Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

A-Star-RV e PA-Star-RV: Visualização da Execução das Ferramentas de Busca A-Star e PA-Star

Eduardo Freire Martins

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2024

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

A-Star-RV e PA-Star-RV: Visualização da Execução das Ferramentas de Busca A-Star e PA-Star

Eduardo Freire Martins

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Marcus Vinicius Lamar
CIC/UnB

Prof.ª Dr.ª Maristela Terto de Holanda
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 09 de setembro de 2024

Dedicatória

À minha família, pelo apoio e pelos sacrifícios incondicionais que recebo desde sempre.

Aos meus amigos de longa data e aos recém-chegados em minha vida, pelos bons momentos... e pelos não tão bons também!

Agradecimentos

Agradeço à minha família, por todo o apoio prestado ao longo deste trabalho.

Agradeço à minha orientadora, Prof.ª Dr.ª Alba de Melo, pela condução deste trabalho e pela paciência com minhas dúvidas e meus horários muitas vezes conflitantes.

Ao Prof. Dr. Daniel Sundfeld agradeço pelo apoio envolvendo o uso do PA-Star, bem como nas modificações de seus *logs* de saída.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

O A-Star é um algoritmo de busca em grafos pelo menor caminho que usa o critério *Best-First* para atingir seu objetivo. É utilizado em diversas aplicações, como definição de rotas otimizadas entre cidades e alinhamentos múltiplos de sequências biológicas, entre outros. O A-Star e suas variações apresentam um padrão irregular de expansão dos nós, o que torna difícil prever, em dado ponto de execução, quais vértices serão expandidos.

O presente trabalho de graduação propõe ferramentas de visualização da execução do A-Star e de sua variante paralela para o alinhamento múltiplo de sequências biológicas, o PA-Star, com o intuito de fornecer suporte à análise do padrão de expansão dos nós ao longo da execução e posterior integração a um mecanismo de previsão de visita dos nós, o que permitiria alocar os recursos computacionais, particularmente a memória, de forma mais eficiente, melhorando o desempenho.

Palavras-chave: Busca em grafos, A-Star, Parallel A-Star, Visualização

Abstract

A-Star is a shortest-path graph search algorithm that uses the Best-First criterion to accomplish its goal. It is used in many applications, such as finding optimal routes between cities and multiple biological sequences alignments. A-Star, as well as its variants, expand a graphs nodes in an irregular fashion, which makes it hard to predict which nodes will be expandaded at a given moment during runtime.

This undergraduate monography proposes visualization tools for both the A-Star algorithm as well as its parallel variant used in multiple biological sequences alignment, PA-Star, in order to support the analysis of the graph nodes expansion pattern during runtime and its future integration into a node expansion prediction mechanism, which would allow computational resources, especially memory, to be more efficiently allocated, improving performance.

Keywords: Graph search, A-Star, Parallel A-Star, Visualization

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O problema do alinhamento múltiplo de sequências biológicas é um problema importante da bioinformática que procura encontrar sequências de DNA, RNA ou de proteínas que sejam semelhantes, com vistas a descobrir possíveis ancestrais evolutivos em comum de seres vivos distintos. Trata-se de um problema computacional complexo, o que motiva a busca por formas eficientes de solucioná-lo.

Existem diversos algoritmos de busca em grafos, dos quais destacam-se os de busca de menor caminho, cujo objetivo é encontrar o caminho de menor custo entre dois nós de um mesmo grafo. Como exemplos, existem os algoritmos de busca em profundidade (DFS - *Depth First Search*), busca em largura (BFS - *Breadth First Search*), o algoritmo de Dijkstra e o algoritmo A-Star [9], objeto deste trabalho. Esses algoritmos têm diversas aplicações, dentre as quais destaca-se o alinhamento múltiplo de sequências biológicas. O A-Star, proposto por Hart *et al.* [9], é um algoritmo *Best-First* que usa uma função f própria para definir a ordem em que os nós serão expandidos.

A Bioinformática é o ramo da ciência da Computação que estuda a modelagem computacional de problemas ligados à biologia, incluindo o de comparação e alinhamento de sequências biológicas de DNA e de proteínas. O alinhamento múltiplo de sequências biológicas é um problema comprovadamente NP-Completo [10], o que significa que sua complexidade aumenta rapidamente com o tamanho e a quantidade de sequências analisadas. Consequentemente, o tempo necessário para encontrar um alinhamento pode ser muito alto, o que torna essencial a busca por algoritmos e ferramentas mais eficientes para resolver o problema.

Uma das ferramentas propostas para solucionar o problema do alinhamento múltiplo é o PA-Star [6], uma implementação paralelizada do algoritmo A-Star que pode gerar grafos de bilhões de nós [6], demandando grande quantidade de memória e, caso ela se esgote, armazenamento em disco. Se houvesse um meio de prever como a busca se dará, os recursos computacionais poderiam ser melhor aproveitados. Por exemplo, se for possível

estimar quais nós serão abertos, podem-se manter os vértices com maior chance de serem explorados na memória principal, minimizando o tempo gasto com operações de *swap* (operações em que são colocados em disco parte dos dados que ficariam na RAM e que comprometem o desempenho devido aos tempos de acesso maiores do disco).

Tendo em vista a dificuldade de determinar quais serão os próximos nós explorados tanto pelo A-Star quanto pelo PA-Star, o objetivo do presente trabalho de graduação é propor, implementar e avaliar duas ferramentas de visualização da execução dos algoritmos A-Star e PA-Star, guardando quais nós foram visitados a cada iteração, com vistas a subsidiar a criação futura de uma ferramenta capaz de estimar quais nós serão explorados em dado momento.

Desse modo, foram desenvolvidas duas ferramentas em linguagem Python: o A-Star-RV e o PA-Star-RV (A-Star *Runtime Visualizer* e PA-Star *Runtime Visualizer*, respectivamente). A primeira busca obter dados de execução do algoritmo A-Star ao passo que a segunda não só coleta dados de execução do PA-Star como também mostra uma visualização gráfica dos nós explorados e o melhor valor de f em cada iteração. Optou-se por desenvolver duas ferramentas distintas devido às características particulares de cada algoritmo, mas ambas têm em comum as funcionalidades de cálculo de métricas e de registro dos nós explorados.

Os resultados obtidos com a ferramenta A-Star-RV mostram que ela calcula adequadamente as métricas propostas, indicando que a ferramenta é adequada para coletar dados da execução do A-Star de forma concomitante. Quanto aos resultados obtidos com a ferramenta PA-Star-RV, a visualização dos dados da execução do PA-Star está de acordo com a redução do espaço de busca proposta por Carrillo e Lipman [11]. Além disso, mostrou-se que existe uma quantidade ótima de *threads* a serem executadas pelo PA-Star para minimizar o tempo de execução.

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta o algoritmo A-Star, o Capítulo 3 faz uma revisão teórica sobre o alinhamento de sequências biológicas, o Capítulo 4 apresenta o funcionamento da ferramenta PA-Star, o Capítulo 5 mostra as decisões tomadas para projetar as ferramentas e suas respectivas arquiteturas, o Capítulo 6 mostra os resultados obtidos e o Capítulo 7 apresenta as conclusões obtidas e propõe trabalhos futuros.

Capítulo 2

O Algoritmo A-Star

2.1 Introdução

Proposto por Hart *et al.* [9], o A-Star é um algoritmo usado para encontrar caminhos ótimos em grafos. Diferentemente do algoritmo de Dijkstra, que leva em consideração apenas os pesos das arestas dos grafos, o A-Star leva em consideração também uma estimativa da distância de cada vértice até o destino. Se essa estimativa for admissível, o algoritmo comprovadamente retornará um caminho ótimo [9].

2.2 Passos do algoritmo

O A-Star recebe como entrada um grafo, um vértice inicial e um conjunto de vértices finais possíveis e entrega o caminho ótimo do vértice inicial até um dos vértices finais, caso ele exista. Seja G um grafo composto pelos conjuntos V, de vértices, e E, de arestas. Seja $v_0 \in V$ o vértice inicial e seja V_f o subconjunto de V composto por todos os vértices aceitos como vértices finais. Por fim, suponha que cada aresta de G tem um custo representado por uma função $w : E \to \mathbb{R}$.

O Algoritmo A-Star (Algoritmo 1) segue os seguintes passos [9]:

1. marcar v_0 como 'aberto' (linha 1);
2. se não houver vértices abertos, não há caminho ótimo e a busca é encerrada (linha 39). Se houver, escolher, dentre eles, um vértice v segundo uma função de priorização $f : V \to \mathbb{R}$ (linhas 6 até 11);
3. se $v \in V_f$, o algoritmo é encerrado (linha 39). Caso contrário, v é marcado como 'fechado' e o valor de f para cada um de seus vizinhos é calculado. Se um vizinho não estiver fechado, ele passa a ser marcado como 'aberto'. Se estiver, ele mudará para

'aberto' apenas se o valor de f calculado agora for menor do que o valor calculado anteriormente. Por fim, se o vizinho já estava aberto mas o valor de f calculado para ele foi menor do que o anterior, ele continua aberto mas o valor de f é atualizado (linhas 17 até 34). Feito isso, retorne para a etapa 2.

Algoritmo 1 Implementação padrão do algoritmo A-Star

```
 1: open.add(v_0)
 2: v_0.predecessor ← NULL
 3: v_0.previous_f ← f(v_0)
 4: while open.length > 0 do
 5:     /* v armazena o nó com o menor valor de f */
 6:     v ← open[0]
 7:     for (i = 1; i < open.length; i++) do
 8:         if f(open[i]) < f(v)) then
 9:             v ← open[i]
10:         end if
11:     end for
12:     if v ∈ V_f then
13:         /* Se v é um dos nós finais, retorna o caminho encontrado. */
14:         return PATH TO v
15:     end if
16:     /* Caso contrário, varre os vizinhos de v */
17:     for all n ∈ v.neighbors() do
18:         /* Os vizinhos de v na lista open só serão modificados se o valor de f encontrado para eles
    nesta iteração for menor do que o valor encontrado anteriormente. */
19:         if n ∈ open and f(n) < n.previous_f then
20:             n.previous_f ← f(n)
21:             n.predecessor ← v
22:             /* Se o caminho encontrado até um nó na lista closed for menor do que o caminho anterior,
    o valor de f do nó e atualizado e ele é recolocado na lista open. */
23:         else if n ∈ closed and f(n) < n.previous_f then
24:             closed.remove(n)
25:             open.add(n)
26:             n.previous_f ← f(n)
27:             n.predecessor ← v
28:             /* Se o nó não estava em nenhuma lista, ele é colocado na lista open. */
29:         else
30:             open.add(n)
31:             n.previous_f ← f(n)
32:             n.predecessor ← v
33:         end if
34:     end for
35:     open.remove(v)
36:     closed.add(v)
37: end while
38: /* Se a lista open estiver vazia, não há caminho entre v_0 e algum nó de V_f. */
39: return NULL
```

2.3 A função de priorização

Estabelecido o algoritmo, falta escolher uma função f tal que o A-Star consiga encontrar um caminho ótimo. A função proposta por Hart *et al.* [9] é definida como:

$$f(v) = g(v) + h(v) \tag{2.1}$$

Onde $g(v)$ é o custo total do caminho menos custoso de v_0 até v encontrado até o momento e $h(v)$ é uma estimativa do custo de v até algum vértice pertencente a V_f. Por fim, quanto menor $f(v)$, maior a prioridade de v.

Assim, conclui-se que a função $f(v)$ proposta é uma estimativa do custo de v_0 até algum vértice final passando por v e que os vértices com menores custos estimados serão priorizados. Convém destacar que empates na escolha de v podem ser resolvidos arbitrariamente, mas devem priorizar vértices pertencentes a V_f.

Dado que encontrar $g(v)$ é trivial, uma vez que a função de custos w é conhecida, a escolha da função $f(v)$ se resume à escolha de uma função de estimativa $h(v)$ adequada. Hart *et al.* demonstraram que uma função $h(v)$ será admissível se, para qualquer $v \in V$, $h(v)$ for menor ou igual ao verdadeiro custo total do caminho ótimo de v até algum vértice de V_f[9]. Tal condição ficou conhecida na literatura como condição de admissibilidade.

Seja, por exemplo, o problema clássico de encontrar o menor caminho terrestre entre duas cidades, modelado por um grafo no qual os vértices correspondem às cidades e as arestas representam as rodovias. A função de custo de cada aresta é a extensão da rodovia representada por ela. Neste caso, uma função $h(v)$ admissível seria a distância euclidiana entre a cidade representada por v e a cidade de destino, pois essa é a menor distância possível entre as cidades.

2.4 Exemplos

2.4.1 Grafo previamente conhecido

O primeiro exemplo [1] consiste em encontrar um caminho ótimo do vértice A até o vértice K no grafo da Figura 2.1. Nela, os números acima das arestas são seus pesos e os números acima dos vértices são seus valores de h.

No início da análise, temos apenas o vértice A marcado como aberto e nenhum vértice fechado. Assim, o vértice inicial será expandido, ou seja, seus vizinhos serão analisados:

Passo 1: expansão de A (os valores de f de cada vértice estão entre parenteses)

- $f(B) = 18,4 + 46,3 = 64,7$
- $f(C) = 12,7 + 40,8 = 53,5$

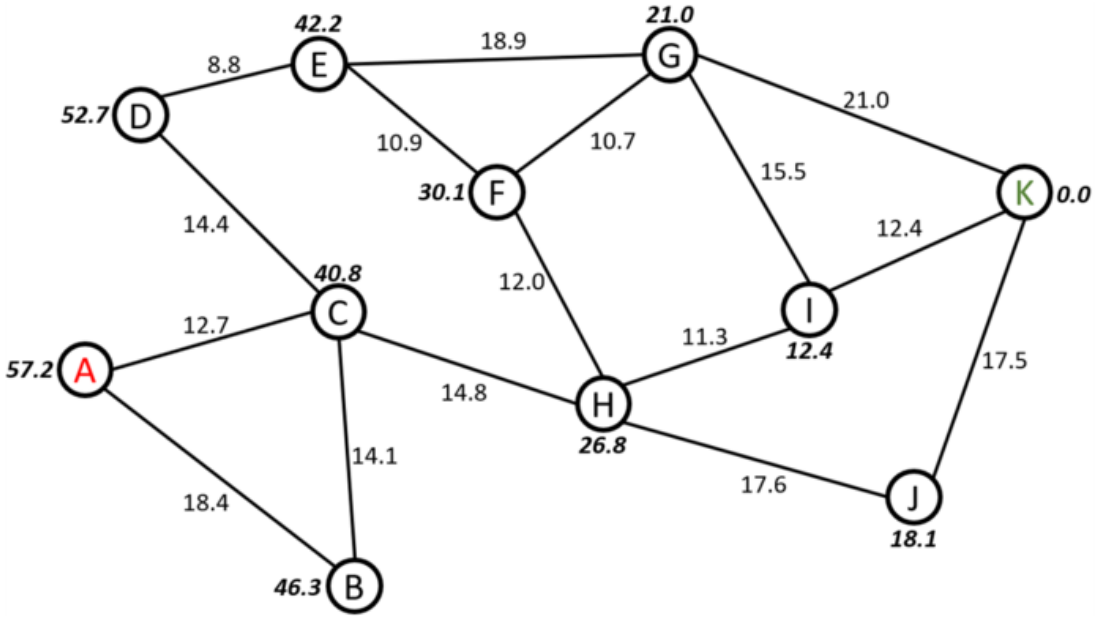


Figura 2.1: Primeiro exemplo. Fonte: `https://optimization.cbe.cornell.edu` [1].

- Vértices abertos: B (64,7), *C (53,5)*
- Vértices fechados: A (57,2)

Ao final deste passo, o próximo vértice a ser expandido será C, pois ele é o vértice aberto com menor valor de f.

Passo 2: expansão de C

- $f(A) = (12,7 + 12,7) + 57,2 = 82,6 \rightarrow$ Não sairá do conjunto fechado
- $f(B) = (12,7 + 14,1) + 46,3 = 73,1 \rightarrow f$ encontrado é maior do que o f achado antes
- $f(D) = (12,7 + 14,4) + 52,7 = 79,8$
- $f(H) = (12,7 + 14,8) + 26,8 = 54,3$
- Vértices abertos: B (64,7), D (79,8), *H (54,3)*
- Vértices fechados: A (57,2), C (53,5)

Repare que os vértices A e B são vizinhos de C e, por isso, seus valores de f foram calculados novamente, mas não houve mudanças nas marcações de A e B porque o f encontrado neste passo foi maior do que os valores encontrados anteriormente. Contudo, se, por exemplo, $f(B)$ tivesse apresentado valor menor do que 64,7, este novo valor de f seria registrado. Nos passos seguintes, por simplicidade, serão omitidos os cálculos dos vértices cujas marcações não serão alteradas. O próximo vértice a ser analisado é H.

Passo 3: expansão de H

- $f(F) = (12,7 + 14,8 + 12,0) + 30,1 = 69,6$
- $f(I) = (12,7 + 14,8 + 11,3) + 12,4 = 51,2$
- $f(J) = (12,7 + 14,8 + 17,6) = 63,2$
- Vértices abertos: B (64,7), D (79,8), F (69,6), *I (51,2)*, J (63,2)
- Vértices fechados: A (57,2), C (53,5), H (54,3)

Passo 4: expansão de I

- $f(G) = (12,7 + 14,8 + 11,3 + 15,5) + 21,0 = 75,3$
- $f(K) = (12,7 + 14,8 + 11,3 + 12,4) + 0,0 = 51,2$
- Vértices abertos: B (64,7), D (79,8), F (69,6), G (75,3), J (63,2), *K (51,2)*
- Vértices fechados: A (57,2), C (53,5), H (54,3), I (51,2)

O vértice a ser expandido no quinto passo é o vértice K, que é o vértice de destino. Com isso, o algoritmo é encerrado e o caminho ótimo encontrado foi $A \to C \to H \to I \to K$.

2.4.2 Grafo não conhecido inicialmente

No exemplo anterior, o grafo analisado já era inteiramente conhecido antes do início da execução do algoritmo, isto é, todos os seus vértices e arestas eram conhecidos. Contudo, também é possível que a análise seja feita em cima de um grafo não conhecido por completo ou então cuja expansão total seja evitada para poupar recursos computacionais.

O segundo exemplo [2], mostrado na Figura 2.2, é um quebra-cabeças composto por 8 peças numeradas e mais um espaço em branco dispostos numa matriz 3×3. Cada peça pode ser deslocada para o espaço em branco apenas se for adjacente a ele. O objetivo é deslocar as peças a partir da configuração inicial até chegar ao estágio final solicitado.

Cada configuração possível das peças corresponde a um vértice do grafo representativo do problema. Dois vértices serão vizinhos se e somente se for possível chegar de um até o outro deslocando apenas uma peça de lugar. Por fim, $g(v)$ é definido como o número de deslocamentos do estado inicial até v e $h(v)$ é definido como a quantidade de peças em posições incorretas em relação ao vértice final.

A Figura 2.3 [2] mostra a solução do segundo exemplo. Em vez de armazenar previamente o grafo inteiro, ele foi expandido de forma gradual a partir do nó inicial até que se encontrasse o vértice de destino. Tal abordagem exige que seja possível deduzir quais são todos os vizinhos de um vértice sem conhecimento prévio dos demais componentes do

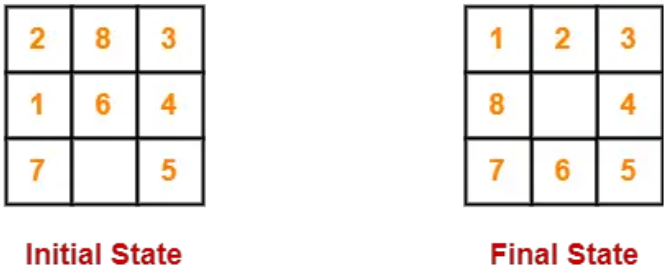


Figura 2.2: Segundo exemplo. Fonte: `https://www.gatevidyalay.com/` [2].

grafo, mas em compensação permite que sejam poupados recursos computacionais, visto que nem todos os nós do grafo serão analisados. Neste exemplo, o grafo tem $9! = 362880$ nós, mas apenas 14 foram analisados ($\approx 0,004\%$ dos nós).

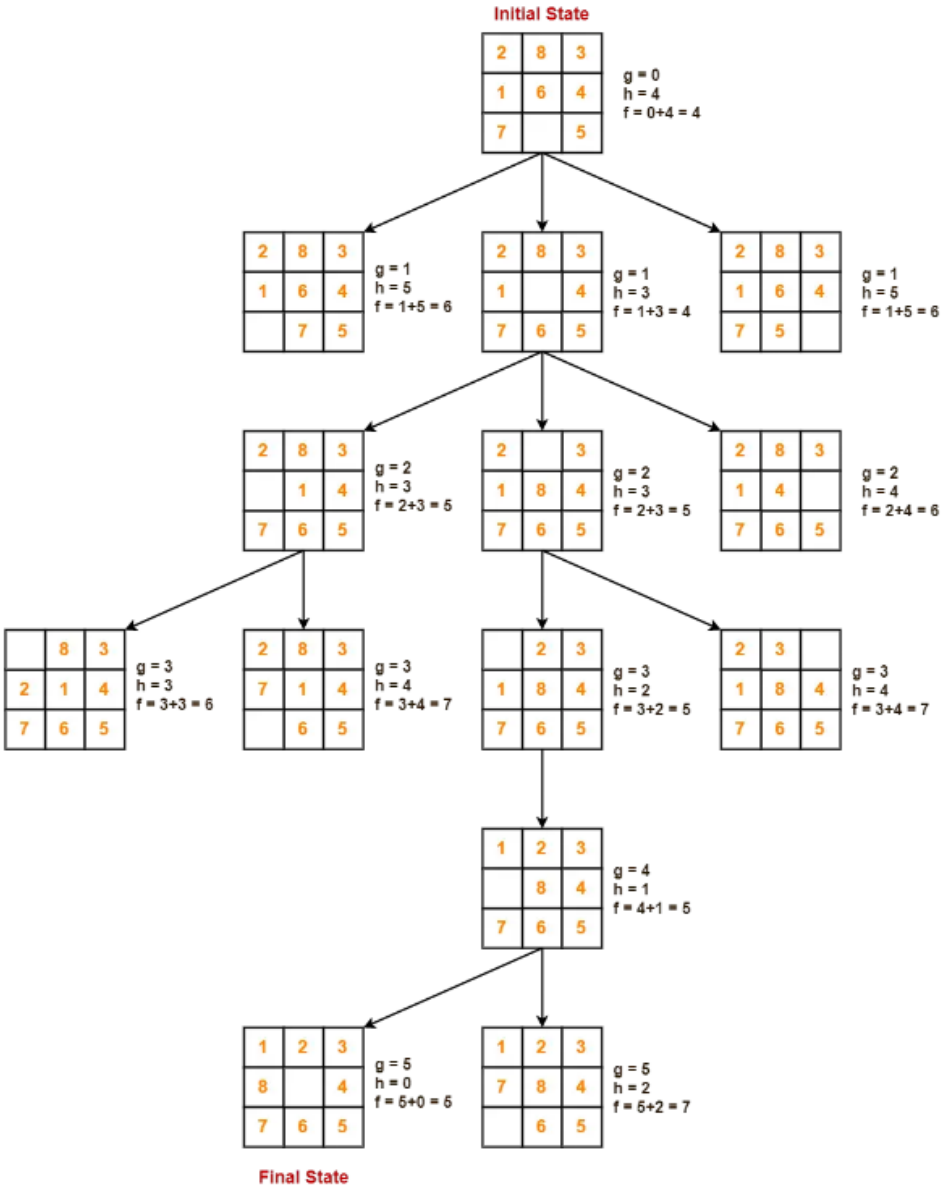


Figura 2.3: Solução do segundo exemplo Fonte: `https://www.gatevidyalay.com/` [2].

Capítulo 3

Alinhamento de Sequências Biológicas

3.1 Introdução

Sequências biológicas são sequências de DNA, RNA ou de proteínas encontradas nos seres vivos. Sequências semelhantes provavelmente exercem funções semelhantes e, se foram encontradas em seres vivos diferentes, significa que eles podem ter um ancestral em comum [3]. Assim, define-se o problema do alinhamento como a comparação entre duas ou mais sequências de modo a atribuir uma pontuação (escore) que mensura a semelhança entre elas. Trata-se de um problema que pode ser modelado computacionalmente mas que, dependendo dos tamanhos das sequências analisadas, pode ficar muito complexo.

Para definir se determinadas sequências são semelhantes ou não, é necessário usar um sistema de pontuação adequado. Esse sistema atribui valores para cada combinação de elementos possível da sequência e, com base nisso, atribui um valor final ao alinhamento. Convém destacar também que sequências biológicas podem perder ou ganhar elementos durante o processo evolutivo. Consequentemente, sequências derivadas de um ancestral em comum podem não ter mais o mesmo número de elementos, o que se manifesta na forma de lacunas que surgem durante o processo de alinhamento, isto é, alguns elementos das sequências em análise não terão um elemento correspondente em outra sequência. Tais lacunas também devem ser consideradas pelo sistema de pontuação.

3.2 Alinhamento de duas sequências

O alinhamento de sequências mais básico envolve duas sequências de DNA ou de proteínas [5]. Seja, por exemplo, o alinhamento da Figura 3.1, que mostra um alinhamento entre duas sequências hipotéticas de proteínas.

Figura 3.1: Exemplo de alinhamento de duas sequências. Fonte: Mount, D. [3].

A Figura 3.1 apresenta um alinhamento entre duas sequências de proteínas com pares de aminoácidos idênticos alinhados (*matches*) e pares com aminoácidos distintos alinhados (*mismatches*). Além disso, há também a introdução de lacunas (*gaps*) nas sequências, representadas pelo caractere '-'. Na Figura 3.1, os *matches* estão indicados também por linhas ligando os aminoácidos idênticos.

3.2.1 Escore

O escore (pontuação) do alinhamento entre duas sequências leva em consideração as operações de substituição, inserção e deleção. A operação de substituição consiste no pareamento entre dois caracteres das sequências e é feita geralmente por meio de matrizes de substituição, que atribuem uma pontuação para cada par de elementos possível, caso sejam pareados. Essa pontuação é definida com base na probabilidade de que cada elemento apareça. As operações de inserção (introdução de novo elemento) ou de deleção (retirada de um elemento), por sua vez, introduzem *gaps* em uma das sequências.

Deve-se destacar também que a pontuação é diferente para sequências de proteínas e de DNA, visto que cada elemento das primeiras é um aminoácido e que existem 20 aminoácidos encontrados na natureza (além de mais 6 aminoácidos sintéticos, até o momento), ao passo que as sequencias de DNA têm um alfabeto de apenas quatro nucleotídeos (G, A, T e C). A pontuação das sequências de proteínas é definida por meio de matrizes de substituição de tamanho 20×20, sendo as mais usadas a PAM e a BLOSUM [3]. A PAM leva em consideração a probabilidade de cada aminoácido aparecer na outra sequência, enquanto que a BLOSUM avalia blocos de aminoácidos mais comumente encontrados na natureza. A Figura 3.2 mostra a matriz PAM150.

A pontuação das sequências de DNA é mais simples, visto que seus alfabetos têm apenas 4 elementos. Um exemplo de matriz de substituição para sequências de DNA é a matriz NUC 4.0, mostrada na Figura 3.3. Observe que ela simplesmente atribui 5 pontos para pares com o mesmo nucleotídeo e desconta 4 pontos para as demais combinações. Assim, diferentemente das matrizes PAM e BLOSUM, a matriz NUC 4.0 não leva em consideração a probabilidade de ocorrência de cada nucleotídeo para definir o escore.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	3	-2	0	0	-2	-1	0	1	-2	-1	-2	-2	-1	-4	1	1	1	-6	-3	0	0	0	-1	-7
R	-2	6	-1	-2	-4	1	-2	-3	1	-2	-3	3	-1	-4	-1	-1	-2	1	-4	-3	-2	0	-1	-7
N	0	-1	3	2	-4	0	1	0	2	-2	-3	1	-2	-4	-1	1	0	-4	-2	-2	3	1	-1	-7
D	0	-2	2	4	-6	1	3	0	0	-3	-5	-1	-3	-6	-2	0	-1	-7	-4	-3	3	2	-1	-7
C	-2	-4	-4	-6	9	-6	-6	-4	-3	-2	-6	-6	-5	-5	-3	0	-3	-7	0	-2	-5	-6	-3	-7
Q	-1	1	0	1	-6	5	2	-2	3	-3	-2	0	-1	-5	0	-1	-1	-5	-4	-2	1	4	-1	-7
E	0	-2	1	3	-6	2	4	-1	0	-2	-4	-1	-2	-6	-1	-1	-1	-7	-4	-2	2	4	-1	-7
G	1	-3	0	0	-4	-2	-1	4	-3	-3	-4	-2	-3	-5	-1	1	-1	-7	-5	-2	0	-1	-1	-7
H	-2	1	2	0	-3	3	0	-3	6	-3	-2	-1	-3	-2	-1	-1	-2	-3	0	-3	1	1	-1	-7
I	-1	-2	-2	-3	-2	-3	-2	-3	-3	5	1	-2	2	0	-3	-2	0	-5	-2	3	-2	-2	-1	-7
L	-2	-3	-3	-5	-6	-2	-4	-4	-2	1	5	-3	3	1	-3	-3	-2	-2	-2	1	-4	-3	-2	-7
K	-2	3	1	-1	-6	0	-1	-2	-1	-2	-3	4	0	-6	-2	-1	0	-4	-4	-3	0	0	-1	-7
M	-1	-1	-2	-3	-5	-1	-2	-3	-3	2	3	0	7	-1	-3	-2	-1	-5	-3	1	-3	-2	-1	-7
F	-4	-4	-4	-6	-5	-5	-6	-5	-2	0	1	-6	-1	7	-5	-3	-3	-1	5	-2	-5	-5	-3	-7
P	1	-1	-1	-2	-3	0	-1	-1	-1	-3	-3	-2	-3	-5	6	1	0	-6	-5	-2	-2	-1	-1	-7
S	1	-1	1	0	0	-1	-1	1	-1	-2	-3	-1	-2	-3	1	2	1	-2	-3	-1	0	-1	0	-7
T	1	-2	0	-1	-3	-1	-1	-1	-2	0	-2	0	-1	-3	0	1	4	-5	-3	0	0	-1	-1	-7
W	-6	1	-4	-7	-7	-5	-7	-7	-3	-5	-2	-4	-5	-1	-6	-2	-5	12	-1	-6	-5	-6	-4	-7
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-2	-2	-4	-3	5	-5	-3	-3	-1	8	-3	-3	-4	-3	-7
V	0	-3	-2	-3	-2	-2	-2	-2	-3	3	1	-3	1	-2	-2	-1	0	-6	-3	4	-2	-2	-1	-7
B	0	-2	3	3	-5	1	2	0	1	-2	-4	0	-3	-5	-2	0	0	-5	-3	-2	3	2	-1	-7
Z	0	0	1	2	-6	4	4	-1	1	-2	-3	0	-2	-5	-1	-1	-1	-6	-4	-2	2	4	-1	-7
X	-1	-1	-1	-1	-3	-1	-1	-1	-1	-1	-2	-1	-1	-3	-1	0	-1	-4	-3	-1	-1	-1	-1	-7
*	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	1

Figura 3.2: Matriz de substituição PAM 150. Fonte: `https://github.com/biogo/` [4]

	A	T	G	C
A	5	-4	-4	-4
T	-4	5	-4	-4
G	-4	-4	5	-4
C	-4	-4	-4	5

Figura 3.3: Matriz de substituição NUC 4.0, usada em alinhamentos de sequências de DNA Fonte: `https://github.com/biogo/` [4]

Figura 3.4: Alinhamentos global e local de um mesmo par de sequências hipotéticas de proteínas. Fonte: Mount, D. [3].

Uma vez escolhida a matriz de substituição, está definida a pontuação que será atribuída para cada par de elementos alinhados (um elemento pode ser tanto um aminoácido ou nucleotídeo) e também a pontuação negativa que expressa a penalidade por introduzir um *gap* em uma das sequências. Para encontrar o escore do alinhamento, calcula-se a soma das pontuações de cada par de elementos alinhados. Um alinhamento entre sequências é dito ótimo quando seu escore é o maior possível para as duas sequências em análise.

3.2.2 Encontrando o alinhamento ótimo

Alinhamentos locais e globais

Existem dois tipos básicos de alinhamentos de sequências: globais e locais. O tipo de alinhamento desejado define a forma como o escore será calculado.

Alinhamentos globais são aqueles cuja pontuação leva em consideração todos os elementos das sequências. Alinhamentos locais, por sua vez, avaliam apenas as subsequências mais semelhantes, sendo desprezadas as partes com mais diferenças. A Figura 3.4 mostra um exemplo de alinhamento global e um de alinhamento local de um mesmo par de sequências de proteínas [3].

Existem dois algoritmos básicos para encontrar o alinhamento ótimo entre duas sequências: o algoritmo Needleman-Wunsch, usado para alinhamentos globais, e o Smith-Waterman, usado para alinhamentos locais [5].

Algoritmo Needleman-Wunsch

Este algoritmo usa programação dinâmica para encontrar o alinhamento ótimo entre duas sequências. Sejam S_1 e S_2 as sequências que serão alinhadas e n_1 e n_2 os comprimentos de S_1 e S_2, respectivamente. Assim, monta-se uma matriz F de tamanho $(n_1 + 1) \times (n_2 + 1)$

tal que o elemento $F(i, j)$ representa o escore do melhor alinhamento encontrado entre os segmentos $S_1[1..i]$ e $S_2[1..j]$ [12].

A construção da matriz F é feita recursivamente. Sejam d a penalidade imposta por incluir um *gap* e $s(a, b)$ o escore do pareamento dos nucleotídeos/aminoácidos a e b de acordo com a matriz de substituição escolhida. Inicialmente, define-se $F(0, 0) = 0$. Em seguida, os elementos da primeira linha e da primeira coluna são computados subtraindo d do valor do elemento imediatamente anterior, conforme as regras $F(i, 0) = -id, \quad 1 \leq i \leq n_1$ e $F(0, j) = -jd, \quad 1 \leq j \leq n_2$ [5]. Os demais elementos são obtidos a partir da equação de recorrência 3.1:

$$F(i,j) = max \begin{pmatrix} F(i-1, j-1) + s(S_1[i], S_2[j]) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{pmatrix}, \quad \begin{matrix} 1 \leq i \leq n_1 \\ 1 \leq j \leq n_2 \end{matrix} \tag{3.1}$$

A Equação 3.1 define que deve-se escolher a maior pontuação possível dentre três equações que representam, respectivamente, os seguintes casos [5]:

- Pareamento dos elementos $S_1[i]$ e $S_2[j]$,
- colocação de um *gap* em S_2,
- colocação de um *gap* em S_1.

Finalizado este procedimento, a célula $F(n_1, n_2)$ conterá o valor do escore do alinhamento ótimo de acordo com a matriz de substituição utilizada. Para encontrar o alinhamento em si, parte-se para a segunda parte do algoritmo, o *traceback* [5].

Conforme o valor de cada célula $F(i, j)$ é calculado, registra-se também qual das células anteriores ($F(i-1, j-1)$, $F(i, j-1)$ ou $F(i-1, j)$) foi usada no cálculo de seu valor, conforme a Equação 3.1. Essa informação permite que o alinhamento ótimo seja montado de trás para frente. Partindo da célula $F(n_1, n_2)$, verifica-se qual foi a célula anterior que contribuiu para o cálculo. Se foi a célula $F(n_1 - 1, n_2 - 1)$, coloca-se no alinhamento o par $S_1[n1], S_2[n_2]$. Se foi a célula $F(n_1 - 1, n_2)$, coloca-se o elemento $S_1[n_1]$ pareado com um *gap* e, se foi a célula $F(n_1, n_2 - 1)$, coloca-se o elemento $S_2[n_2]$ pareado com um *gap*. O procedimento é repetido recursivamente para as células anteriores até que se chegue à célula $F(0, 0)$ [5]. A Figura 3.5 mostra um exemplo de alinhamento encontrado segundo o algoritmo Needleman-Wunsch.

Algoritmo Smith-Waterman

O algoritmo Smith-Waterman, usado para encontrar alinhamentos locais, é similar ao Needleman-Wunsch. A diferença se dá no acréscimo de mais uma possibilidade à Equa-

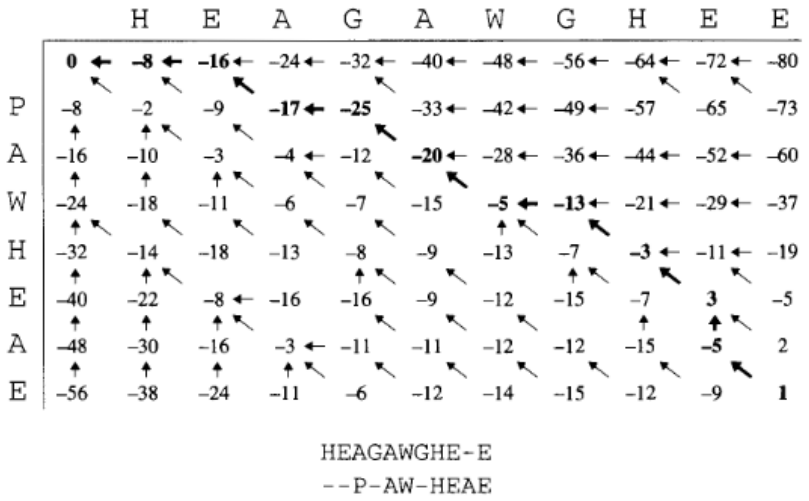


Figura 3.5: Exemplo de matriz de alinhamento feita seguindo o algoritmo Needleman-Wunsch. As setas mostram a etapa do *traceback*. Fonte: Durbin, R. [5]

ção 3.1, de modo a impedir que sejam armazenados valores negativos na matriz F. Com isso, a equação de recorrência usada é a 3.2. Outra consequência dessa mudança é que todos os elementos da primeira linha e da primeira coluna de F serão iguais a 0 [13].

$$F(i,j) = max \begin{pmatrix} 0 \\ F(i-1,j-1) + s(S_1[i], S_2[j]) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{pmatrix}, \quad \begin{matrix} 1 \leq i \leq n_1 \\ 1 \leq j \leq n_2 \end{matrix} \tag{3.2}$$

A escolha pelo valor zero equivale a iniciar um novo alinhamento de segmentos de S_1 e S_2. Intuitivamente, diz-se que é melhor iniciar um novo alinhamento do que estender um que tenha um escore negativo [5].

Como os pontos inicial e final de um alinhamento não são mais fixos, a etapa de *traceback* também sofre alterações. Agora, o *traceback* começará na célula de maior valor de F e continuará até encontrar uma célula de valor igual a 0, que representa o início de um alinhamento. A Figura 3.6 mostra uma matriz de alinhamento construída segundo o algoritmo Smith-Waterman e usada para alinhar as mesmas sequências da Figura 3.5 [5].

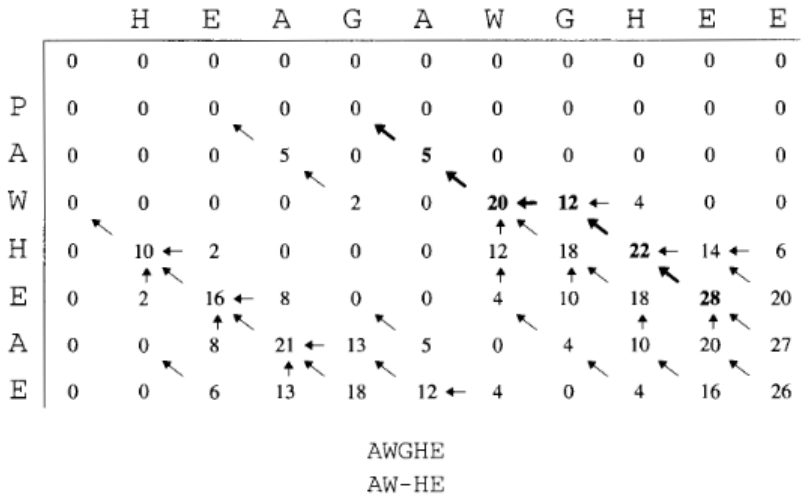


Figura 3.6: Matriz de alinhamento construída conforme Smith-Waterman. Fonte: Durbin, R. [5]

3.3 Alinhamento múltiplo de sequências

3.3.1 Alinhamento ótimo

O alinhamento múltiplo é aquele que considera mais de duas sequências biológicas. O escore de um alinhamento múltiplo é definido como a soma dos escores dos alinhamentos par-a-par de cada sequência (*sum-of-pairs*), obtidos com os algoritmos de comparação de duas sequências (Seção 3.2).

Podem-se estender os algoritmos Needleman-Wunsch e Smith-Waterman para o caso de múltiplas sequências. Para tanto, as equações de recorrência 3.1 e 3.2 são atualizadas para incluírem mais sequências. Seja um alinhamento de N sequências, seja $\alpha_{x_1,x_2,...,x_N}$ uma das células da matriz de alinhamento e seja $s(x_1, x_2, ..., x_N)$ o escore correspondente ao alinhamento dos itens $x_1, x_2, ..., x_n$ de cada sequência conforme a matriz de substituição escolhida, temos que [5]:

$$\alpha_{x_1,x_2,...,x_N} = max \begin{pmatrix} \alpha_{x_1-1,x_2-1,...,x_N-1} + s(x_1, x_2, ..., x_N) \\ \alpha_{x_1,x_2-1,...,x_N-1} + s(-, x_2, ..., x_N) \\ \vdots \\ \alpha_{x_1,x_2,...,x_N-1} + s(-, -, ..., x_N) \\ \vdots \end{pmatrix} \tag{3.3}$$

A Equação 3.3 avalia o maior escore entre os casos envolvendo um alinhamento de todas as sequências e também todas as possíveis implementações de *gaps*. Trata-se de

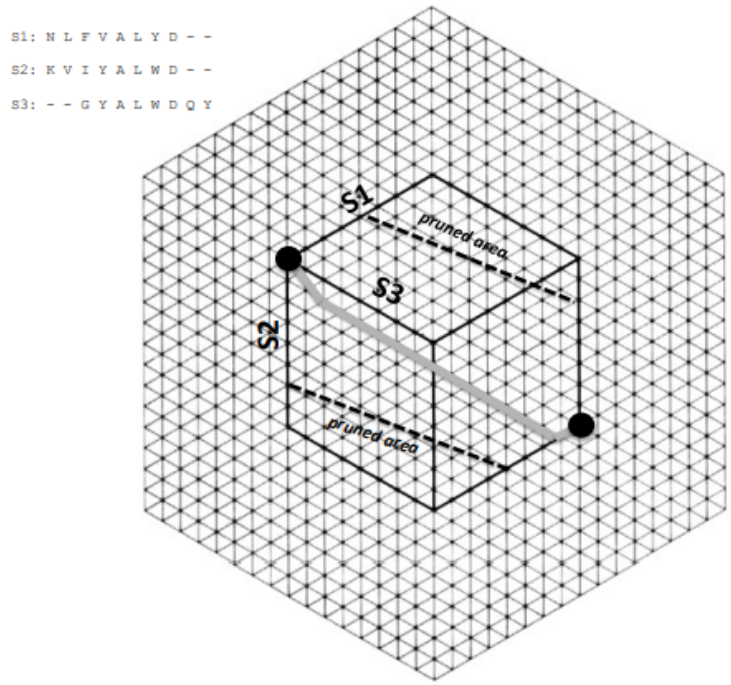


Figura 3.7: Diagrama do espaço de busca de um alinhamento de 3 sequências. Fonte: Sundfeld, D. *et al.* [6].

um problema que foi comprovado NP-Completo [10], o que significa que um pequeno aumento no tamanho ou na quantidade de sequências pode elevar significativamente a complexidade computacional do problema. Assim, são usados métodos para diminuir a complexidade do problema, dos quais destacaremos o método de Carrillo-Lipman [11] e o uso do algoritmo A-Star [9].

3.3.2 Carrillo-Lipman

Embora a matriz de alinhamento varrida pelo método da Seção 3.3.1 seja muito grande, Carrillo e Lipman demonstraram que não é necessário varrer a matriz por completo [11] para encontrar um alinhamento ótimo. Em vez disso, é possível encontrar limites superiores e inferiores para os valores das células da matriz de alinhamento. Caso uma célula tenha um valor fora desses limites, ela não faz parte do alinhamento ótimo e pode ser ignorada.

A Figura 3.7 mostra o espaço de busca de um alinhamento de 3 sequências [6]. Com a definição de limites superiores e inferiores, é possível retirar parte do espaço de busca (as "*pruned areas*") da análise, reduzindo a complexidade do problema.

3.3.3 A-Star

Conforme explicado no Capítulo 2, o algoritmo A-star [9] é usado para encontrar caminhos ótimos em grafos. O problema do alinhamento múltiplo de sequências, por sua vez, pode ser modelado como um grafo de modo que encontrar o alinhamento ótimo passa a ser equivalente a encontrar o caminho ótimo entre o nó de coordenada $(0, 0, ..., 0)$ e o de coordenada $(n_1, n_2, ..., n_N)$ [6].

Com isso, existe também a redução do espaço de busca, no caso médio. Além disso, foi mostrado que, se uma heurística admissível é utilizada quando o algoritmo A-Star é aplicado ao problema do Alinhamento Múltiplo, a redução do espaço de busca é, em média, equivalente à redução conseguida pelo algoritmo Carrillo-Lipman (Seção 3.3.2) [14].

Capítulo 4

Ferramenta PA-Star

4.1 Introdução

O algoritmo PA-Star é uma variação do A-Star (Capítulo 2) usada para resolver o problema de alinhamento de sequências biológicas e que faz uso de paralelismo para completar uma varredura mais rapidamente [6]. Para tanto, a lista de vértices abertos é dividida entre vários *threads* que rodam uma versão alterada do A-Star. Para minimizar o *overhead* oriundo da comunicação entre as *threads*, os vértices são distribuídos de acordo com uma função *hash* sensível à localidade.

4.2 Projeto básico

O PA-Star recebe como entrada um conjunto de sequências biológicas e entrega o alinhamento ótimo entre elas. Ele gera t *threads* que farão a busca A-Star em paralelo no grafo a ser analisado, sendo que cada um deles contém suas próprias listas de vértices abertos e fechados. A Figura 4.1 mostra um esquema básico do funcionamento do algoritmo.

Primeiramente, o algoritmo usará a função de *hash* para determinar qual *thread* receberá o nó inicial em sua respectiva lista de vértices abertos. Em seguida, o *thread* fará a varredura de forma semelhante ao algoritmo A-Star mas, ao encontrar os vizinhos do nó inicial, eles serão distribuídos para as listas abertas dos demais *threads* de acordo com a função de *hash* [6]. Esta etapa mostra a vantagem de usar uma função sensível à localidade, uma vez que haverá uma probabilidade maior de um nó explorado ser alocado para a lista de vértices abertos do mesmo *thread* que o abriu, minimizando o *overhead* decorrente da escrita nas listas de *threads* distintos.

As listas abertas são preenchidas paralelamente por cada *thread* de acordo com o método acima. Contudo, os nós finais são manipulados de forma um pouco diferente [6], visto que, ao se encontrar um nó final, é necessário verificar se não há nenhum outro nó

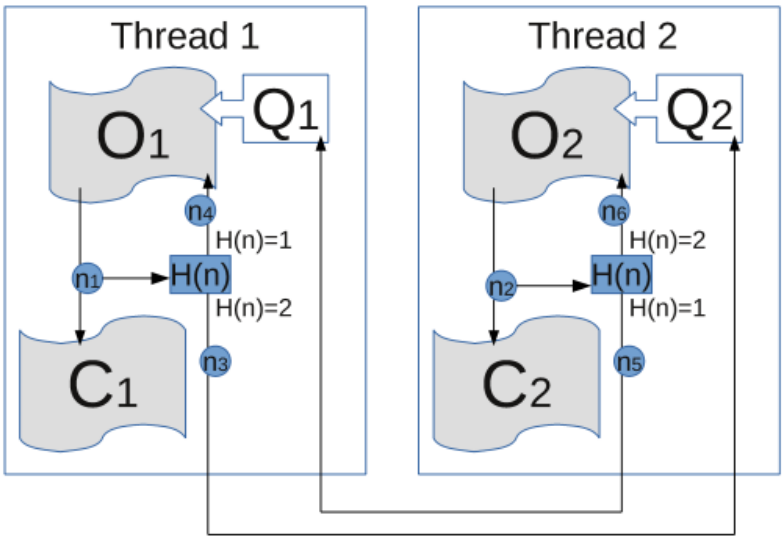


Figura 4.1: Estratégia básica do algoritmo PA-Star usando 2 *threads*. O representa a lista aberta, Q representa a fila de nós e C representa a lista fechada. $H(n)$ é a função de *hash*. Fonte: Sundfeld, D. *et al.* [6].

com valor de f inferior a ele não só na lista do *thread* que o encontrou, mas também nas listas dos demais *threads*. Por isso, quando um *thread* encontra um nó final, ele é colocado nas listas de todas os *threads*.

O *thread* que receberá determinado vértice é definido de acordo com a Equação 4.1 [6]:

$$(H(C_1, C_2, ..., C_n) \gg S) \bmod t = t_i \tag{4.1}$$

Onde $(C_1, C_2, ..., C_n)$ são as coordenadas do nó em análise, H é a função de *hash* sensível à localidade, S é uma constante que determina a quantidade de bits que serão deslocados, t é a quantidade de *threads* que fazem a busca e t_i é o *thread* que receberá o nó em análise em sua lista.

Além de usar uma função H sensível à localidade, também é usada uma operação de deslocamento de bits para a direita, para descartar os bits menos significativos do *hash*. A implementação realizada usa duas funções de *hash*: a função *SUM*, que simplesmente soma as coordenadas do vértice, e a função *ZORDER* [15], composta por operações de deslocamentos (*shifts*) para intercalar os *bits* que representam as coordenadas. Ela é executada rapidamente e preserva a localidade [6].

4.3 Detalhamento do algoritmo

O funcionamento do PA-Star é apresentado no Algoritmo 2. Primeiramente, um *thread* inicial calculará o *hash* do nó inicial e o acrescentará na lista do *thread* de busca correspondente (linha 1). Convém destacar que o *thread* inicial não realiza buscas e, portanto, não faz partes dos t *threads* presentes na Equação 4.1. Após a inicialização, cada *thread* executa uma busca de acordo com o algoritmo A-Star (*search_step*, linha 7) e com base em suas respectivas listas e retorna o nó final. Se necessário, durante a expansão um *thread* pode escrever nas listas de outro [6].

Algoritmo 2 Implementação do PA-Star (adaptado de [6])

1: $h \leftarrow hash(v_0)$
2: $OpenList_h \leftarrow v_0$
3: $end\_condition \leftarrow false$
4: $n \leftarrow \infty$ /* Inicializa a variável que guarda o nó final */
5: **while** $end\_condition is false$ **do**
6: /* Execuções em paralelo: uma para cada thread */
7: $n \leftarrow search\_step(i, V_f)$ /* O *thread* i executa a busca*/
8: $end\_condition \leftarrow verify\_end\_condition(i)$ /* Verifica a condição de parada */
9: **end while**

Se um dos nós finais for expandido por um *thread*, deve-se verificar se esse nó tem valor de f inferior a todos os nós nas listas de nós abertos de todas os *threads*. Tal verificação é feita em duas etapas: uma assíncrona e outra síncrona [6]. Na etapa assíncrona, o contador c_t, que registra a quantidade de *threads* que encontraram um nó final, é incrementado e, em seguida, é verificado se o valor de f do nó final é menor do que todos os outros nas listas do *thread* que o encontrou. Se sim, o nó encontrado é colocado nas listas de nós abertos de todos os demais *threads*.

Quando c_t for igual à quantidade de *threads* de busca, a função *search_step* é encerrada e todos os *threads* vão para a etapa seguinte (*verify_end_condition*, linha 8), na qual as listas de vértices abertos são consumidas para verificar se há algum nó com valor de f inferior ao do nó final encontrado. Se houver, *verify_end_condition* retornará $falso$, fazendo com que o laço da linha 5 seja retomado, reiniciando as buscas. Além disso, o nó final será recolocado na lista do *thread* que o expandiu. Se não houver f inferior, *verify_end_condition* retornará *verdadeiro*, indicando que a solução ótima foi encontrada[6], encerrando o laço e retornando o nó final encontrado e o caminho até ele.

Existem também dois casos especiais que devem ser tratados [6]. Primeiro, é possível que a lista de nós abertos de determinado *thread* fique vazia por algum tempo. Nesse caso, o *thread* deve aguardar até que novos vértices sejam colocados nele por outra *thread*. Além disso, é possível que, após um dos nós finais tenha sido encontrado, um *thread* encontre o mesmo nó ou então um nó final diferente com valor inferior de f. Se isso ocorrer, o

processo de verificação do nó final deve ser reiniciado, isto é, c_t recebe o valor 1 e o novo nó é colocado nas listas dos demais *threads*.

4.4 Estrutura de dados e funções

Esta seção detalha como os nós são armazenados em memória e como foram definidas as funções e estruturas de dados usadas na implementação.

Os nós da lista de vértices abertos são ordenados por dois valores diferentes: o valor f de cada nó e também pelo valor da coordenada que ele representa [6]. Na implementação do PA-Star usada neste trabalho, foi usado o template *multi-index* da biblioteca Boost C++ (http://www.boost.org), que permite a criação de estruturas com mais de um índice [6] (no caso deste trabalho, o valor de f e a coordenada).

As funções e estruturas usadas na implementação apresentam desempenho fortemente dependente da quantidade de sequências em análise, de modo que é comum encontrar estruturas de dados específicas para certas quantidades de sequências [6]. A implementação do PA-Star usada neste trabalho [16] usou *templates* de C++ para implementar as estruturas de dados e funções de acordo com a quantidade de sequências em análise, de tal forma que a definição da estrutura é escolhida durante a compilação.

4.5 Uso de disco

Trabalhos com sequências biológicas frequentemente exigem quantidade de memória maior do que a RAM disponível. Assim, a implementação do PA-Star tem uma solução para armazenar nós também em disco. Tal solução foi chamada de *Disk-Assisted PA-Star*, ou DAPA [6].

Resultados empíricos mostram que a lista de nós fechados costuma ser maior do que a de nós abertos. Por isso, optou-se por dividir a lista de nós fechados em regiões [6], que podem estar ativas ou inativas. As ativas contém nós que estão sendo processados no momento pelo PA-Star ao passo que as inativas guardam nós que não estão. Quando a quantidade de nós fechados ultrapassa um limite definido pelo usuário, os nós das regiões inativas passam a ser armazenados em disco.

Quando se usa o DAPA, cada *thread* pode conter mais de uma região, e cada região terá suas próprias listas. Assim, a Equação 4.1 deve ser adaptada para incluir as r regiões geridas por cada *thread*:

$$(H(C_1, C_2, ..., C_n) \gg S) \bmod (t \times r) = t_i \tag{4.2}$$

O DAPA é usado em aplicações cujo uso de RAM é elevado. Portanto, pode-se concluir que as listas de nós abertos e fechados serão grandes [6]. Por isso, foi incluída uma terceira lista, chamada de *holding list*, que guarda os nós que serão inseridos em alguma lista de nós abertos. Quando um nó é gerado e será inserido na lista aberto, verifica-se se o nó é ativo. Se for, ele será colocado na lista correspondente. Caso contrário, ele vai para a *holding list* correspondente. Se houver duplicidade, apenas o nó com maior prioridade, isto é, o nó com o menor valor de f, é mantido [6].

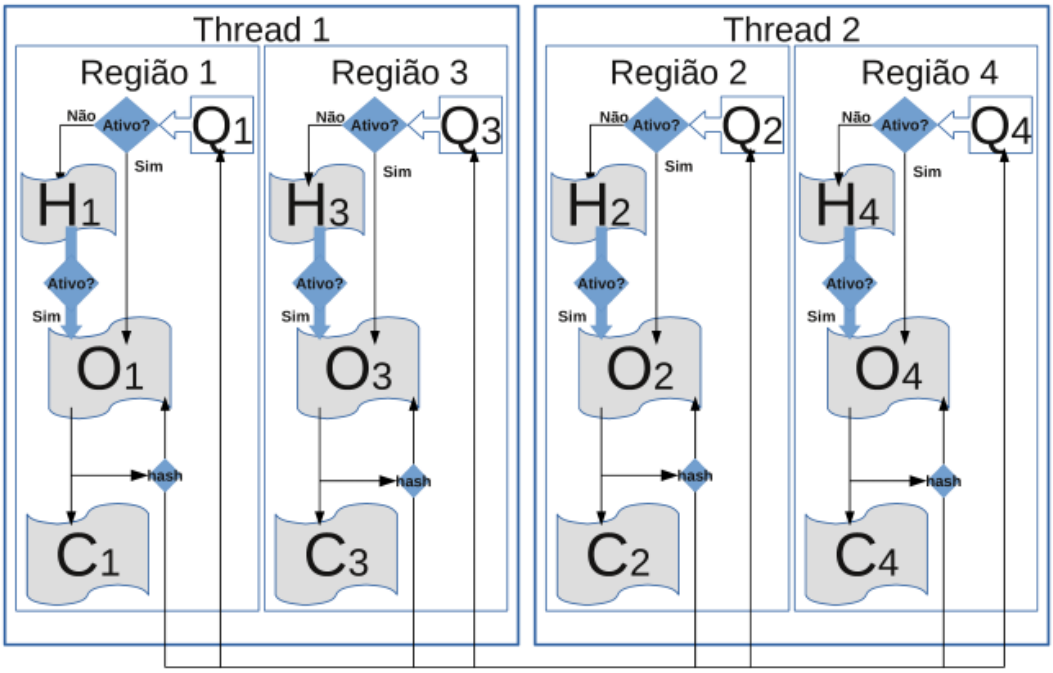


Figura 4.2: DAPA funcionando com dois *threads* divididos em duas regiões. Fonte: Sundfeld, D. *et al.* [6].

Feito isso, os *threads* verificam em qual região está o nó com maior prioridade e classificam essa região como ativa. Nessa etapa, o *thread* resolve eventuais duplicidades nas listas fechadas e na *holding list* da região ativada e, em seguida, os nós na *holding list* da região ativada são colocados na respectiva lista aberta. Se a lista fechada da região ativada estiver no disco, ela volta para a RAM. Por fim, se a quantidade limite na RAM for atingida, a lista fechada da região que contém o nó com a pior prioridade é colocada em disco. Esse processo se repete até que o limite seja respeitado novamente. Feito esse procedimento, as etapas *search_step* e/ou *verify_end_condition* do algoritmo 2 são executadas como explicado na Seção 4.3.

A Figura 4.2 mostra um esquema em que o espaço de busca foi dividido em quatro regiões e cada *thread* opera em duas regiões. Na figura, cada *thread* tem suas próprias

filas (Q) e suas próprias listas *open* (O), *closed* (C) e *holding* (H).

Usar uma estratégia para armazenamento em disco, como o DAPA, se mostra necessário sempre que a memória RAM for insuficiente para armazenar todos os nós do grafo que representa o alinhamento das sequências, o que é uma situação comum. No entanto, essa abordagem leva a uma queda de desempenho por conta da velocidade menor de acesso à memória permanente, no caso o disco. O DAPA usa como critério para determinar os nós a serem guardados em disco o valor de f deles (nós com valores maiores de f são armazenados primeiro). Este projeto de graduação objetiva a visualização da expansão dos nós, de maneira que seja possível determinar quais nós têm mais chance de serem abertos e usar essa análise como critério para definir quais nós serão armazenados em disco.

Capítulo 5

Projeto das Ferramentas A-Star-RV e PA-Star-RV

5.1 Introdução

Este capítulo apresenta o projeto das ferramentas de visualização A-Star-RV (*A-Star Runtime Visualizer*, Figura 5.1) e PA-Star-RV (*PA-Star Runtime Visualizer*, Figura 5.2).

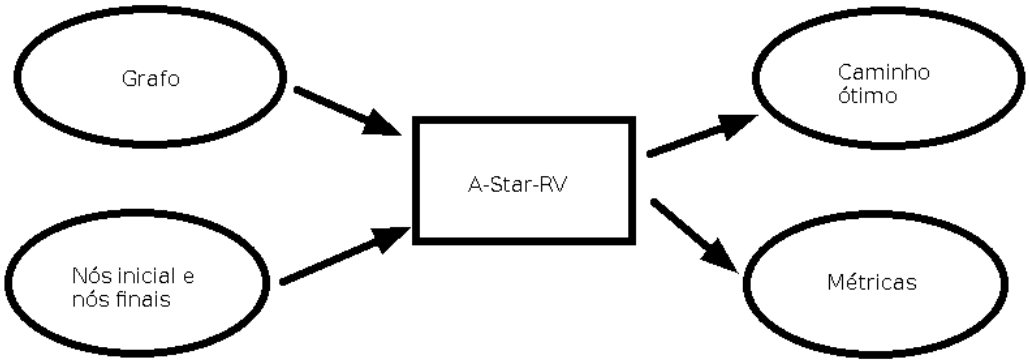


Figura 5.1: Diagrama da ferramenta A-Star-RV. Fonte: próprio autor

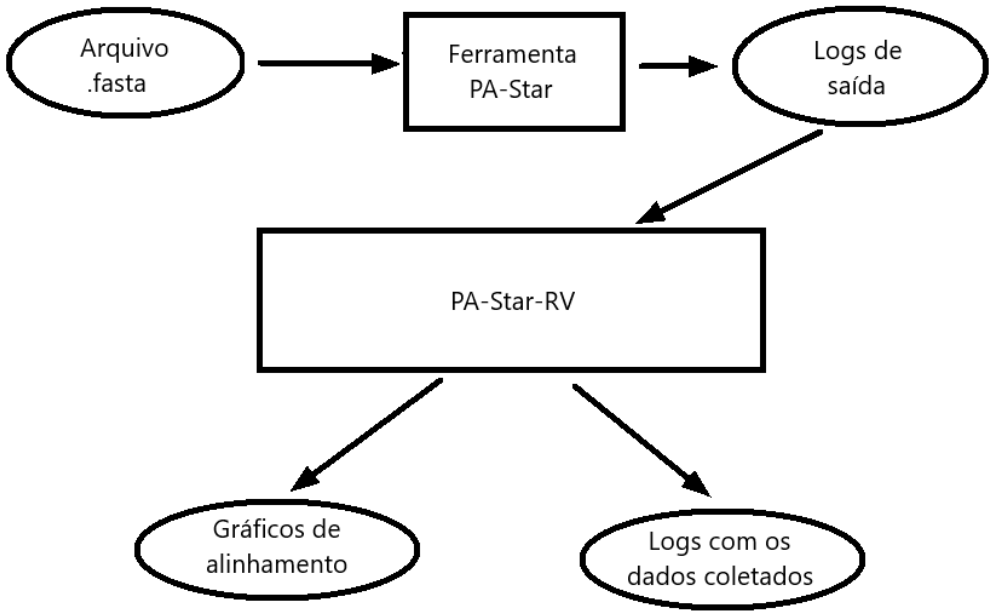


Figura 5.2: Diagrama da ferramenta PA-Star-RV. Fonte: próprio autor

Conforme a Figura 5.1, a ferramenta A-Star-RV é executada concomitantemente com o A-Star, visto que ela recebe as mesmas entradas e entrega, além do caminho ótimo, as métricas solicitadas. O PA-Star-RV, por outro lado, não é executado conjuntamente com o PA-Star. Em vez disso, conforme Figura 5.2, ele recebe como entrada o arquivo log gerado pelo PA-Star e gera duas saídas: um gráfico com a visualização da execução da varredura e um outro arquivo de log com as métricas coletadas.

A implementação do PA-Star usada neste trabalho está disponível em `https://github.com/danielsundfeld/astar_msa` [16] e foi desenvolvida em C++. As ferramentas A-Star-RV e PA-Star-RV, por sua vez, foram desenvolvidas em Python.

Primeiramente, esta seção apresentará as decisões de projeto tomadas para desenvolver as ferramentas. Em seguida, suas arquiteturas serão detalhadas.

5.2 Decisões de projeto

5.2.1 Métricas consideradas

As ferramentas calculam as seguintes métricas:

- quantidade de vezes que cada vértice foi expandido e em qual ordem;

- quantidades de saltos, que foram definidos como ocorrências em que o vértice expandido não é um vizinho do vértice expandido anteriormente;
- lista de saltos, que mostra quais vértices foram expandidos antes e depois de cada salto;
- média do valor de h dos vizinhos de cada vértice;

5.2.2 Visualização de alinhamentos de 3 sequências com o PA-Star-RV

Deseja-se que a ferramenta desenvolvida seja capaz de mostrar como o PA-Star explora os nós do grafo do processo de alinhamento das sequências biológicas ao longo do tempo. Para tanto, optou-se por visualizar alinhamentos de três sequências biológicas. Desse modo, montaram-se gráficos tridimensionais em que cada eixo representa os índices dos caracteres de uma das sequências do alinhamento. Para representar o tempo, que seria a quarta dimensão, foi usada uma escala de cores que determina em qual iteração do PA-Star cada nó foi expandido. A Figura 5.3 mostra um exemplo de gráfico usado para analisar o alinhamento das sequências do conjunto de sequências PF03426, que faz parte da base PFAM.

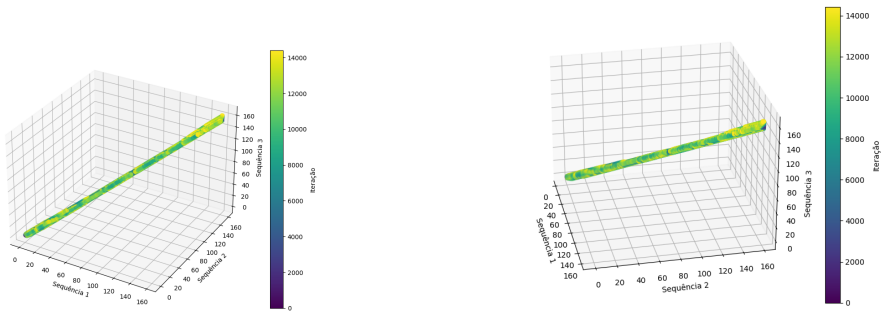


Figura 5.3: Visualização do alinhamento de sequências biológicas do conjunto PF03426 da base PFAM.

Além de visualizar as iterações em que cada nó foi expandido, também foram elaborados gráficos mostrando o valor da variável f do PA-Star escolhida em cada iteração. A Figura 5.4 diz respeito ao mesmo alinhamento da Figura 5.3, mas mostra o valor de f do nó escolhido em cada iteração.

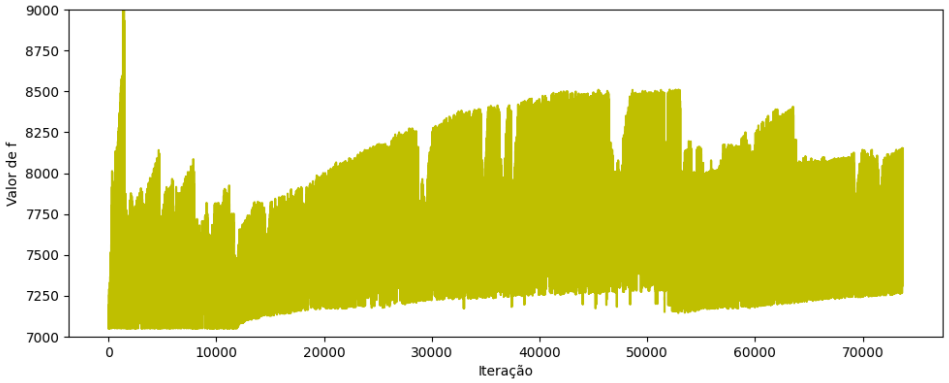


Figura 5.4: Mostra os valores de f para cada nó expandido durante a análise do alinhamento das sequências do conjunto PF03426, base PFAM.

Tentou-se visualizar também o alinhamento de 4 sequências biológicas. Para tanto, uma das dimensões do grafo, a escolha do usuário, seria mostrada de forma discretizada, ou seja, apenas alguns pontos dela seriam mostrados. Contudo, essa forma de representar não se mostrou satisfatória, pois não foi possível visualizar a expansão de todos os nós do grafo. Como não se conseguiu encontrar um meio de visualizar a expansão de alinhamentos de 4 ou mais sequências, optou-se por restringir a análise aos alinhamentos triplos.

A Figura 5.5 mostra uma tentativa de representação de um alinhamento de 4 sequências. Nela, temos 4 gráficos que mostram os nós expandidos com o valor da coordenada da quarta sequência (chamada de "dimensão 3") fixado, respectivamente, em 10, 20, 30 e 40.

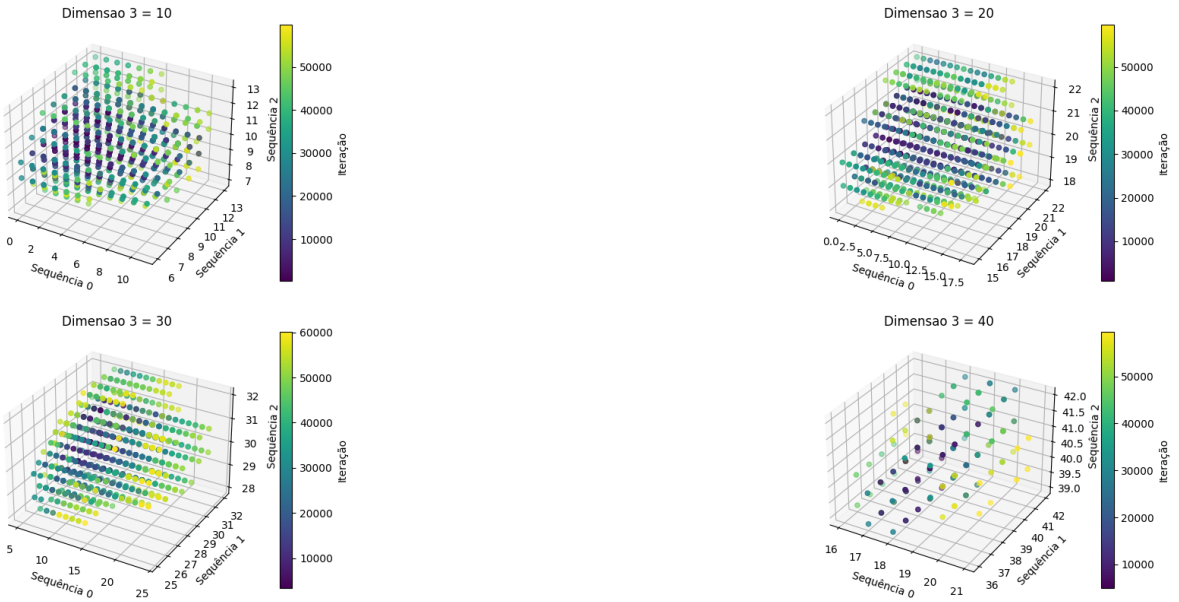


Figura 5.5: Exemplo de visualização de alinhamento de 4 sequências (conjunto PF03913, base PFAM).

5.3 Arquitetura do A-Star-RV

Por ser uma ferramenta que é executada concomitantemente com o A-Star, o A-Star-RV é baseado no Algoritmo 1 (Seção 2.2), com modificações para coletar as métricas. O Algoritmo 3 mostra o funcionamento da ferramenta e destaca em azul as mudanças feitas em relação ao Algoritmo 1.

Nas linhas 1 a 12 são declaradas as variáveis iniciais: a lista *open* recebe o nó inicial (v_0), cujo antecessor ($v_0.predecessor$) é inicializado como NULL e cujo valor inicial de f é calculado. Além disso, são também declaradas as variáveis novas *iterations*, *jumps* e *previous_node*. A primeira é usada para contar a quantidade de iterações e a segunda é usada para contar a quantidade de saltos. A terceira, por fim, registra qual nó foi aberto anteriormente para verificar se houve um salto.

Na linha 9 é iniciado um laço que percorre todos os nós para inicializar mais duas variáveis usadas pela ferramenta: a variável *exp_count*, que calcula a quantidade de vezes que aquele nó foi expandido, e a *avg_h*, que recebe a média dos valores de h dos vizinhos daquele nó.

Após a inicialização das variáveis, a partir da linha 13, é iniciado o *loop* principal da ferramenta. Esse laço será executado enquanto ainda houver algum nó na lista de nós abertos, da mesma forma que ocorre na execução da implementação padrão do A-Star vista no Capítulo 2. Em seguida, na linha 14, a quantidade de iterações é atualizada e, a depender do valor da variável INTERVAL, as listas *open* e *closed* daquela iteração do laço são impressas. A variável INTERVAL é definida pelo usuário previamente à execução da ferramenta e determina de quantas em quantas iterações haverá uma impressão.

Nas linhas 18 a 23 existe um laço para escolher o nó da lista *open* com o menor valor de f, conforme explicado no Capítulo 2. Em seguida, na linha 24, a quantidade de expansões do nó em análise é atualizada. Na linha 26 a ferramenta verifica se houve um salto e, em caso afirmativo, registra o salto e atualiza a contagem deles.

Na linha 30 é verificado se o nó aberto é um dos nós finais. Se sim, a ferramenta é encerrada, o *log* com as métricas calculadas é impresso e o caminho ótimo encontrado é retornado. Caso contrário, os vizinhos do nó são varridos e seus valores de f são calculados e atualizados (linhas 34 a 51) conforme explicado no Capítulo 2. Na linha 52, a variável *previous_node* é atualizada com o valor do vértice atual.

Por fim, se o laço for encerrado sem que um caminho até algum dos nós finais seja encontrado (linha 53), o *log* com as métricas será impresso e a ferramenta retornará NULL, visto que não há caminho ótimo.

Algoritmo 3 A-Star-RV

```
1: open.add(v_0)
2: v_0.predecessor ← NULL
3: v_0.previous_f ← f(v_0)
4: /* Variáveis que guardarão as quantidades de iterações e de saltos. A variável previous_node registra o nó expandido anteriormente e é usada para verificar se houve um salto. */
5: iterations ← 0
6: jumps ← 0
7: previous_node ← NULL
8: /* Variáveis que guardarão, para cada nó do grafo, a quantidade de vezes que ele foi expandido e a média dos valores de h de seus vizinhos. */
9: for all v ∈ V do
10:     v.exp_count ← 0
11:     v.avg_h ← avg_h(v)
12: end for
13: while open.length > 0 do
14:     iterations ← iterations + 1
15:     if iterations mod INTERVAL = 0 then
16:         print_sets()
17:     end if
18:     v ← open[0]
19:     for (i = 1; i < open.length; i++) do
20:         if f(open[i]) < f(v)) then
21:             v ← open[i]
22:         end if
23:     end for
24:     exp_order.add(v)
25:     v.exp_count ← v.exp_count + 1
26:     if previous_node ≠ NULL and previous_node ∉ v.neighbors() then
27:         jumps ← jumps + 1
28:         jump_set.add(previous_node, v)
29:     end if
30:     if v ∈ V_f then
31:         print_log()
32:         return PATH TO v
33:     end if
34:     for all n ∈ v.neighbors() do
35:         if n ∈ open and f(n) < n.previous_f then
36:             n.previous_f ← f(n)
37:             n.predecessor ← v
38:         else if n ∈ closed and f(n) < n.previous_f then
39:             closed.remove(n)
40:             open.add(n)
41:             n.previous_f ← f(n)
42:             n.predecessor ← v
43:             /* If the node was neither open nor closed, it will be opened. */
44:         else
45:             open.add(n)
46:             n.previous_f ← f(n)
47:             n.predecessor ← v
48:         end if
49:     end for
50:     open.remove(v)
51:     closed.add(v)
52:     previous_node ← v
53: end while
54: print_log()
55: return NULL
```

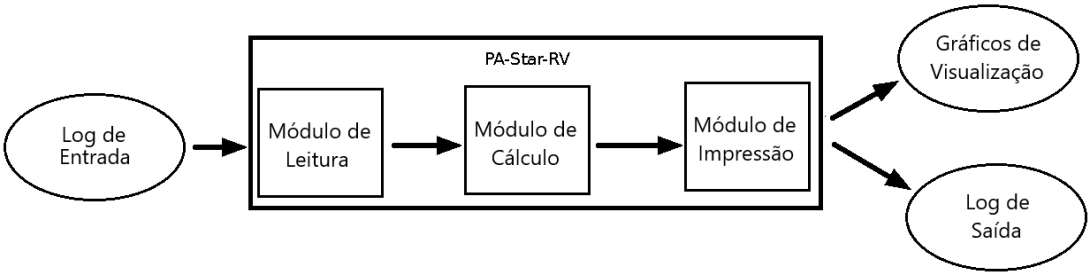


Figura 5.6: Esquema dos módulos do PA-Star-RV

5.4 Arquitetura do PA-Star-RV

5.4.1 Visão geral

A ferramenta PA-Star-RV é dividida em três módulos (Figura 5.6): o módulo de leitura do arquivo de entrada, que extrai os dados relevantes do log; o módulo de cálculo, que usa os dados coletados pelo módulo anterior para calcular as métricas definidas na Seção 5.2.1; e o módulo de impressão, que imprime o log de saída com as métricas calculadas e que também mostra o gráfico de visualização.

5.4.2 Alterações do arquivo de log do PA-Star

A implementação usada da ferramenta PA-Star é a encontrada em [16] com pequenas modificações, uma vez que os *logs* da ferramenta original mostravam apenas os nós expandidos e os valores de f, g e h de cada nó, mas deseja-se que eles mostrem também o *thread* que executou cada expansão e a ordem em que as expansões aconteceram. Para registrar a ordem, acrescentou-se um contador global e atômico de iterações que é lido e incrementado por cada *thread* no momento em que um nó é expandido.

5.4.3 Algoritmo do PA-Star-RV

O Algoritmo 4 mostra o funcionamento da ferramenta PA-Star-RV:

Módulo 1: Leitura do log

O módulo 1 vai das linhas 2 a 6. Nele, o log gerado pela ferramenta PA-Star ($PA-Star\_log$) é lido e dele os seguintes dados são extraídos e guardados em variáveis os nós expandidos, na ordem em que foram expandidos (na linha 6 ocorre a ordenação em função da iteração em que o nó foi expandido), e os valores de f, g e h de cada nó.

Algoritmo 4 PA-Star-RV

```
 1: /* Módulo de leitura */
 2: nodes ← get_nodes(PA − Star_log)
 3: f_values ← get_f(PA − Star_log)
 4: g_values ← get_g(PA − Star_log)
 5: h_values ← get_h(PA − Star_log)
 6: sort(nodes)
 7: /* Módulo de cálculo */
 8: previous_node ← NULL
 9: jumps ← 0
10: jump_set ← ∅
11: for all v ∈ nodes do
12:     expansions[v] ← expansions[v] + 1
13:     if previous_node ≠ NULL & v ∉ previous_node.neighbors then
14:         jumps ← jumps + 1
15:         jump_set.add(previous_node, v)
16:     end if
17:     v.avg_h ← get_avg_h(v.neighbors, H_values)
18:     previous_node ← v
19: end for
20: /* Módulo de impressão */
21: print_log(f_values, g_values, h_values, nodes)
22: create_graphs(nodes, f_values)
23: return
```

Módulo 2: Cálculo das métricas

O segundo módulo vai das linhas 8 a 19 e consiste no cálculo das métricas definidas na Seção 5.2.1 a partir das informações geradas no primeiro módulo. Nas linhas 8 a 10 são definidas três variáveis usadas para verificar os saltos: *previous_node* armazena o nó expandido anteriormente, *jumps* conta a quantidade de saltos e *jump_set* é o conjunto dos saltos ocorridos, ou seja, armazena o nó anterior e o nó atual.

Na linha 12 a quantidade de expansões daquele nó é atualizada e, em seguida, o algoritmo verifica se houve um salto. Se sim, a contagem de saltos é atualizada e a tupla com o nó anterior e o nó atual (*previous_node* e v, respectivamente) é adicionada ao conjunto de saltos.

Por fim, na linha 17 é calculada a média dos valores de h dos vizinhos do vértice e o valor de *previous_node* é atualizado.

Módulo 3: Emissão do log e do gráfico

O último módulo abrange as linhas de 21 a 23 e compreende simplesmente a impressão do *log* com as métricas calculadas (linha 21) e a elaboração do gráfico mostrando as coordenadas de todos os nós expandidos e do gráfico que mostra o valor de f para cada iteração (linha 22). Em seguida, a ferramenta é encerrada (linha 23).

Os gráficos de visualização foram gerados com a biblioteca *pyplot* da linguagem *Python*.

Capítulo 6

Resultados

6.1 Introdução

Este capítulo mostra os resultados obtidos para as ferramentas A-Star-RV e PA-Star-RV. Cada ferramenta foi sujeita a testes distintos. O A-Star-RV foi testado em grafos oriundos de exercícios didáticos ao passo que o PA-Star-RV foi testado em alinhamentos de sequências biológicas.

6.2 Especificações

Os testes foram conduzidos num computador com as seguintes especificações:

- CPU AMD Ryzen 7 3800X com 8 cores (16 cores lógicos), 3893 MHz
- SSD de 120 GiB
- 32 GiB de RAM, DDR4
- Sistema operacional Linux Mint 21.3 64-bit

Os códigos fontes das ferramentas usadas estão disponíveis no repositório a seguir: `https://github.com/edufrma/pa-star-rv`

6.3 Resultados do A-Star-RV

Foram feitos dois tipos de testes com o A-Star-RV: um com grafos estáticos e outro com grafos dinâmicos.

6.3.1 Resultados com Grafos Estáticos

Grafos estáticos são aqueles cujos nós são todos gerados em memória previamente à execução do algoritmo. O código usado para esta primeira análise foi baseado no código escrito em Python e disponível em [7]. Os grafos analisados nesta parte são relativamente pequenos e os dados colhidos foram impressos diretamente no terminal, da seguinte forma:

- mostra os vértices abertos e fechados para cada iteração, além do vértice que será expandido;
- se houve um salto durante a iteração, ele também é mostrado;
- mostra o caminho encontrado no formato "Path found: [Lista dos vértices do caminho ótimo em ordem]";
- contagem de vezes que os vértices foram expandidos;
- quantidade e listagem dos saltos ocorridos;
- média dos valores de h dos vizinhos de cada vértice;
- ordem de análise

Grafo 1

A Figura 6.1 mostra o primeiro grafo analisado. O ponto de partida é A e o de chegada é G.

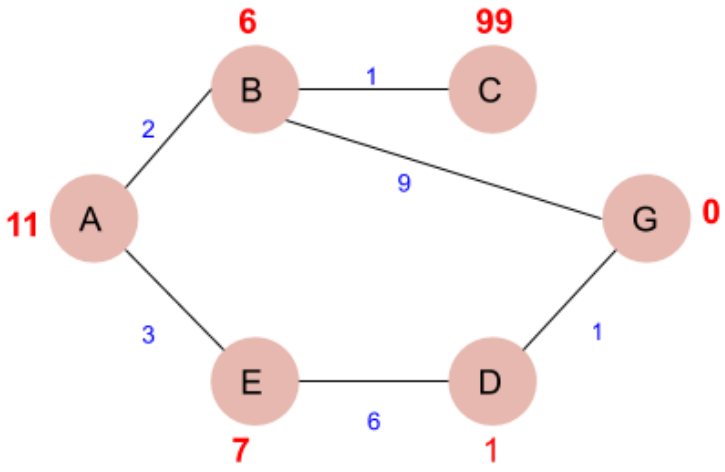


Figura 6.1: Grafo 1. Fonte: `https://www.mygreatlearning.com` [7].

Os resultados obtidos foram:

```
    Iteracao 0
Conjunto aberto:
        {'A'}
Conjunto fechado:
        set()
Vertice escolhido: A

Iteracao 1
Conjunto aberto:
        {'B', 'E'}
Conjunto fechado:
        {'A'}
Vertice escolhido: B

Salto de B para E.
Iteracao 2
Conjunto aberto:
        {'C', 'E', 'G'}
Conjunto fechado:
        {'B', 'A'}
Vertice escolhido: E

Iteracao 3
Conjunto aberto:
        {'C', 'G', 'D'}
Conjunto fechado:
        {'B', 'E', 'A'}
Vertice escolhido: D

Iteracao 4
Conjunto aberto:
        {'C', 'G'}
Conjunto fechado:
        {'D', 'B', 'E', 'A'}
Vertice escolhido: G

Path found: ['A', 'E', 'D', 'G']
Contagem:
{'A': 1, 'B': 1, 'C': 0, 'E': 1, 'D': 1, 'G': 1}
Saltos:
1
[('B', 'E')]
Media das heuristicas vizinhas:
{'A': 6.5, 'B': 36.67, 'C': 6.0, 'E': 6.0, 'D': 3.5, 'G': 3.5}
Ordem de analise:
['A', 'B', 'E', 'D', 'G']
```

Grafo 2

O próximo grafo, na Figura 6.2, é o mesmo grafo do primeiro exemplo mostrado na Seção 2.4.1. O ponto de partida é A e o de chegada é K.

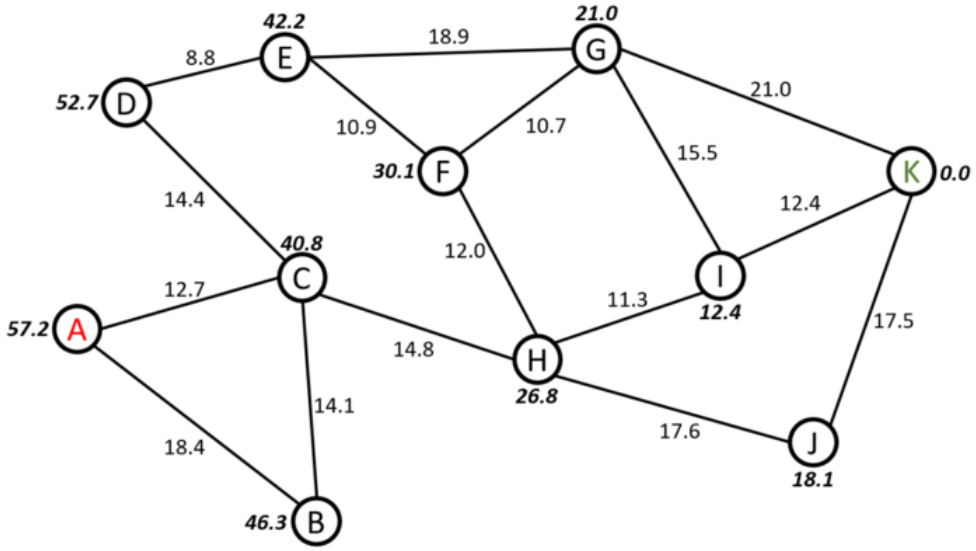


Figura 6.2: Grafo 2. Fonte: `https://optimization.cbe.cornell.edu` [1].

Os resultados obtidos foram:

```
    Iteracao 0
Conjunto aberto:
        {'A'}
Conjunto fechado:
        set()
Vertice escolhido: A

Iteracao 1
Conjunto aberto:
        {'B', 'C'}
Conjunto fechado:
        {'A'}
Vertice escolhido: C

Iteracao 2
Conjunto aberto:
        {'H', 'B', 'D'}
Conjunto fechado:
        {'A', 'C'}
Vertice escolhido: H

Iteracao 3
Conjunto aberto:
        {'D', 'I', 'F', 'J', 'B'}
Conjunto fechado:
        {'A', 'C', 'H'}
Vertice escolhido: I

Iteracao 4
Conjunto aberto:
        {'D', 'K', 'F', 'G', 'J', 'B'}
Conjunto fechado:
        {'A', 'I', 'C', 'H'}
```

```
Vertice escolhido: K

Path found: ['A', 'C', 'H', 'I', 'K']
Contagem:
{'A': 1, 'B': 0, 'C': 1, 'D': 0, 'E': 0, 'F': 0, 'G': 0, 'H': 1, 'I': 1, 'J': 0, 'K': 1}
Saltos:
0
[]
Media das heuristicas vizinhas:
{'A': 43.55, 'B': 49.0, 'C': 45.75, 'D': 41.5, 'E': 34.6, 'F': 30.0, 'G': 21.18, 'H': 25.35, 'I': 15.93,
↪  'J': 13.4, 'K': 17.17}
Ordem de analise:
['A', 'C', 'H', 'I', 'K']
```

Grafo 3

O próximo grafo está na Figura 6.3. O ponto de partida é S e o de chegada é G.

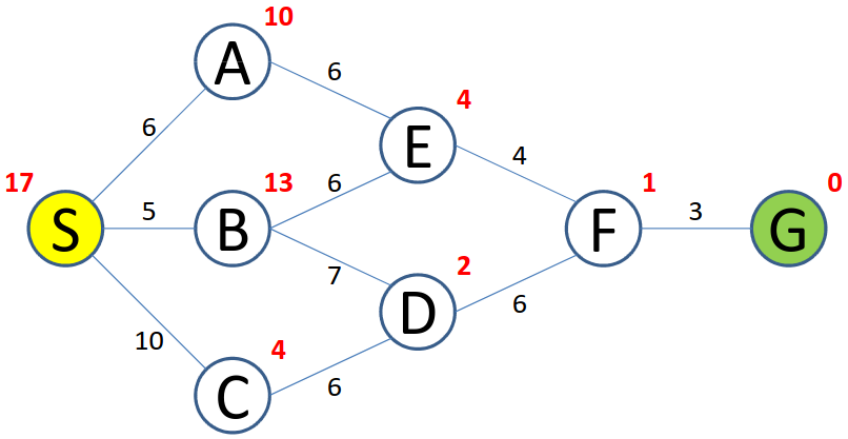


Figura 6.3: Grafo 3. Fonte: `https://www.mygreatlearning.com` [8].

Os resultados obtidos foram:

```
    Iteracao 0
Conjunto aberto:
        {'S'}
Conjunto fechado:
        set()
Vertice escolhido: S

Iteracao 1
Conjunto aberto:
        {'C', 'A', 'B'}
Conjunto fechado:
        {'S'}
Vertice escolhido: C

Salto de C para A.
Iteracao 2
```

```
Conjunto aberto:
        {'D', 'B', 'A'}
Conjunto fechado:
        {'S', 'C'}
Vertice escolhido: A

{...}

Iteracao 10
Conjunto aberto:
        {'G'}
Conjunto fechado:
        {'D', 'C', 'S', 'E', 'F', 'B', 'A'}
Vertice escolhido: G

Path found: ['S', 'B', 'E', 'F', 'G']
Contagem:
{'S': 1, 'A': 1, 'B': 1, 'C': 1, 'D': 2, 'E': 2, 'F': 2, 'G': 1}
Saltos:
2
[('C', 'A'), ('D', 'E')]
Media das heuristicas vizinhas:
{'S': 9.0, 'A': 10.5, 'B': 7.67, 'C': 9.5, 'D': 6.0, 'E': 8.0, 'F': 2.0, 'G': 1.0}
Ordem de analise:
['S', 'C', 'A', 'E', 'F', 'D', 'B', 'D', 'E', 'F', 'G']
```

Grafo 4

O último grafo desta seção está mostrado na Figura 6.4. Deve-se partir de A e chegar em J.

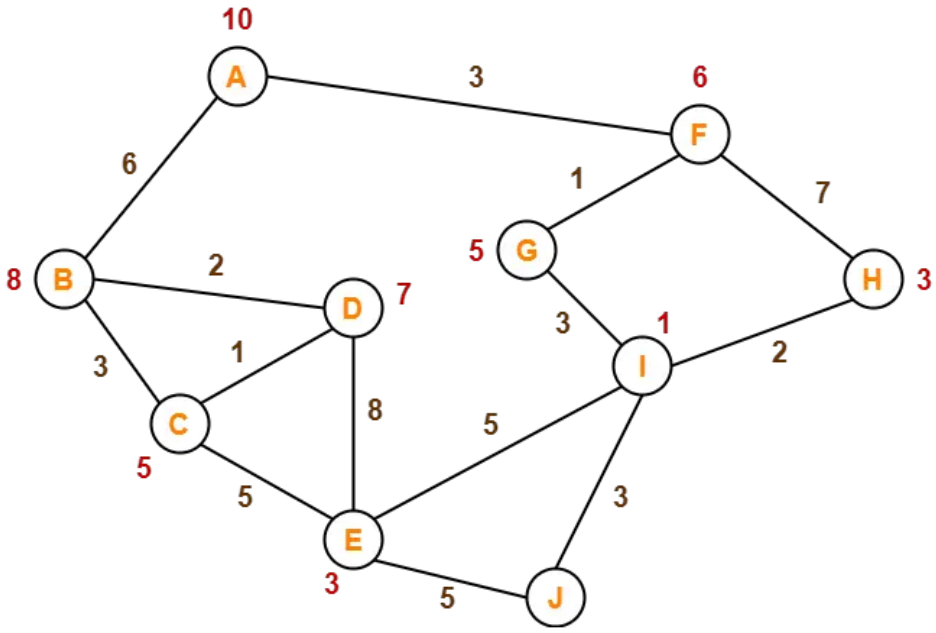


Figura 6.4: Grafo 4. Fonte: `https://www.gatevidyalay.com/` [2].

Os resultados obtidos foram:

```
    Iteracao 0
Conjunto aberto:
        {'A'}
Conjunto fechado:
        set()
Vertice escolhido: A

Iteracao 1
Conjunto aberto:
        {'F', 'B'}
Conjunto fechado:
        {'A'}
Vertice escolhido: F

Iteracao 2
Conjunto aberto:
        {'H', 'B', 'G'}
Conjunto fechado:
        {'F', 'A'}
Vertice escolhido: G

Iteracao 3
Conjunto aberto:
        {'I', 'H', 'B'}
Conjunto fechado:
        {'F', 'G', 'A'}
Vertice escolhido: I

Iteracao 4
Conjunto aberto:
        {'H', 'E', 'B', 'J'}
Conjunto fechado:
        {'I', 'F', 'G', 'A'}
Vertice escolhido: J

Path found: ['A', 'F', 'G', 'I', 'J']
Contagem:
{'A': 1, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 1, 'G': 1, 'H': 0, 'I': 1, 'J': 1}
Saltos:
0
[]
Media das heuristicas vizinhas:
{'A': 7.0, 'B': 7.33, 'C': 6.0, 'D': 5.33, 'E': 3.25, 'F': 6.0, 'G': 3.5, 'H': 3.5, 'I': 2.75, 'J': 2.0}
Ordem de analise:
['A', 'F', 'G', 'I', 'J']
```

6.3.2 Resultados com Grafos Dinâmicos

Os exemplos anteriores eram todos com grafos pequenos e estáticos. Na presente seção, analisaremos os resultados obtidos a partir do grafo proposto na Seção 2.4.2 [2], que é um grafo dinâmico, isto é, antes da execução apenas os nós de início e de fim são gerados em memória. Os demais nós são gerados durante a execução conforme se tornam necessários.

Assim, para trabalhar com um grafo dinâmico, é necessário que o algoritmo seja capaz de deduzir todos os vizinhos de um único nó sem consultar outra base de dados.

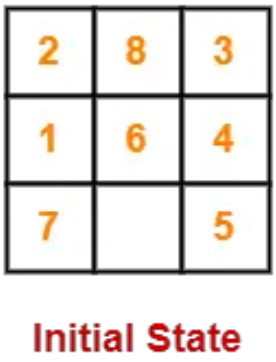
O código usado para analisar estes grafos é diferente do código usado para os grafos estáticos, uma vez que deve incluir um módulo de geração de nós a serem expandidos, e imprime dois arquivos de saída. O primeiro mostra as iterações do algoritmo e os vértices marcados como abertos e fechados em cada uma delas, o valor de g e de h para cada vértice e também o vértice escolhido para ser expandido naquela iteração. O outro arquivo mostra as demais métricas, na seguinte ordem:

1. quantidade de vértices na lista *open*;
2. quantidade de vértices expandidos;
3. quantidade de iterações;
4. caminho ótimo encontrado;
5. contagem de vezes em que cada vértice foi expandido;
6. quantidade e lista de saltos;
7. ordem de análise, que mostra os vértices expandidos em ordem;
8. médias dos valores de h dos vizinhos de cada vértice aberto.

Por fim, cada vértice foi representado por uma tupla composta por três tuplas, que representam, respectivamente, a primeira, a segunda e a terceira linhas dos nós do grafo. O espaço em branco é representado por zero enquanto as peças numeradas de 1 a 8 são representadas por seus próprios números. Assim, por exemplo, o vértice inicial da Figura 6.5 é representado pela tupla $((2, 8, 3), (1, 6, 4), (7, 0, 5))$.

Primeira análise

Na primeira análise, foram usados os mesmos vértices de início e de fim do grafo da Seção 2.4.2, conforme Figura 6.5.



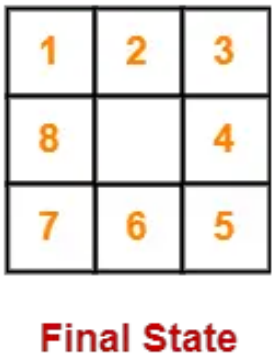


Figura 6.5: Vértices incial e final da primeira análise. Fonte: `https://www.gatevidyalay.com/` [2].

Iterações (primeiro arquivo):

```
Iteracao 1
Conjunto aberto (1 elementos):
((2, 8, 3), (1, 6, 4), (7, 0, 5))        g = 0       h = 4

Conjunto fechado (0 elementos):

Vertice escolhido: ((2, 8, 3), (1, 6, 4), (7, 0, 5))

Iteracao 2
Conjunto aberto (3 elementos):
((2, 8, 3), (1, 6, 4), (7, 5, 0))        g = 1       h = 5
((2, 8, 3), (1, 0, 4), (7, 6, 5))        g = 1       h = 3
((2, 8, 3), (1, 6, 4), (0, 7, 5))        g = 1       h = 5

{...}

Iteracao 6
Conjunto aberto (7 elementos):
((1, 2, 3), (8, 0, 4), (7, 6, 5))        g = 5       h = 0
((2, 8, 3), (1, 4, 0), (7, 6, 5))        g = 2       h = 4
((2, 8, 3), (1, 6, 4), (7, 5, 0))        g = 1       h = 5
((1, 2, 3), (7, 8, 4), (0, 6, 5))        g = 5       h = 2
((2, 3, 0), (1, 8, 4), (7, 6, 5))        g = 3       h = 4
((2, 8, 3), (1, 6, 4), (0, 7, 5))        g = 1       h = 5
((2, 8, 3), (0, 1, 4), (7, 6, 5))        g = 2       h = 3

Conjunto fechado (5 elementos):
((2, 0, 3), (1, 8, 4), (7, 6, 5))
((2, 8, 3), (1, 6, 4), (7, 0, 5))
((1, 2, 3), (0, 8, 4), (7, 6, 5))
((0, 2, 3), (1, 8, 4), (7, 6, 5))
((2, 8, 3), (1, 0, 4), (7, 6, 5))

Vertice escolhido: ((1, 2, 3), (8, 0, 4), (7, 6, 5))
```

Dados (segundo arquivo):

```
12 vertices na lista open.
0.00 % do total.

6 vertices expandidos.
0.00 % do total.

6 iteracoes.
Caminho encontrado:
((2, 8, 3), (1, 6, 4), (7, 0, 5))
((2, 8, 3), (1, 0, 4), (7, 6, 5))
((2, 0, 3), (1, 8, 4), (7, 6, 5))
((0, 2, 3), (1, 8, 4), (7, 6, 5))
((1, 2, 3), (0, 8, 4), (7, 6, 5))
((1, 2, 3), (8, 0, 4), (7, 6, 5))

Contagem:
((2, 8, 3), (1, 6, 4), (7, 0, 5)) : 1
((2, 8, 3), (1, 0, 4), (7, 6, 5)) : 1
((2, 8, 3), (1, 6, 4), (0, 7, 5)) : 0
((2, 8, 3), (1, 6, 4), (7, 5, 0)) : 0
((2, 0, 3), (1, 8, 4), (7, 6, 5)) : 1
((2, 8, 3), (0, 1, 4), (7, 6, 5)) : 0
((2, 8, 3), (1, 4, 0), (7, 6, 5)) : 0
((0, 2, 3), (1, 8, 4), (7, 6, 5)) : 1
((2, 3, 0), (1, 8, 4), (7, 6, 5)) : 0
((1, 2, 3), (0, 8, 4), (7, 6, 5)) : 1
((1, 2, 3), (7, 8, 4), (0, 6, 5)) : 0
((1, 2, 3), (8, 0, 4), (7, 6, 5)) : 1

Saltos: 0

Ordem de analise:
((2, 8, 3), (1, 6, 4), (7, 0, 5))
((2, 8, 3), (1, 0, 4), (7, 6, 5))
((2, 0, 3), (1, 8, 4), (7, 6, 5))
((0, 2, 3), (1, 8, 4), (7, 6, 5))
((1, 2, 3), (0, 8, 4), (7, 6, 5))
((1, 2, 3), (8, 0, 4), (7, 6, 5))

Media das distancias estimadas dos vizinhos:
((1, 2, 3), (8, 0, 4), (7, 6, 5)) : 1.00
((2, 8, 3), (1, 4, 0), (7, 6, 5)) : 4.33
((2, 8, 3), (1, 6, 4), (7, 5, 0)) : 5.00
((1, 2, 3), (7, 8, 4), (0, 6, 5)) : 2.00
((2, 0, 3), (1, 8, 4), (7, 6, 5)) : 3.00
((2, 3, 0), (1, 8, 4), (7, 6, 5)) : 4.00
((2, 8, 3), (1, 6, 4), (7, 0, 5)) : 4.33
((2, 8, 3), (0, 1, 4), (7, 6, 5)) : 3.33
((1, 2, 3), (0, 8, 4), (7, 6, 5)) : 1.33
((0, 2, 3), (1, 8, 4), (7, 6, 5)) : 2.00
((2, 8, 3), (1, 0, 4), (7, 6, 5)) : 3.50
((2, 8, 3), (1, 6, 4), (0, 7, 5)) : 4.50
```

Segunda análise

Para a segunda análise, foi usado o mesmo grafo do exemplo anterior, mas com o vértice de saída modificado para $((3,2,1),(7,0,5),(4,6,8))$. Como desta análise resultaram 5277 iterações, serão mostradas apenas partes dos arquivos de saída. Além disso, nesta análise, apenas as iterações múltiplas de 200 foram registradas.

Iterações (primeiro arquivo):

```
Iteracao 200
Conjunto aberto (137 elementos):
((8, 3, 4), (0, 2, 6), (1, 7, 5))         g = 8        h = 8
((8, 3, 4), (2, 7, 6), (1, 0, 5))         g = 8        h = 8
{...}
((8, 6, 3), (2, 7, 4), (1, 5, 0))         g = 7        h = 8
((2, 4, 8), (6, 0, 3), (1, 7, 5))         g = 7        h = 8

Conjunto fechado (199 elementos):
((2, 3, 4), (1, 8, 5), (0, 7, 6))
((1, 2, 3), (7, 8, 4), (6, 0, 5))
{...}
((1, 2, 8), (7, 6, 3), (0, 5, 4))
((6, 2, 3), (0, 8, 4), (1, 7, 5))

Vertice escolhido: ((2, 0, 3), (6, 8, 5), (1, 4, 7))

Iteracao 400
Conjunto aberto (258 elementos):
((2, 8, 3), (0, 5, 1), (7, 6, 4))         g = 10        h = 7
((1, 3, 4), (8, 6, 2), (7, 0, 5))         g = 10        h = 8
((6, 2, 3), (1, 8, 5), (0, 4, 7))         g = 11        h = 6
{...}
```

Dados (segundo arquivo):

```
8357 vertices na lista open.
2.30 % do total.

5277 vertices expandidos.
1.45 % do total.

5277 iteracoes.
Caminho encontrado:
((2, 8, 3), (1, 6, 4), (7, 0, 5))
((2, 8, 3), (1, 0, 4), (7, 6, 5))
((2, 8, 3), (1, 4, 0), (7, 6, 5))
((2, 8, 3), (1, 4, 5), (7, 6, 0))
((2, 8, 3), (1, 4, 5), (7, 0, 6))
((2, 8, 3), (1, 0, 5), (7, 4, 6))
((2, 0, 3), (1, 8, 5), (7, 4, 6))
((2, 3, 0), (1, 8, 5), (7, 4, 6))
((2, 3, 5), (1, 8, 0), (7, 4, 6))
((2, 3, 5), (1, 0, 8), (7, 4, 6))
((2, 3, 5), (0, 1, 8), (7, 4, 6))
((0, 3, 5), (2, 1, 8), (7, 4, 6))
((3, 0, 5), (2, 1, 8), (7, 4, 6))
```

```
((3, 1, 5), (2, 0, 8), (7, 4, 6))
((3, 1, 5), (0, 2, 8), (7, 4, 6))
((3, 1, 5), (7, 2, 8), (0, 4, 6))
((3, 1, 5), (7, 2, 8), (4, 0, 6))
((3, 1, 5), (7, 2, 8), (4, 6, 0))
((3, 1, 5), (7, 2, 0), (4, 6, 8))
((3, 1, 0), (7, 2, 5), (4, 6, 8))
((3, 0, 1), (7, 2, 5), (4, 6, 8))
((3, 2, 1), (7, 0, 5), (4, 6, 8))

Contagem:
((2, 8, 3), (1, 6, 4), (7, 0, 5)) : 1
((2, 8, 3), (1, 0, 4), (7, 6, 5)) : 1
{...}
((2, 8, 4), (3, 5, 0), (7, 1, 6)) : 0
((3, 8, 0), (2, 7, 4), (1, 6, 5)) : 1
{...}
((3, 4, 8), (2, 0, 6), (1, 7, 5)) : 1
((2, 8, 3), (0, 5, 4), (1, 6, 7)) : 1
{...}
((0, 3, 1), (7, 2, 5), (4, 6, 8)) : 0

Saltos: 4798
(((3, 2, 4), (1, 0, 5), (8, 7, 6)), ((6, 1, 2), (7, 3, 8), (0, 5, 4)))
{...}
(((2, 3, 6), (5, 8, 4), (1, 7, 0)), ((8, 4, 6), (2, 7, 3), (1, 5, 0)))
(((4, 1, 2), (3, 8, 5), (7, 6, 0)), ((8, 3, 4), (1, 2, 7), (0, 5, 6)))

Ordem de analise:
((2, 8, 3), (1, 6, 4), (7, 0, 5))
((2, 8, 3), (1, 0, 4), (7, 6, 5))
{...}
((3, 2, 1), (7, 0, 5), (4, 6, 8))

Media das distancias estimadas dos vizinhos:
((8, 4, 5), (2, 3, 7), (1, 0, 6)) : 7.67
((2, 0, 3), (8, 4, 6), (1, 7, 5)) : 7.67
{...}
```

6.4 Resultados do PA-Star-RV

Para testar o PA-Star-RV, foram feitos alinhamentos de dois conjuntos de sequências diferentes: as sequências do conjunto PF03426, da base PFAM, disponível em [17], e um conjunto de sequências sintéticas cujo alinhamento se mostrou mais difícil. O escore dos alinhamentos foi calculado com base na matriz PAM-250, mostrada na Figura 6.6.

Conforme explicado na Seção 5.2.2, optou-se por analisar o alinhamento de conjuntos de três sequências biológicas, visto que não se conseguiu encontrar uma forma satisfatória de visualizar o alinhamento de conjuntos com 4 ou mais sequências. Sendo assim, apesar do conjunto PF03426 possuir 25 sequências, somente 3 foram utilizadas.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	2	-2	0	0	-2	0	0	1	-1	-1	-2	-1	-1	-3	1	1	1	-6	-3	0	0	0	0	-8
R	-2	6	0	-1	-4	1	-1	-3	2	-2	-3	3	0	-4	0	0	-1	2	-4	-2	-1	0	-1	-8
N	0	0	2	2	-4	1	1	0	2	-2	-3	1	-2	-3	0	1	0	-4	-2	-2	2	1	0	-8
D	0	-1	2	4	-5	2	3	1	1	-2	-4	0	-3	-6	-1	0	0	-7	-4	-2	3	3	-1	-8
C	-2	-4	-4	-5	12	-5	-5	-3	-3	-2	-6	-5	-5	-4	-3	0	-2	-8	0	-2	-4	-5	-3	-8
Q	0	1	1	2	-5	4	2	-1	3	-2	-2	1	-1	-5	0	-1	-1	-5	-4	-2	1	3	-1	-8
E	0	-1	1	3	-5	2	4	0	1	-2	-3	0	-2	-5	-1	0	0	-7	-4	-2	3	3	-1	-8
G	1	-3	0	1	-3	-1	0	5	-2	-3	-4	-2	-3	-5	0	1	0	-7	-5	-1	0	0	-1	-8
H	-1	2	2	1	-3	3	1	-2	6	-2	-2	0	-2	-2	0	-1	-1	-3	0	-2	1	2	-1	-8
I	-1	-2	-2	-2	-2	-2	-2	-3	-2	5	2	-2	2	1	-2	-1	0	-5	-1	4	-2	-2	-1	-8
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6	-3	4	2	-3	-3	-2	-2	-1	2	-3	-3	-1	-8
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5	0	-5	-1	0	0	-3	-4	-2	1	0	-1	-8
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	6	0	-2	-2	-1	-4	-2	2	-2	-2	-1	-8
F	-3	-4	-3	-6	-4	-5	-5	-5	-2	1	2	-5	0	9	-5	-3	-3	0	7	-1	-4	-5	-2	-8
P	1	0	0	-1	-3	0	-1	0	0	-2	-3	-1	-2	-5	6	1	0	-6	-5	-1	-1	0	-1	-8
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	2	1	-2	-3	-1	0	0	0	-8
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3	-5	-3	0	0	-1	0	-8
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17	0	-6	-5	-6	-4	-8
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-4	-2	7	-5	-3	-3	0	10	-2	-3	-4	-2	-8
V	0	-2	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	-1	0	-6	-2	4	-2	-2	-1	-8
B	0	-1	2	3	-4	1	3	0	1	-2	-3	1	-2	-4	-1	0	0	-5	-3	-2	3	2	-1	-8
Z	0	0	1	3	-5	3	3	0	2	-2	-3	0	-2	-5	0	0	-1	-6	-4	-2	2	3	-1	-8
X	0	-1	0	-1	-3	-1	-1	-1	-1	-1	-1	-1	-1	-2	-1	0	0	-4	-2	-1	-1	-1	-1	-8
*	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	1

Figura 6.6: Matriz de substituição PAM 250. Fonte: `https://github.com/biogo/` [4]

As Seções 6.4.1 e 6.4.2 mostram os gráficos confeccionados para cada alinhamento. Para cada conjunto de sequências foram feitos quatro testes: um com apenas 1 *thread*, um com 2, um com 4 e um com 8.

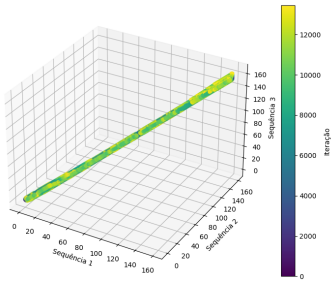
6.4.1 Gráficos do conjunto PF03426

O conjunto PF03426 é composto pelas seguintes sequências de aminoácidos:

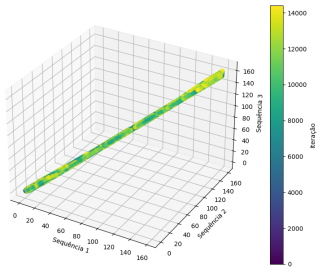
`Sequência 1:`
`IEVDMANGWRGNASGSTSHSGITYSADGVTFA`
`ALGDGVGAVFDIARPTTLEDAVIAMVVNVSAE`
`FKASEANLQIFAQLKEDWSKGEWDCLAASSEL`
`TADTDLTLTCTIDEDDDKFNQTARDVQVGIQA`
`KGTPAGTITIKSVTITLAQEAYSANVDHLRD`

`Sequência 2:`
`IEVDMANGWRGNASGSTSHSGITYSADGVTFA`
`ALGDGVGAVFDIARPTTLEDAVIAMVVNVSAE`
`FKASEANLQIFAQLKEDWSKGEWDCLAASSEL`

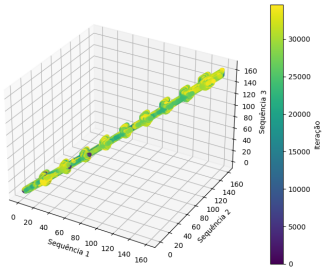
```
TADTDLTLTCTIDEDDDKFNQTARDVQVGIQA
KGTPAGTITIKSVTITLAQEAYSANVDHLRD

Sequência 3:
LTLDMTSGWRGNGTGNSGVTLNSDGVSFTASG
DNIGAVTDMLKPIQLEDAIITMVVNVSSEFKA
SGASLQPIVQIKGGSYPGEWGCWAGNELFTAG
EDATISCTVTEGDKKFNQTEFDVQVGVQAKGT
PTGVVTIKSVTVTLAPAASSSSAQSSSTGSVY
SAN
```



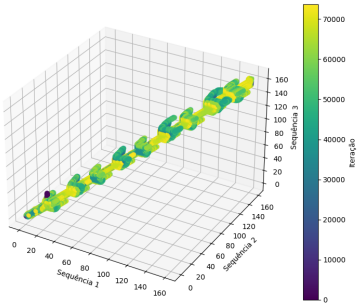
(a) 1 thread



(b) 2 threads



(c) 4 threads



(d) 8 threads

Figura 6.7: Gráfico (número da iteração vs nós explorados) da execução do conjunto PF03426 feito com 1, 2 4 e 8 *threads*

A Figura 6.7 mostra os nós que foram explorados pela ferramenta PA-Star trabalhando com 1, 2, 4 e 8 *threads*, ao passo que a Figura 6.8 mostra os valores de f do nó expandido em cada iteração. Por fim, a Tabela 6.1 mostra a quantidade de iterações, o tempo de execução do alinhamento e o tamanho do *log* gerado em função da quantidade de *threads*

Para este alinhamento, a expansão dos nós ficou próxima à diagonal do cubo (Figura 6.7), com maior quantidade de nós expandidos pelas proximidades nos casos em que

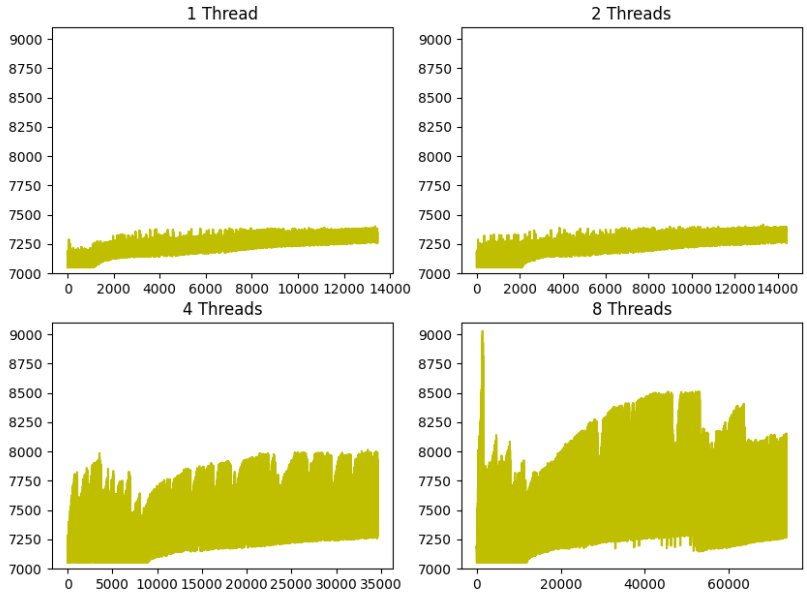


Figura 6.8: Valores de f expandidos em cada iteração do alinhamento do conjunto PF03426

Tabela 6.1: Iterações, tempo de execução e tamanho dos *logs* para o conjunto PF03426

Threads	Iterações	Tempo de Execução	Tamanho do *log*
1	13.454	0,023 s	754,6 KiB
2	14.418	0,014 s	809,6 KiB
4	34.614	0,026 s	2,0 MiB
8	73.732	0,061 s	4,2 MiB

mais *threads* foram executados. As verificações com mais *threads* executam uma quantidade maior de iterações, o que explica por que uma quantidade maior de nós é expandida. Contudo, não há grandes desvios em relação à diagonal do cubo, o que indica se tratar de um alinhamento mais simples, isto é, um alinhamento de sequências relativamente parecidas.

Quanto ao tempo de execução, ele foi menor com 2 *threads* e depois voltou a subir, provavelmente em decorrência da quantidade maior de iterações (a quantidade de iterações com 2 *threads* aumenta em aproximadamente 7%, ao passo que esse aumento supera 100% com 4 e com 8 *threads*). Assim, isso sugere a existência de uma quantidade ótima de *threads* em relação ao tempo de execução.

6.4.2 Gráficos do conjunto sintético

O conjunto sintético é composto pelas seguintes sequências de aminoácidos:

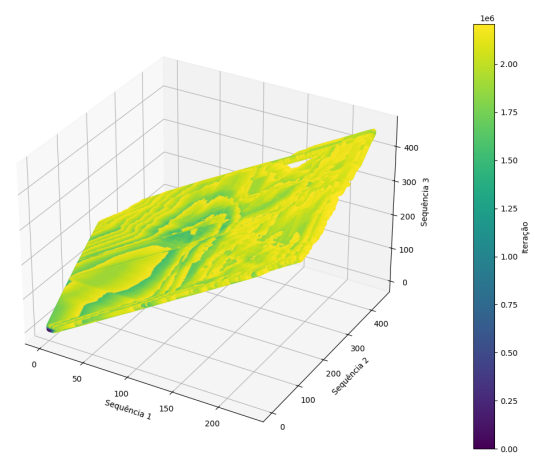
```
Sequência 1:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
KVADALTNAVAHVDDMPNALSALSDLHAHK
LRVDPVNFKLLSHCLLVTLAAHLPAEFTPA
VHASLDKFLASVSTVLTSKYR

Sequência 2:
AAAAAPPPPPPPPPPPPPPPPPPPPPP
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
PPPPPPPPPEKSAVTALPPPPPPPPPPPPP
VHLTPEEKSAVTALWGKVNVDEVGGEALGR
LLVVYPWTQRFFESFGDLSTPDAVMGNPKV
KAHGKKVLGAFSDGLAHLDNLKGTFATLSE
LHCDKLHVDPENFRLLGNVLVCVLAHHFGK
EFTPPVQAAYQKVVAGVANALAHKYH
DDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
```

```
KVADALTNAVAHVDDMPNALSALSDLHAHK
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Sequência 3:
DDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VASAPADKTNVKAAWGKVGAHAGEYAAAAA
GLSDGEWQLVLNVWGKVEADIPGHGQEVLI
RLFKGHPETLEKFDKFKHLKSEDEMKASED
LKKHGATVLTALGGILKKKGHHEAEIKPLA
AAAAAPPPPPPPPPPPPPPPPPPPPPPPP
QSHATKHKIPVKYLEFISECIIQVLQSKHP
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAPPPPPPPPPPPPPPPPPPPPPPPP
GDFGADAQGAMNKALELFRKDMASNYKELG
FQG
```

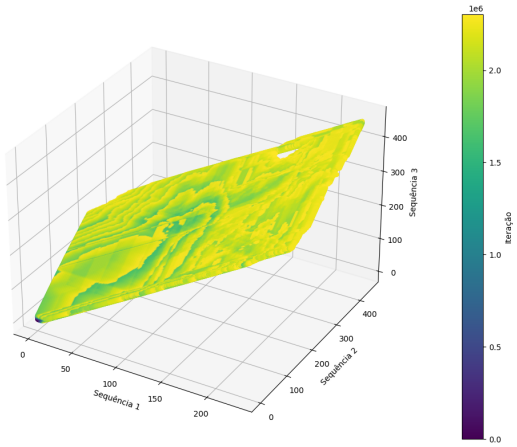
A Figura 6.9 mostra os nós explorados pelo PA-Star rodando com 1, 2, 4 e 8 *threads*. A Figura 6.10, por sua vez, mostra os valores de f do nó expandido em cada iteração e a Tabela 6.2 mostra a quantidade de iterações, o tempo de execução e os tamanhos dos *logs* gerados em função da quantidade de *threads*.

O alinhamento do conjunto sintético ocupou uma região do espaço de busca similar a um plano, indicando desvios muito maiores da diagonal, representando uma quantidade maior de nós explorados, o que indica que se trata de um alinhamento mais complexo. Novamente, as execuções com mais *threads* apresentaram mais desvios, o que pode ser observado pelo aumento da 'rugosidade' dos gráficos conforme a quantidade de *threads* cresce.

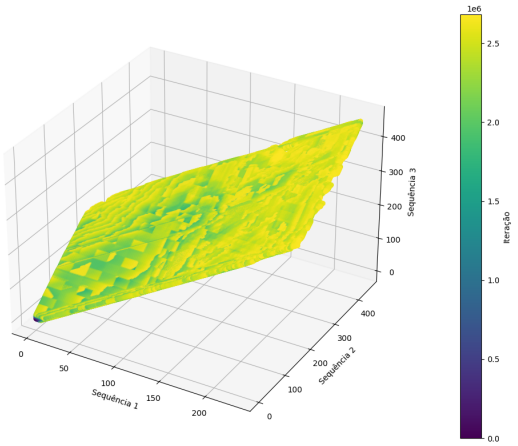
Assim como aconteceu com o alinhamento do conjunto PF03426, o menor tempo de execução foi obtido ao rodar o alinhamento com 2 *threads* ($\approx 71\%$ do maior tempo), o que reforça a ideia de que há um ponto ótimo para a quantidade de *threads* devido ao aumento da quantidade de iterações que acompanha o crescimento do paralelismo.



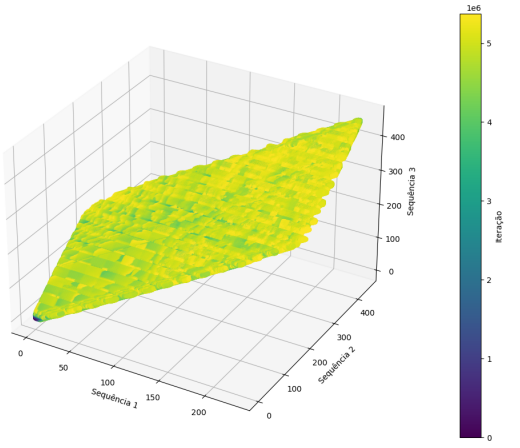
(a) 1 thread



(b) 2 threads



(c) 4 threads



(d) 8 threads

Figura 6.9: Gráfico (número da iteração vs nós explorados) da execução do conjunto sintético feito com 1, 2, 4 e 8 *threads*

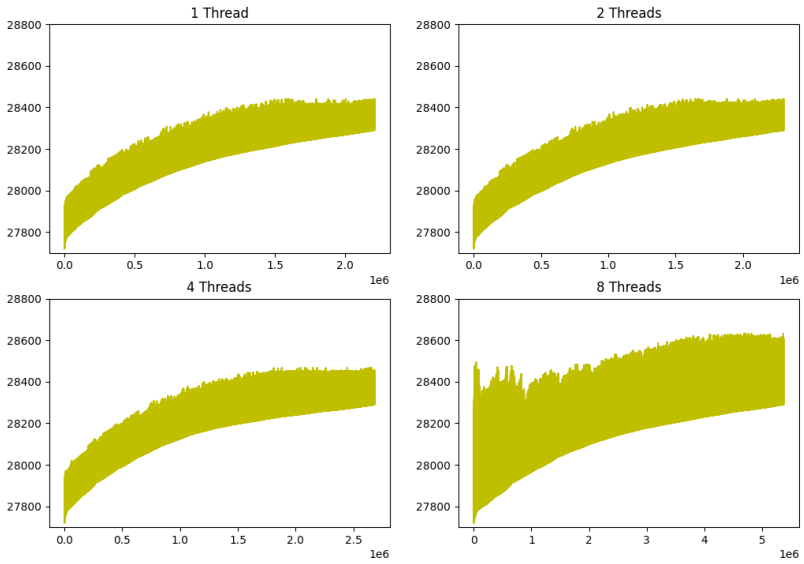


Figura 6.10: Valores de f expandidos em cada iteração do alinhamento do conjunto sintético

Tabela 6.2: Iterações, tempo de execução e tamanho dos *logs* para o conjunto sintético

Threads	Iterações	Tempo de Execução	Tamanho do *log*
1	2.209.751	02,516 s	136,8 MiB
2	2.304.415	01,572 s	142,8 MiB
4	2.683.259	02,125 s	166,6 MiB
8	5.376.697	03,545 s	335,2 MiB

6.4.3 Análise

Os resultados de ambos os conjuntos, mas particularmente do sintético, aparentam similaridades com os resultados obtidos pelo método de Carillo-Lipman [11] descrito na Seção 3.3.2. A forma similar a um plano obtida na análise do alinhamento do conjunto sintético pode apresentar correspondência com a redução do espaço de busca proposta pelos referidos autores. Com efeito, se forem comparadas a Figura 6.9 e a Figura 3.7, que apresenta um exemplo dos limites superior e inferior do Carrillo-Lipman, pode-se ver a correspondência entre os dois métodos (Carrillo-Lipman e A-Star) na exploração do espaço de busca.

Embora se pudesse esperar que aumentar a quantidade de *threads* indefinidamente fosse diminuir o tempo de execução, os resultados também mostraram que esse aumento não garante, necessariamente, uma redução do tempo de execução, uma vez que o cresci-

mento do número de *threads* é acompanhado de uma maior quantidade de iterações e de um aumento da comunicação entre eles. Assim, existe uma quantidade ótima de *threads* que garante o menor tempo de execução. Nos testes realizados, que se resumiram a 3 sequências pequenas, o melhor número de *threads* é 2.

Além disso, os resultados evidenciaram problemas para registrar a quantidade de saltos (ocorrem quando o nó expandido numa iteração não é vizinho do nó expandido anteriormente) com o PA-Star-RV, pois é possível que duas iterações seguidas sejam executadas por *threads* diferentes, o que aumenta as chances de gerar um registro de salto, tendo em vista que a ferramenta PA-Star procura respeitar a localidade ao distribuir os nós para cada *thread*. Contudo, se nessa situação um mesmo *thread* expandiu dois nós vizinhos, o registro do salto será indevido. Isso se dá por causa do não-determinismo que advém da execução de múltiplos threads.

Outra dificuldade do registro de saltos ocorre quando não são gerados *logs* de todas as iterações. Devido a grande quantidade de iterações que podem ser geradas, o usuário pode limitar a quantidade de iterações registradas nos *logs* (ex.: registrar somente as iterações múltiplas de 512). Contudo, se isso for feito, o *log* não conterá informações relativas a nós expandidos consecutivamente, impedindo que a métrica de saltos seja utilizada.

Capítulo 7

Conclusão e Trabalhos Futuros

7.1 Conclusões

O presente trabalho de graduação propôs, implementou e avaliou duas ferramentas de visualização de execuções, uma para o algoritmo A-Star e outra para a ferramenta PA-Star aplicada à solução do problema do alinhamento múltiplo de sequências biológicas.

Primeiramente, ambas as ferramentas têm funcionalidades em comum: o cálculo das métricas definidas na seção 5.2.1 e também o registro de todos os nós que foram explorados.

A ferramenta A-Star-RV é executada concomitantemente com o A-Star e entrega seus resultados em formato textual. O PA-Star-RV, por sua vez, é executado após o encerramento do PA-Star e com base nos *logs* de saída entregues por ele e, além das funcionalidades em comum com o A-Star, também mostra uma visualização dos nós expandidos e da ordem em que eles foram expandidos.

A ferramenta A-Star-RV foi testada com grafos estáticos e dinâmicos usados predominantemente para fins didáticos e se mostrou capaz de registrar as métricas solicitadas.

Com os resultados obtidos pelo PA-Star-RV, mostrou-se que aumentar a quantidade de *threads* do PA-Star não garante, necessariamente, uma redução do tempo de execução, visto que ela é acompanhada de um aumento na quantidade de iterações do algoritmo e na comunicação entre os *threads*. Assim, existe uma quantidade ótima de *threads* para serem executados.

Além disso, não se conseguiu registrar de forma fidedigna os saltos com a ferramenta PA-Star-RV, visto que ela não leva em conta quais *threads* estão executando quais iterações ao verificar se houve saltos. Desse modo, se dois nós expandidos consecutivamente tiverem sido avaliados por *threads* distintos, eles dificilmente serão vizinhos, uma vez que os vértices são distribuídos conforme uma função sensível à localidade, o que resulta num registro provavelmente indevido de um salto.

Os resultados obtidos indicam que há semelhança entre a redução do espaço de busca proposta por Carrillo e Lipman [11] e a região formada pelos nós que foram expandidos pelo PA-Star. A nosso conhecimento, essa é a primeira vez que a possível semelhança entre os dois métodos é apresentada de maneira gráfica, conforme Figura 6.9.

7.2 Trabalhos Futuros

Como trabalhos futuros, sugere-se primeiramente resolver a questão da medida dos saltos. Uma possível alternativa poderia ser considerar se houve saltos ou não apenas em iterações consecutivas de um mesmo *thread*, em vez de avaliar a ordem de expansão global.

Além disso, sugere-se também executar o PA-Star-RV de concomitantemente ao PA-Star, em vez de executá-lo de forma separada com base nos *logs* de saída. Com isso, seria possível registrar os saltos mesmo se o usuário optar por não imprimir todas as iterações no *log*. Contudo, recomenda-se que a execução simultânea seja opcional, visto que o cálculo das métricas pode aumentar o tempo de execução.

Em seguida, propõe-se a criação de vídeos que mostrem a expansão dos nós registradas nas Figuras 6.7 e 6.9, o que permitiria a observação da expansão de cada nó no decorrer do tempo. Também é proposta a marcação nos gráficos de visualização dos limites de Carrillo e Lipman, para verificar se eles de fato foram respeitados, como apontam os indícios encontrados.

Outro trabalho seria a análise das métricas obtidas de modo a tentar encontrar um padrão para prever o comportamento do A-Star e do PA-Star. Ferramentas de análise de dados poderiam ser usadas para encontrar padrões de comportamento, levando em conta a ordem na qual os nós foram expandidos para um dado conjunto de sequências. Ao formar uma base de dados com muitas execuções diferentes, técnicas de aprendizado de máquina poderiam ser utilizadas para definir conjuntos de sequências que possuem o mesmo comportamento, por exemplo.

Referências

[1] *A-star algorithm.* https://optimization.cbe.cornell.edu/index.php?title=A-star_algorithm#Numerical_Example, acessado em 25/09/2023. ix, x, 5, 6, 37

[2] *A* algorithm | a* algorithm example in ai.* https://www.gatevidyalay.com/a-algorithm-a-algorithm-example-in-ai/, acessado em 25/09/2023. ix, x, 7, 8, 9, 39, 40, 42

[3] Mount, David W.: *Bioinformatics: Sequence and Genome Analysis*. CSHL Press, 2004. ix, 10, 11, 13

[4] https://github.com/biogo/biogo/blob/master/align/matrix/matrices, acessado em 19/07/2024. ix, x, 12, 46

[5] Durbin, Richard, Sean Eddy, Anders Krogh e Graeme Mitchison: *Biological Sequence Analysis*. Cambridge University Press, 1998. ix, 10, 13, 14, 15, 16

[6] Sundfeld, Daniel, Caina Razzolini, George Teodoro, Azzedine Boukerche e Alba Cristina Magalhaes Alves de Melo: *Pa-star: A disk-assisted parallel a-star strategy with locality-sensitive hash for multiple sequence alignment*. Journal of Parallel and Distributed Computing, 112:154–165, 2018, ISSN 0743-7315. https://www.sciencedirect.com/science/article/pii/S0743731517301508, Parallel Optimization using/for Multi and Many-core High Performance Computing. ix, 1, 17, 18, 19, 20, 21, 22, 23

[7] *What is a* search algorithm? 2023.* https://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence, acessado em 25/09/2023. x, 35

[8] *Exercises: Artificial intelligence.* https://dtai.cs.kuleuven.be/education/ai/Exercises/Session2/Solutions/solution.pdf, acessado em 26/09/2023. x, 38

[9] Hart, Peter E., Nils J. Nilsson e Bertram Raphael: *A formal basis for the heuristic determination of minimum cost paths*. IEEE Transactions on Systems Science and Cybernetics, 4(2):100–107, 1968. 1, 3, 5, 17, 18

[10] L.Wang e T.Jiang: *On the complexity of multiple sequence alignment*. Journal of Computational Biology, 1994. 1, 17

[11] Carrillo, H. e D.Lipman: *The multiple sequence alignment problem in biology*. SIAM Journal o Applied Mathematics, 1988. 2, 17, 52, 55

[12] Needleman, S. B. e C.D.Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biology, 1970. 14

[13] Smith, T. F. e M. S. Walterman: *Identification of common molecular subsequences.* Journal of Molecular Biology, 1981. 15

[14] Lermen, M. e K. Reinert: *The practical use of the a* algorithm for exact multiple sequence alignment.* Journal of Computational Biology, 2000. 18

[15] Morton, G.M.: *A computer oriented geodetic data base and a new technique in file sequencing.* 1966. 20

[16] *astar_msa.* `https://github.com/danielsundfeld/astar_msa`, acessado em 17/07/2024. 22, 26, 31

[17] *Carbohydrate binding domain (family 15) (pf03426).* `https://www.ebi.ac.uk/interpro/entry/pfam/PF03426/protein/UniProt/#table`, acessado em 25/09/2023. 45