UNIVERSIDADE FEDERAL DE MINAS GERAIS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME BARBOZA MENDONÇA

HEURÍSTICA *BEST-FIRST* PARA COMPUTAR O ALINHAMENTO DE SEQUÊNCIAS

BELO HORIZONTE
2024

GUILHERME BARBOZA MENDONÇA

HEURÍSTICA *BEST-FIRST* PARA COMPUTAR O ALINHAMENTO DE SEQUÊNCIAS

Projeto Orientado em Computação para o curso de graduação em Ciência da Computação na Universidade Federal de Minas Gerais

Orientadora: Raquel Minardi

BELO HORIZONTE

2024

RESUMO

O alinhamento de sequências é um problema clássico porém ainda relevante e com várias aplicações práticas na área da Bioinformática. Soluções robustas e eficientes já foram propostas, entretanto, a quantidade crescente de dados gerados por projetos de sequenciamento progressivamente mais complexos exige o desenvolvimento de algoritmos e estratégias cada vez mais eficientes. Considerando tal demanda, esse projeto visa realizar um estudo sistemático das atuais técnicas para alinhamento de sequências e desenvolver uma ferramenta que reinterprete o problema para construir uma heurística maximal que faça um *tradeoff* entre o tempo de execução, espaço de memória consumido e qualidade da solução.
Palavras-Chave: Heurística, Alinhamento de Sequências, *Needleman-Wunsch*.

Lista de figuras

Lista de tabelas

Sumário

1) INTRODUÇÃO

O alinhamento de sequências é um desafio central na bioinformática que envolve fazer um rearranjo ou reordenação entre os elementos de duas ou mais sequências genéticas, visando encontrar similaridades que podem representar uma consequência funcional, estrutural ou evolutiva do relacionamento entre elas [1, 2]. Problemas de alinhamento podem ser subdivididos e classificados com base em suas propriedades e objetivos: em relação à quantidade de sequências sob análise, temos o alinhamento par-a-par e o de múltiplas sequências, enquanto que, quando se estuda a correlação e a qualidade do casamento, temos o alinhamento global *versus* local.

Técnicas de alinhamento são utilizados para análise e comparação filogenética [3], predição de estruturas secundárias [4] e montagem de genomas [5], mas seus usos não estão restritos à bioinformática, afinal, algoritmos de alinhamento já foram aplicados no Processamento de Linguagem Natural [6] e até na análise de transações em pesquisas relacionadas ao marketing e negócios [7, 8].

Embora soluções robustas já tenham sido propostas para esse problema, encontrar o relacionamento genético entre sequências é o objetivo de diversos estudos e artigos até hoje: ao passo que quantidades crescentes de dados estão sendo gerados por projetos de sequenciamento cada vez mais complexos, a escalabilidade e performance dos métodos de alinhamento se tornam fatores decisivos para o sucesso desses trabalhos, exigindo o desenvolvimento de soluções engenhosas que aproveitam de recursos e ferramentas modernos para implementar técnicas cada vez mais eficientes [9].

2) OBJETIVO

2.1) OBJETIVO GERAL

Dadas as considerações acima, a proposta geral para esse projeto é inicialmente estudar o algoritmo *Needleman-Wunsch*, uma técnica clássica que resolve o alinhamento global par-a-par de sequências com custo ótimo [10]. A partir dessa estratégia, iremos reinterpretar a definição do problema, propor uma estrutura alternativa de resolução e desenvolver uma heurística de alinhamento de sequências que prioriza reduzir o tempo de execução e a memória utilizada em detrimento da otimalidade e acurácia da solução.

Iremos realizar testes estatísticos e apresentar diversas variações dessa heurística para permitir uma melhor compreensão de suas propriedades e também para comparar seus resultados com outros métodos já consolidados.

2.2) OBJETIVOS ESPECÍFICOS

→Estudar e implementar o algoritmo Needleman-Wunsch em suas formas recursiva e iterativa.

→Reinterpretar o algoritmo Needleman-Wunsch como um grafo, abordando o problema do alinhamento como um caminhamento nesse grafo.

→Otimizar esse caminhamento ao definir uma heurística "*best-first*", gulosa e maximal;

→Propor uma generalização da heurística que permita o alinhamento global de múltiplas sequências;

→Implementar uma extensão que altera a condição de parada da heurística para prolongar o processamento e explorar outros caminhos, visando sair de máximos locais;

→Construir uma versão paralela da heurística com *threads* para tentar diminuir o tempo de processamento;

3) REFERENCIAL TEÓRICO

Diversos autores na literatura buscaram implementar soluções aproximativas mais eficientes para o alinhamento de sequências. Dox explica em sua tese como o alinhamento de sequências pode ser feito eficientemente com uma *string* e um gráfico de De Brujin [11]. Liu e Steinegger construíram um algoritmo que utiliza instruções *Single Instruction Multiple Data* (SIMD) com extensões vetoriais para paralelizar o cálculo da tabela de Programação Dinâmica do alinhamento par-a-par [12]. Júnior apresenta uma abordagem para o alinhamento múltiplo de sequências por meio da otimização de uma função objetivo utilizando Evolução Diferencial [13].

Higgins e Sharp propuseram a implementação do Clustal, um algoritmo para alinhamento múltiplo com uma estratégia de alinhamento progressivo que executa uma sequência de alinhamentos par-a-par, seguidos por uma composição no ramo de uma estrutura de árvore [14]. Notredame, Higgins e Heringa implementaram o T-Coffee, uma solução similar ao Clustal que também usa um alinhamento progressivo, mas faz um pré-processamento das sequências para obter uma biblioteca de informações preliminares que guiam o alinhamento progressivo, ajudando a evitar problemas oriundos da natureza gulosa desses algoritmos [15].

4) DESENVOLVIMENTO

4.1) Needleman-Wunsch original

O algoritmo Needleman-Wunsch é um método utilizado para o alinhamento de sequências, frequentemente aplicado na bioinformática para alinhar sequências de proteínas ou nucleotídeos (DNA ou RNA). Ele foi desenvolvido por Saul Needleman e Christian Wunsch em 1970 como uma técnica para encontrar o melhor alinhamento global entre duas sequências, maximizando a pontuação de similaridade entre elas [10].

Antes de darmos a sua implementação, precisamos definir alguns hiperparâmetros para sua execução:

4.1.1) Matriz de pontuação

Dependendo do problema (alinhamento de aminoácidos, proteínas, sequências abstratas, etc), o algoritmo precisa de uma matriz que defina a **pontuação** recebida ao fazer o alinhamento entre dois elementos, sejam eles idênticos (*match*) ou não (*mismatch*).

Nos exemplos que iremos estudar a seguir, desejamos fazer o alinhamento entre proteínas, compostas por sequências de 20 possíveis aminoácidos. Na literatura, vários autores já propuseram diversas matrizes de pontuação, chamadas de ***BLOSUM*** (***BLO****cks of Amino Acid* ***SU****bstitution* ***M****atrix*) e a principal versão que usaremos nesse projeto será a BLOSUM62, definida a seguir.

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	−1	5																		
Asn	−2	0	6																	
Asp	−2	−2	1	6																
Cys	0	−3	−3	−3	9															
Gln	−1	1	0	0	−3	5														
Glu	−1	0	0	2	−4	2	5													
Gly	0	−2	0	−1	−3	−2	−2	6												
His	−2	0	1	−1	−3	0	0	−2	8											
Ile	−1	−3	−3	−3	−1	−3	−3	−4	−3	4										
Leu	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4									
Lys	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5								
Met	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5							
Phe	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6						
Pro	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7					
Ser	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4				
Thr	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5			
Trp	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11		
Tyr	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	
Val	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4

Figura 1: Matriz de pontuação BLOSUM62

4.1.2) Penalidade *Indel*

No alinhamento entre sequências, podemos incluir ***indels*** que representam *gaps* ou intervalos de **in**serção/**del**eção nas *strings*, ou seja, em vez de "casar" um elemento com outro de cada sequência, insere-se um "espaço em branco" em uma sequência e faz o *match* com um elemento da outra sequência. Assim como para o *match* de elementos, essa operação tem um custo fixo pré-definido. Para padronizar os testes que realizaremos a seguir, definimos o valor dessa penalidade como **0**.

4.1.3) Implementação recursiva

O algoritmo Needleman-Wunsch aplica o paradigma da Programação Dinâmica para encontrar o alinhamento global ótimo entre duas sequências. Usualmente, uma sequência é representada como uma *string* e cada caractere é um elemento que pode ser casado com outro elemento ou com um *gap* na outra sequência, assumindo um valor de penalidade ou ganho pré-definido.

A estratégia do algoritmo é subdividir o problema original em subproblemas e resolvê-los individualmente e recursivamente, agregando seus resultados para compor a solução final do alinhamento. Os subproblemas não são mutuamente exclusivos e podem se sobrepor, logo, podemos armazenar seus resultados em uma matriz de memoização e reutilizá-los quando necessário.

Formalmente, o objetivo do algoritmo é encontrar o alinhamento que maximiza a somatória total dos custos/ganhos de *match*, portanto, podemos definir a seguintes regras de implementação: dadas duas strings v e w, seus tamanhos são $|v|$ e $|w|$ e podemos acessar seus elementos com $v[i]$ ($0 \leq i < |v|$) e $w[j]$ ($0 \leq j < |w|$). A operação v[:-1] retorna a *string* v sem o seu último caractere, sendo que se v tiver apenas um elemento, a regra retorna a *string* vazia. Seja pontuacao[v[i], w[j]] o ganho ou custo de se fazer o casamento entre o i-ésimo elemento de v e o j-ésimo elemento de w (*match* quando são iguais e *mismatch* quando são diferentes), e seja PENALIDADE_INDEL um valor constante que representa o custo ou penalidade de se realizar um *indel*, conforme definido acima.

Podemos, portanto, definir a **relação de recorrência**:

$$nw\_rec(a,b) = \begin{cases} 0 \text{ se } a = \text{""} \text{ e } b = \text{""} \\ PENALIDADE\_INDEL + nw\_rec(a, b[:-1]) \text{ se } a = \text{""} \text{ e } b \neq \text{""} \\ PENALIDADE\_INDEL + nw\_rec(a[:-1], b) \text{ se } a \neq \text{""} \text{ e } b = \text{""} \\ \text{se } a \neq \text{""} \text{ e } b \neq \text{""}: max \begin{cases} pontuacao[a[|a|-1], b[|b|-1]] + nw\_rec(a[:-1], b[:-1]) \\ PENALIDADE\_INDEL + nw\_rec(a[:-1], b) \\ PENALIDADE\_INDEL + nw\_rec(a, b[:-1]) \end{cases} \end{cases}$$

Figura 2: Relação de recorrência para o algoritmo Needleman-Wunsch

Por fim, basta resolver o problema para v e w ao chamar esse algoritmo recursivo como nw_rec(v, w). Uma implementação concreta desse algoritmo pode ser encontrado aqui.

4.1.3) Implementação iterativa

As regras de implementação e a sua relação de recorrência permitem que ele possa ser implementado de forma direta e intuitiva como um algoritmo recursivo, porém, sua implementação é geralmente dada de forma iterativa, pois o código dessa versão é menor, mais simples de implementar e fácil de depurar. Além disso, é comum salvar informações adicionais que permitam definir a sequência de passos que devem ser feitos para gerar o alinhamento final, além do resultado da somatória total. A implementação da versão iterativa desse algoritmo em Python pode ser acessada nesse *link*.

Tanto para sua versão recursiva quanto para sua versão iterativa, podemos afirmar que o principal objetivo do *loop* de execução envolve preencher a matriz de similaridade (ou tabela de memoização), logo, a complexidade de tempo e espaço de ambas as versões do algoritmo é $O(|v| \times |w|)$. Como um exemplo de aplicação, na tabela a seguir, descrevemos a composição final da matriz de similaridade após executar o Needleman-Wunsch para as sequências v=”EQW” e w=”EAQ”:

0	0	0	0
0	5	5	5
0	5	5	10
0	5	5	10

Tabela 1: Matriz de similaridade do Needleman-Wunsch para v=”EQW” e w=”EAQ”

Ainda no exemplo acima, vale destacar que o ganho total absoluto do alinhamento para os parâmetros configurados foi de 10 (valor na célula do canto inferior direito), e o alinhamento final em si é igual a:

→ E-QW

→ EAQ-

4.2) Reinterpretação do problema

Como dito anteriormente, no algoritmo original do Needleman-Wunsch, o objetivo do *loop* de execução é preencher a matriz de similaridade (equivalente à matriz de memoização da versão recursiva) com os valores acumulados dos custos e ganhos de fazer casamentos com os elementos. Mais especificamente, a posição [i, j] nessa tabela equivale ao ganho acumulado de fazer o alinhamento par-a-par das *substrings* v[0:i] e w[0:j] e o valor dessa posição depende da penalidade de *indel*, da pontuação a ser adicionada/removida em um casamento e dos valores já existentes nas posições [i−1,j], [i,j−1] e [i−1,j−1] (por isso começamos na posição [0,0], afinal, fazer o alinhamento entre duas strings vazias tem custo 0).

Por fim, a escolha de qual decisão tomar (*indel* em v[i], *indel* em w[j] ou casamento) depende do maior valor agregado possível, portanto, podemos afirmar que esse algoritmo busca encontrar a **sequência de passos** que **maximizam** a pontuação acumulada final (valor na posição [|v|, |w|] na matriz).

Nessa análise, os termos "Sequência de passos" e "maximização" remetem a um outro tipo de problema: **Caminhamento ótimo em grafos**. De fato, podemos remapear as estruturas auxiliares do algoritmo original como um grafo direcionado e reinterpretar esse problema como um caminhamento nessa estrutura:

Dadas strings v e w, a matriz de similaridade é uma matriz numérica de dimensões (|v|+1, |w|+1) (índices começando em 0). Podemos construir um **grafo com (|v|+1 × |w|+1) nós**, um para cada elemento na matriz, ou seja, o elemento na posição matriz_similaridade[i][j] dará origem ao nó (*i*, *j*). Visando facilitar a visualização de sua estrutura, seria conveniente dispô-los em uma posição próxima à posição original em que um elemento de índices equivalentes ficaria na tabela original.

Cada nó está associado a um valor numérico correspondente à pontuação acumulada que é inicialmente desconhecido, com exceção do nó (0,0), que inicia com 0 (não há custo ou ganho para alinhar duas sequências vazias). Sabemos que o calculo desse valor na matriz de similaridade depende dos elementos imediatamente acima, à

esquerda e na diagonal superior esquerda, logo, para cada nó (i,j), devemos adicionar **3 arestas direcionadas e ponderadas**, uma direcionada para o nó (i+1,j), uma para o nó (i,j+1) e uma terceira apontando para (i+1,j+1) (com exceção dos nós nas "bordas" inferior e direita do grafo).

Caminhar de (i,j) para (i+1,j) (abaixo) equivale a fazer uma inserção em w e forçar o *match* com v (ou seja, casar v[i] com um *indel* em w). De forma similar, caminhar de (i,j) para (i,j+1) (à direita) equivale a fazer uma inserção em v e forçar o *match* com w (ou seja, casar w[j] com um *indel* em v), logo, o peso de todas as arestas "verticais" e "horizontais" deverão ter peso igual à penalidade de *indel*. O peso da aresta diagonal que leva (i,j) para (i+1,j+1) (diagonal inferior direita) deverá ter peso igual ao custo ou ganho de se fazer o casamento entre os elementos $v[i]$ e $w[j]$.

A partir dessa nova estrutura, podemos redefinir o objetivo do problema como sendo encontrar o caminho de **maior custo** nesse grafo, saindo do nó fonte f=(0,0) e chegando no nó sumidouro s=(i,j). Para permitir uma melhor interpretação, a figura a seguir representa a matriz de similaridade inicial (ainda não preenchida) do Needleman-Wunsch e essa nova estrutura equivalente como um grafo para as sequências v=”EQW” e w=”EAQ”:

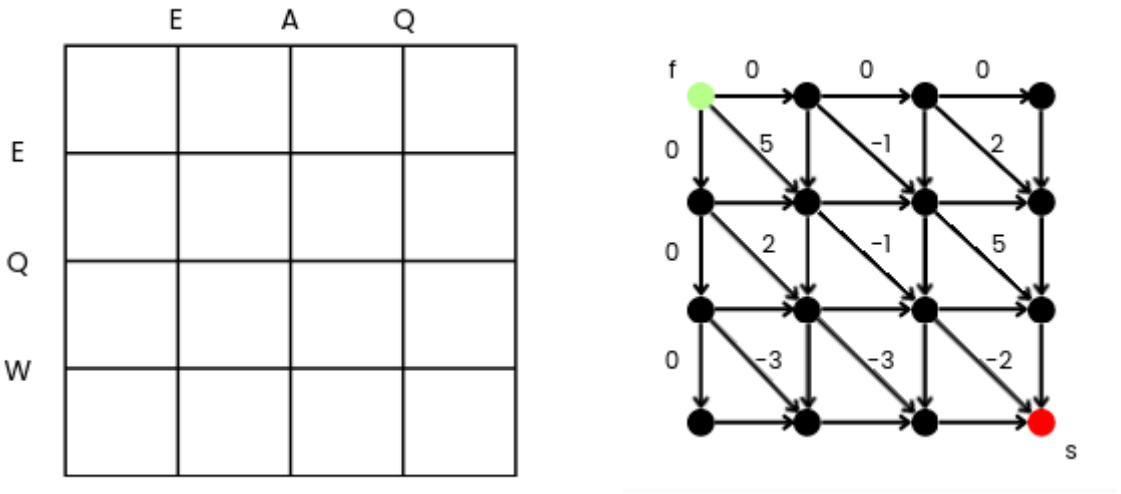


Figura 3: A matriz de similaridade inicial (à esquerda) do Needleman-Wunsch original e o grafo equivalente (à direita) para v=”EQW” e w=”EAQ”

4.2.1) Solução por Bellman-Ford

A solução mais comum para solucionar problemas de caminho de custo ótimo envolve aplicar o algoritmo de Dijkstra, porém, na definição acima, algumas arestas podem conter pesos negativos. Dada essa restrição, iremos implementar uma variação do algoritmo menos eficiente de Bellman-Ford (pois tanto Dijkstra quanto Bellman-Ford originais resolvem o caminho de custo mínimo, mas desejamos encontrar o caminho de custo máximo). O código desse algoritmo em *Python* pode ser visualizado nesse *link*.

Para essa implementação, devemos exaustivamente percorrer todos os vértices e relaxar todas as arestas para gerar a distância acumulada aproximada em relação à fonte. Um ponto interessante relacionado à sua implementação é que, no algoritmo original, deveríamos ter codificado um passo adicional ao final do algoritmo que verifica se há ciclos com peso acumulado negativo, porém, no capítulo a seguir, discutiremos por que essa etapa não é necessária.

Os alinhamentos gerados pelo Bellman-Ford são tão bons quanto os computados pelo Needleman-Wunsch original, tanto é que a matriz de similaridade é exatamente a mesma em ambos os algoritmos. Por outro lado, a performance daquele é muito pior do que a deste: O Bellman-Ford tem complexidade $O(|V|\times|E|)$, onde $|V|$ é a quantidade de vértices e $|E|$ é a quantidade de arestas no grafo.

No problema em questão, podemos computar *a priori* a quantidade de nós e arestas em função do tamanho das *strings*. Dadas duas sequências v e w, como o grafo é gerado a partir da matriz de similaridade definida no Needleman-Wunsch original, teremos $|v|+1\times|w|+1$ nós. Para cada nó, com exceção dos que se encontram nas "bordas" inferior e direita, teremos que criar 3 arestas que levam aos nós abaixo, à direita e na diagonal inferior direita, logo, teremos $3\times(|v|+1\times|w|+1)-2\times|v|-2\times|w|+1$ arestas (o $-2\times|w|$ desconta os nós inferiores, o $-2\times|v|$ desconta os da borda à direita e o +1 compensa pelo nó na ponta inferior direita que foi contado duas vezes).

Portanto, a complexidade do Bellman-Ford nessa implementação é

$$O(|V|\times|E|) =$$
$$O((|v|+1\times|w|+1)\times(3\times(|v|+1\times|w|+1)-2\times|v|-2\times|w|+1)) =$$
$$O(|v|^2\times|w|^2)$$

que é bem superior à complexidade $O(|v|\times|w|)$ do Needleman-Wunsch. Sob uma outra perspectiva, se $|v|\approx|w|$, o Needleman-Wunsch teria complexidade quadrática enquanto que o Bellman-Ford teria complexidade polinomial à quarta potência.

4.3) Best-first Needleman-Wunsch

Concluímos no capítulo anterior que o Bellman-Ford entrega o mesmo resultado que o Needleman-Wunsch original, mas com uma performance consideravelmente pior. Se, porém, reavaliarmos a estrutura do grafo gerado nessa estratégia, iremos notar que todas as arestas levam a um único nó sumidouro. Outra forma de se interpretar isso é que, pela sua construção, todas as arestas apontam para uma de três direções: para baixo, para a direita ou para a diagonal inferior direita. A partir dessa análise, podemos concluir que um grafo construído para solucionar o alinhamento de sequências é **acíclico**, formando um **DAG** (*Directed Acyclic Graph*).

A presença de pesos negativos implica a possibilidade da existência de ciclos negativos, o que impõe preocupações e limitações severas em algoritmos de caminhamento, porém, se o grafo não possui ciclos, obviamente, ele não possui ciclos negativos. Considerando esse fato, podemos otimizar o algoritmo desenvolvido na seção anterior se, em vez de nos inspirarmos no Bellman-Ford, fazermos uma adaptação Heurística do Dijkstra.

Nesse capítulo, iremos desenvolver o **BFNW** - **B**est-**F**irst **N**eedleman-**W**unsch, uma Heurística que usa uma estratégia gulosa "*best-first*" para encontrar o alinhamento global de um par de sequências. A ideia envolve simular o caminhamento no grafo ponderado, direcionado e acíclico descrito no capítulo anterior, porém, o caminho de custo máximo será computado usando uma técnica de priorização similar à de Dijkstra: a cada iteração, o próximo nó a ser expandido será aquele que leva ao maior caminho imediato. A implementação desse algoritmo em Python pode ser encontrada nesse link.

Alguns detalhes de implementação merecem destaque nesse algoritmo: Primeiramente, o uso da estrutura de dados *max-heap* permite que o algoritmo priorize caminhos mais promissores (que equivalem a alinhamentos com um ganho maior) ao sempre processar primeiro o nó com o maior ganho acumulado na *heap* (por isso o nome *best-first*). Vale ressaltar, porém, que esse caminho/alinhamento é computado de forma gulosa (*greedy*) e não necessariamente é o melhor caminho possível. O fato dessa *heap* ter complexidade assintótica de inserção e acesso $O(log(n))$ é um fator de otimização essencial para essa implementação. Por fim, outra decisão importante para essa estratégia é que o algoritmo está configurado para imediatamente parar assim que o nó sumidouro (na última posição, (i,j)) for alcançado pela primeira vez.

Para entendermos a sua ordem de complexidade, podemos analisar seu fluxo de execução. Para simplificar, seja $k=|v|\times|w|$. A criação do grafo em si é proporcional à

multiplicação dos tamanhos das sequências ($O(k)$). Sua iteração principal é realizada enquanto a *heap* não estiver vazia e o número de elementos a serem inseridos nela também é proporcional ao tamanho multiplicado das *strings*, logo, o número de iterações é da ordem $O(k)$. As funções de extração (feita uma vez) e inserção (feita 3 vezes) na *heap* são ambas $O(log_2(k))$. O restante das operações são constantes e atômicas, com complexidade $O(1)$, logo, a complexidade total do algoritmo é:

$$O(k)+O(k\times(O(log_2(k)+3\times O(log_2(k))+O(1)))=$$
$$O(k\times log_2(k))=$$
$$O(|v|\times|w|\times log_2(|v|\times|w|))$$

A partir desse cálculo, verificamos que a complexidade assintótica do BFNW é maior que a do Needleman-Wunsch original e, a princípio, poderíamos argumentar que não há nenhum tipo de ganho ao usar essa heurística. Entretanto, nos testes que executaremos a seguir, encontraremos um resultado positivamente contraditório.

Na tabela a seguir, disponibilizamos os valores encontrados nos nós expandidos do grafo após realizar o BFNW para v=”EQW” e w=”EAQ”, organizados de maneira tabular. Note que há nós que não foram percorridos.

0			
	5	5	
	5		10
			10

Tabela 2: Valores nos nós do grafo do BFNW para v=”EQW” e w=”EAQ”

4.3.1) Análise estatística e comparativa dos algoritmos

Para fazer uma análise mais profunda e que permita uma comparação real entre o Needleman-Wunsch original e o BFNW implementado acima, usaremos uma metodologia estatística que envolve os seguintes passos:

1) Defina um tamanho fixo para as strings de teste. Nesse trabalho, definiremos como 100.

2) Gere uma *string* aleatória de tamanho 100, dado um dicionário específico. Nesse caso, usaremos o dicionário dos 20 aminoácidos disponíveis.

3) A partir da *string* original gerada em 2), aplique uma mutação de tal forma que k % da *string* original fique mutada.

4) Usando a *string* original gerada em 2) e a *string* mutada em 3), compute o alinhamento dessas sequências usando o Needleman-Wunsch original, armazenando a quantidade de células geradas na tabela de programação dinâmica, o tempo levado para executar o algoritmo, o valor ótimo do alinhamento (valor na posição [última linha, última coluna] da matriz de similaridade) e a porcentagem de identidade.

5) Ainda usando essas *strings*, compute o alinhamento com o algoritmo BFNW, armazenando a quantidade de nós expandidos, o tempo levado para executar essa Heurística, o melhor valor do alinhamento (valor do nó (última linha, última coluna) da matriz de similaridade equivalente) e a porcentagem de identidade.

6) Repita os passos 3)-5) variando o grau de mutação de 0 (nenhuma mutação) a 100 (totalmente mutado), armazenando os valores de células calculadas/nós expandidos, tempo levado e valor ótimo obtido em cada algoritmo para cada valor de mutação.

7) Para evitar vieses e gerar uma análise estatística mais confiável, execute os passos 2)-6) 1000 vezes e tire as médias de todos os resultados.

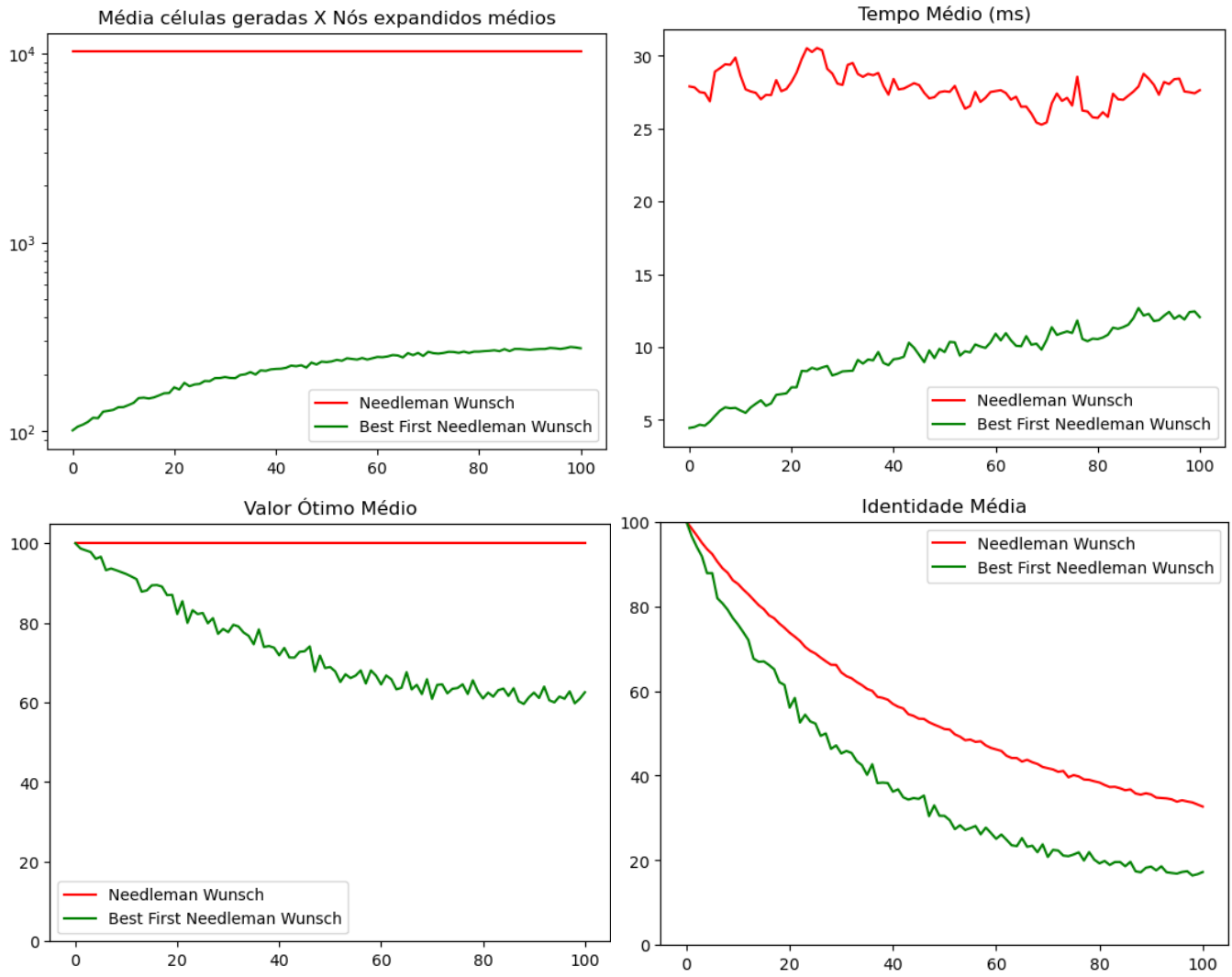


Figura 4: Resultados dos testes comparativos entre o Needleman-Wunsch e o BFNW

Para os parâmetros definidos (sequências de tamanho 100), a quantidade de células processadas na matriz de memoização do NW é sempre a mesma: (100+1)×(100+1)=10201. Por outro lado, para todos os graus de mutação, a quantidade de nós expandidos pelo BFNW foi várias ordens de grandeza menor que a quantidade de células computadas (note que os valores no eixo de células geradas/nós expandidos está em escala logarítmica). A partir desse resultado, podemos concluir que o BFNW é mais eficiente no uso de memória que o Needleman-Wunsch original.

Ao visualizar o gráfico de tempo médio levado, fica evidente que o BFNW foi mais rápido que o Needleman-Wunsch para todos os graus de mutação - a execução mais demorada do BFNW ainda levou menos da metade do tempo da mais rápido do Needleman-Wunsch. Ainda assim, é interessante analisar e comparar o formato das

curvas: No Needleman-Wunsch, o tempo levado é quase constante, não sendo influenciado pelo grau de mutação, o que é um comportamento esperado, afinal, independente do grau de mutação, o tamanho da matriz de similaridade a ser processada é sempre o mesmo.

Para o BFNW, porém, a mutação teve um impacto evidente no tempo: quanto maior o grau de mutação, mais tempo é necessário para processar o alinhamento. Esse comportamento pode ser explicado se revisitarmos a matriz de pontuação BLOSUM62 que estamos utilizando: casamentos entre elementos idênticos (*matches*) levam a um ganho positivo alto, enquanto que casamentos entre elementos diferentes (*mismatches*) resultam em penalidades negativas.

Assim, ao compararmos sequências com poucas mutações, *matches* perfeitos ocorrerão com mais frequência e a pontuação acumulada constantemente aumentará, gerando um efeito dominó que leva o algoritmo a uma finalização rápida e com poucos nós expandidos. Por outro lado, em sequências com alta mutação, vários pesos negativos serão introduzidos graças a casamentos com *mismatches* e isso fará com que mais nós correspondentes a *indels* sejam expandidos e caminhos indiretos sejam percorridos/explorados, aumentando o tempo de execução.

O objetivo principal de ambos os algoritmos é gerar o alinhamento entre as sequências, porém, um subproduto de suas execuções é o Valor Ótimo (dado numérico encontrado na posição/nó $[i,j]$ após a execução) que corresponde ao total de pontos acumulados (ou penalidades subtraídas) para o alinhamento gerado. Podemos usar o valor relativo para comparar a qualidade dos resultados: sabemos que o Needleman-Wunsch é um algoritmo ótimo, sempre entregando o melhor alinhamento possível, logo, o Valor Ótimo Médio para esse algoritmo é sempre 100%.

Se usarmos o resultado gerado pelo Needleman-Wunsch como linha de base e calcularmos o Valor Ótimo encontrado pelo BFNW, concluiremos que o grau de mutação também influencia na qualidade da solução gerada, afinal, *strings* similares geram pontuações muito boas, mas ao passo que o grau de mutação aumenta, as soluções obtidas ficam piores em relação ao alinhamento ótimo. O resultado começa extremamente acurado em 100% com nenhuma mutação, mas decresce rapidamente até estabilizar por volta de 65% de otimalidade com mutação total. Esse resultado enfatiza ainda que o BFNW tem natureza heurística e seus resultados não são perfeitos.

Outra métrica de comparação interessante é a medida de identidade: dadas duas sequências já alinhadas, a identidade é a porcentagem relativa que indica o quão

similares ou parecidas essas *strings* são. A computarmos essa métrica nos testes acima, podemos observar uma correlação importante: a identidade média de ambos os algoritmos é total quando não há mutação (o que faz sentido, afinal, o alinhamento de sequências idênticas é a própria sequência) mas cai rapidamente quando aumentamos o grau de mutação (o que também faz sentido, pois o alinhamento de *strings* muito diferentes será muito ruim).

Ainda assim, é importante notar que as medidas de identidade para o BFNW sempre são iguais ou menores que as identidades do Needleman-Wunsch original (começam iguais, mais se separam cada vez mais à medida que o grau de mutação aumenta), o que é esperado, afinal, o BFNW não é um algoritmo ótimo.

A partir desses resultados, argumentamos que o BFNW se sai extremamente bem quando está fazendo o alinhamento de sequências que sabemos que são relativamente similares, pois, nessas condições, os resultados são acurados, são gerados com extrema rapidez e ocupam pouquíssima memória durante o processamento. Por outro lado, para sequências intencionalmente dissimilares, o BFNW ainda é rápido, continua levando pouca memória de processamento, mas a qualidade dos resultados é relativamente pior em relação ao alinhamento ótimo encontrado pelo Needleman-Wunsch tradicional.

Ainda assim, nessa última situação, mesmo que o resultado do BFNW seja ruim, o alinhamento ótimo do Needleman-Wunsch também será ruim na prática (isso é corroborado pelo último gráfico acima, pois a Identidade Média resultada pelo NW para altas mutações alcança níveis tão baixos quanto 40%, o que não difere muito das piores identidades do BFNW, que alcançam 20%).

4.3.2) Testes com proteínas reais

Em um teste final, desejamos comparar a performance entre esses algoritmos ao alinhar sequências reais. As *strings* que iremos usar para testes correspondem às glicoproteínas *spike* dos vírus SARS-CoV e SARS-CoV-2 [17, 18].

Métrica/Algoritmo	**Needleman-Wunsch**	**BFNW**
Células geradas/nós expandidos	1470144 células	3683 nós
Tempo levado (ms)	11848.431 ms	502.239 ms
Valor ótimo	4898	1638
Identidade (%)	63.24%	12.05%

Tabela 3: Resultados comparativos entre Needleman-Wunsch e BFNW para glicoproteínas reais

É evidente que o BFNW foi bem mais rápido e necessitou de menos memória para processar as *strings*, porém, a qualidade do alinhamento foi muito ruim: Enquanto o Needleman-Wunsch alcançou uma identidade razoável de 63%, o BFNW não conseguiu acertar nem 20% dos elementos nas sequências alinhadas.

4.4) Global-multiple BFNW

O algoritmo definido no capítulo anterior faz o alinhamento global entre um par de sequências, porém, dado um conjunto $S=\{s_1, s_2, ..., s_{k-1}, s_k\}$ com k *strings*, a mesma ideia pode ser expandida e generalizada para alinhar esse conjunto simultaneamente. Isso pode ser feito ao modificar a estratégia de geração do grafo inicial para que, em vez de ter um aspecto bidimensional fixo, ele possa ser expandido para uma estrutura dinâmica k-dimensional.

A partir desse novo grafo com estrutura similar a um tensor, basta executar exatamente a mesma estratégia de priorização, mas agora o caminhamento deve ser feito a partir do nó fonte $(0,0,...,0,0)_k$ até o nó sumidouro $(|s_1|,|s_2|,...,|s_{k-1}|,|s_k|)_k$. A implementação em Python para essa versão do algoritmo pode ser encontrada nesse link.

Como um exemplo didático de execução, a tabela a seguir apresenta o resultado do alinhamento múltiplo global gerado para 5 sequências:

Alinhamento global múltiplo das *strings* a seguir: CQWKSKVKCLSPIANGVLKARPFECWVYQC CQWKSKVKCGSPITTGVLKARPFVCWMYQC CQWQSKVKCFSPEANGVAKARPIECWVYQC CQWKSKVHCFEPHANGVLKARVFECWNYQC CVWKSKMKCLSNIANGVLKARPIEAWVYYC
Maior Pontuação: 564
Alinhamento: CQWKSKVKC----L--S-----PI-AN----GVLKARP-F-E---C-WVYQC-------------------- CQWKSKVKC----------G-SPI--------------------------TTGVLKAR-PF-VCW-MYQC CQWQSKVKCF-S--PEAN-GVA--KARPI-----E---CWV--YQC------------------------- CQWKSKV----------HC----FE--P--H----A----------------NGVLKARV-FE-CWN-YQC C-----V--WKSKM-K--C-LS----N-IANGVLKARP--IEAW--VY-Y-C--------------------

Tabela 4: Resultados do alinhamento simultâneo de 5 sequências com a versão múltipla do BFNW

4.5) Delayed Stop BFNW

A partir das análises feitas nas seções anteriores, concluímos que o BFNW faz um *tradeoff* entre tempo de execução e qualidade do alinhamento: o tempo economizado em relação ao NW original é muito grande, porém, dependendo das sequências sendo alinhadas, a qualidade da solução fica muito prejudicada.

Para explicar porque essa perda na qualidade ocorre, devemos revisitar a versão de alinhamento múltiplo desenvolvido no capítulo anterior (os algoritmos são tecnicamente diferentes, mas a estratégia de execução é exatamente a mesma), afinal, para esse algoritmo, é mais fácil visualizar um efeito negativo dessa implementação: especialmente no início da execução, toda vez que o algoritmo encontra um "sub-alinhamento" minimamente bom (mesmo que seja relativamente ruim na prática), ele "aceita" esse resultado e avança rapidamente a execução até chegar na sua condição de parada. Nesse caso, ele tende a não voltar para buscar por soluções melhores, simulando algo como cair em um máximo local.

Na sua implementação padrão, o BFNW usa uma *heap* para caminhar do nó fonte até o sumidouro e sua execução dura até a *heap* ficar vazia ou até o **nó sumidouro ser alcançado pela 1ª vez**. Se a *heap* ficar vazia, não há mais o que processar. Por outro lado, interromper imediatamente assim que se alcança o sumidouro é uma condição de parada discutivelmente precoce e abrupta.

Portanto, neste capítulo, iremos construir o **Delayed Stop Best-First Needleman-Wunsch** (ds-BFNW) que "atrasa" ou adia a condição de parada em algumas iterações para incentivar uma busca mais expressiva por soluções melhores ao permitir que o algoritmo expanda mais nós antes de finalizar. A implementação dessa adaptação do algoritmo com Python pode ser encontrada nesse link.

A principal característica desse código é a inserção de um novo parâmetro “*num_stops*”, que atua como um contador para a quantidade de vezes que o nó sumidouro deve ser atingido até que o algoritmo pare o processamento. Nas tabelas a seguir, executamos uma série de testes isolados ao alinhar as sequências v=”EQW” e w=”EAQ” com o número de paradas (valor do parâmetro num_*stops*) variando de 1 a 4:

0			
	5	5	
	5		10
			10

num_stops=1

0			
	5	5	5
	5	5	10
	5	5	10

num_stops=2

0			
	5	5	5
	5	5	10
	5	5	10

num_stops=3

0	0	0	0
0	5	5	5
0	5	5	10
0	5	5	10

num_stops=4

Tabela 5: Resultado dos alinhamentos entre v=“EQW” e w=”EAQ” do ds-BFNW para “num_*stops”* variando de 1 a 4

Nos testes acima, quando num_stops=1, o algoritmo funciona exatamente como a implementação do BFNW padrão. Para num_stops igual a 2, notamos que novos caminhos foram explorados, porém, nenhum resultado melhor foi encontrado e colocado nos nós já expandidos. Quando num_stops=3, nenhuma modificação foi feita na matriz, o que indica que, mesmo que um novo caminho seja percorrido, esse algoritmo não garante uma melhora constante dos resultados a cada iteração.

Por fim, ao alinharmos as sequências com num_stops=4, toda a matriz de similaridade foi preenchida, o que é uma consequência do algoritmo rodar até expandir todos os nós exaustivamente (até a *heap* estar vazia). Isso é um evento problemático, pois, por mais que o alinhamento seja perfeito (mesmo que o gerado pelo Needleman-Wunsch original), o tempo de execução é ainda pior que o Bellman-Ford implementado anteriormente, afinal, além de expandir e processar os mesmos nós e arestas que o Bellman-Ford, o ds-BFNW também tem o custo de gerenciar a *heap*.

4.5.1) Análise estatística e comparativa

Dadas sequências de tamanho qualquer, infelizmente não há como saber *a priori* até quantos passos devemos fazer para garantir uma solução melhor mas sem cair na situação de busca total exaustiva descrita acima, logo, nessa subseção, iremos realizar uma análise estatística para compreender como esse novo parâmetro altera as métricas de execução.

1) Defina um tamanho fixo para as *strings* de teste. Nesses exemplos, usaremos sequências de tamanho 25. Defina também uma quantidade de máxima de paradas a serem feitas. A seguir, faremos até 5 paradas.

2) Gere uma *string* aleatória de tamanho 25, dado um dicionário específico. Nesse caso, usaremos o dicionário dos 20 aminoácidos disponíveis.

3) A partir da *string* original gerada em 2), aplique uma mutação de tal forma que k % da *string* original fique mutada.

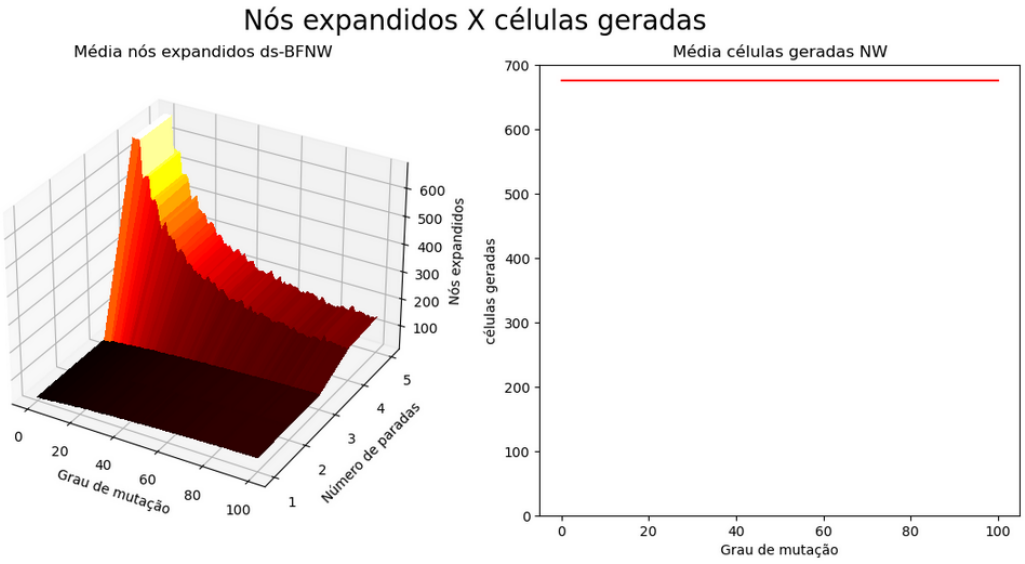
4) Usando a *string* original gerada em 2) e a *string* mutada em 3), compute o alinhamento dessas sequências usando o Needleman-Wunsch original, armazenando as suas métricas.

5) Ainda usando essas *string*, compute o alinhamento com o algoritmo ds-BFNW variando o número de paradas de 1 até 5, armazenando as suas métricas.

6) Repita os passos 3)-5) variando o grau de mutação de 0 (nenhuma mutação) a 100 (totalmente mutado).

7) Para evitar vieses e gerar uma análise estatística mais confiável, execute os passos 2)-6) 1000 vezes e tire as médias de todos os resultados.

As métricas descritas são as mesmas coletadas no teste realizado para comparar o NW original e o BFNW padrão no capítulo 4.3.1) deste trabalho.



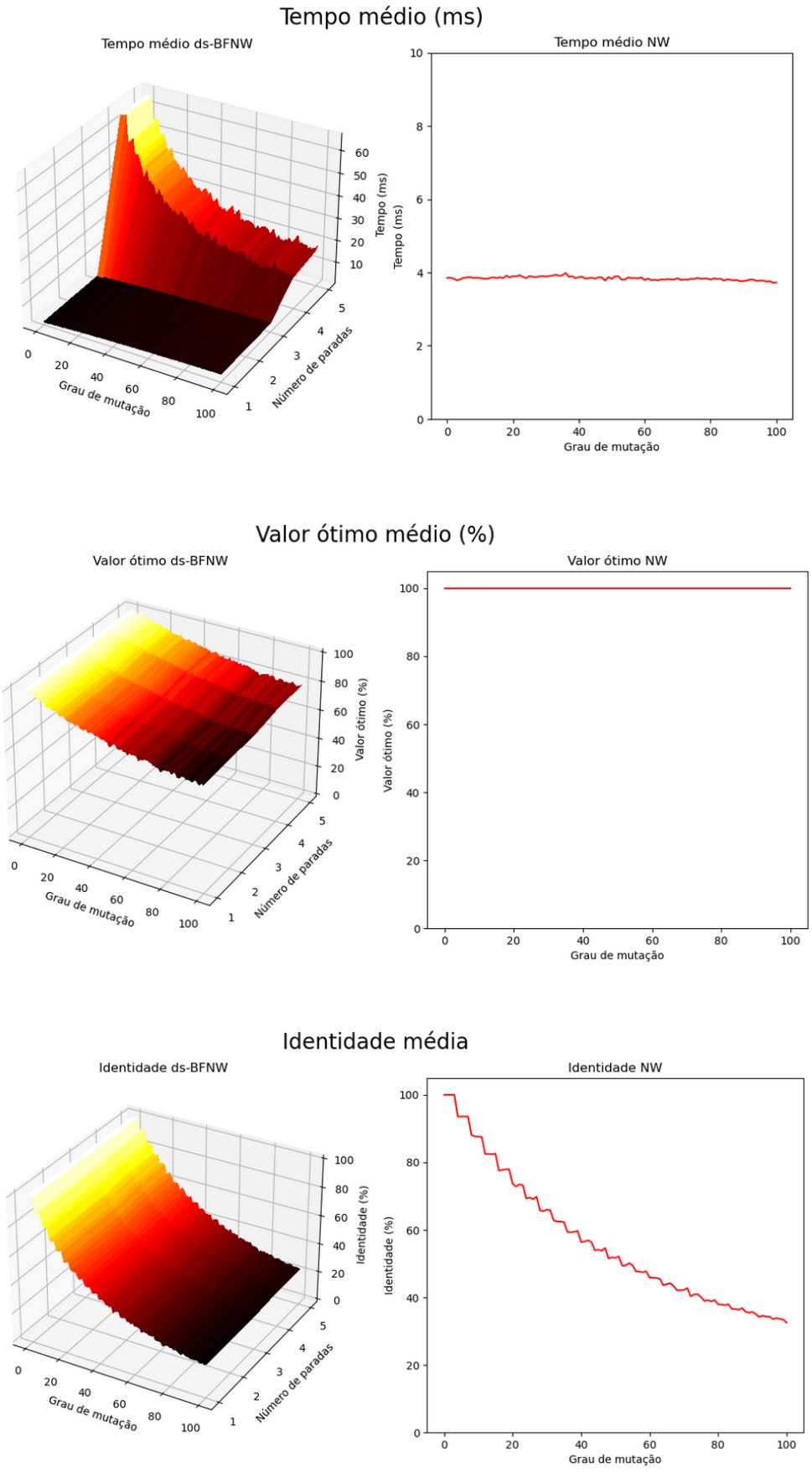


Figura 5: Resultados dos testes comparativos entre o Needleman-Wunsch e o ds-BFNW

Pela análise dos resultados acima, o que mais chama atenção é que, para todos os gráficos, há uma quebra abrupta nas tendências para o eixo do número de paradas quando o algoritmo começa a computar o ds-BFNW com 4 paradas. Esse evento fica mais evidente no gráfico de Nós expandidos e isso acontece pois, na média, para os parâmetros configurados, 4 paradas são suficientes para que o algoritmo já encontre a solução máxima real e comece a excessivamente procurar outros caminhos sem melhora, tanto é que, para alguns valores, ele expande literalmente todos os nós, ou seja, a condição de parada não foi o numero de *stops* alcançados, mas sim a ausência de nós na *heap* para expandir.

Esse evento é indesejado pois, por mais que os resultados do alinhamento sejam absolutamente ótimos (nos gráficos, esses valores não estão exatamente no 100%/máximo possível pois eles representam uma média geral), assim como discutido acima, expandir todos os nós é completamente ineficiente e seria preferível só realizar o Needleman-Wunsch original em vez disso.

Por outro lado, se compararmos os resultados obtidos com o número de paradas apenas entre 1 e 3, concluiremos que o ds-BFNW com 3 paradas computou um alinhamento absolutamente melhor em relação ao BFNW padrão (número de paradas igual a 1) para todos os graus de mutação, afinal, o número de nós expandidos não aumentou muito, mas o ganho na porcentagem do valor ótimo e na identidade são notáveis. Além disso, o *tradeoff* de tempo foi relativamente barato para uma solução que conseguiu se aproximar um pouco mais aos resultados ótimos do NW original.

Finalmente, por mais que a qualidade estatística desse método seja superior ao BFNW básico, a utilidade prática dessa solução é questionável no sentido de que pode ser custoso descobrir a quantidade ideal de paradas para um problema qualquer.

4.5.2) Testes com proteínas reais

Por fim, iremos novamente avaliar esse novo algoritmo ao alinhar sequências reais. As *strings* que usaremos serão as mesmas glicoproteínas *spike* dos vírus SARS-CoV e SARS-CoV-2 usadas em testes anteriores.

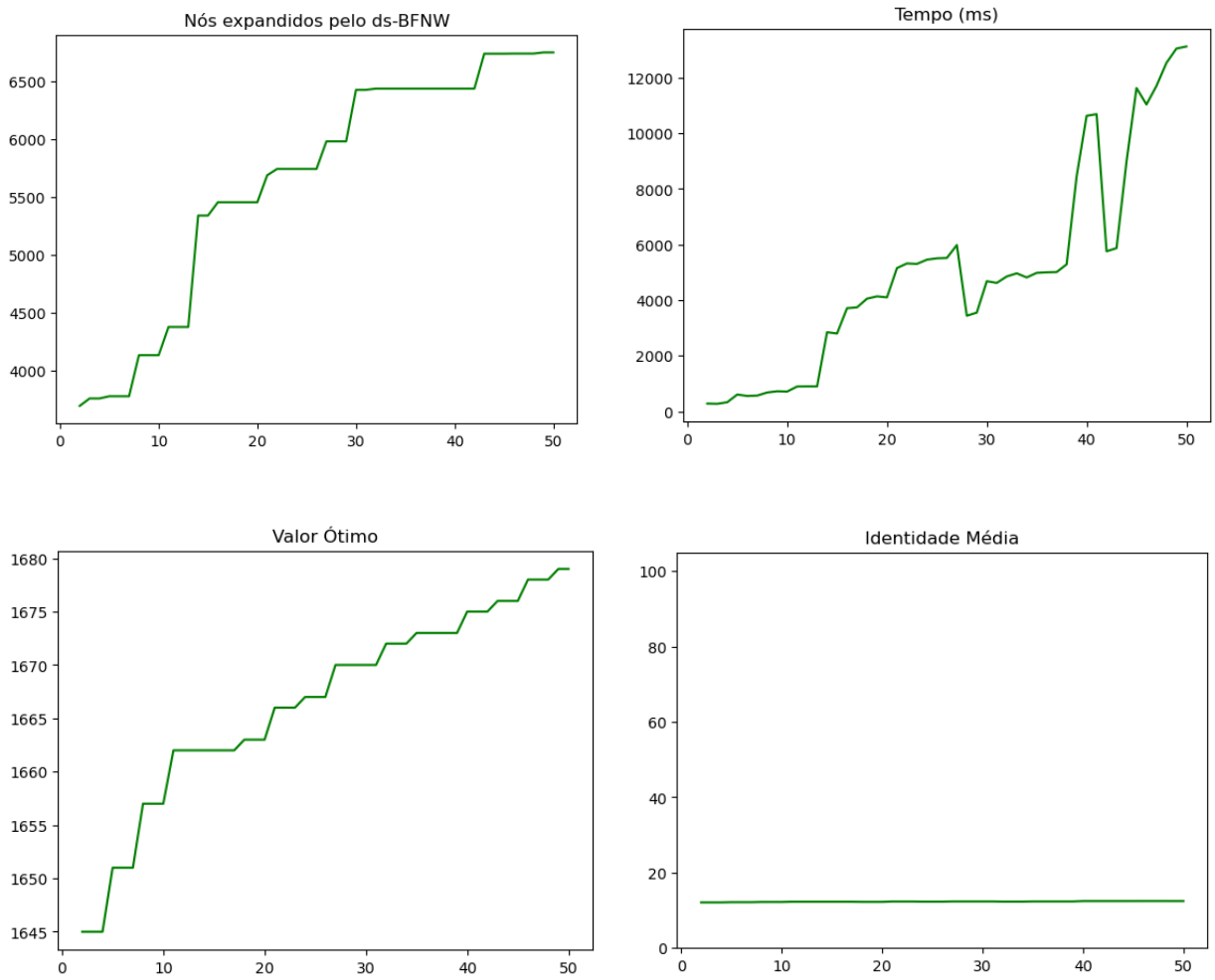


Figura 6: Resultados comparativos entre Needleman-Wunsch e ds-BFNW para glicoproteínas reais

Métrica/Algoritmo	**Needleman-Wunsch**	**BFNW**	**ds-BFNW 5 stops**	**ds-BFNW 10 stops**
Células /nós	1470144 células	3683 nós	3776 nós	4131 nós
Tempo levado (ms)	11848.431 ms	502.239 ms	558.667 ms	758.3 ms
Valor ótimo	4898	1638	1651	1657
Identidade (%)	63.24%	12.05%	12.16%	12.22%

Tabela 6: Resultados comparativos entre Needleman-Wunsch, BFNW e ds-BFNW (5 e 10 paradas) para glicoproteínas reais

Novamente, pela análise dos gráficos gerados e dos resultados sintéticos apresentados na tabela acima, concluímos que o ds-BFNW não representa uma melhora tão significativa em relação ao BFNW padrão, entregando soluções com quase a mesma identidade mas consumindo um pouco mais de tempo.

Mesmo que esse algoritmo seja preferível, ainda teríamos o custo de deduzir qual o número de paradas ideal para que se tenha um *tradeoff* razoável entre tempo e qualidade, afinal, pelos gráficos acima, mesmo com 50 *stops*, os resultados ainda não melhoraram muito, indicando que deveríamos/poderíamos explorar ainda mais o espaço de busca do grafo.

4.6) BFNW Paralelo

Em uma última tentativa de diminuir ainda mais o tempo de execução do algoritmo, implementamos uma adaptação que aplica conceitos de programação concorrente e *threads* para fazer o processamento de forma paralela. A implementação dessa versão na linguagem C++ encontra-se disponível nesse link.

Nesse algoritmo, alguns detalhes especiais precisaram ser observados: primeiramente, a *heap* se tornou uma variável global, utilizada por todas as *threads* configuradas, logo, todo acesso (seja para inserção ou consulta) deve ser feito em exclusão mútua para evitar condições de corrida. Em especial, foi usada uma *mutex* dedicada para restringir o acesso de leitura e inserção na *heap* e também uma variável de condição para verificar se a *heap* está vazia (o que não é mais uma condição de parada).

Além disso, também foi necessário criar uma *mutex* para configurar e verificar se a condição de parada foi alcançada, afinal, se uma *thread* chegar no nó sumidouro, todas as *threads* podem interromper suas trilhas de execução. Por fim, uma *mutex* adicional foi configurada para gerenciar o acesso à tabela correspondente ao grafo de similaridade e garantir a consistência dos valores resultantes.

Dada a natureza não determinística dos algoritmos paralelos e a complexidade intrínseca das linguagens de baixo nível como C++ (se comparada, por exemplo, com Python), a implementação e o teste dessa versão do algoritmo foram significativamente mais difíceis em relação às versões implementadas nos capítulos anteriores.

Nos testes a seguir, utilizamos as mesmas glicoproteínas *spike* dos vírus SARS-CoV e SARS-CoV-2 usadas em testes anteriores para comparar o tempo de execução entre o Needleman-Wunsch original, o BFNW padrão e o BFNW Paralelo. Além disso, para essa última versão do algoritmo, configuramos a quantidade de *threads* simultâneas

para variar de 1 até 16. Esses testes foram realizados em um *notebook* Acer Aspire E5-574, com um processador Intel Core i7-6500U 2.50GHz, 4096 de cache, 4 núcleos com 2 *threads* cada e 8 GB de RAM.

Para evitar vieses no sistema de execução (Python é conhecido por ser mais lento que C++, mesmo que esteja executando o mesmo algoritmo), traduzimos todos os algoritmos sendo comparados para C++. Por fim, para garantir a confiabilidade estatística dos resultados, executamos o teste em cada algoritmo 1000 vezes e comparamos a média final.

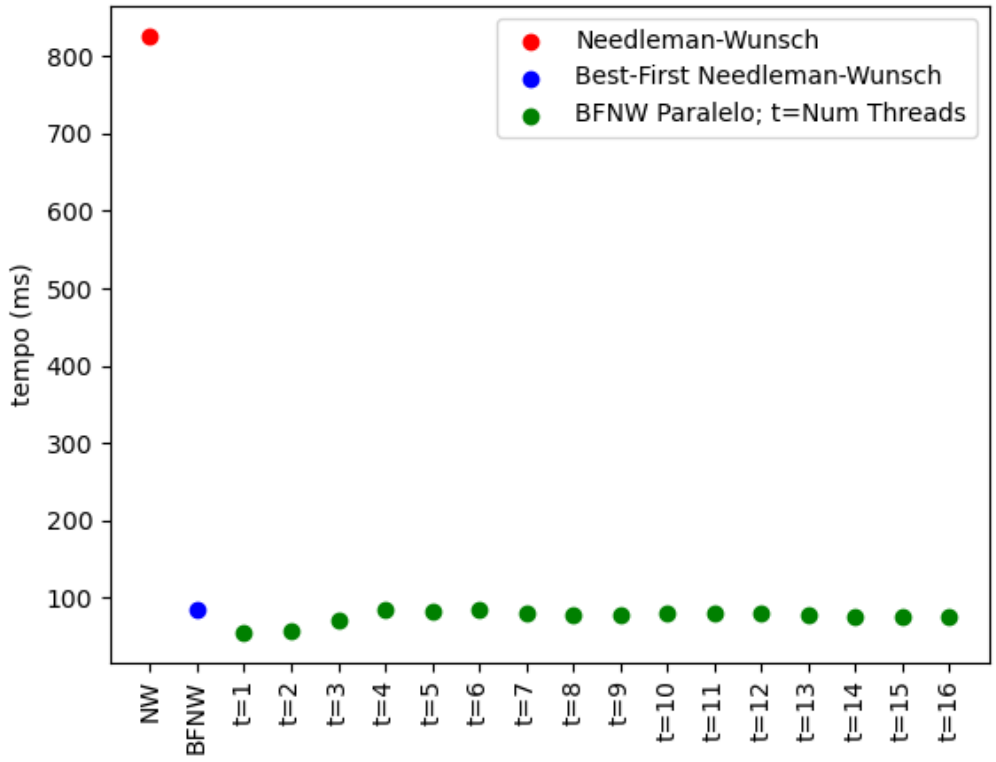


Figura 7: Resultados comparativos entre Needleman-Wunsch, BFNW padrão e BFNW paralelo para glicoproteínas reais

A partir dos resultados obtidos acima, concluímos que, como esperado, as versões sequencial e paralela do BFNW foram bem mais rápidas que o Needleman-Wunsch original, entretanto, a versão paralela não apresentou uma melhora significativa no tempo de execução quando comparada com a sua implementação padrão. Podemos argumentar que esse resultado foi obtido devido à baixa proporção do código que é paralelizável e também ao *overhead* ou custo de execução das funções relacionadas às *mutexes* e à variável de condição usadas na implementação.

5) CONCLUSÃO

Neste projeto, desenvolvemos e implementamos o algoritmo Best-First Needleman-Wunsch (BFNW), uma nova abordagem para o alinhamento de sequências. As análises comparativas realizadas, utilizando dados sintéticos e reais, indicam que o BFNW é significativamente mais rápido e eficiente no uso de memória em comparação com o algoritmo Needleman-Wunsch original. No entanto, a qualidade do alinhamento gerado pelo BFNW é consideravelmente inferior à solução ótima produzida pelo método original.

Além disso, diversas variações e versões do BFNW foram propostas e implementadas na tentativa de otimizar ainda mais o tempo de execução e a qualidade dos alinhamentos, contudo, os resultados obtidos permaneceram similares ao do BFNW padrão. Assim, concluímos que este projeto foi parcialmente bem-sucedido, pois, embora tenhamos conseguido descrever, interpretar e implementar a heurística proposta de maneira correta, sua performance real não atingiu as expectativas iniciais.

Apesar dessas limitações, este trabalho explorou apenas uma fração das possibilidades de melhorias e otimizações que podem ser aplicadas à técnica proposta. Portanto, acreditamos que futuras pesquisas podem expandir, refinar e potencialmente superar as limitações identificadas, contribuindo ainda mais para o desenvolvimento de métodos eficientes de alinhamento de sequências.

6) REFERÊNCIAS BIBLIOGRÁFICAS

1) MOUNT, D. W.; EBRARY, I. **Bioinformatics : sequence and genome analysis**. Editorial: Cold Spring Harbor, N.Y.: Cold Spring Harbor Laboratory Press, 2001.

2) SUNG, W.-K. **Algorithms in Bioinformatics**. [s.l.] CRC Press, 2009.

3) ORTET, P.; BASTIEN, O. Where Does the Alignment Score Distribution Shape Come from? **Evolutionary Bioinformatics**, v. 6, p. EBO.S5875, jan. 2010.

4) JONES, D. T. Protein secondary structure prediction based on position-specic scoring matrices. Journal of Molecular Biology, [S.l.], v.292, n.2, p.195 202, 1999.

5) BLAZEWICZ, J. et al. Whole genome assembly from 454 sequencing output via modified DNA graph concept. **Computational Biology and Chemistry**, v. 33, n. 3, p. 224–230, jun. 2009.

6) ABBOTT, A.; TSAY, A. Sequence Analysis and Optimal Matching Methods in Sociology. **Sociological Methods & Research**, v. 29, n. 1, p. 3–33, ago. 2000.

7) PRINZIE, A.; VAN DEN POEL, D. Incorporating sequential information into traditional classification models by using an element/position-sensitive SAM. **Decision Support Systems**, v. 42, n. 2, p. 508–526, nov. 2006.

8) PRINZIE, A.; VAN DEN POEL, D. Predicting home-appliance acquisition sequences: Markov/Markov for Discrimination and survival analysis for modeling sequential information in NPTB models. **Decision Support Systems**, v. 44, n. 1, p. 28–45, nov. 2007.

9) KATOH, K.; TOH, H. Recent developments in the MAFFT multiple sequence alignment program. **Briefings in Bioinformatics**, v. 9, n. 4, p. 286–298, 27 mar. 2008.

10) NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. **Journal of Molecular Biology**, v. 48, n. 3, p. 443–453, mar. 1970.

11) DOX, G. **Efficient algorithms for pairwise sequence alignment on graphs /**. Disponível em: <https://lib.ugent.be/catalog/rug01:002494667>. Acesso em: 02 de abril de 2024.

12) LIU, D.; STEINEGGER, M. Block aligner: fast and flexible pairwise sequence alignment with SIMD-accelerated adaptive blocks. **bioRxiv (Cold Spring Harbor Laboratory)**, 8 nov. 2021.

13) JÚNIOR, S.; DA, A. L. V. **Uma abordagem de alinhamento múltiplo de sequências utilizando evolução diferencial**. Disponível em: <https://repositorio.ufpe.br/handle/123456789/16194>. Acesso em: 02 de abril de 2024.

14) HIGGINS, D. G.; SHARP, P. M. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. **Gene**, v. 73, n. 1, p. 237–244, dez. 1988.

15) NOTREDAME, C.; HIGGINS, D. G.; HERINGA, J. T-coffee: a novel method for fast and accurate multiple sequence alignment 1 1Edited by J. Thornton. **Journal of Molecular Biology**, v. 302, n. 1, p. 205–217, set. 2000.

16) HENIKOFF, S.; HENIKOFF, J. G. Amino acid substitution matrices from protein blocks. **Proceedings of the National Academy of Sciences of the United States of America**, v. 89, n. 22, p. 10915–10919, 15 nov. 1992.

17) WALLS, A. C. et al. Structure, function, and antigenicity of the sars-cov-2 spike glycoprotein. **Cell**, v. 181, n. 2, p. 281–292, mar. 2020.

18) ZHU, C. et al. Molecular biology of the SARs-CoV-2 spike protein: A review of current knowledge. **Journal of Medical Virology**, v. 93, n. 10, p. 5729–5741, 14 jun. 2021.