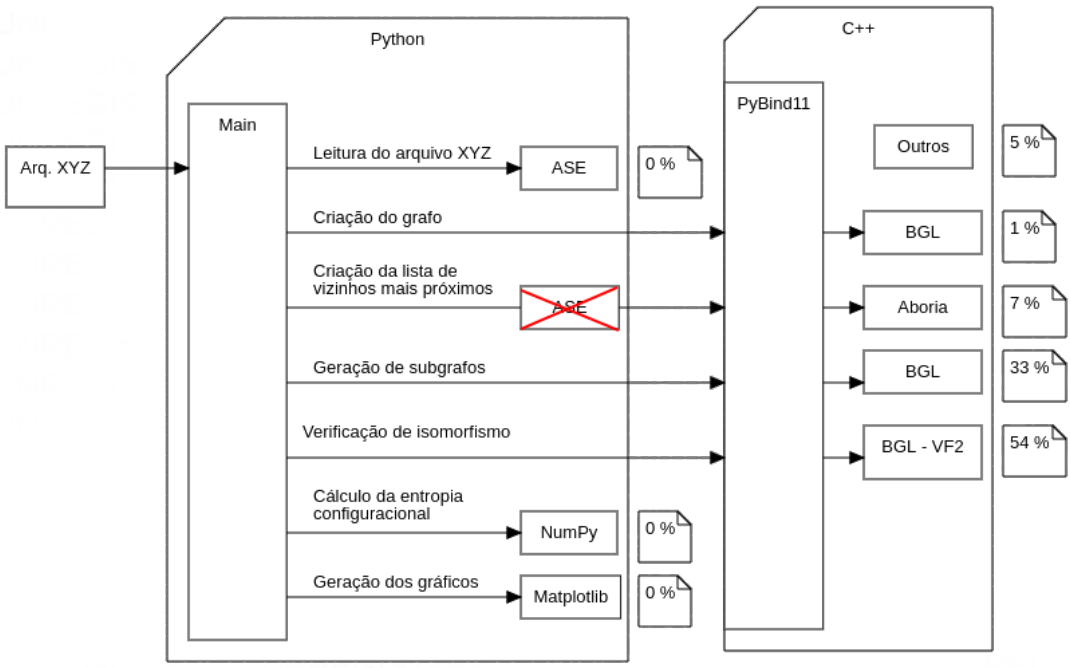
As mudanças realizadas resultaram em resultados inferiores, aumentando em torno de 4 vezes o tempo de execução. Esse aumento do tempo de execução deve-se, principalmente, ao alto *overhead* causado pelas inúmeras comunicações entre as linguagens *C++* e *Python*. De forma a reduzir esse *overhead*, outras partes do código foram reimplementadas na linguagem *C++*. Mais especificamente, as funções de geração de subgrafos e as verificações de isomorfismo foram reimplementadas em *C++*. A estrutura da implementação, junto com os novos percentuais obtidos a partir do perfilamento de uma execução, utilizando como entrada uma estrutura de Cobre *FCC* com 48 átomos, podem ser observados na Figura 35. Neste caso, obteve-se uma diminuição de tempo de execução, para uma estrutura de Cobre *FCC* com 48 átomos, de cerca de 71 minutos para 6 minutos, o que representa uma redução de aproximadamente 91%.

Figura 35 – Fluxograma que mostra a estrutura da implementação e os resultados do perfilamento da aplicação após a reimplementação em *C++* das funções de geração de subgrafos e de verificação de isomorfismo (Próprio, 2018).



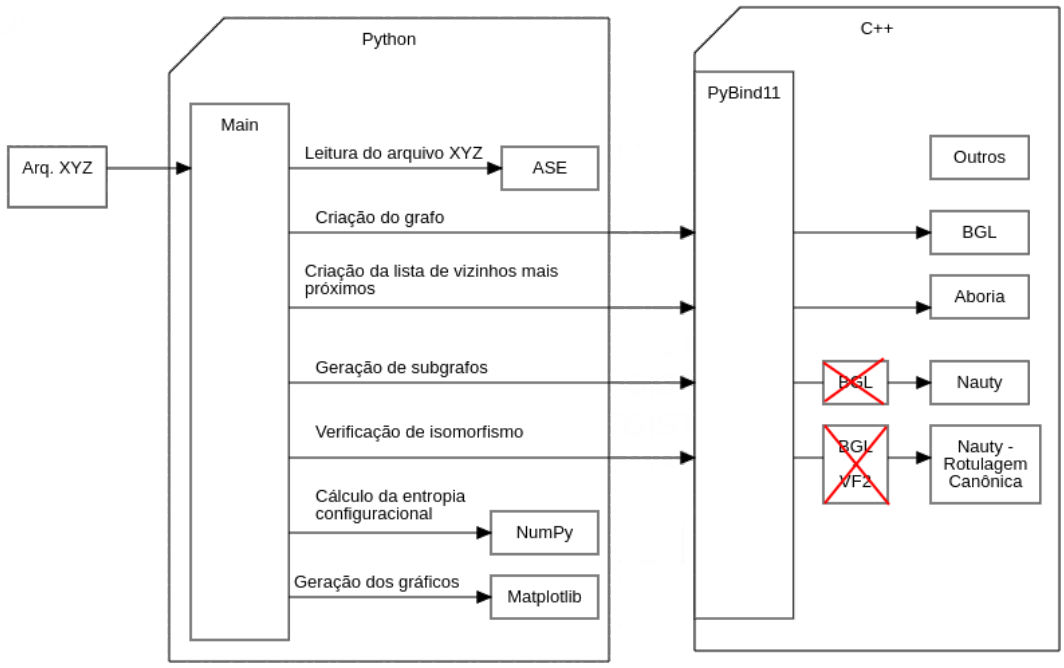
Apesar da redução de tempo de execução para estruturas maiores, o tempo de execução ainda pode ser considerado alto. De fato, para uma estrutura de Cobre *FCC* com 108 átomos, o tempo de execução foi de cerca de 9 minutos. Conforme pode ser observado na Figura 35, a função de verificação de isomorfismo continua responsável por cerca de 64% do tempo de execução. Porém, foi observado que a função que obtém a lista de vizinhos de uma posição aleatória é responsável por 30% do tempo total. Desse modo, a função de busca dos vizinhos foi reimplementada na linguagem *C++*. A solução escolhida foi o uso da biblioteca *Aboria* (ROBINSON, 2018), que é uma biblioteca *C++* criada para tratamento de partículas e utiliza estruturas de dados como *K-d Trees*, *Octrees*, *Cell Lists*, entre outras. Porém, essa ainda está em desenvolvimento, e o uso de células não cúbicas e não ortorrômbicas, como por exemplo, uma estrutura *HCP*, ainda não é suportado. Desta forma, esse tipo de estrutura não é suportada na nova versão. Com essa mudança, obteve-se uma diminuição de tempo de execução, para uma estrutura de Cobre *FCC* com 108 átomos, de cerca de 9 minutos para 26 segundos, o que representa uma redução de, aproximadamente, 95% no tempo de execução. A estrutura da implementação, junto com os percentuais encontrados em um perfilamento de uma estrutura de Cobre *FCC* com 256 átomos podem ser observados na Figura 36.

Figura 36 – Fluxograma que mostra a estrutura da implementação e os resultados do perfilamento da aplicação após a substituição do pacote *ASE* (*Python*) pela biblioteca *Aboria* (*C++*) (Próprio, 2018).



Como pode ser observado na Figura 36, a função de isomorfismo representa cerca de 54 % tempo de execução. Desta forma, optou-se pela utilização da biblioteca *Nauty* (MCKAY; PIPERNO, 2014), que é conhecida como sendo uma das bibliotecas com melhor performance no cálculo de isomorfismo de grafos (CORDELLA et al., 2004). A estrutura nova da implementação pode ser observada na Figura 37.

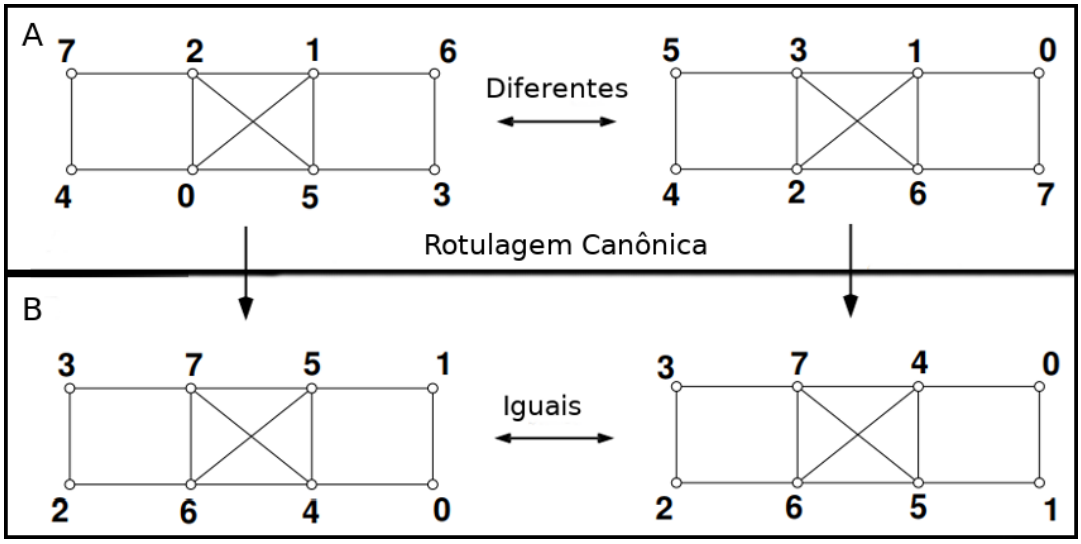
Figura 37 – Fluxograma que mostra a estrutura da implementação após a substituição da biblioteca *BGL* pela biblioteca *Nauty* e do método VF2 pela rotulagem canônica (Próprio, 2018).



A biblioteca *Nauty* utiliza um método de rotulagem canônica (*Canonical Labelling*), que é uma técnica onde se altera os rótulos dos vértices de um grafo de uma maneira que não dependa da posição em que estavam anteriormente. Grafos que são isomorfos (ou seja, iguais, porém, sem levar em conta os rótulos dos vértices) se tornam idênticos depois da rotulagem canônica (MCKAY; PIPERNO, 2018).

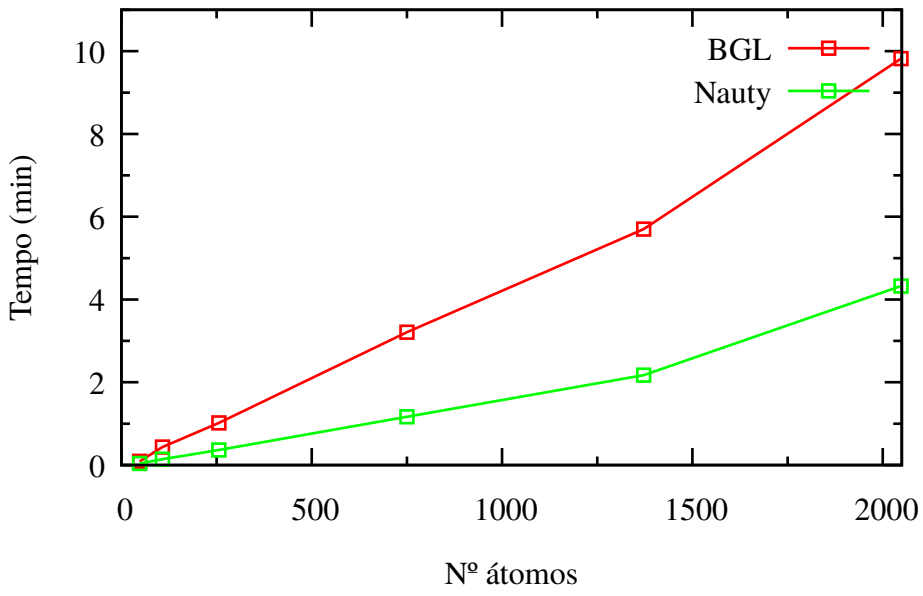
Na Figura 38a, por exemplo, os dois grafos são isomorfos, apesar de não serem idênticos (por exemplo, os vértices 0 e 4 são adjacentes no grafo da esquerda mas não no da direita). Entretanto, após a rotulagem canônica, são produzidos os grafos da Figura 38b, e os resultados são idênticos (os vértices 4, 5, 0 e 1 tem as mesmas arestas adjacentes, apesar de estarem posicionados de forma diferente nas duas imagens). Isso torna a verificação de isomorfismo mais eficiente, uma vez que ambos os grafos acabam sendo representados da mesma forma.

Figura 38 – Representação de dois grafos isomorfos, antes e depois da rotulagem canônica (Adaptado de (MCKAY; PIPERNO, 2018)).



Conforme pode ser visto no gráfico da Figura 39, o uso da biblioteca *Nauty* provocou uma redução significativa no tempo de execução. De fato, houve uma redução de 44% para uma estrutura de Cobre *FCC* de 2048 átomos, passando de quase 10 minutos para menos de 4 minutos e meio.

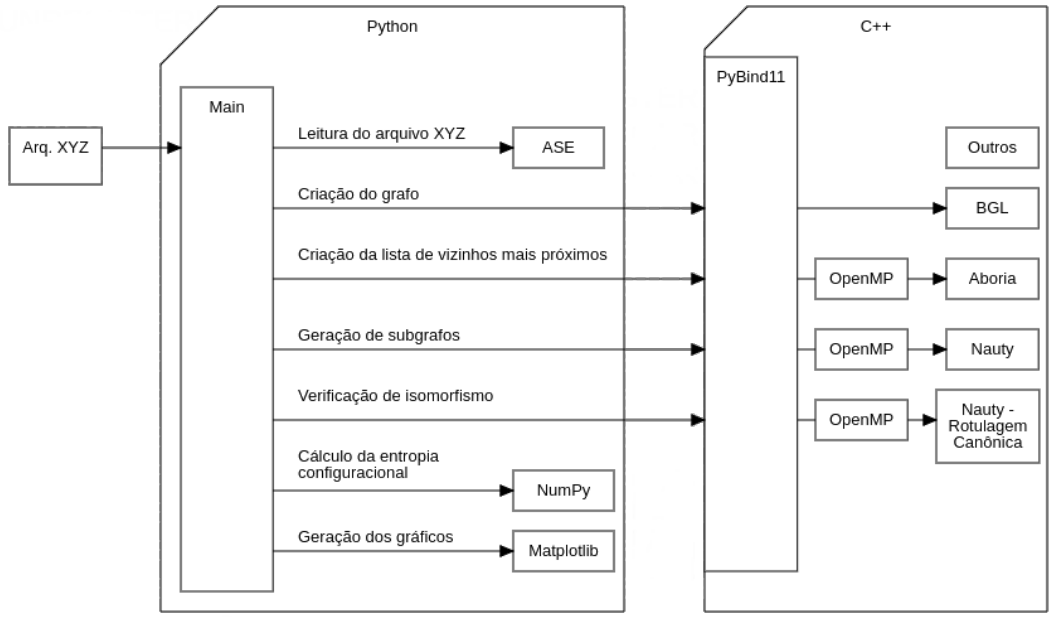
Figura 39 – Comparação do tempo de execução da aplicação usando as bibliotecas de grafos *BGL* e *Nauty*, utilizando como entrada estruturas de Cobre *FCC* com $m = n^2N$ e n de 3 a 25 (Próprio, 2018).



4.1 PARALELIZAÇÃO DA IMPLEMENTAÇÃO

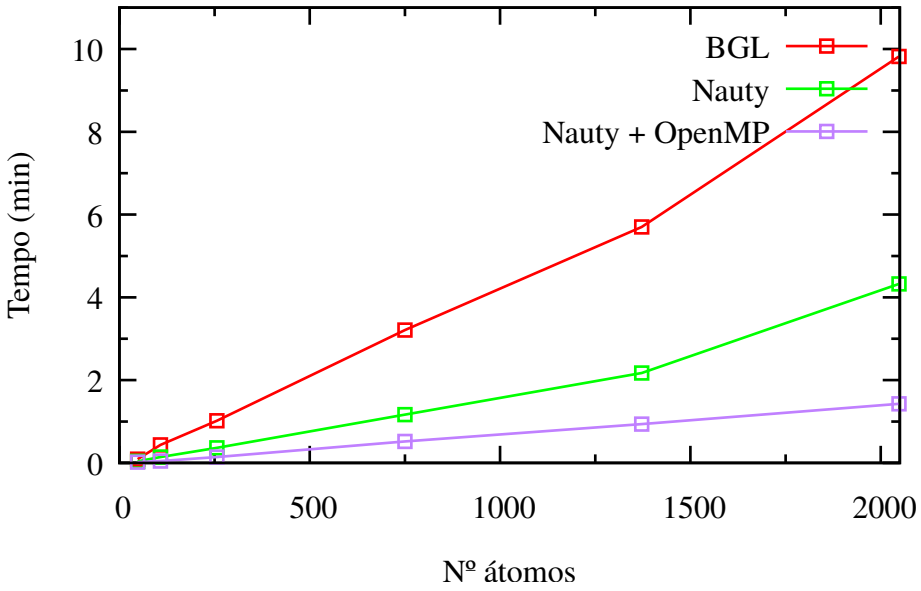
Por fim, de modo a tirar proveito dos recursos de processadores *multi-core*, foi feita a paralelização de algumas partes do programa, sendo elas: busca de vizinhos mais próximos; geração de subgrafos e os correspondentes rótulos canônicos e as verificações de isomorfismo. Para realizar a paralelização da implementação foi utilizada a *API* (*Application Programming Interface*) *OpenMP* (OPENMP, 2018). A estrutura final da implementação pode ser encontrada na Figura 40. O código fonte da implementação final pode ser encontrado no endereço <https://github.com/hlscalon/EntropiaConfiguracional>.

Figura 40 – Fluxograma que mostra a estrutura da versão final, após a utilização da *API OpenMP* (Próprio, 2018).



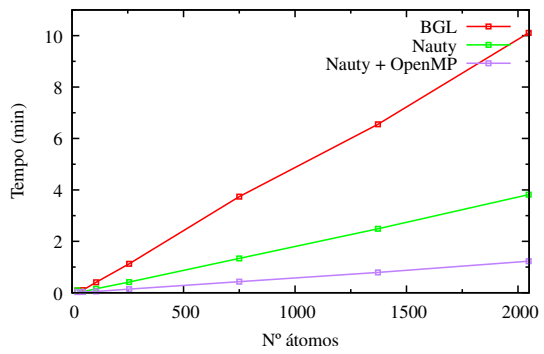
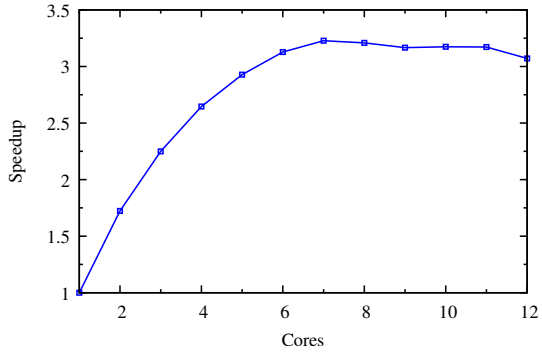
Na Figura 41, tem-se os tempos de execução utilizando a biblioteca *BGL*, a biblioteca *Nauty* e a implementação paralelizada utilizando *OpenMP*.

Figura 41 – Comparação do tempo de execução da aplicação usando a biblioteca *BGL*, a biblioteca *Nauty* e *OpenMP*, utilizando como entrada estruturas de Cobre *FCC* com $m = n^2N$ e n de 3 a 25 (Próprio, 2018).



Na Figura 42a tem-se o mesmo gráfico da Figura 41, porém utilizando um *cluster* com as seguintes especificações: processador Intel Xeon X5660 com 12 *cores*, 2.80GHz de frequência e 12 MB de memória *cache* L3; sistema operacional Fedora 28 e Kernel 4.16.2-300. Já na Figura 42b, tem-se o *Speedup* [1] obtido com a paralelização.

Figura 42 – Na esquerda, tem-se os tempos médios da implementação utilizando a biblioteca *BGL*, a biblioteca *Nauty* e *OpenMP*, em minutos, utilizando como entrada estruturas de Cobre *FCC* com $m = n^2N$ e n de 3 a 25. Já na direita, tem-se o gráfico com o *Speedup* obtido com a paralelização (Próprio, 2018).

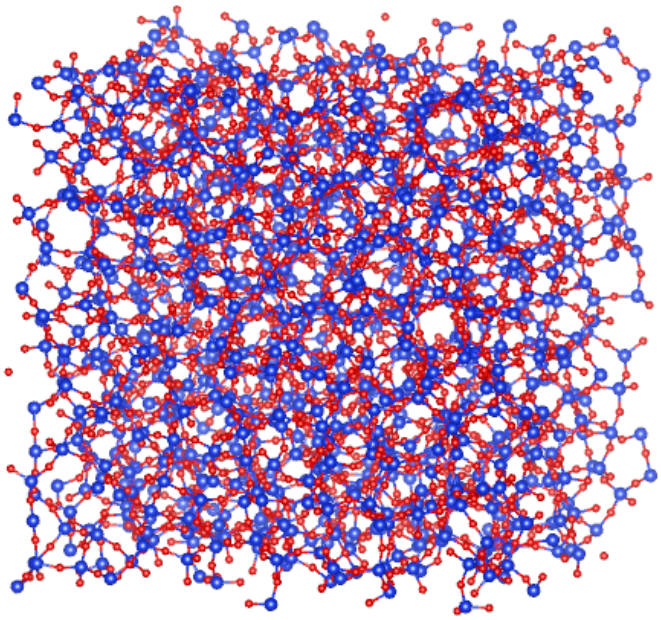
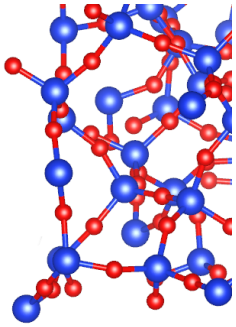
Figura 42a)	Figura 42b)
	

[1] *Speedup* é a razão entre o tempo de execução no programa sequencial e no programa paralelo (COSTA, 2018).

4.2 VALIDAÇÃO DA IMPLEMENTAÇÃO

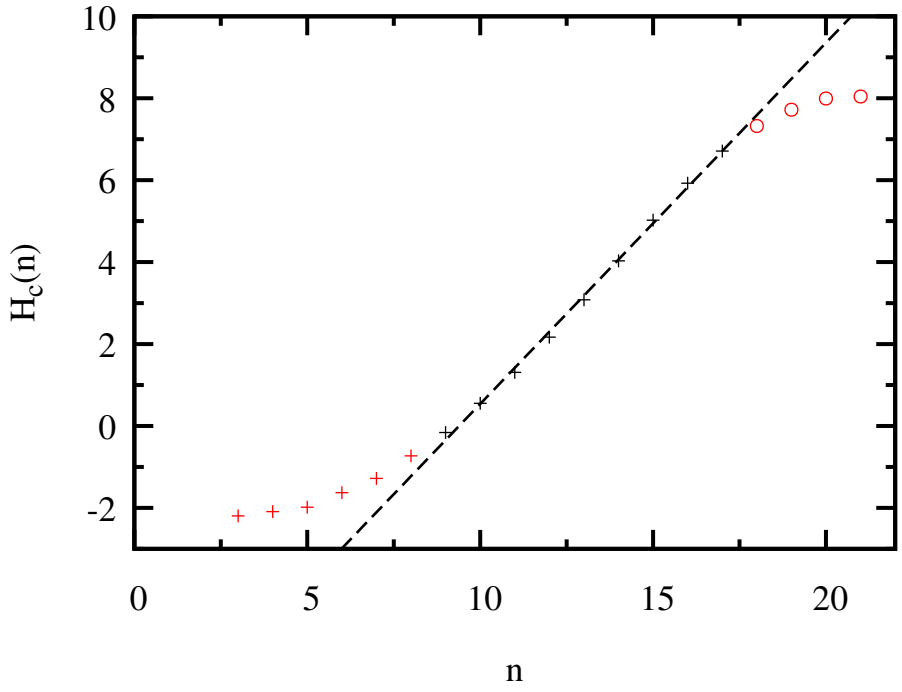
Para a validação final da implementação, foi utilizada a estrutura amorfa que foi disponibilizada pelo pesquisador Richard Vink, e foi utilizada no artigo (VINK; BARKEMA, 2002). Essa é uma estrutura cúbica com 3000 átomos, sendo 1000 átomos de Silício e 2000 de Oxigênio. Nessa estrutura, cada átomo de Silício está conectado a 4 átomos de Oxigênio e cada átomo de Oxigênio está conectado a 2 átomos de Silício. Na Figura 43a, tem-se a representação da estrutura e na Figura 43b um detalhe mostrando as conexões entre os átomos.

Figura 43 – Representação da estrutura amorfa com 1000 átomos de Silício (átomos azuis) e 2000 de Oxigênio (átomos vermelhos) na esquerda e um detalhe mostrando as conexões entre os átomos na direita (Próprio, 2018).

Figura 43a)	Figura 43b)
	

Para estimar a entropia configuracional por átomo de Silício, foi utilizado o mesmo procedimento que foi adotado por Vink (VINK; BARKEMA, 2002), onde cada átomo de Oxigênio (Si-O-Si), foi substituído por uma única aresta ligando dois átomos de Silício (Si-Si). Para a validação, foram utilizados $m = 5n^2N$; *cutoff* do raio covalente = 1,12x e $c = 90$. Com essa configuração, foi obtida a entropia configuracional por átomo de 0,88 k_b, que é idêntico ao resultado obtido por Vink (VINK; BARKEMA, 2002). Na Figura 44, tem-se o gráfico que apresenta os valores calculados para $H_c(n)$ e a inclinação da linha tracejada, a entropia configuracional. Os valores circulados em vermelho foram desconsiderados no cálculo do ajuste de reta, pois ultrapassam o limite de 1% de $H(n)$.

Figura 44 – Gráfico obtido para $H_c(n)$ na estrutura de Silício. A entropia configuracional por átomo, em unidades k_b, é representada pela inclinação da linha tracejada, que inicia em $n = 9$ até $n = 17$, sendo o seu valor igual a 0,88 k_b. Os valores circulados em vermelho foram desconsiderados no cálculo do ajuste de reta, pois ultrapassam o limite de 1% de $H(n)$ (Próprio, 2018).



5 CONCLUSÃO

Esse trabalho apresentou uma implementação do método proposto por Vink e Barkema (VINK; BARKEMA, 2002) para o cálculo da entropia configuracional. Esse método pode ser usado para estruturas cristalinas e amorfas, desde que as posições dos átomos e as suas ligações sejam conhecidas.

Para desenvolvimento da implementação foi utilizada a linguagem de programação *Python* (PYTHON, 2018a), que é uma linguagem com uma grande comunidade envolvida e inúmeros pacotes de terceiros. Mais especificamente, foram utilizados os pacotes: *ASE* (ASE, 2018), que foi utilizado para a leitura e o processamento dos arquivos *Extended XYZ* contendo as estruturas dos materiais utilizados, além de facilitar a construção dos grafos; *NetworkX* (NETWORKX, 2018a), que foi utilizado na representação dos grafos e na verificação de isomorfismo de grafos; *Matplotlib* (MATPLOTLIB, 2018), que foi utilizado na criação dos gráficos e *NumPy* (NUMPY, 2018), que foi utilizado no ajuste da reta para estimar a entropia configuracional.

Para a validação inicial da implementação foram utilizadas as estruturas cristalinas do Cobre e do Titânio, sendo a primeira com 48 átomos e a segunda com 36 átomos. Em ambos os casos, a entropia configuracional encontrada foi igual a 0 (zero), que corresponde ao valor esperado para a entropia configuracional em materiais cristalinos. Essa implementação se mostrou inadequada para trabalhar com estruturas maiores, uma vez que apresentou um alto tempo de execução para estruturas com um elevado número de átomos.

De modo a trabalhar com estruturas maiores, foi criada uma biblioteca na linguagem *C++* para ser usada em conjunto com a implementação em *Python*. Inicialmente, foi feita uma simples substituição da biblioteca *NetworkX* (NETWORKX, 2018a) do *Python* para a biblioteca *BGL* (SIEK; LEE; LUMSDAINE, 2001) do *C++*. Essa alteração resultou em um aumento de cerca de 4 vezes no tempo de execução, devido ao alto *overhead* causado pelas inúmeras comunicações entre as linguagens *C++* e *Python*. De modo a diminuir esse *overhead*, as funções de geração de subgrafos e de verificação de isomorfismo foram reimplementadas em *C++*. Com essa alteração, foi obtida uma melhora de cerca de 91% frente à implementação feita somente em *Python*. De fato, o tempo de execução para uma estrutura de Cobre *FCC* com 48 átomos reduziu de cerca de 71 minutos para 6 minutos.

Porém, o tempo de execução ainda era insatisfatório, já que, por exemplo, a execução utilizando como entrada uma estrutura de Cobre *FCC* com 108 átomos apresentou um tempo de execução de cerca de 9 minutos. Observou-se que a função de criação da lista de vizinhos ocupava cerca de 30% do tempo de execução, então, essa também foi reimplementada em *C++*, utilizando a biblioteca *Aboria* (ROBINSON, 2018). Com essa mudança, obteve-se uma redução de tempo de execução de cerca de 9 minutos para 26

segundos. Poŕem, como essa biblioteca ainda está em desenvolvimento, só são suportadas estruturas com células ortorrômbicas ou cúbicas. Assim, a estrutura *HCP*, por exemplo, não é suportada na versão atual da implementação desenvolvida.

Entretanto, a função de verificação de isomorfismo ainda ocupava cerca de 54% do tempo de execução. Desse modo, a biblioteca *BGL* e o método VF2 foram substituídos pela biblioteca *Nauty* e pelo método da rotulagem canônica. Com essa modificação, obteve-se uma redução do tempo de execução de cerca de 10 minutos para menos de 4 minutos e meio, considerando uma estrutura de Cobre *FCC* com 2048 átomos.

Por fim, foi feita a paralelização de algumas funções. Mais especificamente, foram paralelizadas as funções para criação da lista de vizinhos, geração de subgrafos e verificação de isomorfismo, utilizando a *API OpenMP* (OPENMP, 2018). Com isso, obteve-se um *Speedup* de 3,1x usando 12 *cores*.

Para realizar a validação final da implementação, foi utilizada uma estrutura com 3000 átomos que foi disponibilizada por Vink (VINK; BARKEMA, 2002), sendo formada por 1000 átomos de Silício e 2000 de Oxigênio. A entropia configuracional calculada foi de 0,88 k_b, que é um resultado idêntico ao obtido por Vink (VINK; BARKEMA, 2002).

5.1 TRABALHOS FUTUROS

Várias melhorias podem ser realizadas na implementação, tais como:

- Adicionar suporte a estruturas com células não ortorrômbicas e não cúbicas.
- Verificar se alguma biblioteca derivada do *Nauty* tenha desempenho superior para os grafos gerados nesta implementação e usá-la, se for o caso.
- Melhorar a paralelização para criar menos *threads* e diminuir dependências.
- Verificar a possibilidade de implementar algumas funções em GPU.
- Adicionar um parâmetro na inicialização do *software* que defina a função que calcula o número total de posições aleatórias (m), já que hoje encontra-se codificada.
- Adicionar testes unitários.
- Adicionar documentação.

REFERÊNCIAS

ADAM, G.; GIBBS, J. H. On the Temperature Dependence of Cooperative Relaxation Properties in Glass-Forming Liquids. *The Journal of Chemical Physics*, American Institute of Physics, v. 43, n. 1, p. 139–146, 1965.

ASE. *Atomic Simulation Environment*. 2018. Disponível em: <https://wiki.fysik.dtu.dk/ase/>. Acesso em: 18 abr. 2018.

BABAI, L. Graph Isomorphism in Quasipolynomial Time. *CoRR*, abs/1512.03547, 2015.

BABAI, L.; LUKS, E. M. Canonical Labeling of Graphs. ACM, New York, NY, USA, p. 171–183, 1983.

BENGUIGUI, L. The different paths to entropy. *European Journal of Physics*, v. 34, p. 303, mar. 2013.

CHEMAXON. *XYZ format in Marvin*. 2004. Disponível em: <https://chemaxon.com/marvin-archive/3.3.3/marvin/doc/user/xyz-doc.html>. Acesso em: 09 jun. 2018.

COMBA, J. L. D. *Backtracking*. 2018. Instituto de Informática, UFRGS (Universidade Federal do Rio Grande do Sul), Porto Alegre, Brasil. Disponível em: <http://www.inf.ufrgs.br/~comba/inf1056-files/class07.pdf>. Acesso em: 05 maio 2018.

CORDELLA, L. P. et al. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on pattern analysis and machine intelligence*, v. 26, n. 10, p. 1367–1372, 2004.

CORMEN., T. H. et al. *Algoritmos - Teoria e prática. Tradução de Arlete Simille Marques*. 3. ed. Rio de Janeiro: Elsevier, 2012.

COSTA, L. H. M. K. *Parte 6 Otimizações da Arquitetura*. 2018. GTA (Grupo de Teleinformática e Automação), UFRJ (Universidade Federal do Rio de Janeiro), Rio de Janeiro, Brasil. Disponível em: <https://www.gta.ufrj.br/ensino/EEL580/apresentacoes/Parte6.pdf>. Acesso em: 10 nov. 2018.

DAWES, B.; ABRAHAMS, D.; RIVERA, R. *Boost C++ Libraries*. 2007. Disponível em: <https://www.boost.org/>. Acesso em: 13 out. 2018.

FEOFILOFF, P. *Complexidade: problemas NP-completos*. 2018. Departamento de Ciência da Computação, IME (Instituto de Matemática e Estatística), USP (Universidade de São Paulo), São Paulo, Brasil. Disponível em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto.html>. Acesso em: 09 jun. 2018.

GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1. ed. United States: W. H. Freeman and Company, 1979.

GRAESER, K. A. et al. The Role of Configurational Entropy in Amorphous Systems. *Pharmaceutics*, MDPI AG, v. 2, n. 2, p. 224–244, Mai 2010.

HIREL, P. *XYZ Format*. 2010. Disponível em: <http://atomsk.univ-lille1.fr/doc/en/format_xyz.html>. Acesso em: 14 jun. 2018.

JAKOB, W. *About this project - pybind11*. 2017. Disponível em: <https://pybind11.readthedocs.io/en/master/intro.html>. Acesso em: 13 out. 2018.

JAYNES, E. T. Gibbs vs boltzmann entropies. *American Journal of Physics*, v. 33, n. 5, p. 391–398, 1965.

JONSSONA, P.; LAGERKVISTA, V.; NORDHB, G. Constructing NP-intermediate Problems by Blowing Holes with Parameters of Various Properties. 2015. Department of Computer and Information Science, Linköpings Universitet, Hällekis, Sweden. Disponível em: <http://gustavnordh.com/blowing_holes_revised.pdf>. Acesso em: 19 jun. 2018.

JORNADA, F. H. d. *Propriedades elátiscas de redes contínuas aleatórias de carbono geradas por simulated annealing*. 2010. Dissertação (Mestrado em Física), UFRGS (Universidade Federal do Rio Grande do Sul), Porto Alegre, Brasil.

JR, W. D. C.; RETHWISCH, D. G. *Ciência e engenharia de materiais: Uma introdução*. 9. ed. Rio de Janeiro: LTC, 2018.

KERMODE, J. *extxyz.py*. 2014. Disponível em: <https://gitlab.com/ase/ase/blob/master/ase/io/extxyz.py>. Acesso em: 14 jun. 2018.

KERMODE, J. *AtomsList and AtomsReader objects for I/O*. 2016. Disponível em: <http://libatoms.github.io/QUIP/io.html#module-ase.io.extxyz>. Acesso em: 09 jun. 2018.

LOZANO, A.; RAGHAVAN, V. On the Complexity of Counting the Number of Vertices Moved by Graph Automorphisms. In: *Foundations of Software Technology and Theoretical Computer Science*. Berlin, Heidelberg: Springer, 1998. v. 1530, p. 295–306.

MAKSOV, A.; LI, Y.; BUTLER, R. *Subgraph Isomorphism*. 2015. The University of Tennessee, Knoxville, Estados Unidos. Disponível em: <http://web.eecs.utk.edu/~cphillip/cs594_spring2015_projects/subgraph_isomorphism.pdf>. Acesso em: 09 jun. 2018.

MARTINEZ, F. H. V. *Grau de um vértice*. 2005. FACOM (Faculdade de Computação), UFMS (Universidade Federal de Mato Grosso do Sul), Campo Grande, Brasil. Disponível em: <http://www.facom.ufms.br/~fhvm/disciplinas/anteriores/2005/grafos/aula2.pdf>. Acesso em: 12 jun. 2018.

MATPLOTLIB. *Matplotlib: Python plotting*. 2018. Disponível em: <https://matplotlib.org/>. Acesso em: 17 jun. 2018.

MCKAY, B. D.; PIPERNO, A. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, v. 60, n. 0, p. 94 – 112, 2014. ISSN 0747-7171.

MCKAY, B. D.; PIPERNO, A. *nauty and Traces User's Guide (Version 2.7)*. 2018. Research School of Computer Science, Australian National University, Canberra, Australia and Departimento di Informatica, Sapienza Università di Roma. Disponível em: <http://pallini.di.uniroma1.it/nug27.pdf>. Acesso em: 13 out. 2018.

NETWORKX. *NetworkX*. 2018. Disponível em: <https://networkx.github.io/>. Acesso em: 06 maio 2018.

NETWORKX. *NetworkX isomorphism*. 2018. Disponível em: <https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.isomorphism.is_isomorphic.html>. Acesso em: 06 maio 2018.

NUMPY. *NumPy*. 2018. Disponível em: <http://www.numpy.org/>. Acesso em: 17 jun. 2018.

OPENLABEL. *XYZ Format*. 2007. Disponível em: <http://openbabel.org/wiki/XYZ_%28format%29>. Acesso em: 24 maio 2018.

OPENMP. *OpenMP*. 2018. Disponível em: <https://www.openmp.org/>. Acesso em: 04 nov. 2018.

PINEDA, J. O. d. C. *A entropia segundo Claude Shannon: o desenvolvimento do conceito fundamental da teoria da informação*. 2006. Dissertação (Mestrado em História da Ciência), PUC-SP (Pontifícia Universidade Católica de São Paulo), São Paulo, Brasil.

PYTHON. *About Python*. 2018. Disponível em: <https://www.python.org/about/>. Acesso em: 11 abr. 2018.

PYTHON. *The Python Profilers*. 2018. Disponível em: <https://docs.python.org/2/library/profile.html#module-cProfile>. Acesso em: 21 out. 2018.

ROBINSON, M. *Introduction*. 2018. Disponível em: <https://martinjrobins.github.io/Aboria/aboria/introduction.html>. Acesso em: 13 out. 2018.

SIEK, J.; LEE, L.-Q.; LUMSDAINE, A. *The Boost Graph Library*. 2001. Disponível em: <https://www.boost.org/doc/libs/1_68_0/libs/graph/doc/index.html>. Acesso em: 13 out. 2018.

UFMG. *Propriedades atômicas e tendências periódicas*. 2018. Departamento de Química, UFMG (Universidade Federal de Minas Gerais), Belo Horizonte, Brasil. Disponível em: <http://zeus.qui.ufmg.br/~qgeral/downloads/aulas/aula%2011%20-%20propriedades%20periodicas.pdf>. Acesso em: 09 jun. 2018.

VINK, R. L. C. *Computer Simulations of Amorphous Semiconductors*. 2002. Tese (Doutorado em Física), Universidade de Utrecht, Utrecht, Holanda.

VINK, R. L. C.; BARKEMA, G. T. Configurational Entropy of Network-Forming Materials. *Physical Review Letters*, American Physical Society, v. 89, p. 076405, Jul 2002.

ZHIGILEI, L. V. *Boundary Conditions*. 2018. Department of Materials Science and Engineering, University of Virginia, Charlottesville, Estados Unidos. Disponível em: <http://www.people.virginia.edu/~lz2n/mse627/notes/Boundary.pdf>. Acesso em: 09 jun. 2018.

APÊNDICE A – CÓDIGO FONTE DA IMPLEMENTAÇÃO INICIAL

```
import sys
import networkx as nx
import matplotlib.pyplot as plt

from ase import Atom
from ase.io import read
from ase.data import covalent_radii
from math import log
from operator import itemgetter
from random import uniform
from numpy import asarray
from numpy.polynomial.polynomial import polyfit

def run(G, m, n, slab, c):
   graphs = generateSubgraphs(G, m, n, slab)

   label_total = {}
   iso_label = 1
   for i in range(len(graphs)):
      for j in range(i + 1, len(graphs)):
         iso_label_i = graphs[i].graph["isoLabel"]
         iso_label_j = graphs[j].graph["isoLabel"]

         if iso_label_i == 0 or iso_label_j == 0:
            if nx.is_isomorphic(graphs[i], graphs[j]):
               if iso_label_i == 0 and iso_label_j == 0:
                  graphs[i].graph["isoLabel"] = iso_label
                  graphs[j].graph["isoLabel"] = iso_label
                  label_total[iso_label] = 2
                  iso_label += 1 # label already used
               elif iso_label_i > 0 and iso_label_j == 0:
                  graphs[j].graph["isoLabel"] = iso_label_i
                  label_total[iso_label_i] += 1
               elif iso_label_j > 0 and iso_label_i == 0:
                  graphs[i].graph["isoLabel"] = iso_label_j
                  label_total[iso_label_j] += 1
               elif iso_label_i != iso_label_j:
                  print("Error while checking isomorphism:\nlabelGi %d :
                      labelGj %d" % (iso_label_i, iso_label_j))
```

```
39
40      # get all graphs that are not isomorphic with any other
41      for g in graphs:
42          if g.graph["isoLabel"] == 0:
43              label_total[iso_label] = 1
44              iso_label += 1
45
46      H_n = 0.0
47      H1n = 0.0
48      for i in range(1, iso_label):
49          fi = float(label_total[i])
50          H_n = calcShannonEntropy(H_n, fi, m)
51          if fi == 1.0:
52              H1n = calcShannonEntropy(H1n, fi, m)
53
54      H1nDiv = 0.0
55      if H_n > 0:
56          H1nDiv = (H1n / H_n)
57
58      H_n_extrapolated = H_n + (c * H1nDiv)
59      g_n = 2 * log(n) # (spatial_dimensions - 1)
60      Hc_n = H_n_extrapolated - g_n
61
62      valid = True
63      if H1n > (H_n / 100):
64          print("n: %d. H1(n) exceeds 1% of H(n). Not a valid measurement." % (n))
65          valid = False
66
67      return Hc_n, valid
68
69  def calcShannonEntropy(Hn, fi, m):
70      pi = fi / m
71      Hn -= pi * log(pi)
72      return Hn
73
74  def generateSubgraphs(G, m, n, slab):
75      graphs = []
76
77      (dmin, dmax) = getMaxMinSlab(slab)
78
79      i = 0
80      while i < m:
```

```
81          (x, y, z) = generateRandomPoint(dmin, dmax)
82          n_closest_neighbors = getNClosestNeighborsFromPoint(slab, n, x, y, z)
83          graph = generateSubGraph(G, n, n_closest_neighbors)
84          graphs.append(graph)
85          i += 1
86
87      return graphs
88
89  def getMaxMinSlab(slab):
90      (dmin, dmax) = (
91          { 0: float("Inf"), 1: float("Inf"), 2: float("Inf") }, # x, y, z
92          { 0: -float("Inf"), 1: -float("Inf"), 2: -float("Inf") } # x, y, z
93      )
94
95      positions = slab.get_positions(wrap=True) # wrap atoms back to simulation
            cell
96      for distance in positions:
97          for idx, d in enumerate(distance):
98              if (d > dmax[idx]):
99                  dmax[idx] = d
100             if (d < dmin[idx]):
101                 dmin[idx] = d
102
103     return (dmin, dmax)
104
105 def generateRandomPoint(dmin, dmax):
106     x = uniform(dmin[0], dmax[0])
107     y = uniform(dmin[1], dmax[1])
108     z = uniform(dmin[2], dmax[2])
109
110     return (x, y, z)
111
112 def getNClosestNeighborsFromPoint(slab, n, x, y, z):
113     atomic_numbers = slab.get_atomic_numbers()
114
115     slab.append(Atom(atomic_numbers[0], (x, y, z))) # get the first atom
116     idxAtom = len(slab) - 1
117     all_distances = slab.get_all_distances(mic=True)[idxAtom]
118     slab.pop()
119
120     distances = {}
121     for idx, distance in enumerate(all_distances):
```

```
122         if idx == idxAtom:
123             break
124         distances[idx] = distance
125 
126     n_first = sorted(distances.items(), key=itemgetter(1))[:n] # return list of
        tuples
127     return [i[0] for i in n_first] # return only the first element in list
128 
129 def generateSubGraph(G, n, n_closest_neighbors):
130     graph = nx.Graph(isoLabel=0)
131 
132     for node in n_closest_neighbors:
133         if node in G:
134             if node not in graph:
135                 graph.add_node(node)
136 
137             for neighbor in G[node]:
138                 if neighbor in n_closest_neighbors:
139                     graph.add_edge(node, neighbor)
140 
141     return graph
142 
143 def generateGraphFromSlab(slab, covalent_radii_cut_off):
144     graph = nx.Graph()
145 
146     atomic_numbers = slab.get_atomic_numbers()
147     all_distances = slab.get_all_distances(mic=True)
148     for atom1, distances in enumerate(all_distances):
149         if atom1 not in graph:
150             graph.add_node(atom1) # add nodes not bonded
151 
152         atom1_cr = covalent_radii[atomic_numbers[atom1]]
153         for atom2, distance in enumerate(distances):
154             if atom1 != atom2:
155                 atom2_cr = covalent_radii[atomic_numbers[atom2]]
156                 # if the distance between two atoms is less than the sum of their
                    covalent radii, they are considered bonded.
157                 if (distance < ((atom1_cr + atom2_cr) * covalent_radii_cut_off)):
158                     graph.add_edge(atom1, atom2)
159 
160     return graph
161 
```

```
162 def printGraph(graph):
163     nx.draw(graph, with_labels=True)
164     plt.show()
165 
166 def main():
167     if len(sys.argv) < 6:
168         print("1 parameter: xyz filename\n2 parameter: covalent_radii_cut_off\n3
            parameter: c\n4 parameter: initial n\n5 parameter: final n")
169         return
170 
171     filename = sys.argv[1]
172     covalent_radii_cut_off = float(sys.argv[2]) # 1.12
173     c = float(sys.argv[3])
174     n1 = int(sys.argv[4])
175     n2 = int(sys.argv[5])
176 
177     if n1 > n2:
178         print("Final m cannot be smaller than initial m")
179         return
180 
181     print("Starting script...")
182 
183     slab = read(filename)
184 
185     print("Slab %s read with success" % filename)
186 
187     G = generateGraphFromSlab(slab, covalent_radii_cut_off)
188     total_nodes = len(G)
189     if total_nodes == 0 or G.number_of_edges() == 0:
190         print("No edges found in graph. Check covalent_radii_cut_off")
191         return
192 
193     print("Graph created with success. Nodes found: %d" % total_nodes)
194 
195     hcn_values = []
196     xy_polyfit = []
197     for n in range(n1, n2):
198         m = n * n * total_nodes
199         (hcn, valid) = run(G, m, n, slab, c)
200         hcn_values.append((n, hcn))
201         if valid:
202             xy_polyfit.append((n, hcn))
```

```
203
204     (x_p, y_p) = zip(*xy_polyfit)
205     x_p = asarray(x_p)
206     y_p = asarray(y_p)
207
208     # straight line fit
209     b, m = polyfit(x_p, y_p, 1) # m equals the slope of the line
210     plt.plot(x_p, b + m * x_p, '-')
211
212     x, y = zip(*hcn_values)
213     plt.scatter(x, y)
214
215     plt.axis([n1, n2, -5, 10])
216     plt.show()
217
218     print("Estimated configurational entropy = %f" % (m))
219
220 if __name__ == "__main__":
221     main()
```

ANEXO A – PSEUDOCÓDIGO DO ALGORITMO VF2

Adaptado de (CORDELLA et al., 2004).

```
1  PROCEDURE Match(s)
2    INPUT: um estado intermediário s; o estado inicial s0 possui M(s0)=∅
3    OUTPUT: os mapeamentos entre os dois grafos
4    IF M(s) cobre todos os vértices de G2 THEN
5      OUTPUT M(s)
6    ELSE
7      Computar o set P(s) de pares candidatos para inclusão em M(s)
8      FOREACH (n, m) ∈ P(s)
9        IF as leis de viabilidade retornam sucesso para a inclusão de (n, m) em M(s) THEN
10         Computar o estado s' obtido pela inclusão de (n, m) em M(s)
11         CALL Match(s')
12       END IF
13     END FOREACH
14     Restaurar as estruturas de dados usadas
15   END IF
16 END PROCEDURE
```