PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

OTIMIZAÇÃO E ANÁLISE DE ALGORITMOS DE ORDENAMENTO DE REDES PROTEICAS

FELIPE AUGUSTO KUENTZER

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Alexandre de Morais Amory

Porto Alegre
2014

Dados Internacionais de Catalogação na Publicação (CIP)

K95o Kuentzer, Felipe Augusto
Otimização e análise de algoritmos de ordenamento de redes proteicas / Felipe Augusto Kuentzer. – Porto Alegre, 2014.
78 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Alexandre de Morais Amory.

1. Informática. 2. Biologia Computacional. 3. Proteínas. 4. Genes. 5. Algoritmos (Programação). 6. Otimização Combinatória. I. Amory, Alexandre de Morais. II. Título.

CDD 574.0285

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "Otimização e Análise de Algoritmos de Ordenamento de Redes Protéicas" apresentada por Felipe Augusto Kuentzer como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 25/02/2014 pela Comissão Examinadora:

Prof. Dr. Alexandre de Morais Amory – Orientador — PPGCC/PUCRS

Prof. Dr. César Augusto Missio Marcon – PPGCC/PUCRS

Profa. Dra. Rita Maria Cunha de Almeida – UFRGS

Homologada em 03/04/2014, conforme Ata No. 005 pela Comissão Coordenadora.

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320–3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

OTIMIZAÇÃO E ANÁLISE DE ALGORITMOS DE ORDENAMENTO DE REDES PROTEICAS

RESUMO

A análise por Transcriptograma foi desenvolvida como uma solução para a redução de ruído, comum nas medidas do Transcriptoma provenientes da técnica de microarranjo, e tem demonstrando potencial se aplicada como método para diagnósticos de doenças. A redução do ruído existente nas medidas se dá pelo ordenamento da rede de interações proteicas do organismo, permitindo a análise da expressão gênica em escala de genoma completo. A eficiência do Transcriptograma para a redução do ruído já foi analisada, entretanto, ainda carece a avaliação da qualidade do ordenamento, definindo para isso, a melhor configuração de parâmetros para o algoritmo de ordenamento utilizado pelo Transcriptograma. Até o momento, essa análise é dificultada pelo elevado tempo de execução do algoritmo de ordenamento. Neste trabalho, uma análise das etapas do algoritmo de ordenamento possibilita a realização de otimizações, e consequente redução no tempo de execução, além de permitir a análise mais aprofundada das configurações dos parâmetros que tem maior influência na qualidade do ordenamento. Aplicando o Transcriptograma a um problema de diagnóstico, utiliza-se a medida do diagnóstico para caracterizar a influência dos parâmetros do algoritmo de ordenamento na obtenção de melhores diagnósticos. Observa-se nos resultados, que a rede proteica utilizada em trabalhos anteriores não apresenta os melhores diagnósticos. Além disso, a minimização do ordenamento, alcançada por meio da execução prolongada do algoritmo de ordenamento, não necessariamente aumenta a probabilidade de encontrar um melhor diagnóstico comparado com o ordenamento aleatório. Mesmo que os resultados experimentais com o diagnóstico não diferenciem estatisticamente o ordenamento aleatória do ordenamento otimizado, estes resultados não podem ser considerados conclusivos pois uma única doença foi avaliada.

Palavras Chave: transcriptograma, ordenamento, simulated annealing, otimização de algoritmos, diagnóstico, rede proteica.

OPTIMIZATION AND ANALYSIS OF PROTEIN NETWORKS ORDERING ALGORITHMS

ABSTRACT

Analysis by Transcriptogram was developed as a solution to noise reduction, usually present in the microarray measuring technique of the Transcriptome, and has demonstrated potential to be applied as a method of disease diagnostics. The noise reduction in the measure is achived by the protein interaction network ordering, allowing gene expression analysis in whole genome scale. The Transcriptogram's efficiency to noise reduction was analyzed, however, it still lacks an analisys of the ordering quality, so that the best parameter setting for the ordering algorithm is used by the Transcriptogram. So far, this analysis is hindered by the high runtime of the ordering algorithm. In this work, an analysis of the ordering algorithm stages allows some optimizations, and consequent reduction in execution time, also allowing further analysis on which parameters settings have the greatest influence on the ordering quality. Applying the Transcriptogram to a diagnostic problem, the diagnostic measure is used to characterize the influence of the parameters of the ordering algorithm to achive better diagnoses. The results show that the protein network used in previous works doesn't produce the best diagnostics. Moreover, the ordering minimization, achieved by executing the ordering algorithm for longer periods, does not necessarily increase the probability to find better diagnosis compared to random ordering. Eventhough the experimental diagnostic results could not statistically differentiate random ordering from optimized ordering, these results cannot be considered conclusive since a single disease has been evaluated.

Keywords: transcriptogram, ordering, simulated annealing, optimization of algorithms, diagnostic, protein network.

LISTA DE FIGURAS

Figura 3.1 - Histograma da distribuição de graus da rede proteica do *Homo sapiens* 22
Figura 3.2 - Ordenamento aleatório (a) e versão diagonalizada da matriz (b) [30] 24
Figura 4.1 - Rede proteica e a matriz de adjacência correspondente 26
Figura 4.2 - Contribuição de $M_{6,8}$ no custo da matriz 27
Figura 4.3 - Troca de duas linhas e duas colunas 27
Figura 4.4 - Configuração inicial da matriz (a) e após o ordenamento (b). Os pontos representam interação entre as proteínas. 29
Figura 5.1 - Matriz de adjacência e seleção para troca 31
Figura 5.2 - Exemplo de troca com a vetor de ponteiros 31
Figura 5.3 - Matriz antes e depois da troca 32
Figura 5.4 - Nodos utilizados no cálculo parcial do custo 32
Figura 5.5 - Nodos utilizados no cálculo da diagonal superior 33
Figura 5.6 - Nova representação da matriz 34
Figura 5.7 - Sequência de trocas sobre os vetores de ponteiros 35
Figura 5.8 - Código fonte da função *swap* 37
Figura 5.9 - Código fonte da função *getMatWeight* 37
Figura 5.10 -Código fonte do laço principal 38
Figura 5.11 -Código fonte da função *swap* extraído do algoritmo *pointer_cfm* 38
Figura 5.12 -Código fonte da função *getMatWeight* extraído do algoritmo *diagonal_cfm* . 39
Figura 5.13 -Código fonte do laço principal em *partial_cfm* 40
Figura 5.14 -Tempo de execução das diferentes versões do CFM 42
Figura 5.15 -Medições do custo (verde), do número de trocas (azul) e da temperatura (vermelho) ao longo da execução do algoritmo *platinum_cfm* 43
Figura 5.16 -Medições do custo (verde), do número de trocas (azul) e da temperatura (vermelho) ao longo da execução do algoritmo *platinum_cfm* 44
Figura 6.1 - Menor custo obtido com cada algoritmo para 3000, 5000 e 10000 passos . . . 48
Figura 7.1 - Conjunto de diagnósticos gerados em função *r* 49
Figura 7.2 - Histograma resultante da análise da distribuição dos diagnósticos 51
Figura 7.3 - Comparação entre algoritmos com diagnósticos superior a $0,933$ 52
Figura 7.4 - Conjunto total de diagnóstico gerados a partir das redes do *Homo sapiens* . . 53
Figura 7.5 - Comparação entre *scores* com diagnósticos superiores a $0,933$ 54
Figura 7.6 - Ordenamento com $\alpha = 1$ (a) e ordenamento com $\alpha = 10$ (b) 55
Figura 7.7 - Comparação entre *alphas* (α) com diagnósticos superiores a $0,933$ 56

Figura 7.8 - Comparação entre número de passos (*steps*) com diagnósticos superiores a $0,933$ 56

LISTA DE TABELAS

Tabela 5.1 - Relação de algoritmos e otimizações 36
Tabela 5.2 - Tempo de execução em segundos dos algoritmos em função do número de passos 41
Tabela 6.1 - Resultados estatísticos dos algoritmos *cfm* e *spectrallni* 48
Tabela 6.2 - Resultados estatísticos dos algoritmos *spectralFinal reheat* 48

SUMÁRIO

1 INTRODUÇÃO ... 16

2 CONTEXTUALIZAÇÃO E TRABALHOS RELACIONADOS ... 18

2.1 TRANSCRIPTOGRAMA ... 18

3 REFERENCIAL TEÓRICO ... 20

3.1 BANCO DE DADOS STRING ... 20

3.2 REDES COMPLEXAS ... 21

3.2.1 REDES LIVRES DE ESCALA ... 22

3.3 CLUSTERIZAÇÃO DE DADOS ... 22

3.3.1 CLUSTERIZAÇÃO DE GRAFOS ... 23

3.4 SERIAÇÃO ... 24

3.5 O PROBLEMA MINLA ... 25

4 ALGORITMO CFM ... 26

5 OTIMIZAÇÃO DO MÉTODO E ANÁLISE ... 30

5.1 OTIMIZAÇÕES DE SOFTWARE ... 30

5.1.1 ANÁLISE DE COMPLEXIDADE ... 36

5.1.2 TESTES PRÁTICOS DE DESEMPENHO ... 41

5.2 ORDENAMENTO ESPECTRAL E REAQUECIMENTO ... 42

5.2.1 REAQUECIMENTO ... 43

5.2.2 ORDENAMENTO ESPECTRAL ... 44

6 ANÁLISE QUANTITATIVA DO ORDENAMENTO ... 47

7 ANÁLISE QUALITATIVA DO ORDENAMENTO ... 49

7.1 MÉTODO DE AVALIAÇÃO DOS RESULTADOS ... 50

7.2 EFICIÊNCIA DOS ALGORITMOS DE ORDENAMENTO ... 50

7.3 EFICIÊNCIA DOS PARÂMETROS NO ALGORITMO CFM ... 52

7.3.1 PARÂMETRO SCORE ... 52

7.3.2 PARÂMETRO ALPHA (α) ... 54

7.3.3 PARÂMETRO STEPS (NÚMERO DE PASSOS) ... 55

7.4 CONSIDERAÇÕES FINAIS DA ANÁLISE QUALITATIVA ... 57

8 CONCLUSÃO **58**
8.1 TRABALHOS FUTUROS 59

REFERÊNCIAS **60**

APÊNDICE A – Função do cálculo parcial do algoritmo *partial_cfm* **63**

APÊNDICE B – Função do cálculo parcial do algoritmo *platinum_cfm* **65**

APÊNDICE C – Estatística dos Grupos de Algoritmos **67**

APÊNDICE D – Estatística dos Grupos de *Score* **68**

APÊNDICE E – Estatística dos Grupos de Alpha **71**

APÊNDICE F – Estatística dos Grupos de *Steps* **77**

1. INTRODUÇÃO

Após o mapeamento e sequenciamento do código genético do *Homo sapiens* (Projeto Genoma Humano [23]), um dos desafios da área biológica passa a ser a análise desses dados, determinando quais genes são expressos em uma determinada célula ou determinada condição patológica. As proteínas são as principais coordenadoras das funções biológicas nas células, mas apenas determinar quais proteínas são expressas não é suficiente para entender o seu papel biológico. Uma função biológica necessita de várias proteínas agindo coordenadamente, além disso, uma única proteína pode participar de várias funções celulares diferentes, formando assim uma rede extremamente complexa.

Para determinar o papel de uma dada proteína no contexto total de uma célula, além das dezenas de experimentos de laboratório que devem ser realizados [31], é necessária uma trabalhosa e complicada análise desses dados gerados. Devido à enorme quantidade de informações sobre as proteínas e suas interações, a classificação de conjuntos de proteínas se torna uma tarefa nada trivial, possível apenas com a utilização de ferramentas de bioinformática especializadas. Entretanto, não existem muitos métodos aplicados ao diagnóstico e prevenção de doenças a nível celular. O desenvolvimento de uma ferramenta capaz de analisar perfis de células em diferentes estados, como comparar uma célula saudável com uma célula doente, ou uma célula degenerativa com uma célula em proliferação, permitiria identificar traços relevantes para o diagnóstico celular.

Embora a análise das redes proteicas não seja uma tarefa simples, a complexa rede de interações entre proteínas pode ser modelada computacionalmente. Grafos representam muito bem essas interações, onde os nodos são associados as proteínas e as arestas são associadas as interações entre essas proteínas. A partir dessa modelagem é possível aplicar um algoritmo de minimização de custo como, por exemplo, o CFM (*Cost Function Method*) [27].

O algoritmo CFM modela a interação de proteínas de uma determinada célula através de um grafo representado por uma matriz de adjacência. O objetivo desse algoritmo é ordenar as proteínas, de tal forma que as proteínas que mais interagem entre si fiquem próximas, possibilitando encontrar *clusters*. Cada *cluster* carateriza um módulo funcional, que representa uma ou mais funções celulares, como por exemplo, morte celular, geração de energia, diferenciação celular, entre outras funções. As informações obtidas com a aplicação do algoritmo CFM geram perfis de expressão, também chamados de Transcriptogramas.

Neste momento surge um novo problema. Para analisar cerca de 4000 proteínas e 47 mil interações do microrganismo *Saccharomyces serevisiae* [28], o tempo de execução do algoritmo é da ordem de semanas. O processamento de células do *Homo sapiens* está estimado em alguns meses, uma vez que o mesmo pode ter cerca de 9000 proteínas e 110 mil ligações [8]. Esse exemplo mostra a importância de otimizações algorítmicas e computacionais para tornar o método viável para o diagnóstico de doenças.

Além de tornar viável o diagnóstico de doenças, a aceleração do método permite investigar como os diferentes parâmetros que compõem o algoritmo CFM contribuem para um diagnóstico eficaz. Em outras palavra, é preciso identificar quais parâmetros levam a um melhor ordenamento, e consequentemente a um melhor diagnóstico. Nesse contexto, a análise das diferentes etapas do algoritmo CFM e as subsequentes otimizações passam a ser um dos objetivos deste trabalho. Somado às otimizações, essa dissertação também avalia algumas técnicas presentes na literatura, que ao serem incorporadas nas etapas do CFM, contribuam para a qualidade do ordenamento.

Enquanto [11] analisou a eficiência do Transcriptograma, o presente trabalho tem como objetivo avaliar a eficiência dos ordenamentos produzido pelo algoritmo CFM para a geração de bons diagnósticos. Nesse caso, as variações do algoritmo CFM propostas e os parâmetros que compõem o algoritmo CFM são analisados conforme os diagnósticos gerados a partir de diferentes configurações de algoritmos e parâmetros, caracterizando assim quais configurações são propícias para gerar melhores diagnósticos.

Esta dissertação está organizada de modo a, inicialmente, contextualizar e introduzir a análise por Transcriptograma (Capítulo 2), abordar alguns aspectos relevantes para o desenvolvimento do trabalho (Capítulo 3) e descrever as etapas e parâmetros que compõem o algoritmo CFM (Capítulo 4). Na sequência, são apresentadas as otimizações aplicadas ao CFM, a análise de complexidade do CFM e das otimizações, bem como os testes que demostram a redução no tempo de execução, além das técnicas que visam melhorar a qualidade do ordenamento (Capítulo 5). Por fim, os algoritmos de ordenamento passam por uma análise quantitativa (Capítulo 6) e subsequente análise qualitativa (Capítulo 7), onde os algoritmos e parâmetros do CFM são analisados quanto à eficiência na produção de melhores diagnósticos.

2. CONTEXTUALIZAÇÃO E TRABALHOS RELACIONADOS

Segundo [6], um dos grandes desafios em pesquisas biomédicas com genes é catalogar de forma sistemática todas as moléculas e suas interações em uma célula viva. Existe uma grande necessidade de entender como essas moléculas e as interações entre elas determinam o funcionamento dos organismos complexos, sejam observadas isoladamente ou quando interagindo com outras células.

A análise individual de células foi um grande foco das pesquisas e se mostrou muito eficiente. Porém, a maior parte das características biológicas surgem das complexas interações entre os numerosos componentes das células, como por exemplo as proteínas, o DNA (*Deoxyribonucleic Acid*) e pequenas moléculas. Nesse sentido, novas frentes de pesquisa buscam cada vez mais entender as interações entre elementos de redes complexas, e como elas refletem nas características gerais do sistema.

Redes reais encontradas na natureza e na sociedade podem apresentar duas propriedades genéricas: livres de escala (*scale free*) e elevado grau de clusterização. Em [7], essas propriedades são apresentadas como consequência de uma organização hierárquica. Com base na análise dessas propriedades em diferentes redes reais (e.g. *World Wide Web*, redes sociais e redes metabólicas), o estudo demostra que a hierarquia é uma característica fundamental de muitas redes complexas.

Operando sobre conjuntos de dados oriundos de diferentes áreas (e.g. redes de transporte aéreo, redes de circuitos eletrônicos e redes biológicas), o método não supervisionado proposto em [29] faz a extração de organizações hierárquicas presentes no sistema. O método desenvolvido compreende basicamente dois passos. No primeiro passo é estimada a proximidade entre os nodos da rede, formando assim uma matriz de proximidade, também referenciada como matriz de afinidade. No segundo passo são descobertas as organizações hierárquica gerais de afinidades dos nodos. Para essa segunda etapa o estudo propõe um novo método de clusterização hierárquico (*Box Model Clustering*).

2.1 Transcriptograma

A análise de conjuntos de dados biológicos é objeto dos estudos [28] [8] [25] [11]. O Transcriptograma, apresentado pela primeira vez em [27], é uma técnica para análise de perfis de expressões gênicas. A técnica consiste em um novo método de ordenamento para redes proteicas associado à análise do Transcriptoma.

Transcriptoma é o conjunto de RNA de transcrição presente na célula. Nele está contida a informação de quais proteínas serão produzidas na célula. Entre as técnicas utilizadas para quantificar o Transcriptoma está a técnica de microarranjo [31]. Um dos problema dessa técnica está no ruído presente na medida e na variabilidade da medida, que dependendo do laboratório, apresenta

resultados diferentes. Ainda assim é uma técnica bastante difundida, sendo uma das principais fontes de dados dessa natureza.

Devido à variabilidade e ruídos dos dados gerados pela técnica de microarranjo, surgiu a proposta para o novo método de ordenamento de redes proteicas. Através do ordenamento da rede proteica, cria-se uma lista que relaciona a proximidade de duas proteínas com a probabilidade de existir associação entre elas. Havendo essa relação é possível realizar médias sobre regiões do ordenamento, suavizando assim o ruído. O algoritmo aplicado no ordenamento da rede é descrito em maiores detalhes no capítulo 4.

Com o Transcriptograma é possível extrair informações de células que podem ser aplicadas em diagnósticos, como demostrado em [8]. Nesse trabalho, o Transcriptograma de células cancerosas é comparado com o Transcriptograma de células saudáveis, de modo que a análise possibilita a identificação dos genes afetados pela doença. Mais recentemente, o método de Transcriptograma foi generalizado para duas dimensões [25], e também foi alvo do estudo em [11], que analisou a eficiência do Trancriptograma.

3. REFERENCIAL TEÓRICO

Este capítulo se dedica a descrever alguns aspectos relevantes para o desenvolvimento deste trabalho. Inicialmente será descrito o banco de dado de onde provêm os dados que aqui foram utilizados. Na sequência serão abordadas algumas propriedades das redes complexas, e uma breve introdução à clusterização de dados e à técnica de seriação. Por fim um problema clássico de ordenamento é apresentado, e guia o desenvolvimento de alguns experimentos deste trabalho.

3.1 Banco de Dados STRING

A base de dados STRING [13] [34] [36] (*Search Tool for the Retrieval of InteractingGenes/-Proteins*), acessível em http://string-db.org, disponibiliza informações relativas a interações físicas diretas entre proteínas ou indiretas (rotas-metabólicas), ditas funcionais, onde duas proteínas que não interagem diretamente, contribuem para um determinado processo celular. Além das interações diretas e indiretas, outros métodos de associações são previstos no STRING:

- **Vizinhança:** Uma maior proximidade de genes no genoma indica que as proteínas codificadas por eles participam da mesma rota metabólica, caracterizando assim a interação entre as proteínas.
- **Co-ocorrência:** Essa forma de associação leva em conta a história evolutiva das proteínas em seus genes correspondestes. Nesse caso, a semelhança entre padrões de presença/ausência de genes em diferentes organismos caracteriza a interação.
- **Fusão:** Pela pressão seletiva, dois genes podem se transformar em um único gene. Esse processo de fusão em uma espécie indica uma associação funcional de proteínas em outra espécie.
- **Co-expressão:** Prevê a associação baseado na observação simultânea de padrões de expressão em genes do mesmo organismo.
- **Experimentos** e ***Database*:** São associações extraídas de outras bases de dados. Enquanto os Experimentos fornecem informação relativas a associações físicas diretas, os *Database* indicam associações que estão anotadas em outros bancos de dados.
- ***Textmining*:** Quando genes e proteínas são frequentemente citados em resumos de artigos, mesmo que não possuam nenhuma interação, o STRING considera estatisticamente relevante essa situação para uma associação.

Devido ao grande espaço de associações fornecido pelo STRING, é provável que nem todas ocorram efetivamente, entretanto, uma parte interessante dessa base de dados é a possibilidade de

controlar a qualidade e origem das informação. No STRING, todas as interações recebem um *score* de confiabilidade para cada um dos métodos de associação citados acima. A Equação 3.1 [36] combina os *scores* individuais (S_i) para determinar a probabilidade (S) de que ambas as proteínas estejam associadas a mesma rota metabólica do KEGG [20] (*Kyoto Encyclopedia of Genes and Genomes*). O banco de dados KEGG é considerado um bom padrão de referência, pois seus dados são curados manualmente e cobrem uma grande variedade de organismos.

Por exemplo, um *score* de $0,8$ determina um grau de confiabilidade de no mínimo 80% para associações presentes na rede. Nos trabalhos com Transcriptogramas o *score* combinado é igual a $0,8$, equilibrando assim o número de falsos positivos e falsos negativos [27].

$$S = 1 - \prod_i (1 - S_i) \tag{3.1}$$

As redes proteicas abordadas nesse trabalho foram extraídas das versões 8.2 e 9.05 do STRING, e utilizaram todos os métodos de associação citados acima, com exceção do *textmining*, para compor o *score* total. A versão 9.05 do STRING possui mais de 1000 organismos, o que totaliza mais de 5 milhões de proteínas e cerca de 200 milhões de interações.

3.2 Redes Complexas

O estudo de redes complexas é bastante difundido em diversas áreas, tais como ciência da computação, matemática, biologia e física. O termo redes complexas refere-se a um grafo com estrutura topográfica não trivial. Sendo assim, nem todo grafo pode ser considerado como representação de uma rede complexa. Para um grafo ser considerado uma rede complexa, algumas propriedades topográficas precisam estar presentes. Algumas dessas propriedades serão brevemente descritas a seguir [3].

- **Coeficiente de aglomeração:** Os agrupamentos intrínsecos são quantificados por meio desse coeficiente, também denominado de fenômeno da transitividade. O coeficiente de aglomeração é alto quando um nodo *X* está ligado a um nodo *Y*, e o nodo *Y* está ligado a um nodo *Z*, aumentando a chance do nodo *X* também estar ligado a *Z*.

- **Distribuição de graus:** O grau de um nodo qualquer da rede determina a quantidade de arestas que incidem sobre esse nodo. A distribuição de graus é uma função de distribuição probabilística (p), que indica a chance de um determinado nodo ter grau fixo (k). Em grande número de redes reais, incluindo a *Word Wide Web* [5] e redes metabólicas [18], a distribuição de graus segue uma Lei de Potência, onde $p(k) \sim k^{-\gamma}$ para uma constante γ qualquer que caracteriza a rede em particular.

- **Resistência:** Indica a capacidade que a rede tem para suportar a remoção de alguns vértices, sem que sua funcionalidade seja afetada.

- **Correlação de graus:** Indica se as arestas associam nodos com graus parecidos.

3.2.1 Redes Livres de Escala

São três os principais modelos de redes complexas: aleatórias, pequeno-mundo e livres de escala (*scale free*). Os dois primeiros não caracterizam as redes utilizadas neste trabalho e, portanto não foram abordados. Como demostrado a seguir, as redes proteicas analisadas são classificadas como livres de escala.

Denominadas de livres de escala por [5], são redes que apresentam características bem específicas na ordem da sua estruturação. No modelo proposto, o crescimento da rede é regido pela conexão preferencial, característica que determina a tendência de um novo vértice se conectar a um vértice com grau elevado de conexão. Isso implica em redes com poucos vértices altamente conectados e muitos vértices com poucas conexões.

Redes livres de escala possuem distribuição de grau seguindo a Lei de Potência na forma de $p(k) \sim k^{-\gamma}$, e tipicamente no intervalo $2 < \gamma < 3$. A Figura 3.1 apresenta o graficamente o histograma da distribuição da rede proteica do *Homo sapiens*, foco do presente trabalho. É possível observar o elevado número de nodos que possuem poucas ligações, ou seja, possuem um grau de conectividade baixo. Na outra ponta, em muito menor número, estão os nodos que possuem muitas conexões. Não fica visível no histograma mas uma única proteína da rede do *Homo sapiens* está ligada a outras 816 proteínas.

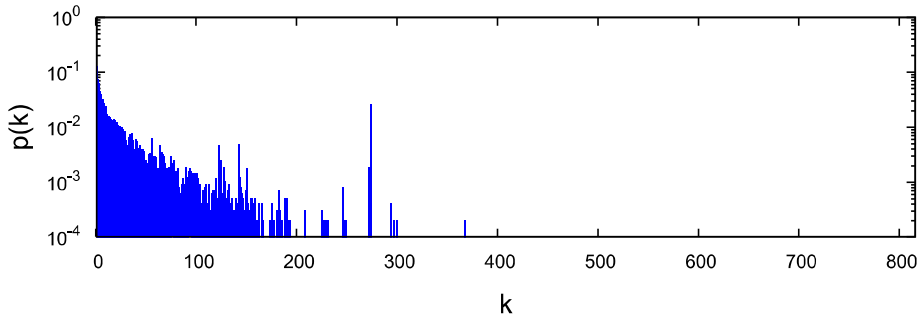


Figura 3.1 - Histograma da distribuição de graus da rede proteica do *Homo sapiens*

3.3 Clusterização de Dados

Grandes conjuntos de dados normalmente dificultam a visualização de informações relevantes. A descoberta de *clusters* pode ser usada para explicar as características da distribuição dos dados subjacentes, e assim servir como base para várias técnicas de análise de dados.

A análise de *clusters* [16] envolve a organização de um conjunto de padrões, normalmente representados na forma de vetores de atributos ou pontos em um espaço multidimensional. O agrupamento dos dados normalmente é baseado em alguma medida de similaridade definida para o conjunto de elementos. A partir dessa medida é possível extrair aspectos relevantes para classificação dos dados para posterior análise.

O ato de agrupar dados pode ser definido como um problema de aprendizagem não-supervisionada, já que o conjunto de dados e as propriedades que os tornam semelhantes não estão rotuladas, ou seja, são desconhecidas. Como não possuem rótulos iniciais, o objetivo da clusterização é encontrar uma organização válida e conveniente de dados, e não separá-los em categorias, como é no processo de reconhecimento e classificação de dados.

Grafos são outra forma para representação de conjuntos de padrões. São estruturas formadas por um conjunto de vértices, também chamados de nodos, e um conjunto de arestas, que são as conexões entre pares de vértices. As pesquisas e publicações propondo diferentes tipos de algoritmos de clusterização de grafos cresceram consideravelmente nos últimos anos [30]. Na seção a seguir serão abordados alguns conceitos relativos a clusterização de grafos e que são de interesse para o presente trabalho.

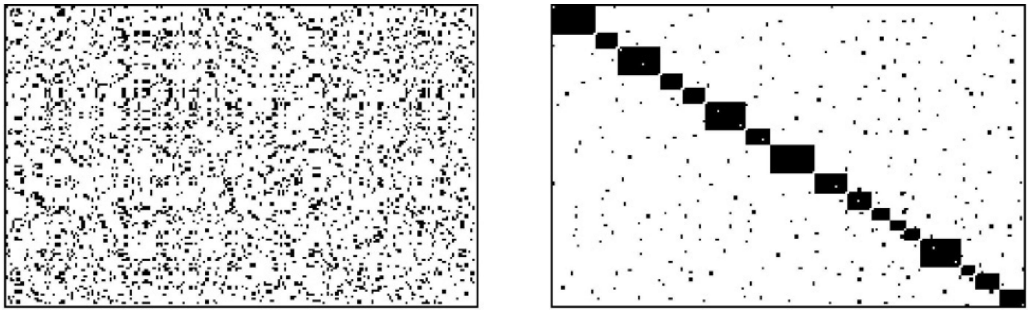
3.3.1 Clusterização de Grafos

Dado um grafo, o objetivo da clusterização é agrupar os vértices similares em *clusters*. Entretanto, nem todos os grafos possuem *clusters* naturais, ou seja, em sua representação não apresentam regiões com alta densidade de vértices separadas de outras regiões com baixa densidade. Independente disso, os algoritmos de clusterização geram os *clusters* para qualquer grafo de entrada.

Se a estrutura do grafo é uniforme, com as arestas distribuídas uniformemente sobre o conjunto de vértices, o agrupamento calculado pelo algoritmo será bastante arbitrário. Medidas de qualidade e formas de visualização ajudam a determinar se existem grupos significativos no grafo. Para entender melhor essa questão, supondo uma matriz de adjacência que representa um determinado grafo, quando os nodos são ordenados aleatoriamente (Figura 3.2(a)) não existe uma estrutura aparente na matriz.

Nessa situação, interpretar a presença, número ou qualidade dos *clusters* não é uma tarefa trivial. No entanto, ao executar um algoritmo de clusterização de grafos e aplicar um novo ordenamento aos vértices conforme seus respectivos *clusters*, a estrutura dos grupos passa a ser evidente (Figura 3.2(b)).

Uma das principais abordagens para identificação de *clusters* consiste em determinar uma valor para cada vértice e classificar os vértices em grupos baseado no valor obtido. Determinar uma medida precisa para classificação não é necessariamente simples, e pode ser uma tarefa ainda mais complexa que o processo de clusterização do grafo.



(a) (b)

Figura 3.2 - Ordenamento aleatório (a) e versão diagonalizada da matriz (b) [30]

Em algumas aplicações a falta de informações relativas às propriedades de cada vértice dificulta ainda mais o cálculo da similaridade. Nesses casos uma alternativa é utilizar a incidência das arestas sobre os vértices, onde a medida de similaridade é baseada exclusivamente nas propriedades estruturais do grafo. Outra alternativa para a falta de características específicas dos vértices é definir os *clusters* a partir da conectividade, obtida calculando o número de caminhos que existem entre todos os pares de vértices. Para alguns vértices pertencerem a um *cluster* eles devem ser altamente conectados entre si.

Como foi demostrado no exemplo da Figuras 3.2, o ordenamento, dependendo das características do grafo, possui um papel muito importante na identificação dos *clusters*. Nesse ponto, a literatura apresenta outro método para identificação de padrões em conjuntos de dados, chamado de Seriação.

3.4 Seriação

Seriação é uma técnica de análise de dados que reordena os objetos em uma sequência unidimensional, permitindo que o melhor ordenamento revele a regularidade e padrões ao longo da série. Segundo [22], a seriação tem uma forte relação com a clusterização, embora não haja um entendimento entre as áreas sobre a sua distinção.

A forma mais comum de representação de dados e visualização de resultados no contexto da seriação é através de uma matrizes de adjacência. A matriz de adjacência é construída a partir de um grafo que representa o conjunto de dados que devem ser analisados. Os objetos da matriz normalmente são ordenados arbitrariamente, e alterar a ordem das linhas e colunas da matriz não altera a estrutura. O objetivo da seriação é encontrar uma permutação que reordene os objetos em uma sequência permitindo identificar padrões no conjunto de dados.

A técnica de permutação da matriz é considerada por [33] uma abordagem com grandes vantagens em relação aos algoritmos de clusterização, já que nenhuma informação é perdida, e o número de *clusters* que pretende-se encontrar não precisa ser fornecido de antemão. Além disso,

[9] demostrou que o problema de ordenamento é usualmente isomorfo ao clássico problema MinLA (*Minimal Linear Arrangement*).

3.5 O Problema MinLA

O problema clássico MinLA, também conhecido como OLA (*Optimal Linear Arrangement*), é bastante difundido em áreas que tratam de problemas de layout, como por exemplo, layout de VLSI (*Very Large Scale Integration*), layout para análise numérica, onde deseja-se reordenar linhas e colunas de matrizes simétricas e esparsas de modo que os elementos não zeros aproximem-se ao máximo da diagonal, layout para desenhos de grafos e problemas de layout na biologia computacional.

Dado um layout σ de um grafo não dirigido $G = (V, E)$, o objetivo do problema MinLA é arranjar linearmente os vértices, mapeando cada vértice i para uma posição $\sigma(i)$, minimizando o custo C. A função custo a ser minimizada é apresentada na Equação 3.2.

$$C_{G,\sigma} = \sum_{ij \in E} |\sigma(i) - \sigma(j)| \tag{3.2}$$

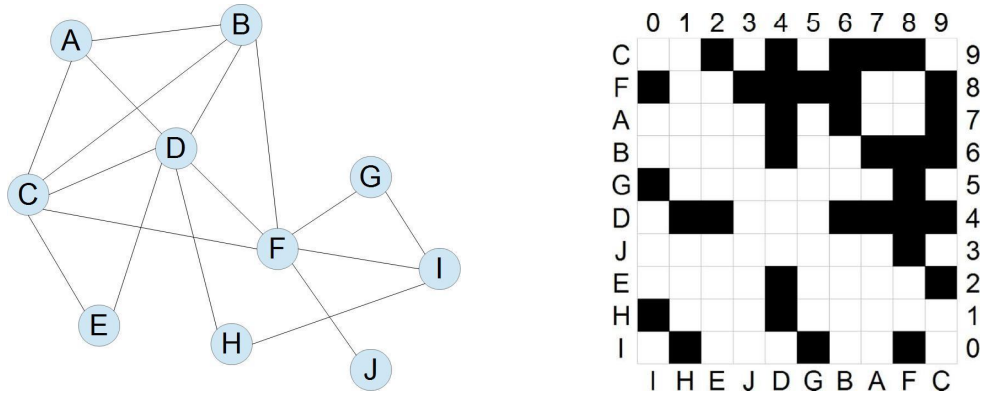
Em [26] são realizados diversos experimentos utilizando métodos dos tipos *Lower Bound* e *Upper Bound*. Entre as heurísticas utilizadas estão a *Breath-first Search*, *Spectral Sequencing* e *Simulated Annealing*. Essas heurísticas são classificadas como heurísticas de aproximação, onde o objetivo não é encontrar o resultado ótimo, mas sim um resultado satisfatório em um tempo hábil. As avaliações foram feitas utilizando grafos de referência da literatura. Além dos testes individuais das heurísticas, o autor propõe uma combinação de técnicas, que produzem bons resultados e motivam alguns experimentos descritos na Seção 5.2.2.

4. ALGORITMO CFM

Este capítulo descreve o algoritmo CFM (*Cost Function Method*), previamente apresentado em [27] e [28], que consiste em minimizar a função custo de uma matriz de interação proteica. A matriz de interação é construída a partir de um conjunto de dados que descreve uma rede de interações entre proteínas. Essas redes estão disponíveis para diversos organismos, e podem ser obtidas através da base de dados STRING [17]. Os exemplos apresentados a seguir são simplificados e não refletem um conjunto de dados reais.

A interação entre as proteínas é modelada por um grafo não direcionado, onde os nodos são proteínas e as arestas representam as interações. Essas informações são então representadas como uma matriz de adjacência.

Inicialmente os nodos são ordenados e distribuídos aleatoriamente na matriz de adjacência. A Figura 4.1(a) ilustra uma rede de interações onde é possível observar, por exemplo, a interação entre a proteína A e a proteína B. Todas as conexões do grafo são transferidas para a matriz de adjacência da Figura 4.1(b). A interação entre as proteínas A e B é percebida pelo preenchimento dos quadrantes AB e BA da matriz.



(a) (b)

Figura 4.1 - Rede proteica e a matriz de adjacência correspondente

Afim de ressaltar a interação entre as proteínas, o algoritmo tenta aproximar os nodos que interagem mais entre si. Isso corresponde a aproximar os núcleos à diagonal da matriz. Para tanto, foi desenvolvida uma função capaz de determinar o custo da matriz, Equação 4.1, onde o custo total H da matriz M corresponde ao somatório do custo individual de cada nodo e seus vizinhos conforme as coordenadas i e j da matriz.

$$H = \sum_{i=1}^{N}\sum_{j=1}^{N}|i-j|^{\alpha}(|M_{i,j}-M_{i+1,j}|+|M_{i,j}-M_{i-1,j}| + |M_{i,j}-M_{i,j+1}|+|M_{i,j}-M_{i,j-1}|) \tag{4.1}$$

O primeiro passo do algoritmo é calcular o custo do estado inicial da matriz. Destacado em vermelho na Figura 4.2 está o quadrante (6,8), e em amarelo os seus vizinhos. Supondo $\alpha = 1$, a contribuição da posição (6,8) para o custo total da matriz é obtida aplicando a equação de custo descrita anteriormente, de modo que $|6-8|^1(1+0+0+0) = 2$. Conforme o cálculo, o custo da posição (6,8) é 2. Nesse exemplo, apenas o vizinho da direita não possui conexão, portanto seu valor é 0, enquanto que os vizinhos superior, inferior e da esquerda contribuem com valor 1 para o cálculo. O custo das outras células é calculado de forma análoga.

Na Equação 4.1, também é possível observar a presença de α. Com o incremento desse parâmetro, interações mais afastadas da diagonal recebem um peso maior. Nesse caso, mover interações mais afastadas da diagonal contribui mais para a redução do custo. O parâmetro α é abordado novamente na Subseção 7.3.2.

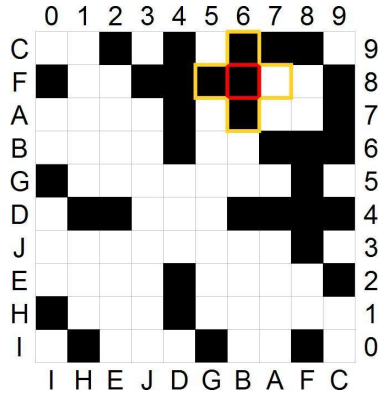


Figura 4.2 - Contribuição de $M_{6,8}$ no custo da matriz

O próximo passo é selecionar dois números inteiros k_1 e k_2. Obtidos aleatoriamente, esses números representam índices de proteínas. Os nodos da posição k_1 e k_2 são então trocados, alterando a posição das proteínas e modificando a matriz de interação. Embora a matriz seja modificada, como a coluna/linha inteira foi movida, nenhuma informação sobre as ligações dos nós k_1 e k_2 é perdida. Essa alteração muda o custo da matriz devido ao novo posicionamento das proteínas. A Figura 4.3 ilustra a troca onde $k_1 = 3$ e $k_2 = 6$.

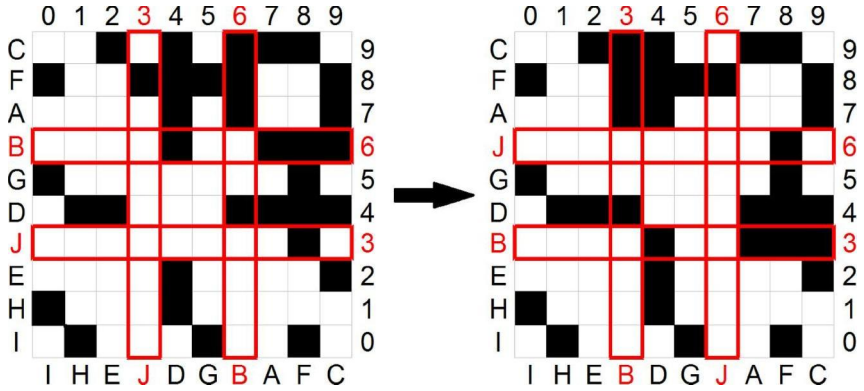


Figura 4.3 - Troca de duas linhas e duas colunas

A função de custo da nova configuração da matriz é calculada (H') e comparada com o valor da configuração anterior (H). Se a diferença entre os custos ($\Delta H = H' - H$) for negativa, a troca das posições é aceita, do contrário, se a diferença for positiva, a troca ainda pode ser aceita com a probabilidade $P = exp(-\frac{\Delta H}{T})$, onde T representa a temperatura, parâmetro utilizado pelo algoritmo *Simulated Annealing* [21], aplicado para evitar mínimos locais, permitindo que a função custo piore temporariamente, mas posteriormente encontre o mínimo global.

A temperatura inicialmente é mantida constante, enquanto que o processo de sorteios e trocas é repetido. Ao longo do tempo, T é reduzida em intervalos constantes (τ) conforme o fator de resfriamento (γ) até o ponto de ser quase nula. Ao final da execução obtém-se uma configuração que se aproxima do custo mínimo, ou seja, quando o sistema estabiliza.

O pseudocódigo (Algoritmo 1) ilustra os passos do algoritmo CFM descritos até o momento. Além disso, é possível observar a existência de dois laços, onde o mais interno é executado conforme o número de nodos do grafo, e o mais externo conforme o número de passos (*steps*) definidos na inicialização do algoritmo, e serve de condição de parada da execução.

Algoritmo 1 Pseudocódigo CFM

```
1: n ← número de nodos
2: M ← matriz de adjacência com distribuição aleatória
3: H ← custo inicial da matriz
4: T ← temperatura inicial
5: for x = 0 to steps do
6:     for y = 0 to n do
7:         a ← valor aleatório em [n]
8:         b ← valor aleatório em [n]
9:         M' ← troca(M, a, b)
10:        H' ← novo custo da matriz
11:        ΔH ← diferença entre os custos
12:        if ΔH ≤ 0 then
13:            H ← H'
14:            M ← M'
15:        else if Probabilidade exp(−ΔH/T) then
16:            H ← H'
17:            M ← M'
18:        end if
19:    end for
20:    if Atingiu intervalo de resfriamento then
21:        T ← resfriamento
22:    end if
23: end for
```

A Figura 4.4(a) ilustra a configuração inicial da matriz gerada a partir do posicionamento aleatório das proteínas, e a Figura 4.4(b) ilustra a configuração final da matriz. Neste exemplo, o algoritmo CFM foi executado sobre uma rede gerada aleatoriamente e com distribuição uniforme, ou seja, não representa uma rede de interação proteica real. Com um total de 500 nodos, o grafo

produz uma matriz de dimensões 500x500. O algoritmo executou 500 iterações até produzir o resultado final.

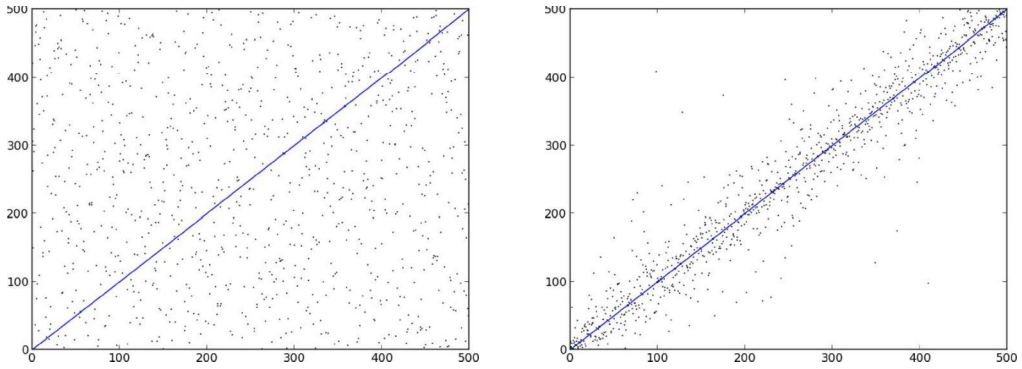


Figura 4.4 - Configuração inicial da matriz (a) e após o ordenamento (b). Os pontos representam interação entre as proteínas.

5. OTIMIZAÇÃO DO MÉTODO E ANÁLISE

Este capítulo descreve inicialmente as otimizações aplicadas ao algoritmo CFM, com o objetivo de reduzir a complexidade e consequentemente levar a redução do tempo de computação do algoritmo. Após a descrição das otimizações e da análise de complexidade do algoritmo CFM, são abordadas e incorporadas ao algoritmo CFM três técnicas encontradas na literatura, gerando três novas versões algoritmos CFM. Essas técnicas somam-se ao CFM para produzir ordenamentos com características diferentes, e possivelmente melhores que o CFM sozinho.

5.1 Otimizações de Software

Com base na descrição do algoritmo CFM, foi desenvolvido um modelo inicial em linguagem Python. É importante salientar que o intuito atual não é obter o melhor desempenho, mas sim compreender as etapas e os elementos que compõem o algoritmo, permitindo visualizar possíveis otimizações.

O modelo descrito tem como parâmetros de entrada a descrição do grafo que representa a rede proteica, o intervalo de resfriamento (τ), o fator de resfriamento (γ), que é aplicado para redução da temperatura a cada iteração, o valor de α, e o número de passos (*steps*), que determina a condição de parada do algoritmo. Como saída, o algoritmo produz um gráfico do estado inicial da matriz (gerada a partir do grafo), um gráfico para o estado final da matriz, o tempo gasto pelo algoritmo, custo inicial e final da matriz, além do mais importante, que é o ordenamento.

Uma vez finalizada essa primeira fase de descrição do modelo, foi traçado um perfil do algoritmo, identificando pontos onde é despendido maior tempo de processamento. Nessa análise foram identificados dois pontos críticos: o processo de troca das colunas e linhas da matriz e o cálculo do custo da matriz. Embora o modelo desenvolvido em Python permita a análise de melhorias em redes pequenas, foi constatada uma limitação da linguagem para a execução de testes com redes complexas (redes proteicas reais).

Dessa forma, foi necessário desenvolver uma versão do algoritmo em uma linguagem que suportasse a execução do algoritmo com redes reais. Após o desenvolvimento do algoritmo CFM na linguagem C foram aplicadas modificações na versão base do CFM. Baseadas no perfil traçado anteriormente, foram aplicadas alterações na estrutura e na forma de manipulação dos dados.

Para exemplificar as otimizações é utilizada a matriz da Figura 5.1(a) juntamente com as linhas e colunas selecionadas para troca da Figura 5.1(b). As alterações foram aplicada incrementalmente e estão descritas em ordem cronológica.

- **Trocas por ponteiros**

  A primeira abordagem foi reduzir o tempo gasto com a manipulação da matriz, alterada sempre que uma troca de linhas e colunas é necessária. Na versão base, sempre que uma troca é feita,

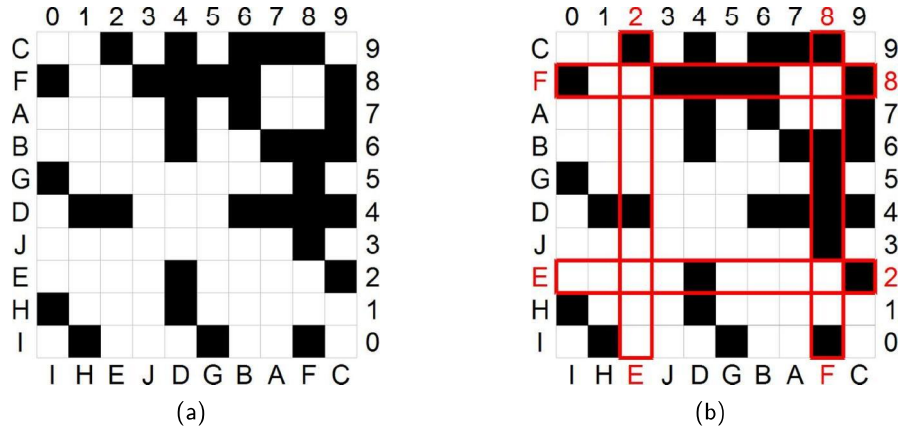


Figura 5.1 - Matriz de adjacência e seleção para troca

os dados das linhas e colunas selecionadas para troca são efetivamente realocados para outra posição. Como a troca é chamado pelo menos uma vez a cada iteração, e é basicamente dependente do número de nodos da rede, o tempo computacional desse processo envolvido é considerável.

Para solucionar esse problema, um vetor descrevendo a posição atual das linhas e colunas foi criado. O vetor basicamente traduz o seu índice, que representa a posição inicial de uma coluna ou linha, e aponta para o índice atualizado, como um ponteiro. Com isso, ao invés de manipular os dados da matriz, as trocas apenas atualizam o índice armazenado no vetor. Por ser um processo rápido, caso o custo obtido com a troca seja superior, o processo de troca é executado novamente sobre os mesmos índices, restaurando assim o estado anterior, ou seja, a matriz não sofre qualquer modificação do início ao fim da execução do algoritmo.

A Figura 5.2 apresenta o estado inicial do vetor produzido a partir do exemplo da Figura 5.1(a), o vetor indicando a seleção antes de troca da Figura 5.1(b) e por fim o vetor com os ponteiros atualizados após o processo de troca. Ao percorrer a matriz, o vetor é utilizado para traduzir a posição atual e traduzir a posição dos vizinhos relativos a posição atual, necessário para o cálculo do custo.

Figura 5.2 - Exemplo de troca com a vetor de ponteiros

- **Cálculo parcial do custo**

A segunda otimização foi aplicada no processo de cálculo do peso, onde originalmente todos os nodos são percorridos e analisados individualmente para obtenção do custo total da matriz. Na análise feita, identificou-se que após a realização das trocas de linhas e colunas, uma parte dos nodos não sofre qualquer alteração no seu custo individual, permitindo a remoção desses nodos do cálculo do novo peso.

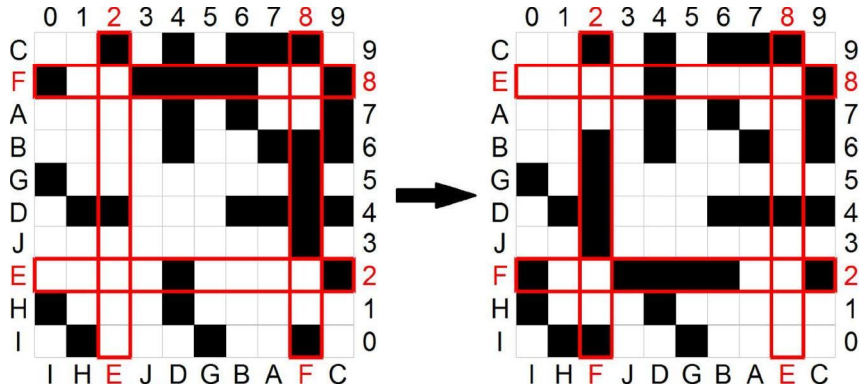


Figura 5.3 - Matriz antes e depois da troca

No início da execução do algoritmo ainda é necessário percorrer todos os nodos para determinar o custo inicial, mas na sequência da execução é possível percorrer parcialmente a matriz. Para o cálculo parcial é preciso determinar o custo dos nodos envolvidos na troca antes e depois da troca. Seguindo o exemplo apresentado anteriormente, a Figura 5.3 mostra o estado da matriz antes da troca e a matriz resultante após a troca. O cálculo parcial considera apenas os vizinhos imediatos de ambas as linhas e colunas envolvidas na troca, ou seja, no caso da coluna 2, além da própria coluna, as conexões das colunas 1 e 3 também devem ser recalculadas.

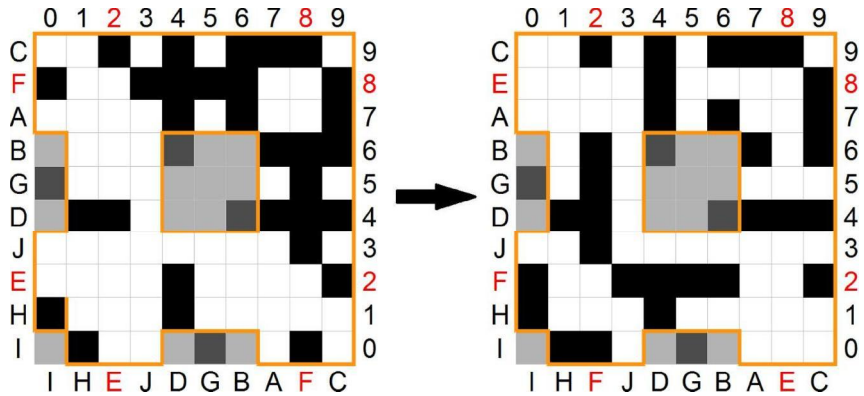


Figura 5.4 - Nodos utilizados no cálculo parcial do custo

Uma vez feito o sorteio dos índices para para troca, a primeira etapa é determinar a contribuição dos nodos das linhas e colunas, juntamente com seus vizinhos imediatos, para o custo total da matriz, chamado de custo parcial (Hp). Em seguida é aplicado o processo de trocas e calcula-se o novo custo parcial (Hp'). A Figura 5.4 ilustra em cinza os nodos desconsiderados

no cálculo parcial. As partes claras correspondem aos nodos das linhas e colunas selecionadas para troca e seus respectivos vizinhos. Estes nodos são percorridos antes e depois de cada troca.

Evidentemente que no exemplo o número de nodos desconsiderados não é grande, e não justificaria as alterações, mas em redes complexas que possuem milhares de nodos e conexões, a quantidade de nodos descartados do cálculo é significativa. Após o cálculo dos dois custos parciais é possível calcular o novo custo total da matriz ($H' = H - Hp + Hp'$).

- **Cálculo da diagonal superior**

A abordagem seguinte faz uso de uma característica da matriz de adjacência que descreve um grafo não-dirigido e que não possui conexões de um nodo com ele mesmo. Nesses casos a matriz possui dados espelhados pela diagonal, é uma matriz simétrica. Haja vista que as informações descritas acima e abaixo da diagonal são redundantes, é possível desconsiderar uma das metades no cálculo do custo. Ao final do cálculo do peso o resultado pode ser multiplicado por dois, gerando assim o mesmo resultado obtido sem essa otimização.

Na Figura 5.5 é possível observar em cinza os nodos não são percorridos pelo procedimento de cálculo do custo da matriz. É importante salientar que o exemplo também incorpora o cálculo parcial do custo, e em função disso pode-se perceber que nodos abaixo da diagonal estão sendo desconsiderados.

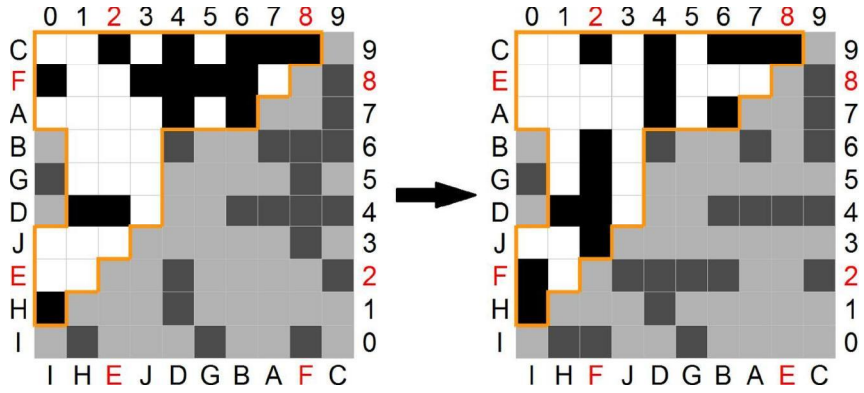


Figura 5.5 - Nodos utilizados no cálculo da diagonal superior

- **Nova representação da matriz**

A última otimização, tira proveito de uma propriedade das redes livres de escala, presente nas redes utilizadas pelo algoritmos CFM. Como demonstrado na Seção 3.2.1, as redes proteicas possuem um número de interações entre proteínas bastante superior ao número total de proteínas da rede. Traduzindo isso para a representação de uma matriz de interação, obtém-se uma matriz com característica esparsa. Como observado na Seção 3.2.1, a maioria das proteínas está conectada a um número reduzido de outras proteínas, abaixo de uma centena na maior parte dos casos. Na rede do *Homo sapiens*, a proteína com mais interconectada possui

aproximadamente 800 ligações, que é muito inferior ao total de proteínas (aproximadamente 10000).

O algoritmo CFM, sempre que precisa calcular o custo, percorrer a matriz em busca de nodos que estejam interconectados. Se dois nodos estão conectados, o seu custo é calculado, caso contrário o cálculo não é feito e segue a busca. Em outras palavras, se o nodo possui valor 0 ele é desconsiderado, mas toda vez que o custo é calculado é necessário verificar se existe interação. A busca pelos nodos que interagem consome bastante tempo, quando na verdade já se sabe desde o início da execução todas as conexões.

Para solucionar esse problema, foi incorporada a estrutura de dados do algoritmo CFM uma nova representação para a matriz. A estrutura utilizada foi uma lista de vetores. A Figura 5.6 mostra a lista correspondente a matriz 5.1(a). Somado a essa estrutura foi adicionado um vetor descrevendo o tamanho dos vetores da estrutura, o que permite saber rapidamente quantas posições pode-se percorrer. Por exemplo, o índice 7 possui tamanho 3.

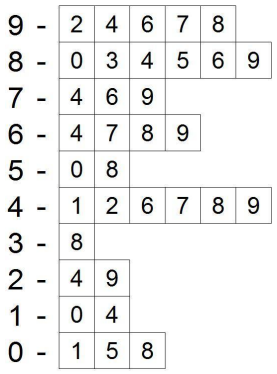


Figura 5.6 - Nova representação da matriz

Em um primeiro momento, o objetivo foi eliminar por completo a matriz de interação, ficando apenas com a nova representação, reduzindo consideravelmente a memória necessária para o armazenamento dos dados. Entretanto, o cálculo do custo com a nova representação elevou a complexidade ao invés de reduzir. Com a nova representação era possível encontrar rapidamente a posição de cada interação, mas identificar se os vizinhos relativos a esse nodos também interagiam com outros nodos não era uma tarefa trivial, sendo necessário incluir funções de busca em vetores vizinhos.

Enquanto que a nova representação permite fácil acesso aos nodos com interação e lenta busca na vizinhança, na representação com a matriz é lento o processo de percorrer a matriz e verificar as interações existentes, mas é fácil a análise dos vizinhos. A solução proposta foi manter as duas representações, utilizando a lista de vetores para percorrer somente os nodos com interação e a matriz para verificar os nodos vizinhos. Claro que essa abordagem precisa de mais memória para armazenamento dos dados, mas como será visto mais adiante, esse acréscimo é justificado pelo ganho de desempenho.

É importante lembrar que as modificações descritas neste item somam-se às outras otimizações descritas anteriormente. A otimização de troca por ponteiros permite que a lista de vetores, assim como a matriz, permaneça inalterada durante toda a execução do algoritmo. Porém, foi necessário incluir um segundo vetor de ponteiros, indicando o índice inicial relativo ao índice atual. A Figura 5.7 ilustra uma sequência de trocas dos valores sorteados para k_1 e k_2 sobre os vetores de ponteiros.

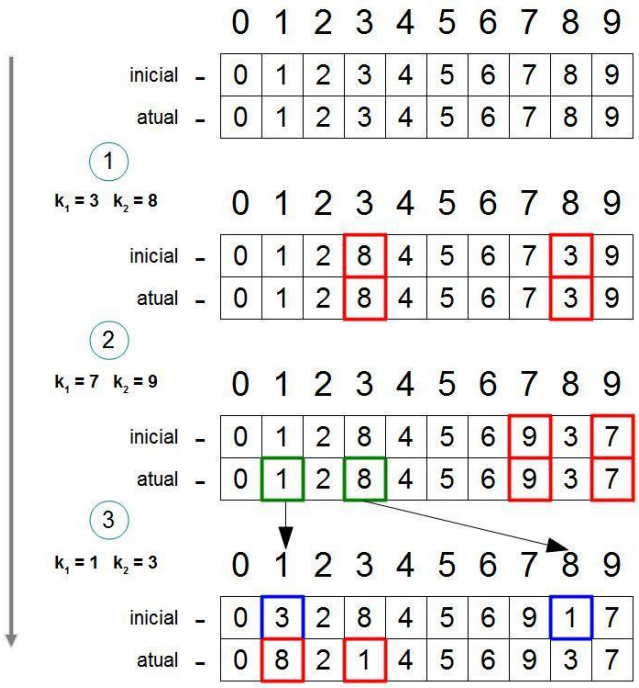


Figura 5.7 - Sequência de trocas sobre os vetores de ponteiros

O vetor de ponteiros **atual** troca de maneira direta os valores armazenados nos índices correspondentes as linhas e colunas selecionadas para troca. Após o instante **1** da Figura 5.3 é possível observar que os valores armazenados nos índices sorteados para troca ($k_1 = 3$ e $k_2 = 8$) foram trocados no vetor **atual**. Já o o vetor de ponteiros **inicial** seleciona os índices traduzidos previamente pelo vetor **atual**, só então os valores armazenados nessas posições são trocados. Em outras palavras, o processo de atualização do vetor **inicial** consulta o vetor **atual** para saber em quais posições deve ser feita a troca.

Na troca ilustrada pelo instante **3** da Figura 5.3, onde $k_1 = 1$ e $k_2 = 3$, fica mais evidente como o processo de troca do vetor **inicial** é feito. Em verde, são os valores utilizados como índice para atualização do vetor *inicial*. As setas apontam para os índices que foram alterados no vetor **inicial**. É possível verificar os valores modificados destacados em azul. Salienta-se que esse processo também ocorre nos instantes **1** e **2**, mas não fica tão evidente, pois os índices selecionados para troca são iguais aos valores armazenados.

Para a correta atualização do vetor **inicial** é preciso cuidar a ordem na qual os vetores são atualizados. A troca dos valores do vetor *atual* só pode ocorrer após a modificação do vetor

inicial. Como já foi dito anteriormente, o processo de troca do vetor **inicial** faz uma consulta ao vetor **atual**. Observando novamente a troca no instante **3** da Figura 5.3, enquanto que k_1 manteve o mesmo índice para troca, k_2 passou a apontar para o índice 8 ao invés do índice 3. Se o vetor **atual** fosse modificado antes do vetor **inicial**, esse processo utilizaria índices inconsistentes.

Através desses vetores de ponteiros é possível traduzir os índices da nova representação, identificando corretamente os índices dos vizinhos que devem ser observados na matriz. Com isso as estruturas da matriz e a lista de vetores mantêm a sua configuração inicial, e permanecem inalteradas durante toda a execução do algoritmo.

5.1.1 Análise de Complexidade

Nesta seção é analisada a complexidade do algoritmo CFM e as otimizações descritas na seção anterior. A partir do CFM base foram desenvolvidos inicialmente três algoritmos (*pointer_cfm*, *partial_cfm* e *diagonal_cfm*), introduzindo individualmente as três primeiras otimizações. Em um segundo momento as três otimizações produziram um quarto algoritmo (*gold_cfm*), e só então foi desenvolvido o algoritmo que incorpora todas as melhorias (*platinum_cfm*). A Tabela 5.1 relaciona o nome dos algoritmos desenvolvidos e as otimizações (Seção 5.1) que cada um possui.

Tabela 5.1 - Relação de algoritmos e otimizações

Nome	**Otimização (ões)**
pointer_cfm	Trocas por ponteiros (TP)
partial_cfm	Cálculo parcial do custo (CPC)
diagonal_cfm	Cálculo da diagonal superior (CDS)
gold_cfm	TP, CPC e CDS
platinum_cfm	TP, CPC e CDS e Nova representação da Matriz

- **Algoritmo CFM base**

  O primeiro passo para a análise de complexidade é determinar a complexidade do algoritmo CFM base, sendo este a referência para comparação com as novas implementações do CFM. A análise do algoritmo CFM base será feita levando em conta as três funções, troca de linhas e colunas (*swap*), cálculo do peso (*getMatWeight*) e o laço principal, que somadas, caracterizam a complexidade do algoritmo.

  No decorrer das análises *n* corresponde ao tamanho da rede proteica. O código fonte da Figura 5.8 foi extraído do CFM base e corresponde à função de trocas. Dependente do número de nodos (*n_nodes*) da rede proteica, o laço itera sobre as posições de memória da matriz que devem ter seus valores trocados. A função *swap* possui complexidade $O(n)$.

  A segunda função analisada é a de cálculo do peso. Também retirada do algoritmo CFM base, a função apresentada na Figura 5.9 percorre todas as posições da matriz, verificando quais

```
1   void swap(char *matrix[], short a, short b){
2     for(x=0;x<n_nodes;x++){
3       if((x != a) && (x != b)){
4         aux = matrix[a][x];
5         matrix[a][x] = matrix[b][x];
6         matrix[b][x] = aux;
7
8         aux = matrix[x][a];
9         matrix[x][a] = matrix[x][b];
10        matrix[x][b] = aux;
11      }
12    }
13    aux = matrix[a][b];
14    matrix[a][b] = matrix[b][a];
15    matrix[b][a] = aux;
16  }
```

Figura 5.8 - Código fonte da função *swap*

nodos possuem interação. Quando necessário analisa-se a vizinhança para o cálculo do custo. Com dois laços encadeados, ambos dependentes de *n_nodes*, a função *getMatWeight* tem complexidade $O(n^2)$.

```
1   double getMatWeight(char *matrix[]){
2     double weight = 0;
3     for(row=(n_nodes-1);row>=0;row--){
4       row_weight = 0;
5       for(col=0;col<n_nodes;col++){
6         if(matrix[row][col] == 1){
7           neighbors = 0;
8           if(col != 0)
9             neighbors = neighbors + (1 - matrix[row][col-1]);
10          if(col != (n_nodes-1))
11            neighbors = neighbors + (1 - matrix[row][col+1]);
12          if(row != 0)
13            neighbors = neighbors + (1 - matrix[row-1][col]);
14          if(row != (n_nodes-1))
15            neighbors = neighbors + (1 - matrix[row+1][col]);
16          if(row < col)
17            abs = col - row;
18          else
19            abs = row - col;
20          row_weight = row_weight + (pow(abs,alpha) * neighbors);
21        }
22      }
23      weight = weight + row_weight;
24    }
25    return weight;
26  }
```

Figura 5.9 - Código fonte da função *getMatWeight*

Por fim, o laço principal do algoritmo CFM base foi analisado. A Figura 5.10 mostra o trecho de código que controla o sorteio dos nodos para troca, e verifica se o novo custo deve ou não ser aceito. O laço mais externo controla a condição de parada do algoritmo (*steps*), passado como parâmetro de entrada. O laço mais interno depende do número de nodos da rede.

Levando em conta que em cada iteração a função de troca pode ser chamada até duas vezes e a função de cálculo do peso é chamada uma vez, e utilizando *m* para representar o número de passos, chega-se a Equação 5.1. Conforme as regras simplificação da notação *Big-Oh* [2], onde as constantes não importam e o termo de maior ordem prevalece, é possível afirmar que a complexidade do algoritmo CFM base é $O(mn^3)$.

$$f(n) = m.(n.(2n + n^2)) = mn^3 + 2mn^2 \tag{5.1}$$

```
1   for(x=0;x<steps;x++){
2     for(y=0;y<n_nodes;y++){
3       /* get the next random nodes  */
4       a = (int)(rand()/(RAND_MAX + 1.0) * n_nodes);
5       b = (int)(rand()/(RAND_MAX + 1.0) * n_nodes);
6       /* skip swap and test if a == b  */
7       while(a == b)
8         b = (int)(rand()/(RAND_MAX + 1.0) * n_nodes);
9       /* swap chosen nodes  */
10      swap(matrix, a, b);
11      /* calculate new weight  */
12      new_weight = getMatWeight(matrix);
13      /* generate delta  */
14      delta_weigth = new_weight - curr_weight;
15      /* if the new weight is better accept and assign  */
16      if(delta_weigth <= 0){
17        curr_weight = new_weight;
18      }else{
19        /* if new weight is worse accept but with a probability level  */
20        expo = expl((long double)-delta_weigth/(long double)temperature);
21
22        if(expo > ((long double)rand()/(long double)RAND_MAX)){
23          curr_weight = new_weight;
24        }else{
25          swap(matrix, a, b);
26        }
27      }
28    }
29    /* cooling process  */
30    if(steps%cooling_interval == 0)
31      temperature = temperature * cooling_factor;
32  }
```

Figura 5.10 - Código fonte do laço principal

- **Algoritmo *pointer_cfm***

O algoritmo *pointer_cfm* incorpora a otimização de trocas por ponteiros apresentada na Seção 5.1. A função de trocas e a função cálculo do custo foram modificadas. No caso da função de trocas do algoritmo *pointer_cfm*, apresentada na Figura 5.11, o laço que antes era necessário para manipular os nodos da matriz foi removido, e substituído pelas trocas sobre o vetor de ponteiros, reduzindo sua complexidade para $O(1)$. Já a função de cálculo do custo foi adaptada para verificar as posições da matriz com base na tradução da posição atual obtida com o vetor de ponteiros, mas sua complexidade não é alterada, visto que ainda precisa percorrer todas as posições da matriz. Em relação ao laço principal, esse não sofreu nenhuma alteração. Mesmo com a redução da complexidade da função de troca, a complexidade do algoritmo *pointer_cfm* permanece a mesma do algoritmo CFM base.

```
1   void swap(graph_ptr *order, short a, short b){
2     aux_swap = order[a].cur;
3     order[a].cur = order[b].cur;
4     order[b].cur = aux_swap;
5   }
```

Figura 5.11 - Código fonte da função *swap* extraído do algoritmo *pointer_cfm*

- **Algoritmo *diagonal_cfm***

Na análise do algoritmo *diagonal_cfm*, apenas a função de cálculo do peso foi alterada. A Figura 5.12 mostra a função modificada. O laço mais interno percorre as linhas da matriz até atingir a diagonal, reduzindo pela metade o número de posições percorridas. Entretanto, a notação *Big-Oh* desconsidera constantes, nesse caso a divisão por dois do número de nodos que devem ser percorridos, permanecendo assim inalterada a complexidade da função *getMatWeight* e do algoritmo.

```
1   double getMatWeight(char *matrix[]){
2     double weight = 0;
3     for(row=(n_nodes-1);row>=0;row--){
4       row_weight = 0;
5       for(col=0;col<row;col++){
6         if(matrix[row][col] == 1){
7           neighbors = 0;
8           if(col != 0)
9             neighbors = neighbors + (1 - matrix[row][col-1]);
10          if(col != (n_nodes-1))
11            neighbors = neighbors + (1 - matrix[row][col+1]);
12          if(row != 0)
13            neighbors = neighbors + (1 - matrix[row-1][col]);
14          if(row != (n_nodes-1))
15            neighbors = neighbors + (1 - matrix[row+1][col]);
16
17          abs =pow((row - col),alpha);
18          row_weight = row_weight + (abs * neighbors);
19        }
20      }
21      weight = weight + row_weight;
22    }
23    return weight*2;
24  }
```

Figura 5.12 - Código fonte da função *getMatWeight* extraído do algoritmo *diagonal_cfm*

Também é possível observar que a função passa a retornar o custo multiplicado por dois. Essa operação é necessária para a comparação do custo final obtido com os algoritmos. Também poderia ser aplicada somente ao final da execução do algoritmo, apenas para fins de comparação, reduzindo o tempo de computação da função, o que não altera a análise de complexidade da função *getMatWeight*.

- **Algoritmo *partial_cfm***

  O cálculo parcial do peso envolve a modificação da função do cálculo do custo e do laço principal do algoritmo. Em função do tamanho, o código fonte da função do cálculo parcial do custo (*getPartWeight*) pode ser visto no APÊNDICE A.

  No algoritmo *partial_cfm* a função do cálculo parcial está dividida em quatro partes: duas responsáveis pelas linhas modificadas pela troca e duas para as colunas alteradas. Cada parte possui dois laços, o mais externo depende do número de linhas ou colunas vizinhas que precisam ser percorridas. No pior dos casos esse laço precisa percorrer três linhas ou colunas: a vizinha da esquerda, a vizinha da direita e a que foi efetivamente trocada. O laço mais externo ainda depende do tamanho da rede, portanto, as quatro partes percorrem no máximo doze vezes a quantidade de nodos que a rede possui, que resulta na complexidade $O(n)$.

  A Figura 5.13 apresenta o laço principal do algoritmo *partial_cfm*. Como já foi dito na descrição da otimização do cálculo parcial, é preciso fazer a chamada da função do custo parcial em dois momentos, antes e depois da troca. Com as duas chamadas da função de troca do pior caso, e as duas chamadas do cálculo parcial, é possível encontrar a Equação 5.2. Simplificando para a notação *Big-Oh* ($O(mn^2)$) observa-se a redução de uma ordem na complexidade em relação ao algoritmo CFM base.

$$f(n) = m.(n.(2n + 2n)) = 4mn^2 \tag{5.2}$$

```
1   for(x=0;x<steps;x++){
2     for(y=0;y<n_nodes;y++){
3       /* get the next random nodes  */
4       a = (int)(rand()/(RAND_MAX + 1.0) * n_nodes);
5       b = (int)(rand()/(RAND_MAX + 1.0) * n_nodes);
6       /* skip swap and test if a == b  */
7       while(a == b)
8         b = (int)(rand()/(RAND_MAX + 1.0) * n_nodes);
9       /* calculate current partial weight  */
10      curr_part_weight = getPartWeight(matrix, a, b);
11      /* swap chosen nodes  */
12      swap(matrix, a, b);
13      /* calculate new partial weight  */
14      new_part_weight = getPartWeight(matrix, a, b);
15      /* calculate new weight  */
16      new_weight = curr_weight - curr_part_weight + new_part_weight;
17      /* generate delta  */
18      delta_weigth = new_weight - curr_weight;
19      if(delta_weigth <= 0){
20        curr_weight = new_weight;
21      }else{
22        /* if new weight is worse accept but with a probability level  */
23        expo = expl((long double) - delta_weigth/(long double)temperature);
24
25        if(expo > ((long double)rand()/(long double)RAND_MAX)){
26          curr_weight = new_weight;
27        }else{
28          swap(matrix, a, b);
29        }
30      }
31    }
32    /* cooling process  */
33    if(steps%cooling_interval == 0)
34      temperature = temperature * cooling_factor;
35  }
```

Figura 5.13 - Código fonte do laço principal em *partial_cfm*

- **Algoritmo *gold_cfm***

  O *gold_cfm* é o algoritmo que incorpora as três otimização presentes nos algoritmos descritos nas subseções anteriores. Como os algoritmos *pointer_cfm* e *diagonal_cfm* não possuem complexidade menor que o algoritmo CFM base, a complexidade do *gold_cfm* é regida pela otimização do cálculo parcial do custo. Dessa forma a complexidade é a mesma do algoritmo *partial_cfm*, $O(mn^2)$.

- **Algoritmo *platinum_cfm***

  O último algoritmo desenvolvido tem como base o algoritmo *gold_cfm*, acrescentando a este a nova representação da matriz. Embora grande parte do código tenha sido alterado para suportar as novas estruturas (lista de vetores, vetor de ponteiros inicial e vetor com tamanho dos vetores da lista), a principal alteração está no cálculo parcial do custo (APÊNDICE B).

  A estrutura da função de cálculo parcial permanece a mesma do algoritmo *partial_cfm*, com o cálculo dividido em quatro etapas: duas para o cálculo das linhas e duas para o cálculo das colunas. Mas, ao invés do laço mais interno depender do número total de nodos da rede proteica, ele passa a depender da quantidade de ligações existentes entre a proteína que a linha ou coluna representa e as demais proteínas. Em outras palavras, o laço mais interno percorre somente as posições da linha ou coluna que tiverem valor não zero (*nz*).

  Sabe-se que as redes utilizadas nesse trabalho produzem matrizes de interação esparsas (Seção 3.2), de modo que a grande maioria dos nodos possuem poucas conexões. Sendo assim, a complexidade da função de custo parcial passa a ser determinada por *nz*, e não mais *n*, resultando na complexidade $O(nz)$. Por fim, refazendo a análise que leva em conta todas as

otimizações, o algoritmo *platinum_cfm* possui complexidade $O(m.n.nz)$, onde tipicamente $nz << n$.

5.1.2 Testes Práticos de Desempenho

Os testes para comparação dos algoritmos com relação ao tempo de execução foram realizados em um computador com processador Intel Core i5-2450M 2.50MHz, 6GB de memória RAM e rodando sistema operacional Linux Ubuntu 12.04 64 bits. A tomada do tempo de execução de cada algoritmo analisado foi feita durante a execução do laço principal, e encerrada assim que a condição de parada fosse atingida.

Para os testes foi utilizada a rede proteica do *Homo sapiens* extraída do banco de dados STRING versão 8.2 com *score* $0,8$, possui 8815 proteínas e 138568 interações. O intervalo de resfriamento foi fixado em 100 e o fator de resfriamento foi configurado em $0,5$. A temperatura inicial corresponde a $0,01\%$ do custo inicial da rede e *alpha* é igual a 1. A semente do gerador de números aleatórios foi mantida a mesma em todas as execuções, produzindo sempre a mesma sequência de valores sorteados, de modo que todos os algoritmos consigam o mesmo ordenamento final.

Tabela 5.2 - Tempo de execução em segundos dos algoritmos em função do número de passos

	Algoritmos				
Passos	**cfm**	**diagonal_cfm**	**partial_cfm**	**gold_cfm**	**platinum_cfm**
1	1965,713	912,513 (2,154)	15,204 (129,289)	2,886 (681,120)	0,195 (10080,579)
2	3933,853	1824,956 (2,156)	31,427 (125,174)	5,763 (682,605)	0,409 (9618,222)
3	5896,279	2706,856 (2,178)	47,479 (124,187)	8,639 (682,519)	0,611 (9650,211)
4	7960,454	3591,725 (2,216)	63,461 (125,439)	11,529 (690,472)	0,844 (9431,818)
5	9822,521	4496,265 (2,185)	79,882 (122,963)	14,443 (680,089)	1,293 (7596,691)
100	-	-	-	288,369	20,826
3000	-	-	-	8664,24	632,708
10000	-	-	-	28880,829	2090,855

A comparação dos tempos foi feita variando o número de passos de 1 até 5 para todos os algoritmos. Na Tabela 5.2 são listados os resultados obtidos com cada algoritmo, e a Figura 5.14 ilustra graficamente os tempos de execução (eixo em escala logarítmica) em relação ao número de passos. O valor entre parênteses representa o *speedup* em relação ao algoritmo CFM base (*cfm*).

O algoritmo *pointer_cfm* não consta nos resultados, pois o tempo de execução foi pior que o algoritmo CFM base. Vale lembrar que a complexidade desse algoritmo é a mesma que do CFM base, mas o ganho de tempo obtido com a simplificação da função de troca foi inferior ao tempo necessário para a tradução das posições da matriz durante o cálculo do custo. A utilização das trocas por ponteiros só contribui para a redução do tempo de execução se associada à otimização do cálculo parcial.

Observa-se que, mesmo possuindo complexidades iguais, o desempenho do algoritmo *diagonal_cfm* é superior *cfm*, já que percorre apenas metade das posições da matriz. O restante dos

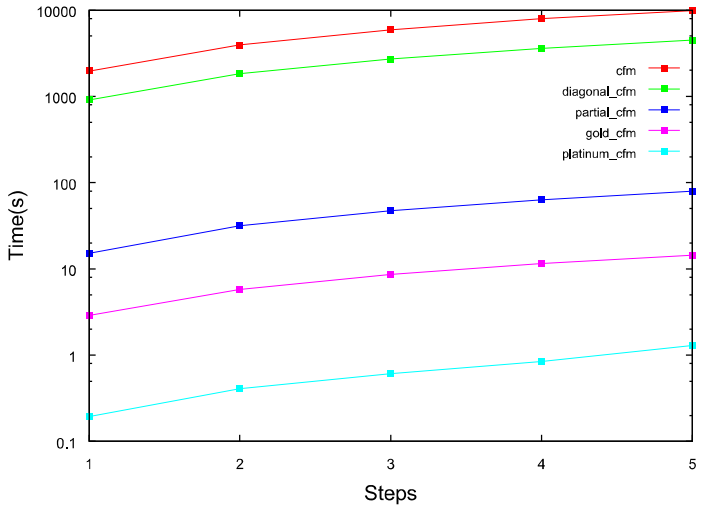


Figura 5.14 - Tempo de execução das diferentes versões do CFM

resultados reflete no tempo de execução a redução da complexidade. O algoritmo *platinum_cfm* é 7598 vezes mais rápido que o *cfm*. Para os algoritmos *gold_cfm* e *platinum_cfm*, a Tabela 5.2 apresenta os resultados para número de passos maiores, e ilustra o ganho computacional dessas implementações. Curiosamente, se o *cfm* fosse executar os mesmos 10000 passos, este levaria aproximadamente 228 dias, e o tempo necessário para computar 10000 passos com o *platinum_cfm* é equivalente a 1 passo do *cfm*.

5.2 Ordenamento Espectral e Reaquecimento

A análise de complexidade e os testes práticos mostraram a eficiência das otimizações aplicadas ao algoritmo *cfm* base no que diz respeito ao tempo computação. A redução obtida com o algoritmo *platinum_cfm* permite a avaliar diferentes cenários de teste, antes limitada pelo longo período de execução. Outra vantagem é a avaliação de novas versões do *platinum_cfm*, que modificam a lógica, inserindo novos estágios ou heurísticas ao fluxo do algoritmo. Nesta seção serão propostas duas modificações para *platinum_cfm*, produzindo três novas versões, mas antes é preciso observar como o *platinum_cfm* se comporta durante a sua execução. Nas discussões e comparações que se seguem, sempre que for feita uma referência ao algoritmo *cfm*, trata-se da versão mais otimizada (*platinum_cfm*).

Durante a execução do algoritmo *cfm* sobre o conjunto de dados da rede do *Homo sapiens* foram realizadas medições do custo, quantidade de trocas aceitas e a temperatura ao longo de 3000 iterações. Na Figura 5.15 pode-se observar que a principal redução do custo ocorre até a iteração 1500, quando a quantidade trocas aceitas praticamente se estabiliza. Em testes com número maior de passos, a redução do custo é imperceptível graficamente. Nas subseções a seguir serão descritas

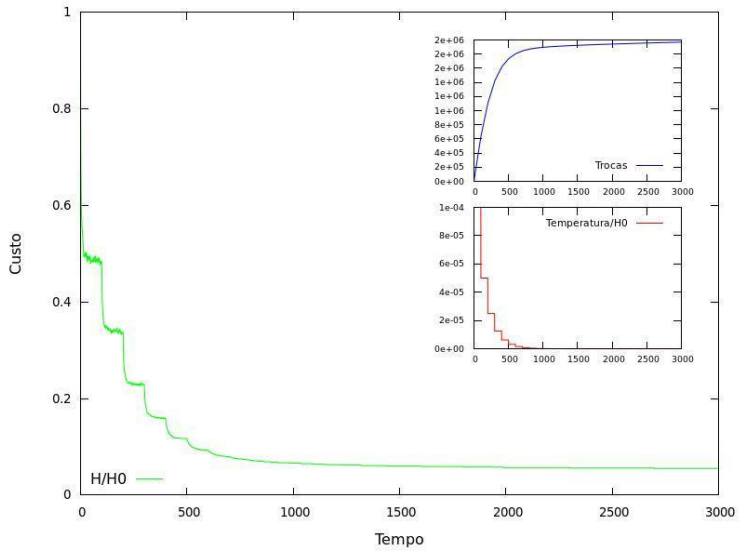


Figura 5.15 - Medições do custo (verde), do número de trocas (azul) e da temperatura (vermelho) ao longo da execução do algoritmo *platinum_cfm*

duas novas abordagens para a o algoritmo *cfm*, que levam em conta resultados observados com as medições.

5.2.1 Reaquecimento

Uma técnica seguidamente associada a problemas que trabalham com *Simulated Annealing* é o reaquecimento. A ideia básica é que, uma vez que o *Simulated Annealing* atinge uma temperatura muito baixa, ele tem dificuldades para fugir do mínimo local, já que a a probabilidade de aceitar soluções que piorem momentaneamente o custo é muito baixa. Com o reaquecimento, a temperatura é elevada, possibilitando novamente ao algoritmo fugir do mínimo local. Vários autores propuseram sistemas de resfriamento que aplicam o reaquecimento [4] [24] [1] [12].

O algoritmo *cfm* proposto por [28] não aplica a técnica de reaquecimento, entretanto, com base no comportamento observado na Figura 5.15, o presente trabalho propõe a inclusão de uma técnica de reaquecimento ao sistema de resfriamento do algoritmo *cfm*. Algumas técnicas de reaquecimento existentes foram analisados, até que o reaquecimento utilizado por [35] foi selecionado. A escolha foi baseada na simplicidade da equação de reaquecimento e por ser de fácil inserção na estrutura do algoritmo *cfm*.

O reaquecimento em questão é definido pela Equação 5.3. A temperatura de reinício T_r inicialmente é igual a temperatura inicial e é aplicada em intervalos definidos na inicialização do algoritmo. Quando o intervalo de reaquecimento é atingido, o maior valor entre a metade da temperatura de reinício e a temperatura atual T é atribuída ao sistema, e passa a ser a referência para o próximo intervalo de reaquecimento.

$$T_r = max(\frac{T_r}{2}, T) \tag{5.3}$$

Poucas foram as modificações no código do algoritmo *cfm*, e sua complexidade não foi alterada. O novo algoritmo passa a ser chamado de *reheat*. A Figura 5.16 apresenta as medições como o novo algoritmo. A diferença para o teste apresentado na Figura 5.15, é a aplicação do reaquecimento em dois momentos. Em todos os gráficos fica bastante evidente o momento em que o sistema é aquecido, o que eleva o número de trocas aceitas.

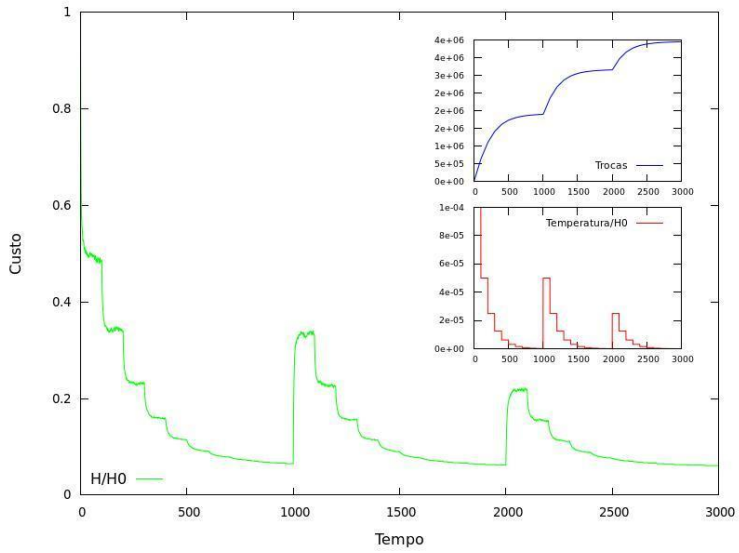


Figura 5.16 - Medições do custo (verde), do número de trocas (azul) e da temperatura (vermelho) ao longo da execução do algoritmo *platinum_cfm*

A definição do intervalo foi baseado nas medidas feitas com o algoritmo *cfm*. Estudos relativos a escolha do melhor intervalo não foram alvos do presente trabalho, mas certamente deve ser levado em conta futuramente. Nesse momento o desempenho do algoritmo *reheat* não será avaliado, ficando essa análise para o capítulo 6.

5.2.2 Ordenamento Espectral

Em [26], são abordadas diferentes técnicas para o problema de ordenamento MinLA. Uma das propostas do autor foi combinar os algoritmos *Spectral Sequencing* (SS) e o *Simulated Annealing* (SA). Ao invés de fornecer um ordenamento inicial aleatório ao algoritmo SA, a entrada passa a ser o ordenamento gerado pelo SS. Os resultados mostram que em média a combinação SS+SA consegue atingir custos menores num tempo menor, se comparado ao melhor resultados de ambas as técnicas executadas individualmente.

Ao combinar o SS com o *cfm*, que faz uso do SA, pretende-se verificar se os resultados encontrados em [26] contribuem para um menor custo do ordenamento proteico.

O *Spectral Sequencing*, ou ordenamento espectral, é tipicamente baseado na computação do autovetor correspondente ao segundo menor autovalor da matriz Laplaciana que representa a estrutura do grafo [19]. Esse autovetor também é conhecido como vetor de *Fiedler*. A seguir são descritos os passos envolvidos no ordenamento espectral:

1. Determinar os vértices vizinhos.
2. Montar a matriz de adjacência.
3. Calcular a matriz Laplaciana.
4. Determinar os menores autovalores e autovetores.
5. Encontra e ranquear o vetor de *Fiedler*
6. Utilizar o vetor ranqueado para ordem de inserção dos vértices.

O ranqueamento do vetor de *Fiedler* pode ser feito por um algoritmo de classificação clássico, como o *Quicksort* ou o *Bubblesort*.

Ficou claro nos experimentos com o *cfm* que a maior redução do custo ocorre logo na fase inicial de sua execução, e estabiliza no restante do tempo. Em [26] o SS é utilizado para gerar um ordenamento inicial para o SA. O presente trabalho, além de utilizar a mesma abordagem onde o ordenamento espectral substitui o ordenamento aleatório, também propõe a utilização do ordenamento espectral sobre o ordenamento produzido pelo *cfm*, ou seja, aplicar o ordenamento espectral após o *cfm* estabilizar.

No desenvolvimento do ordenamento espectral foi utilizada a biblioteca *igraph*, disponível em http://igraph.sourceforge.net. Nela estão implementadas as funções de cálculo da matriz laplaciana, dos autovalores e autovetores. O ranqueamento do vetor de *Fiedler* foi feito com a função *qsort* da biblioteca padrão da linguagem C.

A partir desse desenvolvimento foram criadas duas novas versões para o algoritmo *cfm*: o *spectralIni*, que utiliza o ordenamento espectral no lugar do ordenamento aleatório; e o *spectralFinal*, que aplicar o ordenamento espectral após a execução do *cfm*. Como existe a dependência de funções de bibliotecas externas, a análise de complexidade não será apresentada para esses algoritmos, mas é pelo menos igual a complexidade do *platinum_cfm*, já que serviu de base para ambas as implementações.

Os testes práticos dos algoritmos com ordenamento espectral foram feitos com o mesmo cenário e configurações apresentadas na Subseção 5.1.2. Entretanto, o ordenamento espectral não depende dos parâmetros do *cfm*, e o tempo de execução do ordenamento espectral varia conforme o tamanho da rede, nesse caso a rede do *Homo sapiens* que possui 8815 proteínas.

O tempo de computação do ordenamento espectral foi tomado separadamente, sem incluir o tempo de execução correspondente ao *cfm*. Os resultados indicam um elevado tempo de

processamento, sendo necessários aproximadamente 2595 segundos para o ordenamento do algoritmo *spectralFinal* e 2573 segundos com o *spectralIni*. O resultado semelhante é esperado, pois a função de ordenamento é a mesma para os dois algoritmos.

Se comparado com o algoritmo *platinum_cfm*, o tempo de execução do ordenamento espectral é muito superior, sendo equivalente a uma execução de 10000 passos. Esse resultado seria suficiente para descartar essa solução. Entretanto, se comparado ao *cfm* original, esse tempo seria próximo a 1 passo. De fato, o resultado não foi o mesmo obtido por [26], porém, esse comportamento pode ser decorrente de três razões.

A primeira razão diz respeito a características das redes. Analisando mais criteriosamente as redes utilizadas por [26], nenhuma é verdadeiramente livre de escala. Já a segunda refere-se a utilização das funções da bibliotecas *igraph*, que podem não ser tão otimizadas quanto as utilizada por [26]. Por fim, pode ser uma diferença de característica do problema. Mesmo que o MinLA e o CFM sejam ambos problemas de ordenamento, enquanto o CFM busca minimizar o custo através da diagonalização da matriz de interações, o MinLA busca reduzir a soma total das distância entre os nodos, e não a distância dos nodos em relação a diagonal.

Independente dos resultados obtidos até o momento, ainda é preciso analisar os algoritmos com relação ao custo final do ordenamento. Se os algoritmos com ordenamento espectral conseguirem reduzir o custo de forma expressiva em relação ao *cfm*, o tempo de execução maior pode ser justificado por esse ganho. Essa análise é apresentada no capítulo 7.

6. ANÁLISE QUANTITATIVA DO ORDENAMENTO

Neste capítulo, os ordenamentos produzidos pelos algoritmos descritos anteriormente são avaliados quantitativamente com relação ao resultado final da função custo, observando qual o custo obtido com cada algoritmo e os parâmetros de entrada correspondentes a esse resultado. Com essa comparação é possível determinar qual algoritmo (*cfm*, *spectralnit*, *spectralFinal* e *reheat*) atinge o menor custo.

Para a avaliação quantitativa foram mantidos constantes todos os parâmetros de entrada dos algoritmos (Subseção 5.1.2), com exceção do número de passos, que variou entre 3000, 5000 e 10000. Considerando que alto custo computacional dos algoritmos com ordenamento espectral em comparação com o *cfm* (Seção 5.2.2), optou-se por limitar o escopo da comparação quantitativa destes algoritmos, variando somente o número de passos. Outros parâmetros, como temperatura inicial, intervalo de resfriamento e *alpha*, são considerados constantes.

Nos testes de avaliação quantitativa, foi utilizada a rede proteica do *Homo sapiens*. Extraída do banco de dados STRING versão 9.05, foram consideradas associados todo par de proteínas cujo *score* fosse igual ou superior a $0,8$.

Especificamente no caso do algoritmo *reheat*, o parâmetro que define quantos reaquecimentos devem ser aplicados precisa ser configurado. Para todos os testes com esse algoritmo foram aplicados dois reaquecimentos, que elevam a temperatura em períodos onde poucas trocas são aceitas, e a redução do custo é muito pequena. Já o algoritmo *spectrallni* precisa ser iniciado com uma temperatura inicial baixa, caso contrário a solução inicial será destruída pelo *Simulated Annealing*. A temperatura inicial foi a mesma utilizada por [26].

Devido à aleatoriedade utilizada no fluxo dos algoritmos, seria impraticável a produção de resultados para qualquer semente. Entretanto, uma abordagem estatística denominada *sample size estimation* [15] permite determinar o tamanho da amostra que melhor representa a população. Antes de estimar o tamanho da amostra é necessário executar um teste piloto com uma amostra pequena do custo final dos ordenamentos. Com os resultados dessa amostra é possível calcular a média e o desvio padrão do custo. Esses valores são então aplicados para determinar o tamanho da amostra. Para os algoritmos avaliados, o tamanho mínimo da amostra calculado foi 15, sendo esse número de execuções suficiente para representar a média e o desvio padrão das execuções.

Considerando que somente o número de passos é variado e este possui 3 valores. Considerando também que cada configuração deve ser executada 15 vezes. Então, cada um dos algoritmos é executado $3 \times 15 = 45$ vezes. Nas Tabelas 6.1 e 6.2 são apresentados alguns dados estatísticos de cada algoritmo. É possível observar que o desvio padrão (*std*) se manteve praticamente constante em todos os testes, que os melhores resultados foram encontrados com os algoritmos *cfm* e *spectralFinal*, e que as médias foram bastante semelhantes. O ordenamento de menor custo foi obtido com o algoritmo *cfm* em uma das execuções de 10000 passos.

Tabela 6.1 - Resultados estatísticos dos algoritmos *cfm* e *spectralIni*

	cfm			spectralInit		
steps	**3000**	**5000**	**10000**	**3000**	**5000**	**10000**
mean	2,307e+08	2,198e+08	2,097e+08	3,063e+08	2,863e+08	2,783e+08
std	2,964e+08	2,965e+08	2,964e+08	2,966e+08	2,966e+08	2,966e+08
min	2,183e+08	2,064e+08	1,962e+08	2,886e+08	2,647e+08	2,510e+08
max	2,435e+08	2,471e+08	2,231e+08	3,286e+08	3,139e+08	2,962e+08

Tabela 6.2 - Resultados estatísticos dos algoritmos *spectralFinal reheat*

	spectralFinal			reheat		
steps	**3000**	**5000**	**10000**	**3000**	**5000**	**10000**
mean	2,336e+08	2,205e+08	2,093e+08	2,602e+08	2,381e+08	2,184e+08
std	2,965e+08	2,965e+08	2,964e+08	2,967e+08	2,965e+08	2,964e+08
min	2,195e+08	2,044e+08	1,984e+08	2,449e+08	2,235e+08	2,097e+08
max	2,542e+08	2,362e+08	2,199e+08	2,991e+08	2,548e+08	2,324e+08

A Figura 6.1 mostra o gráfico dos menores custos para cada algoritmo. Novamente é possível observar que os algoritmos *cfm* e *spectralFinal* obtiveram resultados semelhantes. Além disso fica evidente que quanto maior o número de passos, menor e o custo final. Nesse momento surgem algumas questões importantes. Um menor custo absoluto significa um melhor ordenamento? O que é um melhor ordenamento?

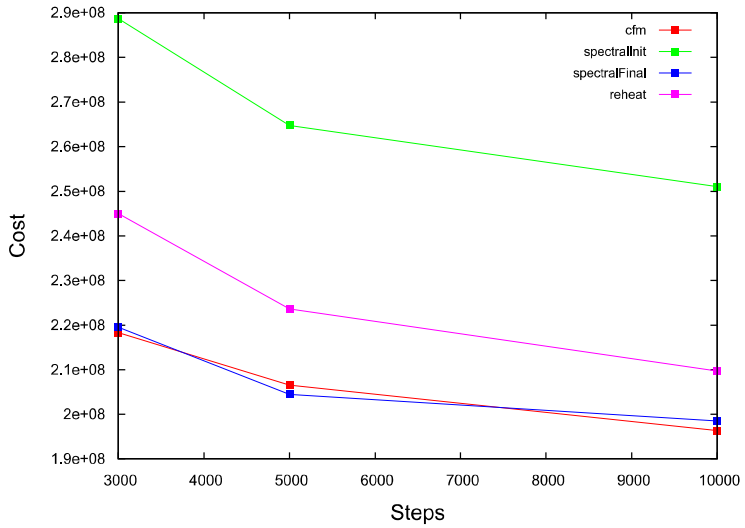


Figura 6.1 - Menor custo obtido com cada algoritmo para 3000, 5000 e 10000 passos

Essas questões permitem que o algoritmo de reaquecimento, e os dois algoritmos com ordenamento espectral ainda não sejam descartados. Amparados pela suposição de que o menor custo define o melhor ordenamento, ainda é preciso avaliá-los utilizando outra métrica. A próxima sessão será dedicada a responder essas questões.

7. ANÁLISE QUALITATIVA DO ORDENAMENTO

Como já foi abordado anteriormente, uma questão importante para ser avaliada é a qualidade do ordenamento, saber se um custo menor corresponde a um melhor ordenamento. A eficiência do Transcriptograma já foi abordada em [11]. Uma das formas de avaliação foi aplicar o método a um problema de diagnóstico, e permite relacionar os parâmetros do método a uma medida de eficiência, neste caso o diagnóstico. Por outro lado, [11] faz novos questionamentos em suas conclusões, sobre como os diferentes parâmetros utilizados pelo algoritmo de ordenamento influenciam no resultado de um diagnóstico.

Para a produção de diagnósticos foi utilizado o método descrito em [11]. Todas as ferramentas e amostras necessárias para o diagnóstico foram fornecidas por [11]. O processo de diagnóstico consiste em, a partir de um ordenamento, gerar o Transcriptograma, aplicar um método de aprendizado de máquina para o diagnóstico de amostras, em específico o método conhecido por LDA *Linear Discriminant Analysis* [14], e por fim avaliar a eficiência do diagnóstico. A eficiência do diagnóstico é obtida com a método CCEM (*Correct Class Enrichment Metric*), e produz um resultado que varia entre 0 e 1, sendo 0 a ausência e 1 a certeza da existência de uma determinada doença na amostra, juntamente com o desvio padrão associado a esse resultado. Detalhes sobre a implementação dessas etapas fogem ao escopo desse trabalho e podem ser encontradas em [11].

O presente trabalho utiliza o diagnóstico da Esclerose Múltipla como medida de qualidade dos ordenamentos e avaliação de como alguns parâmetros dos algoritmos abordados anteriormente influenciam no diagnóstico. A esclerose múltipla é uma doença autoimune que afeta o sistema nervoso central. As amostras foram obtidas de [10], experimento cadastrado no ArrayExpress sob o código E-MATB-69, com 26 amostras de pacientes com esclerose múltipla, sendo 12 em fase recidiva e 14 em fase remitente, e 18 amostras controle.

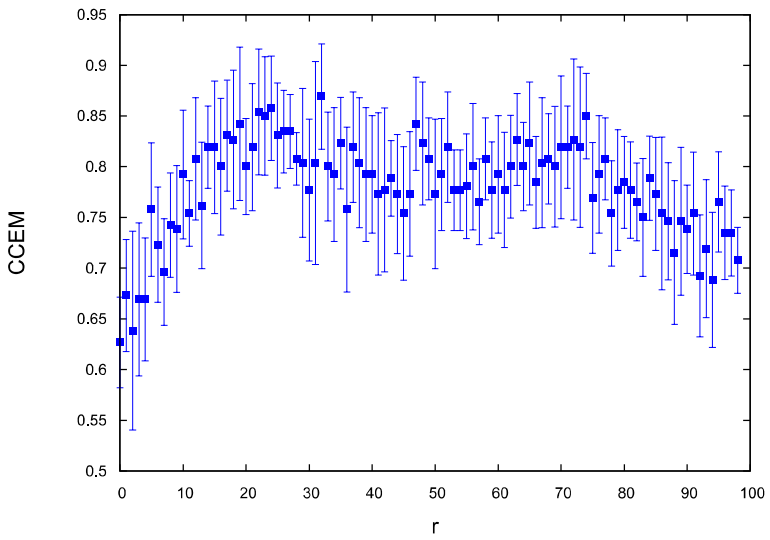


Figura 7.1 - Conjunto de diagnósticos gerados em função *r*

Os diagnósticos gerados diferenciam pacientes com esclerose múltipla em fase recidiva e remitente. A Figura 7.1 exemplifica a eficiência do diagnóstico, utilizanddo um Transcriptograma gerado a partir de um ordenamento qualquer da rede proteica do *Homo sapiens*. No gráfico estão os diagnósticos em função do raio *r*. O raio *r* define qual o tamanho da região do ordenamento utilizada para as médias do Trancriptograma. Além do diagnóstico resultante para cada raio, o gráfico mostra o erro (desvio padrão) de cada medida. A análise do raio que apresenta o melhor diagnóstico está fora do escopo desta dissertação, pois o raio não afeta o ordenamento, mas sim o método de diagnóstico que não é o foco desta dissertação.

7.1 Método de Avaliação dos Resultados

Nas subseções a seguir serão analisadas a eficiência dos algoritmos de ordenamento, a relação existente entre os parâmetros α, número de passos e *score* da rede proteica, e a qualidade do diagnóstico. Mas antes é preciso definir um valor de referência para essa comparação, permitindo assim descartar todos aqueles diagnósticos inferiores a essa medida, restando somente uma quantidade reduzida de diagnósticos que evidencia qual algoritmo ou parâmetro tem o potencial para gerar melhores diagnósticos.

Para determinar um valor de referência foram utilizados diagnósticos obtidos a partir de 45 ordenamentos aleatórios. Utilizar diagnósticos gerados a partir de ordenamentos aleatórios é uma foma de verificar que os algoritmos de ordenamento efetivamente contribuem para um melhor diagnóstico.

O teste de normalidade *Shapiro–Wilk* [32] foi utilizado para analisar o conjunto de diagnósticos gerados a partir destes 45 ordenamentos aleatórios. O *software* Minitab foi utilizado para esta análise. Segundo este método, se $p > 0,05$ a distribuição pode ser considera normal. Para os 45 diagnósticos $p = 0,154$, portanto, os diagnósticos seguem uma distribuição normal. A Figura 7.2 apresenta o gráfico da distribuição.

Dessa forma, o valor de referência foi calculado com base na média dos 45 melhores diagnósticos, somado a ao valor de dois desvios padrões que, segundo as propriedades de uma distribuição normal, $95,44\%$ dos dados estão compreendidos nesse intervalo. Como o índice CCEM médio foi de $0,866$ e o desvio padrão foi de $0,033$, o ponto de corte fica em $0,933$. Ou seja, um índice CCEM maior que $0,933$ se destaca estatisticamente da média dos diagnósticos que utilizaram um ordenamento aleatório.

7.2 Eficiência dos Algoritmos de Ordenamento

A primeira fase de testes foi feita para avaliar e comparar a eficiência dos ordenamentos produzidos pelos algoritmos de ordenamento. Embora os algoritmo com ordenamento espectral (*spectralInit* e *spectralFinal*) e o algoritmo com reaquecimento (*reheat*) não tenham superado o

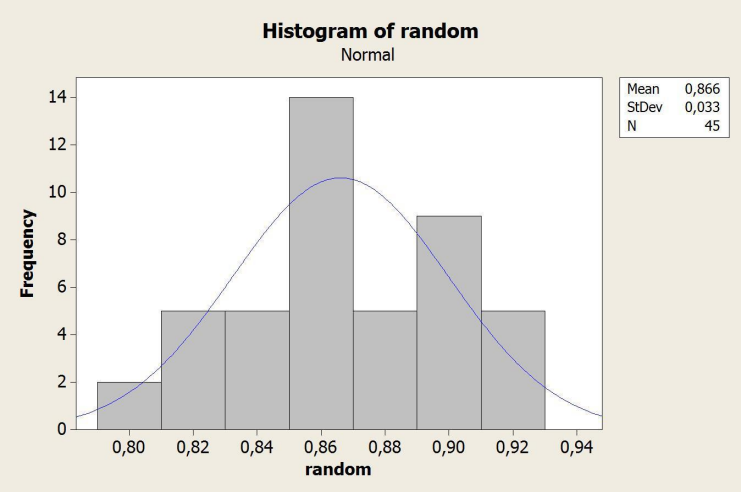


Figura 7.2 - Histograma resultante da análise da distribuição dos diagnósticos

algoritmo de ordenamento *cfm*, seja em relação ao custo computacional ou ao custo final do ordenamento, estes foram avaliados novamente, agora com base na medida do diagnóstico.

Assim como na análise quantitativa, utilizando a rede proteica do *Homo sapiens* com *score* $0,8$, foram gerados 15 ordenamentos para cada configuração, variando apenas o número de passos (3000, 5000 e 10000). O restante dos parâmetros são os mesmos da Subseção 5.1.2. Nesta etapa da avaliação, a número de execuções foi balizado pelos algoritmos *spectralInit* e *spectralFinal*, que possuem uma execução lenta (Seção 5.2.2). No total foram gerados 45 ordenamentos com cada algoritmo. Todos os ordenamento passaram pelo processo de diagnóstico, variando o raio da janela entre 0 e 100. Com raio superior a esse intervalo os diagnósticos não apresentavam melhores resultados. Dos diagnósticos gerados para cada ordenamento foi selecionado o melhor diagnóstico, e em seguida fez-se o descarte dos diagnósticos que não atingiram o resultado mínimo de $0,933$. O gráfico da Figura 7.3 mostra a quantidade de diagnósticos que cada algoritmo possui acima do valor de referência.

Embora não seja um algoritmo, mas a ausência de qualquer ordenamento, foi incluído no gráfico o diagnóstico realizado com o Transcriptograma de raio 0 (*radius0*), que corresponde ao sinal puro, sem a suavização resultante de médias sobre vizinhos. De fato, aplicar o Transcriptograma contribui para a produção de melhores diagnósticos, visto que nenhum diagnóstico com Trancriptograma de raio 0 atingiu o valor de referência, ficando com média inferior a $0,7$, resultado inferior ao obtido com os ordenamentos aleatórios, que tem média igual a $0,866$.

No gráfico da Figura 7.3 não é possível visualizar nenhum diagnóstico aleatório, entretanto é preciso dizer que, mesmo com frequência inferior aos demais ordenamentos, foi possível encontrar bons diagnósticos, com alguns poucos superando o valor de $0,9$. Com relação aos outros algoritmos, todos tiveram pelo menos um resultado acima da referência. Igualmente à análise quantitativa, os algoritmos *cfm* e *spectralFinal* foram os melhores, com uma pequena vantagem para o *spectralFinal*. Porém, como já foi visto anteriormente, o custo computacional dos dois algoritmos que utilizam o

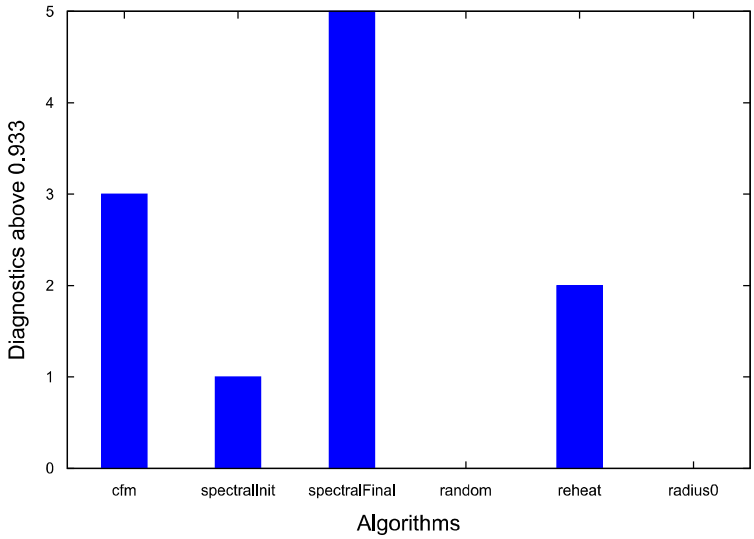


Figura 7.3 - Comparação entre algoritmos com diagnósticos superior a $0,933$

método espectral é muito superior ao *cfm*. Considerando que os dois algoritmos são semelhantes quantitativamente e qualitativamente, mas o *cfm* é mais eficiente computacionalmente, optou-se por aprofundar a análise somente do *cfm*. A análise de parâmetros que se dará nas seções seguintes abordará somente os testes com o algoritmo *cfm*. O APÊNDICE C mostra uma análise da distribuição do conjunto de diagnósticos produzidos por cada algoritmo.

7.3 Eficiência dos Parâmetros no Algoritmo CFM

Em [11], o autor utiliza alguns parâmetros de referência, mas esta escolha dos parâmetros era dificultada pela impossibilidade de se executar múltiplos experimentos devido ao alto tempo de computação. Desta forma, as análises realizadas em [11] podem não estar utilizando os parâmetros mais adequados.

Contudo, com as otimizações realizadas neste trabalho, torna-se possível realizar uma busca mais abrangente no espaço de parâmetros do *cfm*. O objetivo deste capítulo é apresentar este estudo da relação existente entre a qualidade do diagnóstico e os parâmetros do *cfm*.

7.3.1 Parâmetro Score

Uma das análises mais relevantes do presente trabalho foi feita com relação ao tamanho da rede proteica, determinada pelo *score* mínimo que indica se existe interação entre duas proteínas. Até o momento, todas as pesquisas com Transcriptograma utilizaram o *score* igual a $0,8$, acreditando que com esse valor os dados mais relevantes estariam incluídos na rede. É importante verificar se

redes maiores são mais eficientes para o diagnóstico. Por outro lado, a análise de redes maiores só é viável atualmente, com as otimizações implementadas nesta dissertação.

Sendo a rede proteica a base para os diagnósticos, e o *score* o parâmetro que determina o tamanho da rede, quanto menor o *score* maior é a quantidade de interações proteicas aceitas. Afim de observar a eficiência do *score*, além da rede proteica do *Homo sapiens* com *score* $0,8$, foram extraídas do banco de dados STRING versão 9.05 as redes do *Homo sapiens* com *scores* $0,7$, $0,6$ e $0,5$, essa última com 13688 proteínas e 254513 interações. Somada a variação das redes com *score* diferente, os parâmetros α ($0,5$, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) e número de passos (3000, 5000 e 10000) também foram variados, totalizando 132 configurações diferentes.

Assim como na análise quantitativa, cada configuração foi executada 15 vezes, produzindo 1980 ordenamentos diferentes, sendo 495 para cada *score*. Os diagnósticos para esclerose múltipla foram gerados com todos os ordenamentos, variando de acordo com o raio do Transcriptograma (0 a 100), ou seja, são 199980 diagnósticos. Este cenário de testes e os resultados obtidos também foram utilizados na análise de eficiência dos parâmetros α e número de passos.

O gráfico da Figura 7.4, dividido em quatro quadrantes, uma para cada *scores*, mostra o melhor diagnóstico (*Best Diagnostics*) de cada ordenamento em verde, junto com o diagnóstico desses mesmos ordenamentos para o Transcriptograma de raio 0 (*Radius0*) em azul. Novamente é possível observar o benefício oriundo do algoritmo de ordenamento, já que a ausência de um ordenamento (Transcriptograma com raio 0) não produz bons diagnósticos, ficando bem abaixo dos melhores diagnósticos, que sofrem influência do ordenamento.

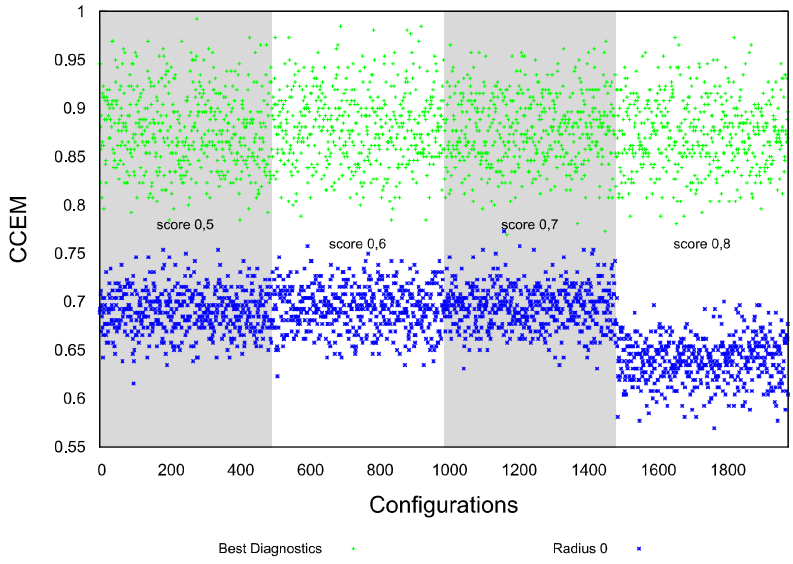


Figura 7.4 - Conjunto total de diagnóstico gerados a partir das redes do *Homo sapiens*

Outro ponto importante a ser observado na Figura 7.4, é a perceptível redução de qualidade dos diagnóstico com raio 0 e *score* $0,8$. Uma explicação para esse comportamento é que esse *score*, ao ser aplicado, subtraiu informações relevantes para o diagnóstico, que não poderiam ser

descartadas. Entretanto, mais uma vez é possível observar que o ordenamento contribuiu para o diagnóstico, elevando a qualidade e equiparando-se aos outros *scores*.

Devido à grande quantidade de dados fica difícil perceber uma diferença de qualidade entre os diagnósticos produzidos com cada rede. A abordagem da análise de eficiência dos algoritmos de ordenamento, apresentada na Seção 7.1, foi novamente utilizada. Através do valor de referência definido pelos diagnóstico com ordenamentos aleatórios é possível contabilizar somente os resultados que sobressaem. A Figura 7.5 apresenta o resultado dessa análise para os melhores diagnósticos.

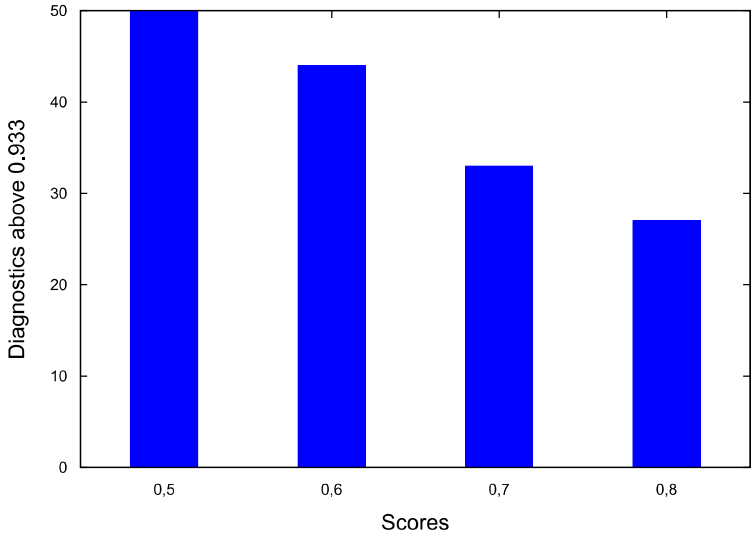


Figura 7.5 - Comparação entre *scores* com diagnósticos superiores a $0,933$

Os valores apresentados correspondem ao número absoluto de diagnóstico que cada *score* produziu acima do valor de referência. Como o total por *score* é de 495 diagnósticos, isso significa que aproximadamente 10% dos diagnósticos com *score* $0,5$ foram superiores à referência.

Fica claro nos resultados que as redes maiores contribuem para a eficiência do diagnóstico. Ao comparar as redes com *score* $0,8$ e *score* $0,5$, observa-se que, utilizar a rede com *score* $0,5$ representa um acréscimo de aproximadamente 85% na quantidade de diagnósticos acima do valor de referência. Entretanto, é preciso lembrar que uma rede com mais nodos representa um custo computacional maior na execução do algoritmo de ordenamento, assim como no processo de diagnóstico. Os gráficos apresentados no APÊNDICE D contribuem para a análise deste parâmetro.

7.3.2 Parâmetro Alpha (α)

Outro parâmetro investigado é o α. É possível observar na Figura 7.6 com um valor alto de α um formato de folha, por assim dizer, é produzido. No caso do ordenamento com $\alpha = 10$ (Figura 7.6(b)), as proteínas que interagem são impedidas de ficar muito distantes no ordenamento, tendo em vista a total ausência ligações longe da diagonal da matriz.

Em contrapartida, ordenamentos com α menor, como na Figura 7.6(a) ($\alpha = 1$), priorizam a formação de grupos que apresentam forte conexão interna, como os blocos pretos próximos à diagonal da matriz. O trabalho [11] investigou ordenamentos produzidos com o parâmetros α diferente de 1, quando demostrou que os melhores diagnósticos eram obtidos com $\alpha = 10$, e não com $\alpha = 1$.

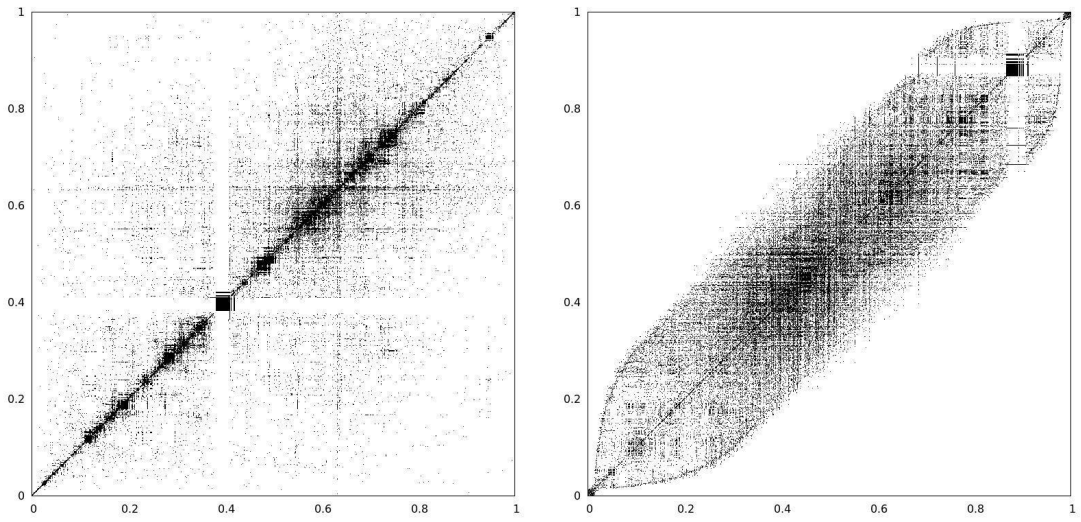


Figura 7.6 - Ordenamento com $\alpha = 1$ (a) e ordenamento com $\alpha = 10$ (b)

O cenário de testes, descrito na Subseção 7.3.1, assim como os resultados obtidos, foram utilizados para a análise do parâmetro α. Diferente de [11], valores intermediários foram avaliados, variando entre $0,5$, 1, 2, até 10. Levando 1980 ordenamentos, e que são 11 valores para α, foram gerados 180 ordenamentos para cada α.

O gráfico da Figura 7.7, também considera o valor de referência dos ordenamentos aleatórios. Levando em conta a quantidade e a capacidade de produzir diagnósticos acima do valor de referência, não é possível observar que $\alpha = 10$ é melhor que $\alpha = 1$, como apresentado em [11]. Com $\alpha = 6$ foram produzidos 21 diagnósticos acima da referência, o que representa aproximadamente 11% do total de diagnósticos (180). Ainda assim, a diferença para outros valores de α não é tão grande a ponto de definir a opção por um valor específico. O APÊNDICE E complementa a análise desta conjunto de dados.

7.3.3 Parâmetro Steps (Número de Passos)

O último parâmetro avaliado foi o número de passos. Junto com o *score*, que determina o tamanho da rede, esse é parâmetro que mais influência no tempo de computação do processo de ordenamento. Sendo assim, é importante visualizar graficamente a eficiência do diagnóstico com relação ao número de passos e determinar a importância desse parâmetro para o diagnóstico.

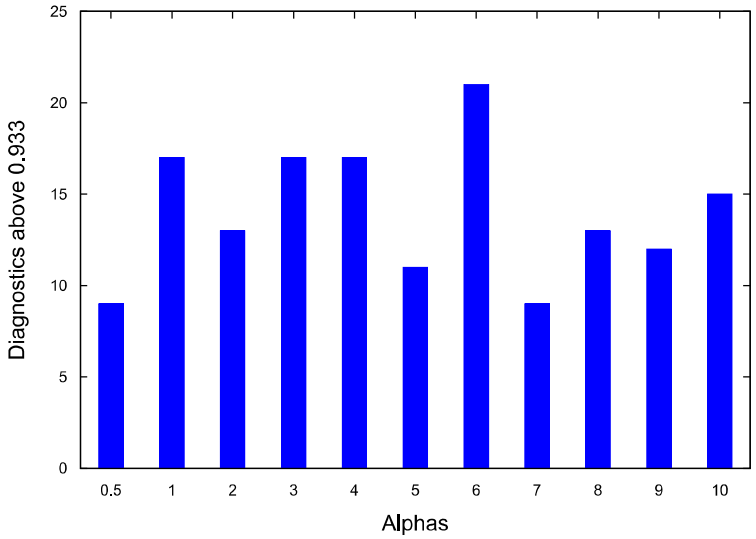


Figura 7.7 - Comparação entre *alphas* (α) com diagnósticos superiores a $0,933$

No cenário de testes descrito anterior, o número de passos variou entre 3000, 5000 e 10000. Número de passos menores que 3000 foram desconsiderados, porque não eram suficientes para reduzir a temperatura a um ponto onde a redução do peso estabilizasse. A Figura 5.15, apresentada anteriormente, ilustra essa questão. De um total de 1980 ordenamentos, 660 para cada número de passos, cada um produzindo um melhor diagnóstico, foram removidos todos aqueles que não atingiram o valor de referência. A quantidade de diagnósticos restante de cada número de passos pode ser visto na Figura 7.8.

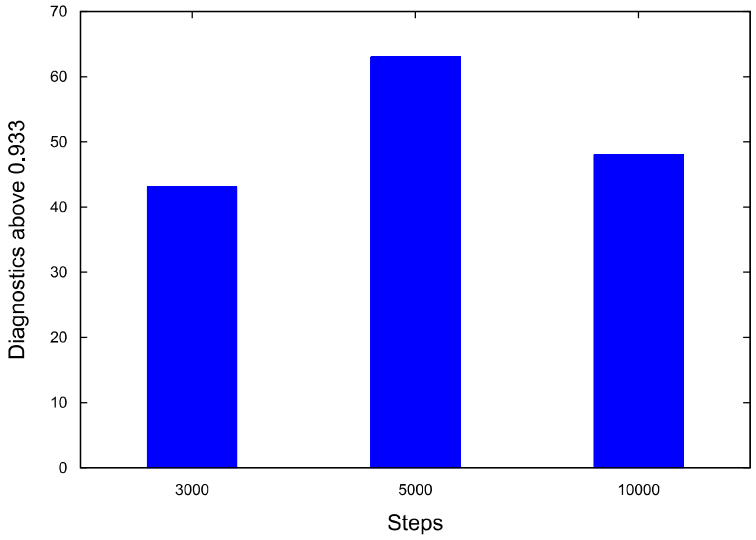


Figura 7.8 - Comparação entre número de passos (*steps*) com diagnósticos superiores a $0,933$

É interessante observar que a quantidade de diagnósticos contabilizados em 5000 aumentou em relação à quantidade de 3000, mas reduziu com 10000 passos. Se levarmos em conta a

análise quantitativa, onde quanto maior o número de passos, menor é o valor da função custo do ordenamento produzido pelo algoritmo *cfm*, isso não se reflete em um melhor diagnóstico, ou seja, o menor custo absoluto não é determinante para a qualidade do ordenamento. Esta análise demonstra que 5000 passos parece ser um parâmetro mais adequado, e que o custo computacional necessário para aumentar o número de passos não se reflete necessariamente em um melhor diagnóstico. Alguns dados referentes a análise estatísticos deste grupo de testes podem ser vistas no APÊNDICE F.

7.4 Considerações Finais da Análise Qualitativa

Através das análises feitas nas seções anteriores, foi possível avaliar os algoritmos de ordenamento e alguns parâmetros, visando identificar qual algoritmo, e quais parâmetros produzem melhores condições para um bom diagnóstico. Diante dos resultados obtidos, é possível recomendar a utilização dos seguintes parâmetros para o diagnóstico do estado (recidiva ou remitente) da doença Esclereose Múltipla:

1. Rede proteica do *Homo sapiens* com *score* igual a $0,5$.
2. Número de passos igual a 5000.
3. Valor de α qualquer (não foi conclusivo).

Os parâmetros avaliados neste trabalho exercem maior influência sobre o tempo de execução do ordenamento, com exceção de α. Entretanto, o algoritmo de ordenamento possui outros parâmetros que não foram analisados, como o intervalo de resfriamento e o fator de resfriamento. Sugere-se um melhor estudo destes em trabalhos futuros por questão de completude, porém, estima-se que estes parâmetros tenham impacto reduzido no diagnóstico.

8. CONCLUSÃO

Este trabalho dedicou-se a fazer a otimização e análise do algoritmo de ordenamento de redes proteicas CFM, além de propor três novas versões do algoritmo, que buscam melhorar a qualidade do ordenamento. Juntamente com a análise quantitativa dos algoritmos, avaliou-se a eficiência dos algoritmos e a eficiência de alguns parâmetros para a geração de bons diagnósticos.

As otimizações a nível de *software* aplicadas ao algoritmo CFM levaram a uma redução significativa da complexidade, passando de um problema com complexidade $O(m.n^3)$ para um problema com complexidade $O(m.n.nz)$, sendo que tipicamente $nz << n$. Essa redução na complexidade tem reflexo direto no tempo de execução do ordenamento. Resultados práticos de tempo de execução mostram que a versão otimizada pode ser até 10000 vezes mais rápida do que a versão original do algoritmo, permitindo realizar um estudo mais completo do impacto dos parâmetros do ordenamento no diagnóstico.

Observou-se nos resultados a importância da realização do ordenamento para a análise por Transcriptograma, uma vez que os diagnósticos feitos com a ausência do ordenamento mostraram-se abaixo dos diagnósticos produzidos com algum algoritmo de ordenamento associado, inclusive do ordenamento aleatório. As novas implementações do algoritmo CFM propostas neste trabalho, incluindo o reaquecimento e o ordenamento espectral, não obtiveram resultados superiores na comparação com o CFM sozinho. Mesmo atingindo resultados semelhantes ao CFM na geração dos diagnósticos, o algoritmo *spectralFinal*, que associa um ordenamento espectral ao CFM, teve um desempenho inferior ao CFM com as otimizações desenvolvidas. Entretanto, sugere-se a análise mais aprofundada do algoritmo *reheat*, pois possui um parâmetro diferente dos demais e que não foi avaliado, o número de reaquecimentos.

Uma das contribuições mais relevantes deste trabalho foi a avaliação da rede proteica com *score* inferior a $0,8$, até então utilizado nos trabalhos contendo Transcriptograma [28] [8] [25] [11]. Embora não fosse impeditiva a utilização de redes com *score* inferior a $0,8$, o tempo necessário para produzir um ordenamento de redes com *scores* menores seria muito elevado, visto que quanto menor é o *score* maior é a rede. Isso respalda mais uma vez a importância das otimizações desenvolvidas.

A partir da análise das redes com *scores* menores que $0,8$ foi possível constatar que, além da quantidade de bons diagnósticos produzidos por essas redes serem superiores ao *score* $0,8$, existe uma aparente perda de informação relevante ao utilizar esse *score*.

Tendo em vista que, quanto maior o tempo de execução do algoritmo de ordenamento, menor é o custo final, outra contribuição do presente trabalho foi observada ao correlacionar o menor custo a geração de melhores diagnósticos. Observou-se que o menor custo absoluto não significa um melhor diagnóstico. Somado a isso, a análise dos parâmetros mostrou que é possível definir um número máximo de interações para o algoritmo de ordenamento, não sendo necessária a busca pela minimização completa do ordenamento.

Contudo, os diagnósticos obtidos com o ordenamento aleatório não se destacaram estatisticamente dos resultados apresentados com o algoritmo de ordenamento CFM otimizado, entretanto estes resultados ainda são inconclusivos, pois o diagnóstico de uma única doença foi avaliado.

8.1 Trabalhos Futuros

Os resultados observados com o diagnóstico da Esclereose Múltipla apenas sugerem a utilização de alguns valores para os parâmetros analisados. É preciso enfatizar que, não é possível generalizar os resultados observados com esses parâmetros e algoritmos para todas as doenças. Levando essa e outras questões abordadas neste trabalho, é possível listar frentes para guiar os próximos trabalhos:

- Investigar se existe uma configuração genérica para o diagnóstico de qualquer doença, ou se os parâmetros ideais precisam ser caraterizados para cada doença. Essa análise pode incluir outros parâmetros utilizados pelos algoritmos de ordenamento e que não foram abordados neste trabalho.
- Explorar paralelismo no CFM otimizado com arquitetura *multicore* e GPU (*Graphics Processing Unit*). Estas e outras possíveis otimizações podem vir a reduzir ainda mais o tempo de execução, possibilitando a execução de redes ainda maiores, e problemas de otimização mais complexos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Abramson, D.; Dang, H.; Krisnamoorthy, M. "Simulated annealing cooling schedules for the school timetabling problem", *Asia-Pacific Journal of Operational Research*, vol. 16, 1999, pp. 1–22.

[2] Aho, A.; Ullman, J. "Foundations of Computer Science: C Edition". W. H. Freeman, 1995.

[3] Alon, U. "An Introduction to Systems Biology". Chapman & Hall/CRC, 2007.

[4] Anagnostopoulos, A.; Michel, L.; Hentenryck, P. V.; Vergados, Y. "A simulated annealing approach to the traveling tournament problem", *J. of Scheduling*, vol. 9–2, Abril 2006, pp. 177–193.

[5] Barabasi, A. L.; Albert, R. "Emergence of scaling in random networks", *Science*, vol. 286–5439, Outubro 1999, pp. 509–512.

[6] Barabasi, A.-L.; Oltvai, Z. N. "Network biology: understanding the cell's functional organization", *Nature Reviews Genetics*, vol. 5, Fevereiro 2004, pp. 101–113.

[7] Barabási, A.-L.; Ravasz, E.; Oltvai, Z. "Hierarchical organization of modularity in complex networks", 2003.

[8] Benetti, F. P. C. "Homo sapiens: análise de expressão gênica por transcriptograma". Trabalho de Conclusão, Instituto de Física, UFRGS, Porto Alegre, RS, Brasil, 2010.

[9] Bhandarkar, S. M.; Chirravuri, S.; Arnold, J. "Parallel computing of physical maps–a comparative study in simd and mimd parallelism", *Journal of Computational Biology*, vol. 3–4, Fevereiro 1996, pp. 503–528.

[10] Brynedal, B.; Khademi, M.; Wallström, E.; Hillert, J.; Olsson, T.; Duvefelt, K. "Gene expression profiling in multiple sclerosis: A disease of the central nervous system, but with relapses triggered in the periphery?", *Neurobiology of Disease*, vol. 37–3, Março 2010, pp. 613–621.

[11] da Silva, S. R. M. "A eficiência do transcriptograma", Tese de Doutorado, UFRGS, Porto Alegre, RS, Brasil, 2013, 84p.

[12] de Castro Gomes Júnior, A.; Souza, M. J. F.; Martins, A. X. "Simulated annealing aplicado à resolução do problema de roteamento de veículos com janela de tempo", *TRANSPORTES*, vol. 13–2, Dezembro 2005, pp. 5–20.

[13] Franceschini, A.; Szklarczyk, D.; Frankild, S.; Kuhn, M.; Simonovic, M.; Roth, A.; Lin, J.; Minguez, P.; Bork, P.; von Mering, C.; Jensen, L. J. "String v9.1: protein-protein interaction networks, with increased coverage and integration", *Nucleic Acids Research*, vol. 41, Janeiro 2013, pp. 808–815.

[14] Hastie, T.; Tibshirani, R.; Friedman, J. "The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition". Springer, 2010.

[15] Hill, T.; Lewicki, P. "Statistics: Methods and Applications : a Comprehensive Reference for Science, Industry, and Data Mining". StatSoft, 2006.

[16] Jain, A. K.; Murty, M. N.; Flynn, P. J. "Data clustering: a review", *ACM Comput. Surv.*, vol. 31–3, Setembro 1999, pp. 264–323.

[17] Jensen, L. J.; Kuhn, M.; Stark, M.; Chaffron, S.; Creevey, C.; Muller, J.; Doerks, T.; Julien, P.; Roth, A.; Simonovic, M.; Bork, P.; von Mering, C. "String 8–a global view on proteins and their functional interactions in 630 organisms", *Nucleic Acids Research*, vol. 37, Janeiro 2009, pp. D412–D416.

[18] Jeong, H.; Tombor, B.; Albert, R.; Oltvai, Z. N.; Barabási, A. L. "The large-scale organization of metabolic networks", *Nature*, vol. 407–6804, Outubro 2000, pp. 651–654.

[19] Juvan, M.; Mohar, B. "Optimal linear labelings and eigenvalues of graphs", *Discrete Applied Mathematics*, vol. 36–2, Janeiro 1992, pp. 153–168.

[20] Kanehisa, M.; Goto, S.; Kawashima, S.; Okuno, Y.; Hattori, M. "The kegg resource for deciphering the genome", *Nucleic Acids Research*, vol. 32, Janeiro 2004, pp. 277–280.

[21] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. "Optimization by simulated annealing", *Science*, vol. 220–4598, Maio 1983, pp. 671–680.

[22] Liiv, I. "Seriation and matrix reordering methods: An historical overview", *Stat. Anal. Data Min.*, vol. 3–2, Abril 2010, pp. 70–91.

[23] of Energy Office of Energy Research, U. S. D.; of Energy Office of Health, U. S. D.; Research, E. "Human genome: program report". U.S. Dept. of Energy, Office of Energy Research, Office of Health and Environmental Research, 1990.

[24] Osman, I. H. "Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem", *Ann. Oper. Res.*, vol. 41–1-4, Maio 1993, pp. 421–451.

[25] Perrone, G. C. "Transcriptograma em duas dimensões", Tese de Doutorado, UFRGS, Porto Alegre, RS, Brasil, 2013, 60p.

[26] Petit, J. "Layout problems", Tese de Doutorado, Polytechnic University of Catalonia, Barcelona, Espanha, 2001, 265p.

[27] Rybarczyk-Filho, J. L. L. "Medidas de performance metabólica usando a expressão gênica de genoma completo", Tese de Doutorado, UFRGS, Porto Alegre, RS, Brasil, 2011, 84p.

[28] Rybarczyk-Filho, J. L. L.; Castro, M. A.; Dalmolin, R. J.; Moreira, J. C.; Brunnet, L. G.; de Almeida, R. M. "Towards a genome-wide transcriptogram: the saccharomyces cerevisiae case", *Nucleic Acids Research*, vol. 39–8, Abril 2011, pp. 3005–3016.

[29] Sales-Pardo, M.; Guimerà, R.; Moreira, A. A.; Amaral, L. A. N. "Extracting the hierarchical organization of complex systems", *Proceedings of the National Academy of Sciences*, vol. 104–39, Setembro 2007, pp. 15224–15229.

[30] Schaeffer, S. E. "Survey: Graph clustering", *Comput. Sci. Rev.*, vol. 1–1, Agosto 2007, pp. 27–64.

[31] Schena, M.; Shalon, D.; Davis, R. W.; Brown, P. O. "Quantitative monitoring of gene expression patterns with a complementary dna microarray", *Science*, vol. 270–5235, Outubro 1995, pp. 467–470.

[32] Shapiro, S. S.; Wilk, M. B. "An analysis of variance test for normality (complete samples)", *Biometrika*, vol. 52–3/4, Dezembro 1965, pp. 591–611.

[33] Späth, H. "Cluster analysis algorithms for data reduction and classification of objects". Ellis Horwood, 1980, 226p.

[34] Szklarczyk, D.; Franceschini, A.; 0004, M. K.; Simonovic, M.; Roth, A.; Minguez, P.; Doerks, T.; Stark, M.; Muller, J.; Bork, P.; Jensen, L. J.; von Mering, C. "The string database in 2011: functional interaction networks of proteins, globally integrated and scored", *Nucleic Acids Research*, vol. 39, Janeiro 2011, pp. 561–568.

[35] Tan, K. C.; Lee, L. H.; Zhu, K. Q.; Ou, K. "Heuristic methods for vehicle routing problem with time windows", *AI in Engineering*, vol. 15–3, 2001, pp. 281–295.

[36] von Mering, C.; Jensen, L. J.; Snel, B.; Hooper, S. D.; Krupp, M.; Foglierini, M.; Jouffre, N.; Huynen, M. A.; Bork, P. "String: known and predicted protein-protein associations, integrated and transferred across organisms.", *Nucleic Acids Research*, vol. 33, Janeiro 2005, pp. 433–437.

APÊNDICE A – FUNÇÃO DO CÁLCULO PARCIAL DO ALGORITMO *PARTIAL_CFM*

```
double getPartWeight(char *matrix[], short a, short b){
  double col_a_weight = 0, col_b_weight = 0, row_a_weight = 0, row_b_weight = 0;
  if(a > b){
    index_a = b;
    index_b = a;
  }
  init_a = index_a;
  init_b = index_b;

  if((index_a+1) == index_b){
    if(index_a == 0){
      index_a_cnt = 2;
    }else{
      index_a_cnt = 3;
      index_a = index_a-1;
    }
    if(index_b == (n_nodes-1)){
      index_b_cnt = 0;
    }else{
      index_b_cnt = 1;
      index_b = index_b+1;
    }
  }else if((index_a+1) == (index_b-1)){
    if(index_a == 0){
      index_a_cnt = 2;
    }else{
      index_a_cnt = 3;
      index_a = index_a-1;
    }
    if(index_b == (n_nodes-1))
      index_b_cnt = 1;
    else
      index_b_cnt = 2;
  }else{
    if(index_a == 0){
      index_a_cnt = 2;
    }else{
      index_a_cnt = 3;
      index_a = index_a-1;
    }
    if(index_b == (n_nodes-1))
      index_b_cnt = 2;
    else
      index_b_cnt = 3;

    index_b = index_b-1;
  }

  index_cnt = index_a + index_a_cnt;
  /* column a  */
  for(col=index_a;col<index_cnt;col++){
    part_weight = 0;
    for(row=(n_nodes-1);row>=0;row--){
      if(matrix[row][col] == 1){
        neighbors = 0;
        if(col != 0)
          neighbors = neighbors + (1 - matrix[row][col-1]);
        if(col != (n_nodes-1))
          neighbors = neighbors + (1 - matrix[row][col+1]);
        if(row != 0)
          neighbors = neighbors + (1 - matrix[row-1][col]);
        if(row != (n_nodes-1))
          neighbors = neighbors + (1 - matrix[row+1][col]);

        if(row < col)
          abs = col - row;
        else
          abs = row - col;
        part_weight = part_weight + (abs * neighbors);
      }
    }
    col_a_weight = col_a_weight + part_weight;
  }
  /* row a  */
```

```
75      for(row=(index_cnt-1);row>=index_a;row--){
76        part_weight = 0;
77        for(col=0;col<n_nodes;col++){
78          if(matrix[row][col] == 1){
79            if((col != init_a-1) && (col != init_a) && (col != init_a+1) && (col != init_b-1) && (col != init_b) && (col !=
                  init_b+1)){
80              neighbors = 0;
81              if(col != 0)
82                neighbors = neighbors + (1 - matrix[row][col-1]);
83              if(col != (n_nodes-1))
84                neighbors = neighbors + (1 - matrix[row][col+1]);
85              if(row != 0)
86                neighbors = neighbors + (1 - matrix[row-1][col]);
87              if(row != (n_nodes-1))
88                neighbors = neighbors + (1 - matrix[row+1][col]);
89
90              if(row < col)
91                abs = col - row;
92              else
93                abs = row - col;
94              part_weight = part_weight + (pow(abs,alpha) * neighbors);
95            }
96          }
97        }
98        row_a_weight = row_a_weight + part_weight;
99      }
100     if(index_b_cnt != 0){
101       index_cnt = index_b + index_b_cnt;
102       /* column b  */
103       for(col=index_b;col<index_cnt;col++){
104         part_weight = 0;
105         for(row=(n_nodes-1);row>=0;row--){
106           if(matrix[row][col] == 1){
107             neighbors = 0;
108             if(col != 0)
109               neighbors = neighbors + (1 - matrix[row][col-1]);
110             if(col != (n_nodes-1))
111               neighbors = neighbors + (1 - matrix[row][col+1]);
112             if(row != 0)
113               neighbors = neighbors + (1 - matrix[row-1][col]);
114             if(row != (n_nodes-1))
115               neighbors = neighbors + (1 - matrix[row+1][col]);
116
117             if(row < col)
118               abs = col - row;
119             else
120               abs = row - col;
121             part_weight = part_weight + (pow(abs,alpha) * neighbors);
122           }
123         }
124         col_b_weight = col_b_weight + part_weight;
125       }
126       /* row b  */
127       for(row=(index_cnt-1);row>=index_b;row--){
128         part_weight = 0;
129         for(col=0;col<n_nodes;col++){
130           if(matrix[row][col] == 1){
131             if((col != init_b-1) && (col != init_b) && (col != init_b+1) && (col != init_a-1) && (col != init_a) && (col !=
                    init_a+1)){
132               neighbors = 0;
133               if(col != 0)
134                 neighbors = neighbors + (1 - matrix[row][col-1]);
135               if(col != (n_nodes-1))
136                 neighbors = neighbors + (1 - matrix[row][col+1]);
137               if(row != 0)
138                 neighbors = neighbors + (1 - matrix[row-1][col]);
139               if(row != (n_nodes-1))
140                 neighbors = neighbors + (1 - matrix[row+1][col]);
141
142               if(row < col)
143                 abs = col - row;
144               else
145                 abs = row - col;
146               part_weight = part_weight + (pow(abs,alpha) * neighbors);
147             }
148           }
149         }
150         row_b_weight = row_b_weight + part_weight;
151       }
152     }
153     return (col_a_weight + col_b_weight + row_a_weight + row_b_weight);
154   }
```

APÊNDICE B – FUNÇÃO DO CÁLCULO PARCIAL DO ALGORITMO *PLATINUM_CFM*

```
1  double getPartWeight(char *matrix[], graph_ptr *order, short a, short b, short *size_column, short *ones_list[]){
2    double col_a_weight = 0, col_b_weight = 0, row_a_weight = 0, row_b_weight = 0;
3
4    if(a > b){
5      index_a = b;
6      index_b = a;
7    }
8    init_a = index_a;
9    init_b = index_b;
10   if((index_a+1) == index_b){
11     if(index_a == 0){
12       index_a_cnt = 2;
13     }else{
14       index_a_cnt = 3;
15       index_a = index_a-1;
16     }
17     if(index_b == (n_nodes-1)){
18       index_b_cnt = 0;
19     }else{
20       index_b_cnt = 1;
21       index_b = index_b+1;
22     }
23   }else if((index_a+1) == (index_b-1)){
24     if(index_a == 0){
25       index_a_cnt = 2;
26     }else{
27       index_a_cnt = 3;
28       index_a = index_a-1;
29     }
30     if(index_b == (n_nodes-1))
31       index_b_cnt = 1;
32     else
33       index_b_cnt = 2;
34   }else{
35     if(index_a == 0){
36       index_a_cnt = 2;
37     }else{
38       index_a_cnt = 3;
39       index_a = index_a-1;
40     }
41     if(index_b == (n_nodes-1))
42       index_b_cnt = 2;
43     else
44       index_b_cnt = 3;
45     index_b = index_b-1;
46   }
47   index_cnt = index_a + index_a_cnt;
48   /* column a  */
49   for(col=index_a;col<index_cnt;col++){
50     part_weight = 0;
51     pos_i = order[col];
52     for(row=0;row<size_column[pos_i.cur];row++){
53       pos_j = order[ones_list[pos_i.cur][row]];
54       if(col < pos_j.ini){
55         neighbors = 0;
56         if(pos_j.ini != 0)
57           neighbors = neighbors + (1 - matrix[pos_i.cur][order[pos_j.ini-1].cur]);
58         if(pos_j.ini != (n_nodes-1))
59           neighbors = neighbors + (1 - matrix[pos_i.cur][order[pos_j.ini+1].cur]);
60         if(col != 0)
61           neighbors = neighbors + (1 - matrix[order[col-1].cur][order[pos_j.ini].cur]);
62         if(col != (n_nodes-1))
63           neighbors = neighbors + (1 - matrix[order[col+1].cur][order[pos_j.ini].cur]);
64
65         abs = pow((pos_j.ini - col), alpha);
66         part_weight = part_weight + (abs * neighbors);
67       }
68     }
69     col_a_weight = col_a_weight + part_weight;
70   }
71   /* row a  */
72   for(row=(index_cnt-1);row>=index_a;row--){
73     part_weight = 0;
74     pos_i = order[row];
```

```
75        for(col=0;col<size_column[pos_i.cur];col++){
76          pos_j = order[ones_list[pos_i.cur][col]];
77          if(row > pos_j.ini){
78            if((pos_j.ini!= init_a-1) && (pos_j.ini != init_a) && (pos_j.ini != init_a+1) && (pos_j.ini != init_b-1) && (pos_j.
                  ini != init_b) && (pos_j.ini != init_b+1)){
79              neighbors = 0;
80              if(pos_j.ini != 0)
81                neighbors = neighbors + (1 - matrix[pos_i.cur][order[pos_j.ini-1].cur]);
82              if(pos_j.ini != (n_nodes-1))
83                neighbors = neighbors + (1 - matrix[pos_i.cur][order[pos_j.ini+1].cur]);
84              if(row != 0)
85                neighbors = neighbors + (1 - matrix[order[row-1].cur][order[pos_j.ini].cur]);
86              if(row != (n_nodes-1))
87                neighbors = neighbors + (1 - matrix[order[row+1].cur][order[pos_j.ini].cur]);
88              abs = pow((row - pos_j.ini), alpha);
89              part_weight = part_weight + (abs * neighbors);
90            }
91          }
92        }
93        row_a_weight = row_a_weight + part_weight;
94      }
95      if(index_b_cnt != 0){
96        index_cnt = index_b + index_b_cnt;
97        /* column b  */
98        for(col=index_b;col<index_cnt;col++){
99          part_weight = 0;
100         pos_i = order[col];
101         for(row=0;row<size_column[pos_i.cur];row++){
102           pos_j = order[ones_list[pos_i.cur][row]];
103           if(col < pos_j.ini){
104             neighbors = 0;
105             if(pos_j.ini != 0)
106               neighbors = neighbors + (1 - matrix[pos_i.cur][order[pos_j.ini-1].cur]);
107             if(pos_j.ini != (n_nodes-1))
108               neighbors = neighbors + (1 - matrix[pos_i.cur][order[pos_j.ini+1].cur]);
109             if(col != 0)
110               neighbors = neighbors + (1 - matrix[order[col-1].cur][order[pos_j.ini].cur]);
111             if(col != (n_nodes-1))
112               neighbors = neighbors + (1 - matrix[order[col+1].cur][order[pos_j.ini].cur]);
113             abs = pow((pos_j.ini - col), alpha);
114             part_weight = part_weight + (abs * neighbors);
115           }
116         }
117         col_b_weight = col_b_weight + part_weight;
118       }
119       /* row b  */
120       for(row=(index_cnt-1);row>=index_b;row--){
121         part_weight = 0;
122         pos_i = order[row];
123         for(col=0;col<size_column[pos_i.cur];col++){
124           pos_j = order[ones_list[pos_i.cur][col]];
125           if(row > pos_j.ini){
126             if((pos_j.ini!= init_b-1) && (pos_j.ini != init_b) && (pos_j.ini != init_b+1) && (pos_j.ini != init_a-1) && (pos_j
                    .ini != init_a) && (pos_j.ini != init_a+1)){
127               neighbors = 0;
128               if(pos_j.ini != 0)
129                 neighbors = neighbors + (1 - matrix[pos_i.cur][order[pos_j.ini-1].cur]);
130               if(pos_j.ini != (n_nodes-1))
131                 neighbors = neighbors + (1 - matrix[pos_i.cur][order[pos_j.ini+1].cur]);
132               if(row != 0)
133                 neighbors = neighbors + (1 - matrix[order[row-1].cur][order[pos_j.ini].cur]);
134               if(row != (n_nodes-1))
135                 neighbors = neighbors + (1 - matrix[order[row+1].cur][order[pos_j.ini].cur]);
136               abs = pow((row - pos_j.ini), alpha);
137               part_weight = part_weight + (abs * neighbors);
138             }
139           }
140         }
141         row_b_weight = row_b_weight + part_weight;
142       }
143     }
144     return (col_a_weight + col_b_weight + row_a_weight + row_b_weight);
145   }
```

APÊNDICE C – ESTATÍSTICA DOS GRUPOS DE ALGORITMOS

A distribuição observada na Figura APÊNDICE C.1 corresponde ao conjunto de diagnósticos gerados pelos testes apresentados na Seção 7.2.

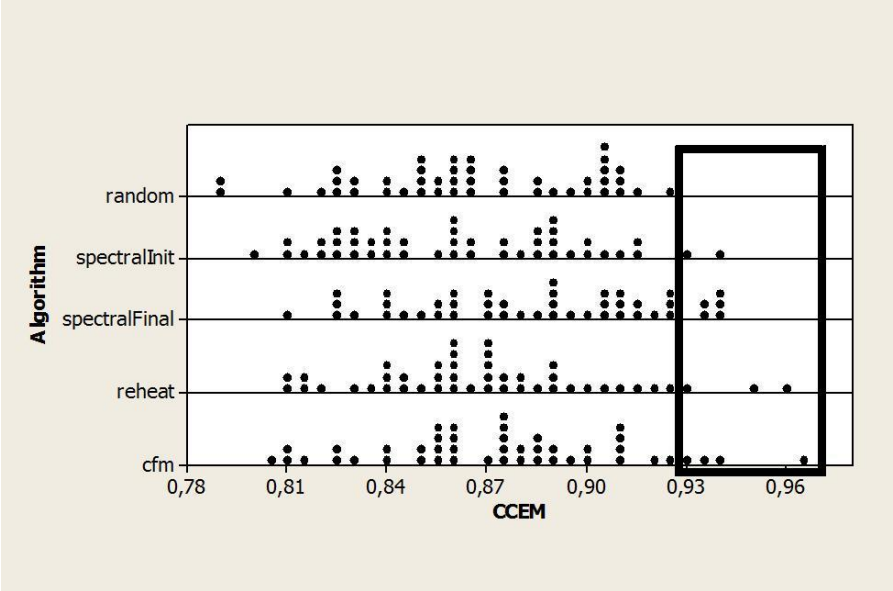


Figura APÊNDICE C.1 - Distribuição dos índices de diagnóstico para os 5 métodos de ordenamento. A área destacada corresponde à área de interesse

APÊNDICE D – ESTATÍSTICA DOS GRUPOS DE *SCORE*

A seguir são apresentados alguns gráficos que complementam a análise feita na Subseção 7.3.1.

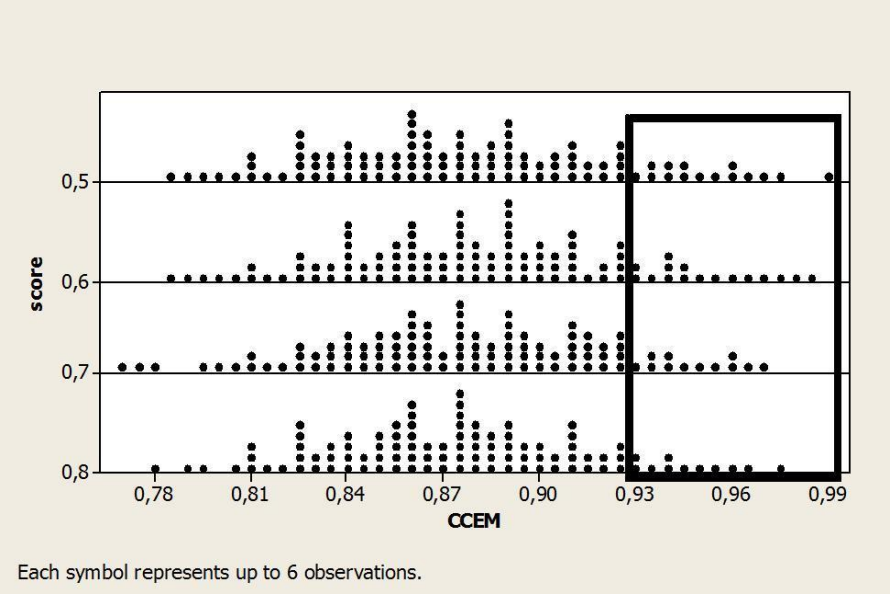


Figura APÊNDICE D.1 - Distribuição dos índices de diagnóstico para os 4 grupos de *score*. A área destacada corresponde à área de interesse

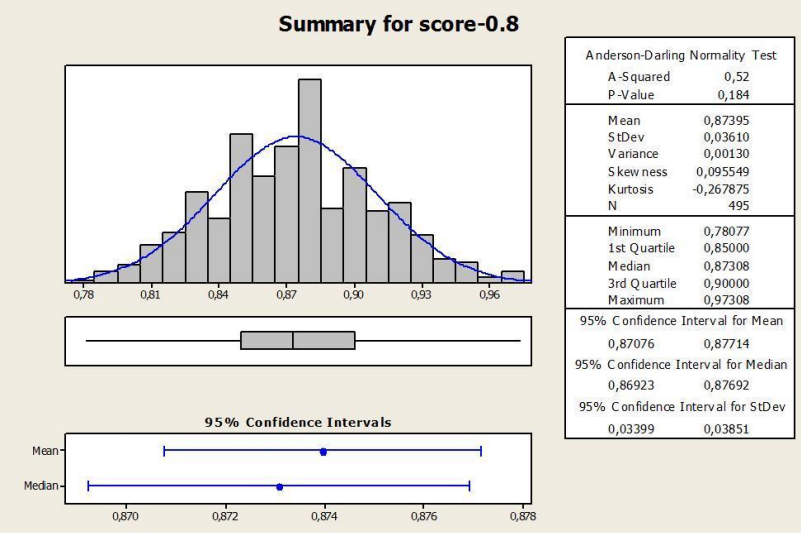


Figura APÊNDICE D.2 - Estatística para o grupo de execuções com *score* $0,8$

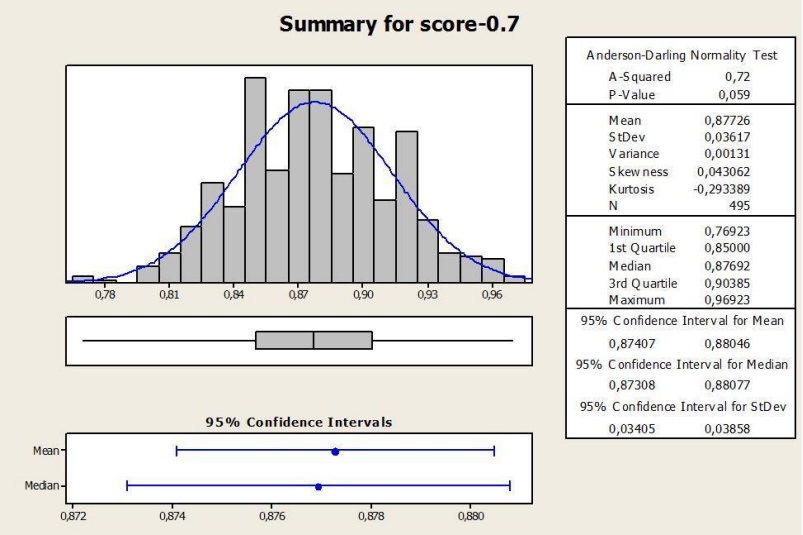


Figura APÊNDICE D.3 - Estatística para o grupo de execuções com *score* $0,7$

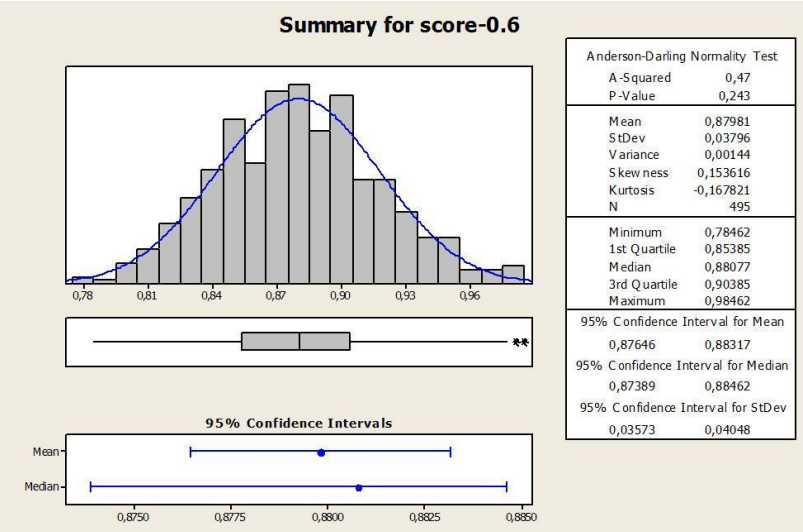


Figura APÊNDICE D.4 - Estatística para o grupo de execuções com *score* $0,6$

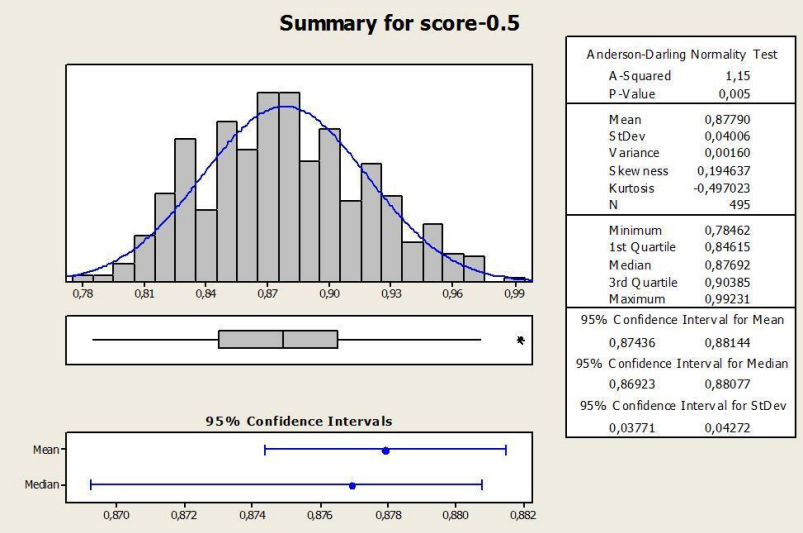


Figura APÊNDICE D.5 - Estatística para o grupo de execuções com *score* $0,5$

APÊNDICE E – ESTATÍSTICA DOS GRUPOS DE ALPHA

A análise da Subseção 7.3.2 é complementada pelos gráficos mostrados a seguir.

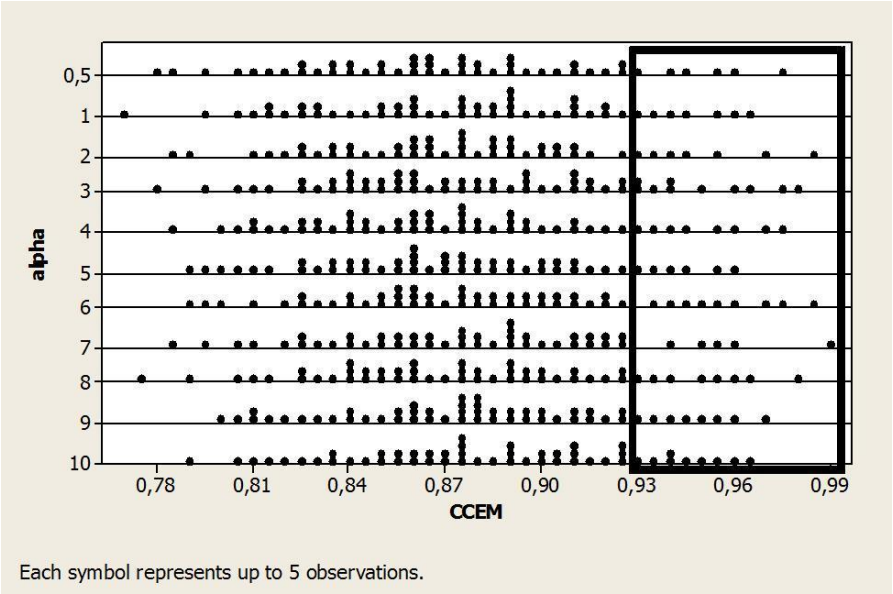


Figura APÊNDICE E.1 - Distribuição dos índices de diagnóstico para os 11 grupos de *alpha*. A área destacada corresponde à área de interesse

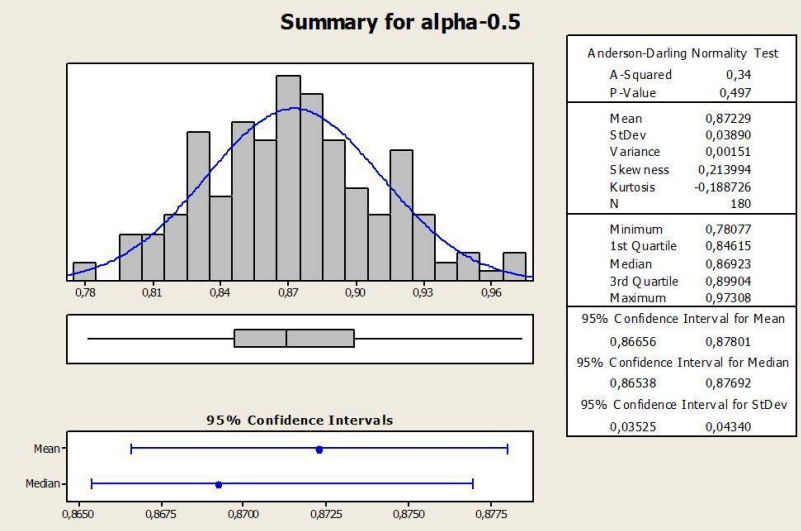


Figura APÊNDICE E.2 - Estatística para o grupo de execuções com *alpha* $0,5$

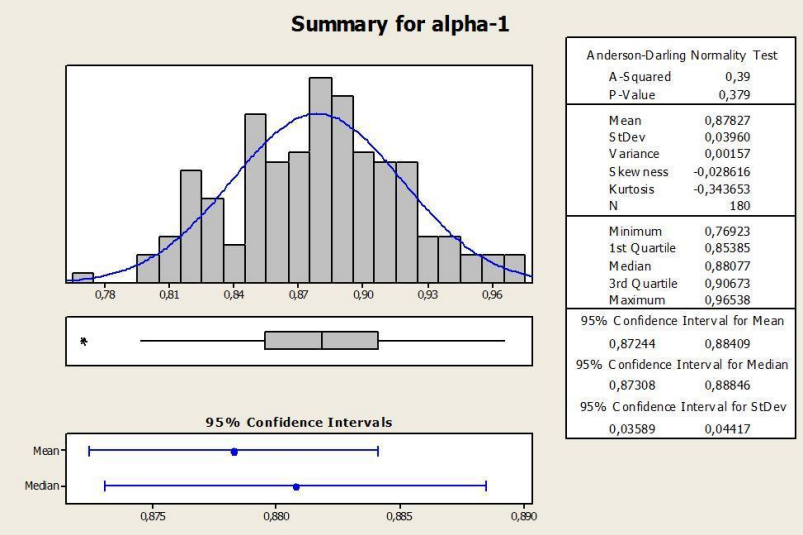


Figura APÊNDICE E.3 - Estatística para o grupo de execuções com *alpha* 1

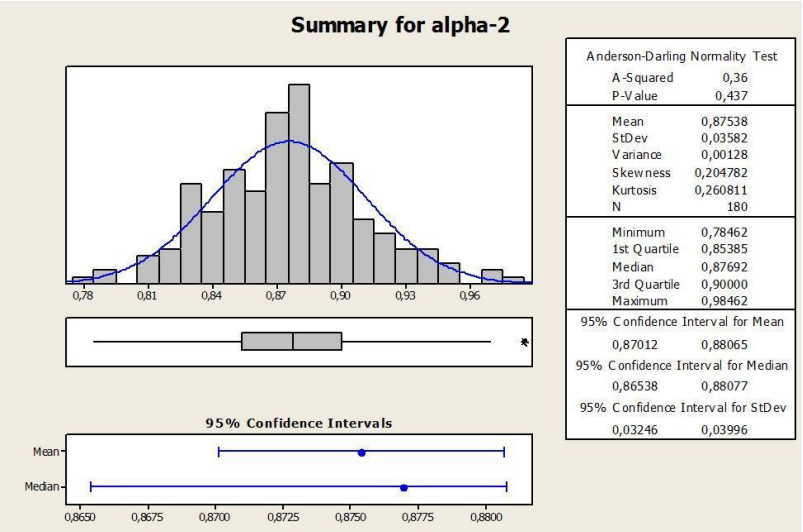


Figura APÊNDICE E.4 - Estatística para o grupo de execuções com *alpha* 2

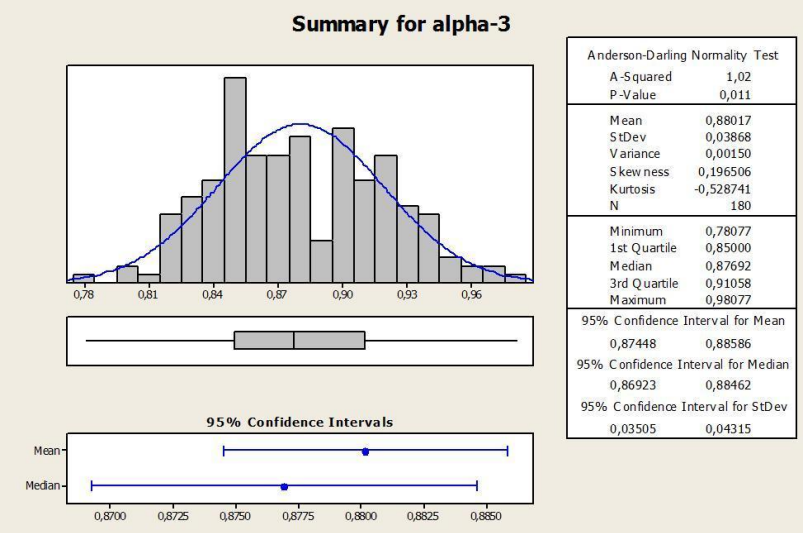


Figura APÊNDICE E.5 - Estatística para o grupo de execuções com *alpha* 3

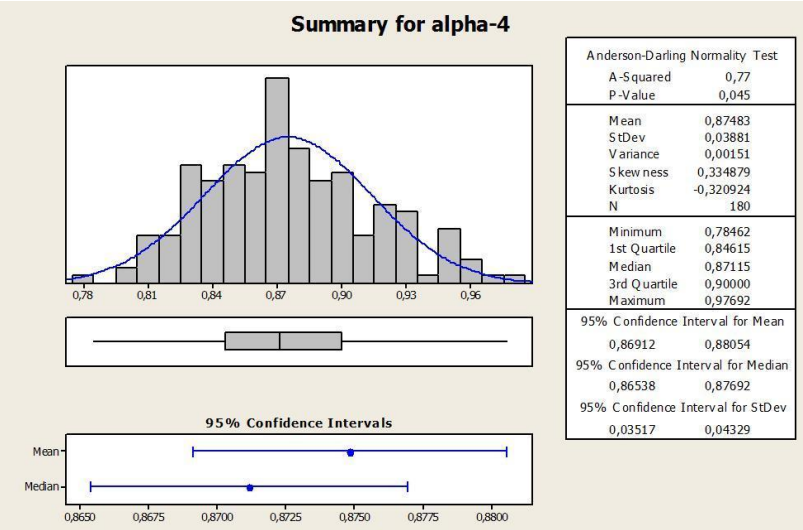


Figura APÊNDICE E.6 - Estatística para o grupo de execuções com *alpha* 4

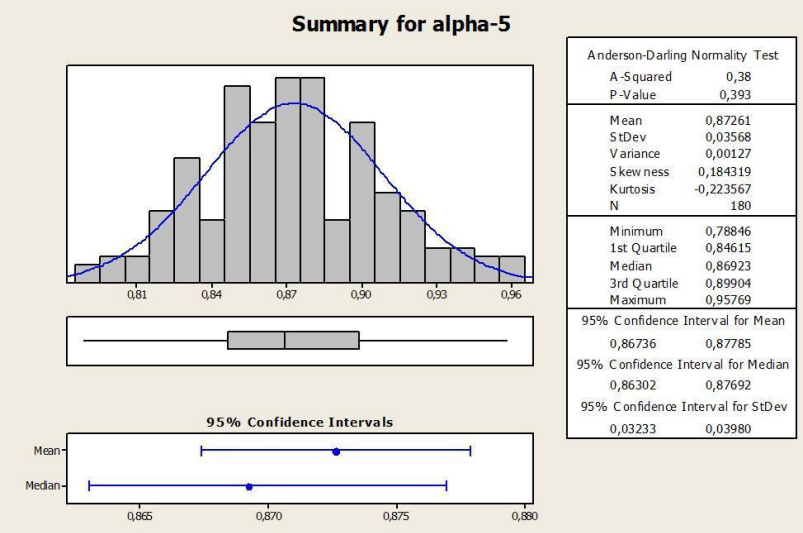


Figura APÊNDICE E.7 - Estatística para o grupo de execuções com *alpha* 5

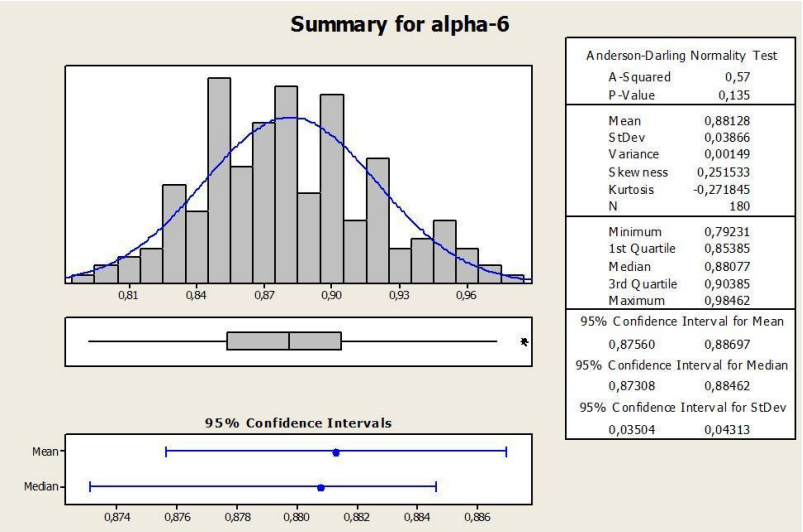


Figura APÊNDICE E.8 - Estatística para o grupo de execuções com *alpha* 6

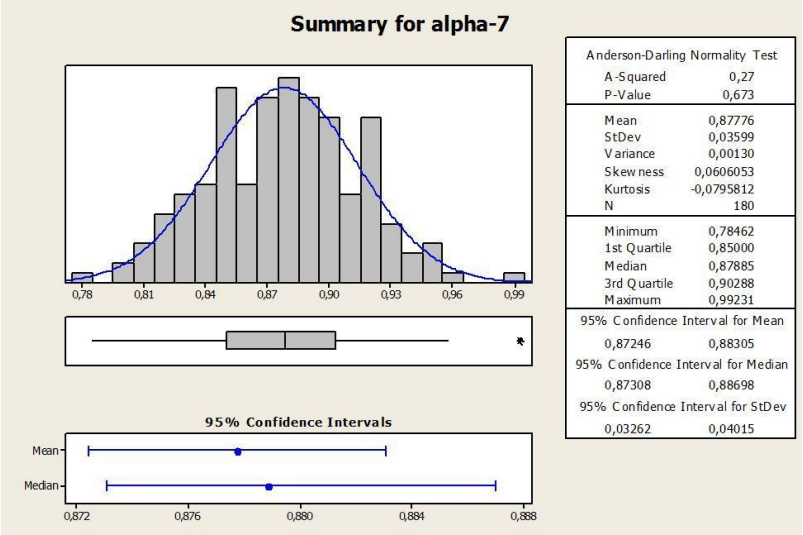


Figura APÊNDICE E.9 - Estatística para o grupo de execuções com *alpha* 7

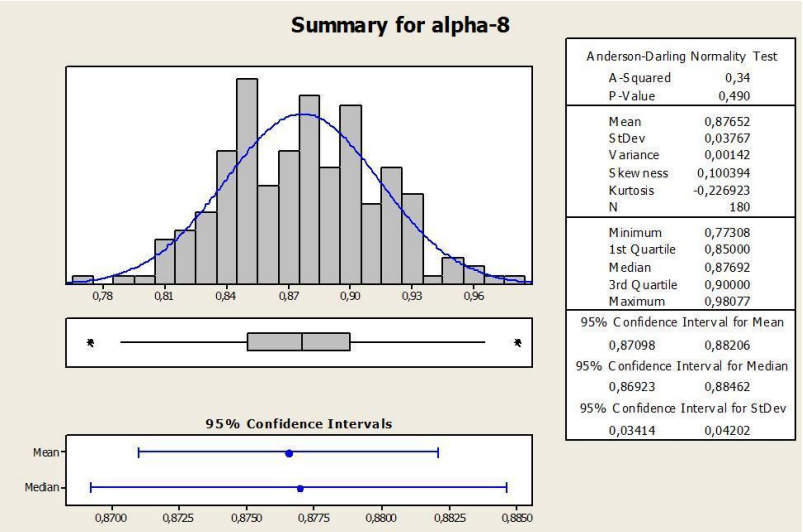


Figura APÊNDICE E.10 - Estatística para o grupo de execuções com *alpha* 8

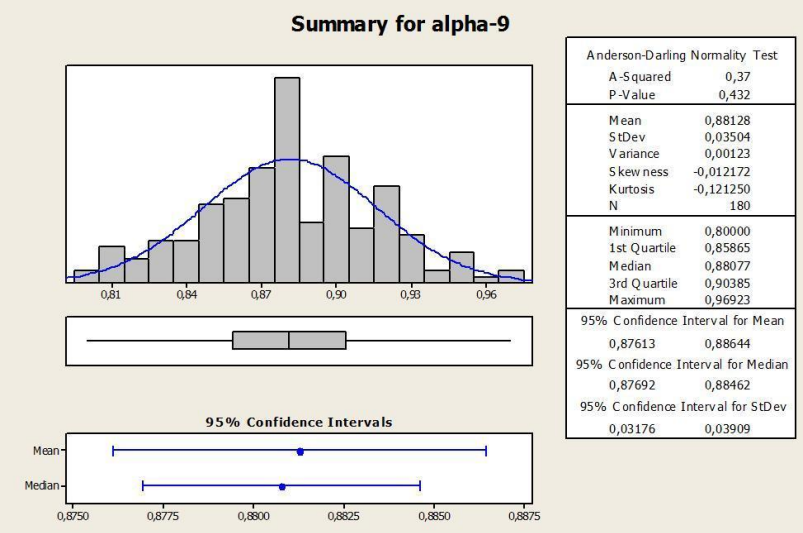


Figura APÊNDICE E.11 - Estatística para o grupo de execuções com *alpha* 9

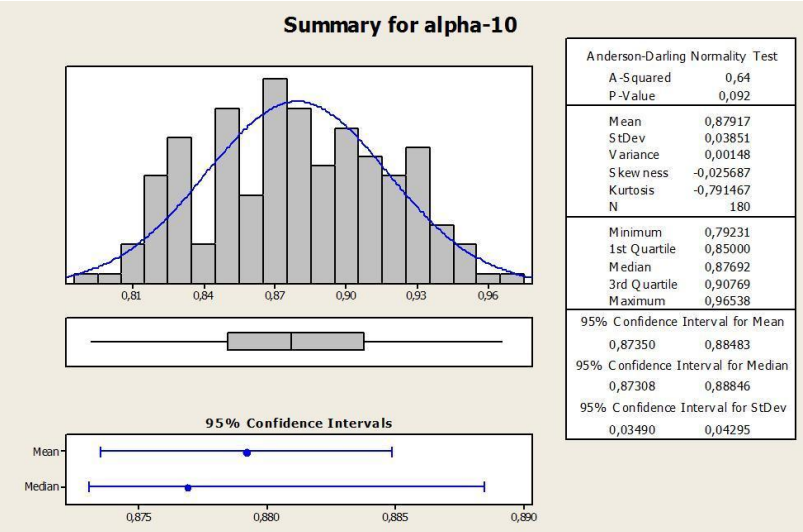


Figura APÊNDICE E.12 - Estatística para o grupo de execuções com *alpha* 10

APÊNDICE F – ESTATÍSTICA DOS GRUPOS DE *STEPS*

Os gráficos a seguir referense-se a análise feita sobre o conjunto de dados abordados na Subseção 7.3.3.

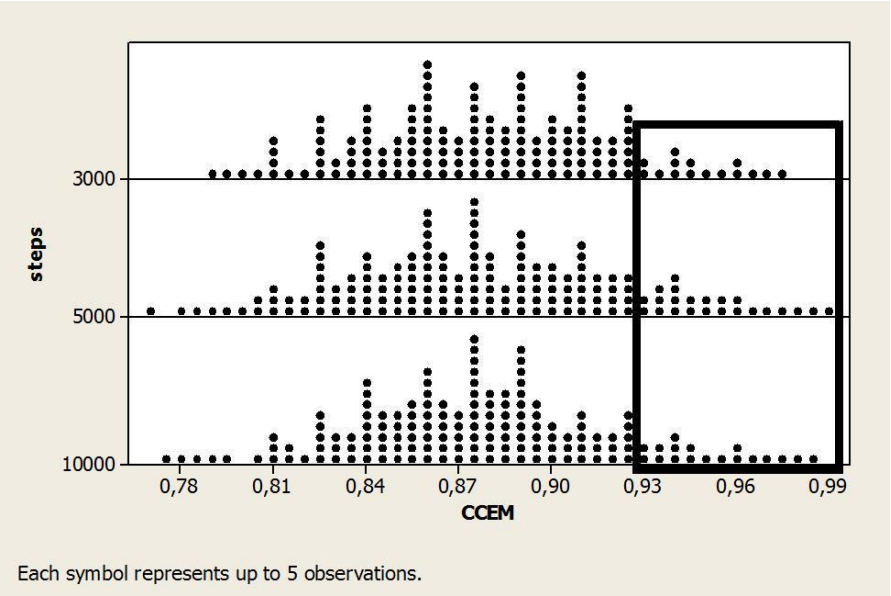


Figura APÊNDICE F.1 - Distribuição dos índices de diagnóstico para os 3 grupos de passos. A área destacada corresponde à área de interesse

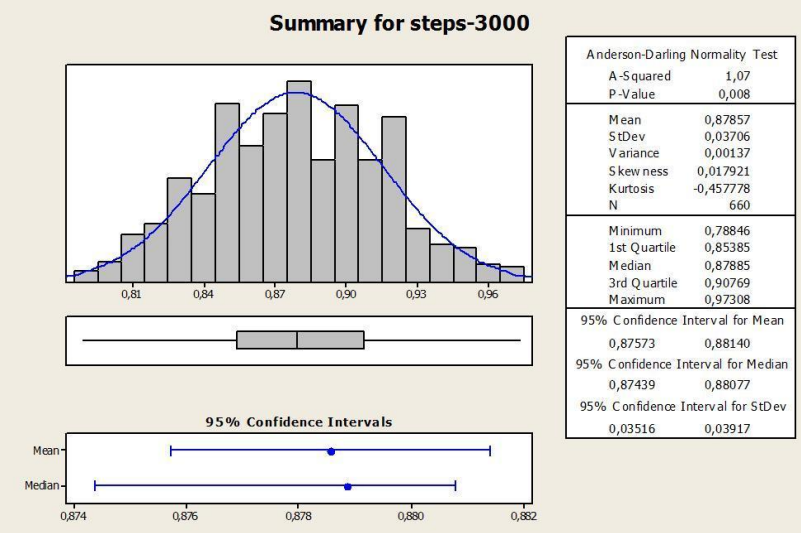


Figura APÊNDICE F.2 - Estatística para o grupo de execuções com 3000 passos

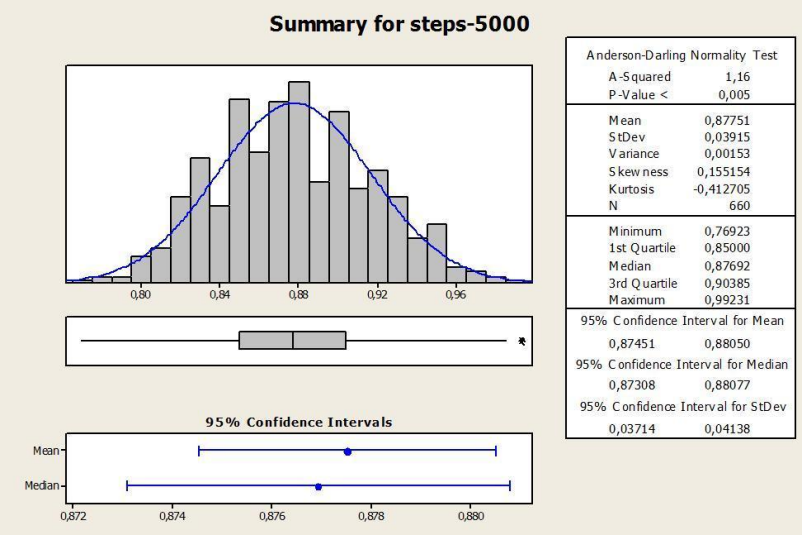


Figura APÊNDICE F.3 - Estatística para o grupo de execuções com 5000 passos

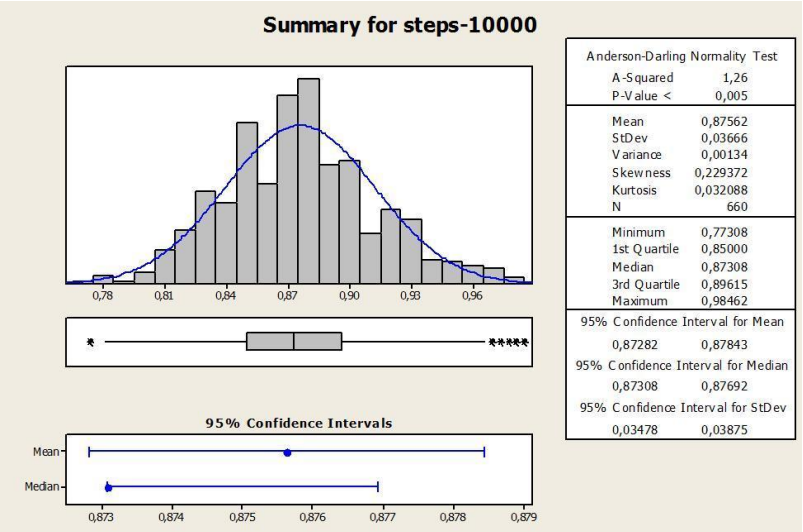


Figura APÊNDICE F.4 - Estatística para o grupo de execuções com 10000 passos