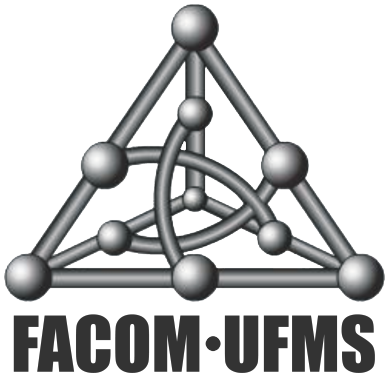
O Problema do Alinhamento de Segmentos

Leandro Ishi Soares de Lima

Dissertação de Mestrado



Orientador: Prof. Dr. Said Sadique Adi

Área de Concentração: Ciência da Computação

Campo Grande, outubro de 2013

O Problema do Alinhamento de Segmentos

Leandro Ishi Soares de Lima

Dissertação apresentada
à
Faculdade de Computação
da
Universidade Federal de Mato Grosso do Sul
para
obtenção do grau de mestre
em
Ciência da Computação

Este exemplar corresponde à redação final da dissertação de mestrado devidamente corrigida e defendida por Leandro Ishi Soares de Lima e aprovada pela Banca Examinadora.

Banca examinadora:

- Prof. Dr. Said Sadique Adi (Orientador) (FACOM/UFMS)
- Prof. Dr. Guilherme Pimentel Telles (IC/Unicamp)
- Prof. Dr. Nalvo Franco de Almeida Junior (FACOM/UFMS)

Agradecimentos

Meus primeiros agradecimentos vão para meus pais, Maria Ilma e Mauro, e para minha namorada, Heloísa, pela paciência, compreensão e apoio incondicional durante a realização deste trabalho. Sem eles, eu não seria capaz de sequer iniciar este estudo.

Sou muito grato também ao meu orientador, professor Said Sadique Adi, que me direcionou corretamente em várias etapas desse projeto, além de me ensinar novos conceitos e técnicas da Ciência da Computação, corrigir meus erros e esclarecer minhas dúvidas (que não eram poucas). Espero ter retribuído toda a atenção dedicada por ele a mim em reuniões que às vezes excediam o limite de tempo e em *e-mails* longos e frequentes, que de vez em quando apareciam em sua caixa de entrada nos sábados à noite e nos domingos pela tarde.

Agradeço também aos vários professores da UFMS que de alguma forma contribuíram para o desenvolvimento desse projeto, em especial: Francisco Elói Soares de Araújo, Vagner Pedrotti, Edna Ayako Hoshino, Cláudio Leonardo Lucchesi, Fábio Henrique Viduani Martinez, Irineu Sotoma, Marcelo Henriques de Carvalho, Marco Aurélio Stefanes, Nalvo Franco de Almeida Junior, Rodrigo Mitsuo Kishi e Ronaldo Fiorilo dos Santos.

Agradeço ainda à Isabel e à Beth pela ajuda provida com os vários documentos entregues durante este mestrado.

Por fim, agradeço à CAPES pelo apoio financeiro.

Resumo

Dentre a variedade de problemas de otimização existentes, aqueles que envolvem sequências destacam-se por sua aplicabilidade em vários campos de pesquisa. Nesta dissertação apresentamos um estudo detalhado de um novo problema de otimização combinatória envolvendo sequências, denominado Problema do Alinhamento de Segmentos (PASG). Esse estudo envolve a definição formal desse problema e a descrição de um algoritmo eficiente, baseado na técnica de programação dinâmica, que o resolve. Ademais, formalizamos uma versão múltipla do PASG, denominada Problema do Alinhamento de Segmentos Múltiplo (PASGM). Para essa versão do problema, nós provamos que ela é NP-Completa e que é muito improvável existir um algoritmo de aproximação com uma boa razão para ela. Com base nesse resultado, propomos três heurísticas para tratar o PASGM e as avaliamos experimentalmente através de testes artificiais. Por fim, as implementações das soluções propostas neste trabalho foram aplicadas na tarefa de identificação de genes. A aplicabilidade dos nossos programas nessa tarefa foi atestada através dos bons resultados obtidos por eles em um conjunto de instâncias de testes reais.

Palavras-chave: Otimização Combinatória, Problema do Alinhamento de Segmentos, Programação Dinâmica, NP-Completa, Algoritmo de Aproximação, Tarefa de Identificação de Genes.

Abstract

Among the variety of existing combinatorial optimization problems, those involving sequences stand out for their applicability in several research fields. In this work we present a detailed study of a new combinatorial optimization problem involving sequences, called Segment Alignment Problem (SAP). This study includes the formal definition of this problem and the description of an efficient dynamic programming algorithm to solve it. Furthermore, we present the formal definition of a multiple version of the SAP, called Multiple Segment Alignment Problem (MSAP). For this version of the problem, we prove its NP-Completeness and that it is very unlikely to exist an approximation algorithm for it with a good factor. With these results in mind, we propose three heuristics for MSAP and we evaluate them experimentally through artificial tests. Finally, we apply the implementations of the solutions proposed in this work in the gene prediction task. The applicability of our programs in this task was attested by the good results achieved by all of them in a set of real test instances.
Keywords: Combinatorial Optimization, Segment Alignment Problem, Dynamic Programming, NP-Completeness, Approximation Algorithm, Gene Prediction Task.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

As diversas áreas do conhecimento, sobretudo a Biologia, Física e Química, incluem uma série de problemas que podem ser modelados computacionalmente e, assim, resolvidos através de conceitos, ferramentas e técnicas da Ciência da Computação. Muitos desses problemas podem ser abordados como problemas de otimização combinatória. Como tal, eles necessitam de algoritmos adequados para serem resolvidos eficientemente.

Com o surgimento e desenvolvimento de algumas subáreas específicas da Ciência da Computação, em especial a Biologia Computacional, os problemas de otimização que envolvem sequências vêm ganhando considerável importância prática. Dentre esses problemas destaca-se o Problema do Alinhamento de Sequências e suas variantes (local e semi-global), que servem como base para vários outros problemas que envolvem essas estruturas.

O objetivo principal deste trabalho é o estudo detalhado de um problema de otimização combinatória (envolvendo sequências) denominado Problema do Alinhamento de Segmentos (PASG) que, dadas duas sequências s_1 e s_2, um conjunto ordenado B de segmentos de s_1, um conjunto ordenado C de segmentos de s_2 e uma função de pontuação ω, determina duas cadeias de segmentos, uma para cada conjunto ordenado de segmentos da entrada, cujas sequências resultantes das suas concatenações possuem a maior similaridade possível. Esse problema pode ser visto como uma variante do Problema do Alinhamento *Spliced* (*Spliced Alignment Problem*, do inglês) [17] e como uma generalização do Problema do Encadeamento Bidimensional (*Two-dimensional Chain Problem*, do inglês) [19]. Até onde sabemos, não existem trabalhos na literatura que abordam o PASG formalmente, o que justifica um estudo detalhado de suas propriedades na busca por soluções eficientes para ele.

No decorrer deste trabalho, apresentamos a definição formal do PASG e um algoritmo eficiente baseado na técnica de programação dinâmica que o soluciona. Mais adiante, aprofundamos o estudo sobre o PASG formulando uma versão múltipla para ele, denominada Problema do Alinhamento de Segmentos Múltiplo (PASGM). A principal diferença entre o PASG e o PASGM é que neste último são dadas como entrada $n > 2$ sequências $s_1, s_2, ..., s_n$, n conjuntos ordenados de segmentos $B_1, B_2, ...,\ B_n$ e uma função de pontuação ω e o objetivo é determinar n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$ tal que a soma par a par das similaridades globais das sequências resultantes das concatenações destas n cadeias é máxima. Para tratar o PASGM, primeiramente determinamos a complexidade desse problema, provando que ele é NP-Completo e que é muito improvável existir um algoritmo de aproximação com uma boa razão para ele. Com base no estudo da complexidade do PASGM, propomos três heurísticas para ele e as avaliamos experimentalmente através de testes artificiais.

Por fim, também como parte dos objetivos deste trabalho, implementamos as soluções propostas e aplicamos os programas desenvolvidos na tarefa de identificação de genes, tarefa essa de cunho prático inserida na área de Biologia Molecular. Sucintamente, nessa tarefa estamos interessados em determinar as regiões que compõem um ou mais genes de uma sequência de DNA. Com isso em mente, modelamos a tarefa de identificação de genes através do PASG, com o objetivo de localizar os éxons que constituem os genes de uma sequência de DNA através da comparação dessa sequência com outra evolutivamente relacionada. Com o PASGM, é possível comparar uma sequência de

DNA com várias outras evolutivamente relacionadas, permitindo também sua aplicação na tarefa de identificação de genes na tentativa de obter melhores resultados. Para atestar a validade da aplicação de nossos programas nessa tarefa, avaliamos a acurácia deles em instâncias de testes reais, onde as principais medidas de avaliação foram as medidas de especificidade e sensibilidade propostas por Burset e Guigó em [6].

1.1 Contribuições

As principais contribuições deste trabalho são:

- Formulação de dois novos problemas de otimização combinatória: o Problema do Alinhamento de Segmentos (PASG), e sua versão múltipla, o Problema do Alinhamento de Segmentos Múltiplo (PASGM);
- Desenvolvimento de um algoritmo eficiente baseado na técnica de programação dinâmica que soluciona o PASG;
- Demonstração da NP-Completude do PASGM;
- Demonstração que o PASGM é um problema difícil de aproximar;
- Proposta de três heurísticas para o PASGM;
- Desenvolvimento de quatro ferramentas para a tarefa de identificação de genes, baseadas nas soluções propostas para os problemas aqui formulados;
- Avaliação das ferramentas desenvolvidas utilizando instâncias de testes reais, atestando sua aplicabilidade na tarefa de identificação de genes.

1.2 Organização do texto

Esta dissertação está dividida em sete outros capítulos, além desta introdução. O Capítulo 2 descreve os conceitos básicos de Ciência da Computação essenciais para a compreensão dos problemas abordados neste trabalho e das soluções desenvolvidas para eles. O Capítulo 3 apresenta as definições formais do PASG e do PASGM e alguns trabalhos da literatura relacionados a esses problemas. No Capítulo 4, é detalhado passo a passo a metodologia que seguimos para desenvolver um algoritmo eficiente baseado na técnica de programação dinâmica que soluciona o PASG. No Capítulo 5, discutimos sobre algumas propriedades do PASGM, provando que ele é um problema NP-Completo e que é muito improvável existir um algoritmo de aproximação com uma boa razão para ele. No Capítulo 6, detalhamos três heurísticas desenvolvidas para o PASGM e as avaliamos experimentalmente através de testes artificiais. O Capítulo 7 introduz a tarefa de identificação de genes e os métodos disponíveis na literatura para abordá-la. Também nesse capítulo, discutimos como os programas desenvolvidos podem ser aplicados na tarefa mencionada e avaliamos a exatidão de nossas ferramentas com instâncias de testes reais. Finalmente, o Capítulo 8 apresenta as considerações finais e o que ainda pode ser explorado em trabalhos futuros.

Capítulo 2

Conceitos básicos

Nesta seção são apresentados os conceitos básicos de Ciência da Computação necessários à compreensão dos problemas abordados neste trabalho e das soluções que propomos para eles. Este capítulo foi escrito com base em [7, 11, 14, 19, 22, 28, 30, 38, 42].

2.1 Definições básicas

Um **alfabeto** Σ é definido como um conjunto finito de caracteres. Um exemplo de alfabeto é o conjunto $\{0,1,2,3,4,5,6,7,8,9\}$, que inclui os caracteres do sistema decimal de numeração. Uma **sequência** s, construída sobre um alfabeto Σ qualquer, é uma concatenação finita de caracteres pertencentes a Σ. Por exemplo, $u = CACHORRO$ e $v = GATO$ são sequências construídas sobre o alfabeto $\{A, ..., Z\}$. Dado um alfabeto Σ qualquer, Σ^* é o conjunto de todas as sequências finitas que podem ser construídas sobre Σ. Por exemplo, se $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$, então Σ^* é o conjunto dos números naturais. O **tamanho** de uma sequência s, denotado por $|s|$, corresponde à quantidade de caracteres de s. Voltando às sequências do nosso exemplo, temos que $|u| = 8$ e $|v| = 4$. Uma sequência cujo tamanho é zero é chamada de **sequência vazia** e é denotada por ϵ.

Dadas uma sequência s qualquer e um inteiro i tal que $1 \leq i \leq |s|$, $s[i]$ denota o caractere na posição i de s. No nosso exemplo, $u[4] = H$, $u[5] = u[8] = O$ e $v[1] = G$. Uma **subsequência** de uma sequência s é uma outra sequência que consiste de uma cópia de s com alguns (ou possivelmente todos ou nenhum) de seus caracteres removidos. Por exemplo, $u' = CARRO$, $u'' = COR$ e $u''' = \epsilon$ são todas subsequências da sequência u tomada como exemplo. Um **segmento** ou **subcadeia** de uma sequência s é um trecho com zero ou mais caracteres consecutivos de s. Denotamos um segmento que começa na posição i e estende-se até a posição j de s por $s[i..j]$. Voltando ao nosso exemplo, $u[3..6] = CHOR$ é um segmento da sequência u. Um **segmento vazio** de uma sequência s é um segmento $s[i..j]$ tal que $i > j$ e corresponde à sequência ϵ. Um **prefixo** de uma sequência s é um segmento $s[1..j]$ tal que $1 \leq j \leq |s|$. Um **prefixo vazio** de s é um segmento $s[1..j]$ tal que $j < 1$ e corresponde à sequência ϵ. Similarmente, um **sufixo** de uma sequência s é um segmento $s[i..|s|]$ tal que $1 \leq i \leq |s|$. Um **sufixo vazio** de s é um segmento $s[i..|s|]$ tal que $i > |s|$ e corresponde à sequência ϵ.

Seja $s_1 = s[i..j]$ um segmento qualquer de uma sequência s. Dada uma posição k de s_1, $1 \leq k \leq j-i+1$, $s_1[k]$ denota o k-ésimo caractere de s_1, ou seja, $s_1[k] = s[i+k-1]$. Denotamos por $first(s_1)$ a posição do primeiro caractere de s_1 em s, ou seja, $s_1[1] = s[first(s_1)] = s[i]$. Analogamente, denotamos por $last(s_1)$ a posição do último caractere de s_1 em s, ou seja, $s_1[|s_1|] = s[last(s_1)] = s[j]$.

Dadas duas sequências s e t quaisquer, a **concatenação** de s com t é uma operação que gera uma terceira sequência, denotada por $s \bullet t$, que consiste dos caracteres de s seguidos pelos caracteres de t. Formalmente, $(s \bullet t)[i] = \begin{cases} s[i], & \text{se } i \leq |s| \\ t[i-|s|], & \text{se } i > |s| \end{cases}$, para $1 \leq i \leq |s| + |t|$. Disso, observe que $|s \bullet t| = |s| + |t|$. Considerando as sequências do nosso exemplo, temos que $u \bullet v = CACHORROGATO$.

Seja $B = \{s_1, s_2, ..., s_p\}$ um conjunto de segmentos qualquer de uma sequência s. Dizemos que B é um **conjunto ordenado de segmentos** se: 1) $first(s_i) < first(s_{i+1})$ ou 2) $first(s_i) =$

$first(s_{i+1})$ e $last(s_i) < last(s_{i+1})$, para $1 \leq i \leq p-1$. Informalmente, B é um conjunto ordenado de segmentos de uma sequência s se os segmentos de B estão ordenados de forma não-decrescente pelas suas primeiras posições na sequência. Os casos de empate são resolvidos por uma ordenação não-decrescente pela última posição dos segmentos em s.

Seja s uma sequência qualquer. Dizemos que um segmento $s_1 = s[i..j]$ de s se sobrepõe a outro segmento $s_2 = s[k..l]$ de s se $i \leq k \leq j$ ou $i \leq l \leq j$ ou $k \leq i \leq l$ ou $k \leq j \leq l$. Neste caso, dizemos que s_1 e s_2 são **segmentos sobrepostos**. Diferentemente, dizemos que um segmento $s_1 = s[i..j]$ de s **precede** outro segmento $s_2 = s[k..l]$ de s se $j < k$, ou seja, se s_1 termina antes de s_2 começar. Esta relação é denotada por $s_1 \prec s_2$. Seja B um conjunto ordenado de segmentos de s. Um subconjunto $\Gamma_B = \{s_1, s_2, ..., s_p\}$ de B é dito uma **cadeia de segmentos** se $s_1 \prec s_2 \prec ... \prec s_p$. Nós denotamos por $\mathbf{before(s_i)}$, onde s_i é um segmento de um conjunto B de segmentos, o subconjunto B' de B tal que $s' \prec s_i$, para todo segmento $s' \in B'$. Denotamos ainda por $\mathbf{\Delta(s_i)}$ o conjunto que inclui todas as possíveis cadeias de segmentos construídas a partir dos elementos de $before(s_i)$, incluindo a cadeia vazia. Por fim, denotamos a concatenação dos segmentos de uma cadeia de segmentos Γ_B por $\Gamma_B^\bullet$. Ou seja, $\Gamma_B^\bullet = s_1 \bullet s_2 \bullet ... \bullet s_p$. No decorrer deste texto, para simplificar sua escrita, chamaremos a sequência resultante da concatenação dos segmentos de uma cadeia de segmentos de **sequência correspondente** àquela cadeia de segmentos.

2.2 Complexidade computacional

Um **algoritmo** é uma sequência finita de passos não ambíguos executados por um certo agente de computação. Em outras palavras, um algoritmo é um processo que transforma a descrição de um valor ou de um conjunto de valores, conhecida como **entrada**, em um valor ou um conjunto de valores conhecidos como **saída**. Dizemos que um algoritmo A **resolve** um problema p se, ao receber uma instância arbitrária I de p como entrada, A devolve como saída uma solução para I ou informa que I não tem solução. O algoritmo A pode consumir uma quantidade de tempo diferente para resolver cada instância I de p. Seja $T_A(I)$ uma função que relaciona a instância I e a quantidade de tempo que A consome para resolver I. A relação entre $T_A(I)$ e o tamanho de I nos fornece uma medida da eficiência do algoritmo [14].

Um problema normalmente possui várias instâncias diferentes com o mesmo tamanho. Com base nesse fato, as medidas de eficiência geralmente consideram as instâncias de pior caso de um problema (as que consomem maior quantidade de tempo para serem resolvidas). Dados um algoritmo A para um problema p e um número natural n qualquer, seja $T_A^*(n)$ o máximo de $T_A(I)$ para todas as instâncias I de tamanho n. Nesse caso, dizemos que T_A^* mede o consumo de tempo de A no pior caso [14]. São as instâncias do pior caso que consideramos toda vez que fazemos medições da eficiência de um algoritmo.

Para facilitar a análise de um algoritmo, é desejável calcular seu consumo de tempo de uma maneira que não dependa da linguagem de programação, nem dos detalhes de implementação ou do computador empregado. Isso nos permite utilizar um modo grosseiro de estimar o consumo de tempo de um algoritmo. Uma outra justificativa para isso é o fato de que, em geral, a precisão extra obtida através de uma análise detalhada não é relevante. Isso ocorre porque quando observamos entradas com tamanhos suficientemente grandes, apenas a ordem de crescimento do tempo de execução se torna pertinente. Quando consideramos este modo grosseiro de medição, estamos estudando a eficiência assintótica dos algoritmos [11, 14].

Para expressar a eficiência assintótica de um algoritmo, podemos utilizar as três notações seguintes: O[1], Ω e Θ [11, 22]. A notação O estabelece um limite assintótico superior do tempo de execução de um algoritmo. Dadas duas funções $f(n)$ e $g(n)$, dizemos que $f(n) = O(g(n))$ se existe uma constante $c_1 > 0$ e um $n_1 \geq 0$ tal que para todo $n \geq n_1$, $f(n) \leq c_1 * g(n)$ [22]. Neste caso, nós dizemos que a função $g(n)$ é um limite assintótico superior da função $f(n)$. A notação O geralmente é utilizada para expressar a complexidade de tempo de um algoritmo no pior caso. Quando dizemos

[1] Comumente conhecida como notação *Big O*, do inglês.

que a complexidade de tempo de um algoritmo é $O(g(n))$, queremos dizer que o tempo de execução do algoritmo é, no máximo, uma constante multiplicada por $g(n)$, para um n suficientemente grande, independentemente de qual entrada específica com tamanho n foi escolhida [11].

Sobre a notação Ω, ela estabelece um limite assintótico inferior do tempo de execução de um algoritmo. Dadas duas funções $f(n)$ e $g(n)$, dizemos que $f(n) = \Omega(g(n))$ se existe uma constante $c_2 > 0$ e um $n_2 \geq 0$ tal que para todo $n \geq n_2$, $f(n) \geq c_2 * g(n)$ [22]. Assim, nós dizemos que a função $g(n)$ é um limite assintótico inferior da função $f(n)$. A notação Ω geralmente é utilizada para expressar o consumo de tempo de um algoritmo no melhor caso. Quando dizemos que a complexidade de tempo de um algoritmo é $\Omega(g(n))$, queremos dizer que o tempo de execução do algoritmo é, no mínimo, uma constante multiplicada por $g(n)$, para um n suficientemente grande, independentemente de qual entrada específica com tamanho n foi escolhida [11].

Por fim, a notação Θ estabelece um limite assintótico justo. Se pudermos provar que o tempo de execução $T_A(n)$ de um algoritmo A é $O(f(n))$ e $\Omega(f(n))$, então dizemos que $T_A(n) = \Theta(f(n))$. Nesse caso, $T_A(n)$ cresce como a função $f(n)$, dentro de um fator constante. Formalmente, dizemos que $f(n) = \Theta(g(n))$ se existem duas constantes $c_1 > 0$ e $c_2 > 0$ e um $n_3 \geq 0$ tal que para todo $n \geq n_3$, $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ [11].

Quando analisamos algoritmos, é amplamente aceito classificar um algoritmo como eficiente se sua complexidade de tempo é $O(n^k)$ no pior caso, onde n denota o tamanho da entrada e k é uma constante. Estes algoritmos são chamados de **algoritmos de tempo polinomial**, ou simplesmente **algoritmos polinomiais**. Um algoritmo é dito **ineficiente** se ele não admite um limite assintótico superior polinomial. Um algoritmo é **exponencial** se consome tempo $\Omega(a^n)$ no pior caso, onde a é um número real maior que 1 e n é o tamanho da entrada [14]. Os algoritmo exponenciais são os exemplos mais comuns de algoritmos ineficientes.

2.3 NP-Completude

É natural questionarmos se todos os problemas podem ser resolvidos eficientemente, ou seja, através de um algoritmo eficiente. A resposta é não. O Problema da Parada de Turing, por exemplo, não pode ser resolvido por nenhum computador, não importa quanto tempo lhe seja fornecido [11]. Nesse contexto, existe uma classe interessante de problemas, chamada **NP-Completa**, cujo *status* dos problemas nela incluídos é desconhecido, ou seja, ainda não foi descoberto nenhum algoritmo de tempo polinomial para resolver um ou mais problemas NP-Completos (apesar de eles admitirem soluções ineficientes), mas também ninguém ainda foi capaz de provar que esse algoritmo não existe.

A teoria da NP-Completude objetiva caracterizar os problemas pertencentes à classe NP-Completa. Ela define três classes de problemas: **P**, **NP** e **NPC**. Informalmente, a classe P consiste dos problemas que podem ser resolvidos por um algoritmo polinomial. A classe NP consiste dos problemas que admitem algoritmos eficientes que verificam a validade de uma solução[2]. Um problema pertence à classe NPC (a classe dos problemas NP-Completos) se ele pertence à NP e é tão "difícil" quanto qualquer outro problema em NP [11].

Uma forma de provar que um problema p é NP-Completo consiste em seguir os dois passos seguintes [11, 22]:

1. provar que $p \in NP$;
2. fazer uma redução polinomial de um problema da classe NP-Completo a ele.

Estes dois passos serão detalhados posteriormente, mas antes discutiremos um pouco sobre problemas de otimização e problemas de decisão.

[2]Sucintamente, isto significa que podemos desenvolver um algoritmo polinomial que, dadas uma instância do problema e uma solução candidata para esta instância, consegue determinar se a solução candidata é uma solução real. Este conceito de algoritmos verificadores será formalizado adiante.

2.3.1 Problemas de otimização e de decisão

Para uma melhor compreensão das etapas da prova da NP-Completude de um problema, é necessário distinguir dois tipos de problemas: problemas de otimização e problemas de decisão.

Nos **problemas de otimização**, temos um conjunto S de possíveis soluções de uma instância I e uma função objetivo f que associa um valor a cada uma das soluções em S. Para resolver um problema de otimização, devemos encontrar uma solução $s \in S$ que minimize ou maximize o valor de f. Dizemos que a solução s encontrada é uma **solução com valor ótimo** (máximo ou mínimo), ou simplesmente **solução ótima**. É importante ressaltar que podem existir várias soluções ótimas para uma instância qualquer de um problema. Como exemplo das definições acima, considere o problema abaixo:

Problema da Subsequência Comum Mais Longa Múltipla (LCSM): dadas $n > 2$ sequências $s_1, s_2, ..., s_n$ construídas sobre um mesmo alfabeto finito Σ, determinar a maior sequência $t \in \Sigma^*$, tal que t é subsequência de $s_1, s_2, ...$ e s_n.

Seja $I = \{ACARO, MACABRO, CANARIO, ABACATEIRO, CARRO\}$ uma instância do LCSM, considerando $\Sigma = \{A, ..., Z\}$. Definindo o valor de uma solução t como $|t|$, a sequência $t_1 = RO$ seria uma solução de valor 2. Já a solução $t_2 = CARO$ é ótima e possui valor 4. Para esta instância em particular, não existe outra solução ótima.

Um problema é de **decisão** se cada uma de suas instâncias admite uma e apenas uma de duas respostas: "SIM" e "NÃO". Dizemos que uma instância é **positiva** se tem solução SIM e **negativa** caso contrário, e que um algoritmo resolve um determinado problema de decisão se, ao receber uma instância arbitrária do problema, devolve SIM se a instância for positiva e NÃO se a instância for negativa [22].

A teoria da NP-Completude restringe sua atenção a problemas de decisão. Isso significa que quando queremos provar que um problema p é NP-Completo, p necessariamente é um problema de decisão e não de otimização. Contudo, existe uma relação conveniente entre problemas de otimização e problemas de decisão. Mais especificamente, podemos formular um determinado problema de otimização como um problema de decisão estabelecendo um limite para o valor a ser otimizado. Desta forma, para formular uma versão de decisão do LCSM, podemos estabelecer um limite inferior no tamanho da sequência t:

Problema da Subsequência Comum Mais Longa Múltipla - versão de decisão (LCSMD): dadas $n > 2$ sequências $s_1, s_2, ..., s_n$ construídas sobre um mesmo alfabeto finito Σ, e um inteiro positivo k, existe uma sequência $t \in \Sigma^*$, com $|t| \geq k$, tal que t é subsequência de $s_1, s_2, ...$ e s_n?

Essa relação entre problemas de otimização e de decisão auxilia no processo de mostrar que um problema de otimização é "difícil" ou, mais formalmente, muito improvável de possuir uma solução eficiente. Isso ocorre porque o problema de decisão é de certo modo "mais fácil" ou, pelo menos, "não mais difícil" que a versão de otimização [11]. Como um exemplo específico, podemos resolver o LCSMD resolvendo o LCSM e depois comparando o tamanho da subsequência comum mais longa, $|t|$, com o valor limite k.

Finalizando, se um problema de otimização é fácil, o problema de decisão relacionado também é fácil. Em contraste, se pudermos fornecer evidências de que um problema de decisão é difícil, também estamos fornecendo evidências de que o problema de otimização relacionado é difícil. Desse modo, embora restrinja a atenção a problemas de decisão, a teoria da NP-Completude tem implicações diretas sobre problemas de otimização [11].

2.3.2 Provando que um problema é NP-Completo

A primeira etapa da prova da NP-Completude de um problema de decisão p consiste em mostrar que $p \in NP$. Para isso, é necessário definir os conceitos de certificados e algoritmos verificadores.

Dados um problema de decisão p e uma instância arbitrária I de p, um **certificado** é uma cadeia de caracteres que pode conter evidências de que I é uma instância positiva de p. A ideia é que o certificado seja uma "prova" de que I é uma instância positiva de p. Um **algoritmo verificador** para um problema de decisão é um algoritmo que recebe dois argumentos, a saber, uma instância arbitrária I do problema e um certificado S e responde SIM, caso S prove que I é uma instância positiva, ou NÃO, caso contrário. Quando a resposta é SIM, dizemos que o algoritmo verificador aceitou o certificado [22].

Um algoritmo verificador polinomial para um determinado problema de decisão satisfaz as seguintes condições, retiradas de [22]:

1. para cada instância positiva do problema, existe um certificado que o algoritmo verificador aceita em uma quantidade de tempo limitada por uma função polinomial no tamanho da instância;

2. para cada instância negativa do problema, não existe certificado que o algoritmo verificador aceite.

Exemplificando, um algoritmo verificador polinomial para o LCSMD é um algoritmo simples que recebe n sequências $s_1, s_2, ..., s_n$, um inteiro positivo k (instância do problema) e uma sequência t (certificado), e devolve SIM se $|t| \geq k$ e se t é subsequência de $s_1, s_2, ...$ e s_n. Caso contrário, o algoritmo devolve NÃO.

A segunda etapa da prova da NP-Completude de um problema de decisão p consiste em achar um outro problema de decisão q que seja NP-Completo e provar que q é redutível polinomialmente a p. Dizemos que um problema q é **redutível polinomialmente** a um problema p se, supondo que temos um algoritmo alg_p que resolve p em tempo polinomial, conseguimos desenvolver um algoritmo polinomial alg_q que resolve q utilizando um número polinomial de chamadas a alg_p [22]. O processo de mostrar que q é redutível polinomialmente a p é chamado de **redução polinomial** de q a p.

A principal estratégia para desenvolver o algoritmo alg_q consiste em transformar uma instância qualquer I_q de q em uma instância I_p de p tal que I_q é positiva se e somente se I_p é positiva [22]. Observe que essa transformação deve ser eficiente, ou seja, consumir tempo polinomial. Em particular, essa estratégia requer apenas uma única chamada ao algoritmo alg_p. Assim, a resposta obtida pela invocação $alg_p(I_p)$ é também a resposta para a instância I_q de q.

2.3.3 Heurísticas e aproximações

A motivação prática para se provar a NP-Completude de um problema está na necessidade de resolvê-lo e, consequentemente, de determinar quais métodos e técnicas computacionais devem ser utilizados para abordá-lo. Se demonstrarmos que um problema é NP-Completo, temos uma indicação forte de que ele não pode ser resolvido por um algoritmo polinomial. Nesse caso, ao invés de investir tempo e esforço buscando um algoritmo eficiente para o problema, deve-se considerar o desenvolvimento de heurísticas e/ou de algoritmos de aproximação para ele. Essas abordagens foram concebidas para o tratamento dos problemas NP-Completos e, apesar de não determinarem necessariamente uma solução ótima, podem apresentar um bom comportamento na prática.

Um **algoritmo de aproximação** é um algoritmo que garante encontrar, em tempo polinomial, uma solução para um problema de otimização combinatória cujo valor possui uma relação pré-estabelecida com o valor da solução ótima [7]. Para explicar melhor o conceito de algoritmos de aproximação, considere um problema de otimização p qualquer. Dada uma instância arbitrária I de p, denotaremos por $S(I)$ o conjunto das possíveis soluções de I, por f uma função objetivo que associa um valor a cada uma das soluções $s \in S(I)$ e por $s^\star \in S(I)$ uma solução ótima para I. Assuma que o valor de uma solução $s \in S(I)$ é sempre maior ou igual a zero ($f(s) \geq 0$). Seja A um algoritmo que, ao receber uma instância arbitrária I de p, sempre devolve uma solução $s_A \in S(I)$. Se p é um problema de minimização e $f(s_A) \leq \alpha * f(s^\star)$, com $\alpha \geq 1$, dizemos que A é uma **α-aproximação** para p e que α é a **razão de aproximação** do algoritmo [7]. Se p é um problema de maximização, a relação entre os valores das soluções encontradas por uma α-aproximação A e

os valores das soluções ótimas é redefinida para $f(s_A) \geq \alpha * f(s^\star)$, com $0 < \alpha \leq 1$. Note que se um algoritmo de aproximação A possuir uma razão de aproximação igual a 1 para um problema de otimização, então dizemos que A é um **algoritmo exato** para o problema [7].

Quando pensamos em desenvolver um algoritmo de aproximação para um problema de otimização NP-Completo, indagamos sobre a existência ou não desse algoritmo e sobre a melhor razão de aproximação para o problema. Para muitos problemas, existem algoritmos de aproximação de razões boas, ou seja, razões constantes próximas a 1. Para outros problemas, os melhores algoritmos de aproximação conhecidos possuem razões que crescem como funções no tamanho da entrada [11]. Com base nisso, podemos classificar os algoritmos de aproximação em três classes de dificuldade, dependendo de suas razões de aproximação. Na **classe de razão constante**, temos os algoritmos de razão $\alpha_1 > 1$, onde α_1 é uma constante, se o problema é de minimização, e de razão $0 < \alpha_2 < 1$, onde α_2 é uma constante, se o problema é de maximização. Na **classe de razão logarítmica**, temos os algoritmos de aproximação de razão $f(n) = \Omega(log(n))$ se o problema é de minimização e de razão $f'(n) = O(\frac{1}{log(n)})$ se o problema é de maximização, onde n é o tamanho da entrada. Por fim, na **classe de razão polinomial**, temos os algoritmos de aproximação de razão n^δ se o problema é de minimização e de razão $\frac{1}{n^\delta}$ se o problema é de maximização, onde $\delta > 0$ é uma constante [42].

Existem problemas de otimização que não temos sequer algoritmos de aproximação na classe de razão polinomial, a não ser que P = NP. Nesses casos, dizemos que o problema é difícil de aproximar que, em outras palavras, significa dizer que é muito improvável existir uma aproximação com uma boa razão para ele. Nessas situações, é comum desconsiderar o desenvolvimento de algoritmos de aproximações e atacar o problema através de heurísticas.

Heurísticas são critérios, métodos e princípios utilizados na escolha de um caminho, dentre vários, que supõe-se ser o mais adequado na busca por algum objetivo [30]. Elas geralmente são desenvolvidas com base em informações obtidas pelo estudo de peculiaridades do problema e são empregadas nos casos em que os algoritmos conhecidos para ele necessitam de um tempo proibitivo e quando os algoritmos de aproximação não são apropriados. Diferentemente desses últimos, as heurísticas não apresentam limites formais em relação ao tempo de execução e/ou à qualidade da solução, mas sempre devolvem soluções válidas para um problema.

2.4 Programação dinâmica

A **programação dinâmica** é uma abordagem para solução de problemas baseada na combinação ou extensão de soluções de subproblemas. Essa abordagem assemelha-se, em alguns aspectos, ao método de divisão e conquista. A diferença principal entre elas está no fato da programação dinâmica ser mais eficiente que a divisão e conquista quando os subproblemas não são independentes, isto é, quando eles compartilham subsubproblemas. Nesses casos, o método de divisão e conquista realiza mais cálculos que o necessário, pois resolve os mesmos subsubproblemas repetidamente, tornando a complexidade de tempo do algoritmo recursivo geralmente exponencial. Diferentemente, a programação dinâmica resolve cada subsubproblema apenas uma vez, gravando sua resposta em uma tabela, evitando assim o trabalho de recalculá-la toda vez que o mesmo subsubproblema for encontrado [11]. A eficiência desta técnica se torna mais perceptível quando a resolução de um subproblema origina uma árvore de recursão de tamanho considerável, com vários subsubproblemas repetidos.

Assim como a grande maioria das estratégias para solução de problemas, a programação dinâmica é aplicada a problemas de otimização. Existem duas características fundamentais que um problema de otimização deve ter para que este método seja aplicável: subestrutura ótima e subproblemas sobrepostos [11].

Um problema possui **subestrutura ótima** se uma solução ótima para o problema contém, em seu interior, soluções ótimas para subproblemas [11]. Como no método de divisão e conquista, na programação dinâmica uma solução ótima para o problema é construída combinando soluções ótimas de subproblemas. Para determinar a subestrutura ótima de um problema, geralmente seguimos um

padrão comum, descrito por Cormen *et al.* em [11] e que consiste das seguintes quatro etapas:

1. mostre que uma solução para o problema consiste em fazer uma escolha; essa escolha exige que um ou mais subproblemas sejam resolvidos;
2. suponha que, para um dado problema, você tem a escolha que conduz a uma solução ótima;
3. dada essa escolha, determine quais subproblemas resultam dela e como caracterizar melhor o espaço de subproblemas resultante;
4. mostre que as soluções para os subproblemas usados dentro da solução ótima para o problema devem elas próprias ser ótimas.

A programação dinâmica geralmente se utiliza da subestrutura ótima para encontrar uma solução ótima do problema seguindo um esquema de "baixo para cima" (comumente conhecido como *bottom-up*, do inglês), onde os subproblemas são resolvidos primeiramente, suas respectivas soluções armazenadas em uma tabela, e então são combinadas para construir uma solução ótima para o problema. Contudo, existe uma variação deste método que oferece a mesma eficiência mantendo uma estratégia de construção de soluções de cima para baixo (*top-down*). Esta variação utiliza a técnica de memoização. Sucintamente, a memoização mantém uma entrada em uma tabela para a solução de cada subproblema. Inicialmente, todas as entradas da tabela contêm um valor especial para indicar que ela ainda tem que ser preenchida e que o subproblema relacionado ainda não foi resolvido. Quando um subproblema é encontrado pela primeira vez durante a execução do algoritmo, sua solução é calculada e depois armazenada em uma posição específica da tabela. Em cada momento subsequente que esse mesmo subproblema é encontrado, o valor armazenado é pesquisado e retornado [11].

A segunda característica fundamental que um problema de otimização deve ter para que o método de programação dinâmica seja aplicável é apresentar **subproblemas sobrepostos**. Um problema apresenta essa característica se um algoritmo recursivo para ele resolve os mesmos subproblemas repetidas vezes, ao invés de sempre gerar subproblemas novos. Geralmente, nessas situações, o espaço de subproblemas (a quantidade de subproblemas distintos gerados durante a execução do algoritmo) é limitado por um polinômio no tamanho da entrada [11]. Essa característica diferencia claramente onde aplicar os algoritmos de divisão e conquista e os algoritmos de programação dinâmica. Um problema para o qual a abordagem de dividir e conquistar é satisfatória quase sempre gera subproblemas novos em cada passo da recursão. Em contraste, os algoritmos de programação dinâmica costumam tirar proveito de subproblemas sobrepostos, resolvendo cada subproblema uma vez e depois armazenando a solução para pesquisas posteriores.

Segundo Cormen *et al.* em [11], o desenvolvimento de um algoritmo de programação dinâmica pode ser desmembrado em uma sequência de quatro etapas ou passos:

1. caracterizar a estrutura de uma solução ótima;
2. definir recursivamente o valor de uma solução ótima;
3. calcular o valor de uma solução ótima, geralmente em um processo de baixo para cima (*bottom-up*);
4. construir uma solução ótima a partir de soluções já calculadas.

Sucintamente, na primeira etapa é identificada a subestrutura ótima do problema, e então a utilizamos para construir uma solução ótima para o problema a partir de soluções ótimas de subproblemas.

A segunda etapa envolve definir uma solução recursiva, na forma de uma recorrência ou de um algoritmo, que resolve o problema. O algoritmo recursivo definido nesta etapa normalmente apresenta um tempo de execução exponencial em relação ao tamanho da entrada, pois ele resolve os mesmos subproblemas repetidas vezes.

O principal objetivo da terceira etapa é melhorar o tempo de execução do algoritmo recursivo. Para fazer isso, aplicamos a característica de subproblemas sobrepostos, desenvolvendo um algoritmo polinomial que resolve cada subproblema apenas na primeira vez que o encontra e então guarda o valor da solução em uma tabela. Em cada momento subsequente que o subproblema é encontrado, o valor armazenado é pesquisado e retornado. Nessa etapa pode-se adotar tanto a estratégia de baixo para cima (*bottom-up*) como a estratégia de cima para baixo (*top-down*), aquela que for mais apropriada.

A última etapa, que pode ser omitida quanto estamos interessados apenas no valor de uma solução ótima, consiste em construir a sequência de escolhas realizadas para obter a solução ótima. Isso é feito geralmente a partir das informações armazenadas na tabela.

2.5 Alinhamento de sequências

Nesta seção é apresentado um dos métodos mais utilizados para comparar sequências, operação básica em diversas áreas da Ciência da Computação. Essa operação é fundamental, por exemplo, na Biologia Computacional, servindo de base para outras tarefas mais complexas como a da identificação de genes por comparação de DNAs [26, 28, 39] e montagem de *reads* provenientes de sequenciadores de larga escala [31, 45]. Aplicações em outras áreas incluem a tarefa de buscas textuais e métodos de correção ortográfica.

Genericamente, comparamos sequências para descobrir quais trechos delas são parecidos e quais trechos são diferentes. Uma maneira de comparar sequências é através do alinhamento delas. O alinhamento de duas sequências pode ser definido como a seguir:

Alinhamento de duas sequências: seja Σ um alfabeto qualquer tal que $'-' \notin \Sigma$, onde o caractere $'-'$ denota um espaço. Chamamos de $\overline{\Sigma}$ o alfabeto contendo os símbolos de Σ mais o símbolo $'-'$. Ou seja, $\overline{\Sigma} = \Sigma \cup \{'-'\}$. Para toda sequência $s \in \overline{\Sigma}^*$, seja $s_{|\Sigma}$ a sequência s restrita ao alfabeto Σ (ou seja, a sequência resultante da remoção de todos os espaços presentes em s). Um alinhamento A de duas sequências s_1 e $s_2 \in \Sigma^*$ é um par $(\overline{s_1}, \overline{s_2})$, com $\overline{s_1}$ e $\overline{s_2} \in \overline{\Sigma}^*$ tal que:

1. $|\overline{s_1}| = |\overline{s_2}|$;
2. $\overline{s_1}_{|\Sigma} = s_1$ e $\overline{s_2}_{|\Sigma} = s_2$;
3. não existe i tal que $\overline{s_1}[i] = \overline{s_2}[i] = {}'-'$.

Informalmente, um alinhamento de duas sequências s_1 e s_2 corresponde a um par contendo as sequências $\overline{s_1}$ e $\overline{s_2}$, que são cópias de s_1 e s_2, respectivamente, com espaços inseridos no interior e/ou extremidades delas. Essa inserção de espaços é feita de forma que $\overline{s_1}$ e $\overline{s_2}$ fiquem com o mesmo tamanho e que não haja uma posição i tal que $\overline{s_1}[i]$ e $\overline{s_2}[i]$ sejam ambos iguais a espaço. Dessa forma, as sequências podem ser dispostas uma em cima da outra, com cada caractere de $\overline{s_1}$ correspondendo a um caractere de $\overline{s_2}$.

Para exemplificar um alinhamento de duas sequências, considere o alinhamento $A_1 = (\overline{s_1}, \overline{s_2})$ das sequências $s_1 = CACHORRO$ e $s_2 = CAVALHEIRO$ representado na Figura 2.1. Observe que as colunas do alinhamento estão enumeradas e não existe nenhuma com dois espaços. As colunas 1 e 2 são colunas com caracteres iguais e são genericamente chamadas de ***match***. As colunas 6 e 7 são colunas com caracteres diferentes e são genericamente chamadas de ***mismatch***. As colunas 3 e 4 são colunas com espaço em $\overline{s_1}$ e $\overline{s_2}$, respectivamente, e são genericamente chamadas de ***space***.

		1	2	3	4	5	6	7	8	9	10	11	12
$\overline{s_1}$	=	C	A	-	C	-	H	O	-	-	R	R	O
$\overline{s_2}$	=	C	A	V	-	A	L	H	E	I	-	R	O

Figura 2.1: *Um exemplo de alinhamento de duas sequências s_1 e s_2.*

Dada uma **função de pontuação** $\omega : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}$, que relaciona cada par de caracteres (c_1, c_2) pertencentes a $\overline{\Sigma}$ com um valor real, a **pontuação de um alinhamento** $A = (\overline{s_1}, \overline{s_2})$ com respeito a ω, denotada por $Score_\omega(A)$ ou por $Score_\omega(\overline{s_1}, \overline{s_2})$, é definida como $\sum_{i=1}^{l} \omega(\overline{s_1}[i], \overline{s_2}[i])$, onde $l = |\overline{s_1}| = |\overline{s_2}|$.

Como exemplo, considere $\Sigma = \{A, C, E, H, I, L, O, R, V\}$ e uma função de pontuação ω_1 sobre $\overline{\Sigma}$ definida pela matriz da Figura 2.2. A pontuação do alinhamento A_1 da Figura 2.1 com respeito a ω_1 é:

$$\sum_{i=1}^{12} \omega_1(\overline{s_1}[i], \overline{s_2}[i]) =$$

$$\omega_1(C,C) + \omega_1(A,A) + \omega_1(-,V) + \omega_1(C,-) + \omega_1(-,A) + \omega_1(H,L) +$$

$$\omega_1(O,H) + \omega_1(-,E) + \omega_1(-,I) + \omega_1(R,-) + \omega_1(R,R) + \omega_1(O,O) =$$

$$2 + 1 - 2 - 2 - 1 + 0 + 2 - 2 - 1 - 1 + 8 + 7 = 11.$$

	A	C	E	H	I	L	O	R	V	-
A	1	-2	-3	1	-4	1	-4	2	2	-1
C		2	1	2	2	-3	2	1	0	-2
E			3	2	1	1	0	0	-1	-2
H				4	-1	0	2	1	2	0
I					5	1	3	0	1	-1
L						6	-1	4	2	-3
O							7	0	2	-1
R								8	2	-1
V									9	-2
-										0

Figura 2.2: *Matriz que define a função de pontuação* ω_1.

Vale notar que a função de pontuação ω_1 definida pela matriz da Figura 2.2 é apenas um exemplo arbitrário e que normalmente as funções de pontuação valorizam *matches* e penalizam *mismatches* e *spaces*.

Dadas as definições anteriores, a **similaridade** de duas sequências s_1 e s_2 com respeito a uma função de pontuação ω, denotada por $sim_\omega(s_1, s_2)$, é definida como a maior pontuação de todos os possíveis alinhamentos de s_1 e s_2 com respeito a ω. Ou seja, $sim_\omega(s_1, s_2) = max\{Score_\omega(A)\}$, tal que $A = (\overline{s_1}, \overline{s_2})$ é um alinhamento de s_1 e s_2.

O problema de encontrar a similaridade de duas sequências, chamado de Problema da Similaridade de Duas Sequências, é um problema de otimização e pode ser assim definido:

Problema da Similaridade de Duas Sequências (PSS): dados um alfabeto Σ, tal que $'-' \notin \Sigma$, duas sequências s_1 e s_2 sobre Σ e uma função de pontuação ω sobre $\overline{\Sigma}$, determinar o valor da similaridade $sim_\omega(s_1, s_2)$ de s_1 e s_2.

Dizemos que um alinhamento $A = (\overline{s_1}, \overline{s_2})$ de s_1 e s_2 é um **alinhamento ótimo** com respeito a uma função de pontuação ω se $Score_\omega(A) = sim_\omega(s_1, s_2)$. Note que pode existir mais de um alinhamento ótimo para duas sequências arbitrárias. Com esta definição podemos formular um segundo problema, que estende o PSS e que consiste em encontrar um alinhamento ótimo de duas sequências:

Problema do Alinhamento de Duas Sequências (PAS): dados um alfabeto Σ, tal que $'-' \notin \Sigma$, duas sequências s_1 e s_2 sobre Σ e uma função de pontuação ω sobre $\overline{\Sigma}$, determinar um alinhamento $A = (\overline{s_1}, \overline{s_2})$ de s_1 e s_2 tal que $Score_\omega(A) = sim_\omega(s_1, s_2)$.

Existem diversas variantes do problema do alinhamento de duas sequências. Se consideramos as duas sequências em sua totalidade, estamos falando do **alinhamento global** de duas sequências. Se consideramos apenas segmentos das duas sequências, estamos falando do **alinhamento local** de duas sequências. Se o interesse for alinhar as duas sequências sem penalizar os espaços inseridos nas suas extremidades (espaços que aparecem antes do primeiro e depois do último caractere das sequências), estamos falando do **alinhamento semi-global** de duas sequências. Para mais detalhes e descrições de algoritmos existentes para as variantes do problema do alinhamento de duas sequências, sugerimos as referências [19, 38].

No decorrer deste trabalho, consideramos apenas o alinhamento global de duas sequências. Sendo assim, toda vez que usarmos o termo "alinhamento de duas sequências", estamos nos referindo ao alinhamento global de duas sequências.

2.5.1 O algoritmo de Needleman-Wunsch

O PSS e o PAS são problemas fortemente relacionados. Para a solução de ambos podemos utilizar o algoritmo de Needleman-Wunsch, proposto por Saul B. Needleman e Christian D. Wunsch em 1970 [28][3]. Esse algoritmo utiliza a técnica de programação dinâmica, introduzida na Seção 2.4, para encontrar um alinhamento ótimo de duas sequências dadas como entrada.

Descreveremos o algoritmo de Needleman-Wunsch de acordo com os quatro passos descritos por Cormen *et al.* em [11] para o desenvolvimento de um algoritmo genérico de programação dinâmica, e apresentados na Seção 2.4. Como veremos, o PSS é resolvido ao chegarmos no terceiro passo, onde é possível determinar o valor da similaridade de duas sequências. Para resolver o PAS, basta continuar esta sequência de passos e então executar o quarto passo, que consiste em construir um alinhamento ótimo a partir das informações calculadas.

Subestrutura ótima do PSS

Para a compreensão do algoritmo de Needleman-Wunsch é necessário, primeiramente, identificar a subestrutura ótima do PSS. Para isso, denotaremos por $Opt(i, j)$, com $0 \leq i \leq |s_1|$ e $0 \leq j \leq |s_2|$, a similaridade do prefixo $s_1[1..i]$ de s_1 e do prefixo $s_2[1..j]$ de s_2. Assim, considerando que $n = |s_1|$ e $m = |s_2|$, $Opt(n, m)$ denota a similaridade de s_1 e s_2, que corresponde à solução do PSS. É importante notar que podemos ter $Opt(i, j)$ tal que $i = 0$ e/ou $j = 0$. Nestes casos, a(s) sequência(s) associada(s) correspondem ao prefixo vazio de s_1 e/ou s_2. Por exemplo $Opt(0, 4)$ denota a similaridade do prefixo vazio de s_1 e $s_2[1..4]$.

O PSS tem a propriedade de subestrutura ótima, como mostra o Lema 2.5.1. Como veremos, a classe de subproblemas que o PSS gera corresponde a pares de prefixos de s_1 e s_2.

Lema 2.5.1 *Sejam s_1 e s_2 duas sequências construídas sobre um alfabeto Σ qualquer, tal que $'-' \notin \Sigma$. Seja ω uma função de pontuação sobre $\overline{\Sigma}$. Seja $A = (\overline{s_1}, \overline{s_2})$ um alinhamento ótimo de s_1 e s_2. Considere $n = |s_1|$, $m = |s_2|$, $\overline{n} = |\overline{s_1}|$ e $\overline{m} = |\overline{s_2}|$.*

1. *Se $\overline{s_1}[\overline{n}] \neq -$ e $\overline{s_2}[\overline{m}] \neq -$, então $A' = (\overline{s_1}[1..\overline{n}-1], \overline{s_2}[1..\overline{m}-1])$ é um alinhamento ótimo de $s_1[1..n-1]$ e $s_2[1..m-1]$;*

2. *Se $\overline{s_1}[\overline{n}] = -$ e $\overline{s_2}[\overline{m}] \neq -$, então $A' = (\overline{s_1}[1..\overline{n}-1], \overline{s_2}[1..\overline{m}-1])$ é um alinhamento ótimo de s_1 e $s_2[1..m-1]$;*

[3] Vale observar que a complexidade de tempo do algoritmo descrito em [28] é cúbica no tamanho da entrada e que o algoritmo que apresentaremos é uma otimização do algoritmo original e possui complexidade de tempo quadrática no tamanho da entrada.

3. *Se* $\overline{s_1}[\overline{n}] \neq -$ *e* $\overline{s_2}[\overline{m}] = -$, *então* $A' = (\overline{s_1}[1..\overline{n}-1], \overline{s_2}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $s_1[1..n-1]$ *e* s_2.

Prova (1) Suponha, por contradição, que $A' = (\overline{s_1}[1..\overline{n}-1], \overline{s_2}[1..\overline{m}-1])$ não é um alinhamento ótimo de $s_1[1..n-1]$ e $s_2[1..m-1]$. Logo, existe um alinhamento $A'' = \left(\overline{\overline{s_1}}, \overline{\overline{s_2}}\right)$ de $s_1[1..n-1]$ e $s_2[1..m-1]$ tal que $Score_\omega(A'') > Score_\omega(A')$. Se adicionarmos $s_1[n]$ a $\overline{\overline{s_1}}$ e $s_2[m]$ a $\overline{\overline{s_2}}$, obteremos o alinhamento $A''' = \left(\overline{\overline{s_1}} \bullet s_1[n], \overline{\overline{s_2}} \bullet s_2[m]\right)$, que é um alinhamento de s_1 e s_2, tal que $Score_\omega(A''') = Score_\omega(A'') + \omega\left(s_1[n], s_2[m]\right) > Score_\omega(A) = Score_\omega(A') + \omega(s_1[n], s_2[m])$, o que é uma contradição da nossa hipótese de que A é um alinhamento ótimo de s_1 e s_2.

(2) Suponha, por contradição, que $A' = (\overline{s_1}[1..\overline{n}-1], \overline{s_2}[1..\overline{m}-1])$ não é um alinhamento ótimo de s_1 e $s_2[1..m-1]$. Logo, existe um alinhamento $A'' = \left(\overline{\overline{s_1}}, \overline{\overline{s_2}}\right)$ de s_1 e $s_2[1..m-1]$ tal que $Score_\omega(A'') > Score_\omega(A')$. Se adicionarmos $-$ a $\overline{\overline{s_1}}$ e $s_2[m]$ a $\overline{\overline{s_2}}$, obteremos o alinhamento $A''' = \left(\overline{\overline{s_1}} \bullet -, \overline{\overline{s_2}} \bullet s_2[m]\right)$, que é um alinhamento de s_1 e s_2, tal que $Score_\omega(A''') = Score_\omega(A'') + \omega\left(-, s_2[m]\right) > Score_\omega(A) = Score_\omega(A') + \omega(-, s_2[m])$, o que é uma contradição da nossa hipótese de que A é um alinhamento ótimo de s_1 e s_2.

(3) A prova do item (3) é análoga à prova do item (2).

■

A maneira que o Lema 2.5.1 caracteriza um alinhamento ótimo nos mostra que um alinhamento ótimo de duas sequências s_1 e s_2 contém, em seu interior, alinhamentos ótimos de prefixos de s_1 e de s_2. Isso significa que uma solução ótima para uma instância qualquer do PSS contém, em seu interior, soluções ótimas de subproblemas desta instância, demonstrando a propriedade de subestrutura ótima do PSS.

Solução recursiva do PSS

Seguindo os passos descritos na Seção 2.4, nesta seção iremos definir uma solução recursiva, na forma de uma recorrência, para o PSS.

Ao identificar a subestrutura ótima do PSS, vimos que um alinhamento ótimo de duas sequências s_1 e s_2 é construído combinando alinhamentos ótimos de prefixos de s_1 e s_2. Dessa maneira, uma solução recursiva para o PSS estabelece um relacionamento recursivo entre os valores de $Opt\,(i,j)$, com $i > 0$ e $j > 0$, e os valores de $Opt\,(k,l)$, com $k < i$ e $l < j$. Ou seja, para resolver o problema de encontrar um alinhamento ótimo de $s_1[1..i]$ e $s_2[1..j]$, com $i > 0$ e $j > 0$, é necessário conhecer os alinhamentos ótimos de seus subproblemas. Quando um problema não precisar das soluções de seus subproblemas para ser resolvido, ele pode ser calculado diretamente e independentemente dos outros problemas. A estes problemas damos o nome de **condições base**. Para o PSS, as condições base são:

$$Opt(0,0) = 0. \tag{2.1}$$

$$Opt(0,j) = \sum_{k=1}^{j} \omega(-, s_2[k]), \quad \text{com } 1 \leq j \leq |s_2|. \tag{2.2}$$

$$Opt(i,0) = \sum_{k=1}^{i} \omega(s_1[k], -), \quad \text{com } 1 \leq i \leq |s_1|. \tag{2.3}$$

A Equação 2.1 está correta já que a similaridade de duas sequências vazias é 0. A condição base 2.2 está correta porque só existe uma maneira de fazer o alinhamento de uma sequência vazia e $s_2[1..j]$, com $1 \leq j \leq |s_2|$, que é alinhar todos os caracteres de $s_2[1..j]$ com espaços. Disso, a pontuação deste alinhamento é $Opt(0,j) = \sum_{k=1}^{j} \omega(-, s_2[k])$. De forma análoga, podemos mostrar que a Equação 2.3 também está correta.

A recorrência que define $Opt\,(i,j)$ quando $i > 0$ e $j > 0$ está definida a seguir:

$$Opt\,(i,j) = max \begin{cases} Opt\,(i-1, j-1) + \omega\left(s_1[i], s_2[j]\right), \\ Opt\,(i, j-1) + \omega\left(-, s_2[j]\right), \\ Opt\,(i-1, j) + \omega\left(s_1[i], -\right). \end{cases} \tag{2.4}$$

O Teorema 2.5.2 e o Corolário 2.5.3 demonstram a correção da solução recursiva definida pela Recorrência 2.4.

Teorema 2.5.2 *$Opt\,(i,j)$ é igual a um dos seguintes valores: $Opt\,(i-1,j-1)+\omega\,(s_1[i],s_2[j])$, ou $Opt\,(i,j-1)+\omega\,(-,s_2[j])$ ou $Opt\,(i-1,j)+\omega\,(s_1[i],-)$. Não existem outras opções.*

Prova Seja $A=(\overline{s_1},\overline{s_2})$ um alinhamento ótimo de $s_1[1..i]$ e $s_2[1..j]$, com $i>0$ e $j>0$. A última coluna do alinhamento A, denotada por $(\overline{s_1}[\overline{n}],\overline{s_2}[\overline{m}])$, deve ser uma das três opções seguintes: (1) o caractere $s_1[i]$ alinhado com o caractere $s_2[j]$, ou (2) um espaço alinhado com o caractere $s_2[j]$ ou (3) o caractere $s_1[i]$ alinhado com um espaço.

1. Na primeira opção, a última coluna do alinhamento $A=(\overline{s_1},\overline{s_2})$ alinha o caractere $s_1[i]$ com o caractere $s_2[j]$, logo $\overline{s_1}[\overline{n}]\neq-$ e $\overline{s_2}[\overline{m}]\neq-$. Pelo Lema 2.5.1, se $\overline{s_1}[\overline{n}]\neq-$ e $\overline{s_2}[\overline{m}]\neq-$, então $A'=(\overline{s_1}[1..\overline{n}-1],\overline{s_2}[1..\overline{m}-1])$ é um alinhamento ótimo de $s_1[1..i-1]$ e $s_2[1..j-1]$. Como por definição $Score_\omega(A')=Opt(i-1,j-1)$, então a opção (1) possui valor $Score_\omega(A')+\omega\,(s_1[i],s_2[j])=Opt(i-1,j-1)+\omega\,(s_1[i],s_2[j])$.

2. Na segunda opção, a última coluna do alinhamento $A=(\overline{s_1},\overline{s_2})$ alinha um espaço com o caractere $s_2[j]$, logo $\overline{s_1}[\overline{n}]=-$ e $\overline{s_2}[\overline{m}]\neq-$. Pelo Lema 2.5.1, se $\overline{s_1}[\overline{n}]=-$ e $\overline{s_2}[\overline{m}]\neq-$, então $A'=(\overline{s_1}[1..\overline{n}-1],\overline{s_2}[1..\overline{m}-1])$ é um alinhamento ótimo de $s_1[1..i]$ e $s_2[1..j-1]$. Como por definição $Score_\omega(A')=Opt(i,j-1)$, então a opção (2) possui valor $Score_\omega(A')+\omega\,(-,s_2[j])=Opt(i,j-1)+\omega\,(-,s_2[j])$.

3. A terceira opção é análoga à segunda opção e possui valor $Opt\,(i-1,j)+\omega\,(s_1[i],-)$.

Como, por definição, dois espaços não podem ser alinhados, nós cobrimos todas as opções e estabelecemos o teorema.

■

Corolário 2.5.3 *$Opt\,(i,j)=max\{Opt\,(i-1,j-1)+\omega\,(s_1[i],s_2[j]),\ Opt\,(i,j-1)+\omega\,(-,s_2[j]),\ Opt\,(i-1,j)+\omega\,(s_1[i],-)\}$.*

Calculando a similaridade de duas sequências

Com base na Recorrência 2.4 e nas condições base 2.1, 2.2 e 2.3, é possível desenvolver facilmente um algoritmo recursivo AR para o PSS. Para saber a similaridade de duas sequências s_1 e s_2, de tamanho n e m respectivamente, bastaria calcular $Opt(n,m)$. No algoritmo recursivo AR, esse cálculo geraria a árvore de recursão especificada na Figura 2.3, cujo número de nós é $\Omega(3^k)$, com $k=min(n,m)$. Como cada nó desta árvore de recursão representa um (sub)problema a ser resolvido, o algoritmo AR é exponencial porque precisa resolver uma quantidade exponencial de (sub)problemas. O Lema 2.5.4 mostra que os $k+1$ primeiros níveis da árvore de recursão gerada pelo algoritmo recursivo AR possuem $3^k+\lfloor\frac{3^k}{2}\rfloor$ nós, com $k=min(n,m)$.

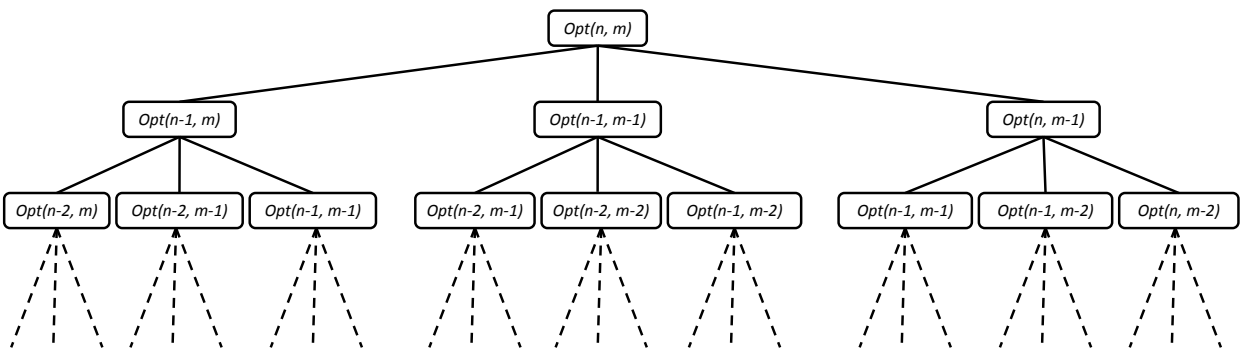


Figura 2.3: *Árvore de recursão para $Opt(n,m)$.*

Lema 2.5.4 *Os $k+1$ primeiros níveis da árvore de recursão gerada pelo algoritmo recursivo AR possuem $3^k + \lfloor \frac{3^k}{2} \rfloor$ nós, com $k = min(n, m)$.*

Prova Considere que a raiz da árvore de recursão está no nível 0, os filhos diretos da raiz no nível 1, e assim por diante. Mostraremos que os $k+1$ primeiros níveis da árvore de recursão gerada pelo algoritmo recursivo AR possuem $3^k + \lfloor \frac{3^k}{2} \rfloor$ nós, com $k = min(n, m)$, por indução no número de nós da árvore. Se $k = 0$ (ou seja, $n = 0$ e/ou $m = 0$), então o primeiro nível da árvore de recursão possui 1 nó e $3^0 + \lfloor \frac{3^0}{2} \rfloor = 1$. Para o passo indutivo, considere que $k = min(n, m)$ e suponha que a hipótese de indução valha até $k-1$, ou seja, os k primeiros níveis da árvore de recursão possuem $3^{k-1} + \lfloor \frac{3^{k-1}}{2} \rfloor$ nós. Mostraremos que a hipótese de indução vale para k, ou seja, os $k+1$ primeiros níveis da árvore de recursão possuem $3^k + \lfloor \frac{3^k}{2} \rfloor$ nós. Primeiramente, observe que a subárvore que inclui apenas os $k+1$ primeiros níveis da árvore de recursão é uma árvore ternária cheia, ou seja, o nível 0 possui 3^0 nós, o nível 1 possui 3^1 nós, o nível 2 possui 3^2 nós,..., o nível k possui 3^k nós. Pela hipótese de indução, os k primeiros níveis da árvore de recursão possuem $3^{k-1} + \lfloor \frac{3^{k-1}}{2} \rfloor$ nós. Logo, os $k+1$ primeiros níveis da árvore de recursão possuem

$$(3^{k-1} + \lfloor \frac{3^{k-1}}{2} \rfloor) + 3^k =$$

$$(\lfloor \frac{2 * 3^{k-1}}{2} \rfloor + \lfloor \frac{3^{k-1}}{2} \rfloor) + 3^k =$$

$$(\lfloor \frac{3 * 3^{k-1}}{2} \rfloor) + 3^k =$$

$$3^k + \lfloor \frac{3^k}{2} \rfloor \text{ nós.}$$

■

Baseando-se no Lema 2.5.4, é trivial mostrar que a árvore de recursão gerada pelo algoritmo recursivo AR possui $\Omega(3^k)$ nós, com $k = min(n, m)$. O Corolário 2.5.5 determina a complexidade de tempo do algoritmo recursivo AR.

Corolário 2.5.5 *O algoritmo recursivo AR para o PSS possui complexidade de tempo $\Omega(3^k)$, com $k = min(n, m)$.*

A abordagem recursiva do problema é simples de programar, porém é extremamente ineficiente para valores grandes de n e de m. A principal causa dessa ineficiência é que a quantidade de subproblemas que o algoritmo deve resolver cresce exponencialmente em função de $min(n, m)$. Contudo, se observarmos melhor o espaço de subproblemas, podemos perceber que existem apenas $(n+1) * (m+1)$ pares de números (i, j) distintos, para $0 \leq i \leq n$ e $0 \leq j \leq m$. Consequentemente, existem apenas $(n+1) * (m+1)$ subproblemas distintos, que é um número polinomial em relação ao tamanho da entrada. Com isso, podemos concluir então que um algoritmo recursivo para o PSS resolve os mesmos subproblemas repetidamente, ao invés de sempre gerar subproblemas novos. Desse fato, podemos dizer que o PSS apresenta a característica de subproblemas sobrepostos. Essa é a segunda característica fundamental que um problema precisa apresentar para que a programação dinâmica seja aplicável sobre ele.

Nesse terceiro passo, iremos apresentar um algoritmo de programação dinâmica que resolve cada subproblema apenas na primeira vez que o encontra, e então guarda o valor da solução em uma tabela, que será consultada nos próximos momentos que o subproblema for encontrado novamente. Fazendo isso, melhoramos o tempo de execução de nosso algoritmo de exponencial para polinomial em função do tamanho da entrada.

Adotando-se a estratégia de baixo para cima, primeiramente devemos calcular $Opt(i, j)$ para os menores valores possíveis de i e j e então calcular $Opt(i, j)$ para valores crescentes de i e j. Para o PSS, esses cálculos serão organizados em uma tabela bidimensional M de dimensões $(n+1)*(m+1)$,

chamada de **matriz de alinhamento**, de tal forma que o valor da solução do subproblema $Opt(i, j)$ esteja armazenado em $M[i][j]$. Note que as células da coluna 0 e da linha 0 de M representam as condições base da recorrência. Portanto, podemos calcular os valores que serão armazenados nas células da coluna 0 e da linha 0 de M diretamente. As $n * m$ células restantes são preenchidas uma linha por vez, em ordem crescente. Para cada linha, as células são preenchidas em ordem crescente das colunas. Dessa maneira, os valores necessário para preencher $M[i][j]$ já estarão disponíveis quando essa célula for alcançada. A similaridade das duas sequências será determinada ao preencher a célula $M[n][m]$. O algoritmo de Needleman-Wunsch [28] segue exatamente esta ideia e está especificado no Algoritmo 1.

Algoritmo 1 Calcula_Similaridade(s_1, s_2, ω)

Entrada: Duas sequências s_1 e s_2 e uma função de pontuação ω.
Saída: O valor da similaridade $sim_\omega(s_1, s_2)$ de s_1 e s_2.

```
1: //Inicialização
2: n ← |s1|;
3: m ← |s2|;

4: //Condições base
5: M[0][0] ← 0;
6: para i ← 1 até n faça
7:    M[i][0] ← M[i − 1][0] + ω(s1[i], −);
8: fim para
9: para j ← 1 até m faça
10:   M[0][j] ← M[0][j − 1] + ω(−, s2[j]);
11: fim para

12: //Recorrência 2.4
13: para i ← 1 até n faça
14:   para j ← 1 até m faça
15:      M[i][j] ← max(M[i − 1][j − 1] + ω(s1[i], s2[j]), M[i][j − 1] + ω(−, s2[j]), M[i − 1][j] + ω(s1[i], −));
16:   fim para
17: fim para

18: devolva M[n][m];
```

Como um exemplo da execução do Algoritmo 1, considere o alfabeto $\Sigma = \{A, C, G, T\}$, e que são dadas como entrada as sequências $s_1 = ACGT$ e $s_2 = ACC$ e a função de pontuação $\omega(a, b)$ como definida a seguir:

$$\omega(a,b) = \begin{cases} 1, & \text{se } a = b \\ -1, & \text{se } a \neq b \\ -2, & \text{se } a = - \text{ ou } b = - \end{cases}.$$

A matriz de alinhamento completa, resultante da execução do Algoritmo 1 neste exemplo, está representada na Figura 2.4. Para esta instância, a similaridade de s_1 e s_2 é -1.

Analisando o Algoritmo 1, podemos perceber que, para preencher qualquer célula $M[i][j]$ da matriz de alinhamento, é necessário um número constante de operações aritméticas e de comparações. Como temos $(n + 1) * (m + 1)$ células que são calculadas em tempo constante, podemos concluir que a complexidade de tempo do Algoritmo 1 é $O(n * m)$.

			A	C	C
		0	1	2	3
	0	**0**	**-2**	**-4**	**-6**
A	1	**-2**	**1**	**-1**	**-3**
C	2	**-4**	**-1**	**2**	**0**
G	3	**-6**	**-3**	**0**	**1**
T	4	**-8**	**-5**	**-2**	**-1**

Figura 2.4: *Exemplo de uma matriz de alinhamento do Algoritmo 1.*

Construindo um alinhamento ótimo

Até agora vimos como calcular a similaridade de duas sequências s_1 e s_2 com respeito a uma função de pontuação ω, resolvendo o PSS. Contudo, ainda temos o problema de encontrar um alinhamento ótimo dessas duas sequências. Para isso, basta continuarmos a sequência de passos anterior adicionando o quarto passo, que será responsável por construir um alinhamento ótimo a partir da matriz de alinhamento já calculada.

A maneira mais fácil de fazer isso é modificar levemente o Algoritmo 1, inserindo ponteiros nas células da matriz de alinhamento. Detalhadamente, logo após calcularmos o valor de $M[i][j]$, com $i > 0$ e $j > 0$, adicionamos um ponteiro:

1. de $M[i][j]$ para $M[i-1][j-1]$ se $M[i][j] = M[i-1][j-1] + \omega(s_1[i], s_2[j])$;
2. de $M[i][j]$ para $M[i-1][j]$ se $M[i][j] = M[i-1][j] + \omega(s_1[i], -)$;
3. de $M[i][j]$ para $M[i][j-1]$ se $M[i][j] = M[i][j-1] + \omega(-, s_2[j])$.

Note que, pelas condições acima, permitimos que uma célula $M[i][j]$ aponte para uma, duas, ou três outras células. Os ponteiros das células das condições base podem ser diretamente calculados. $M[0][0]$ é a única célula da matriz que não aponta para nenhuma outra célula. As células da coluna 0, $M[i][0]$, com $1 \leq i \leq n$, apontam sempre para a célula imediatamente acima. As célula da linha 0, $M[0][j]$, com $1 \leq j \leq m$, apontam sempre para a célula imediatamente à esquerda.

O Algoritmo 2 é o algoritmo de Needleman-Wunsch modificado com a inclusão destes ponteiros. Nele, uma matriz adicional P de dimensões $(n+1) * (m+1)$, que chamaremos de **matriz de ponteiros**, é utilizada para armazenar os ponteiros após o cálculo do valor de uma célula.

Voltando ao nosso exemplo da Figura 2.4, se executarmos o Algoritmo 2 com a instância do exemplo, obtemos a matriz de alinhamento da Figura 2.5, que apresenta não somente o valor de cada célula, mas também os ponteiros, representados por setas, para onde cada célula aponta. Note que, por questões de simplicidade, mesclamos M e P em uma única matriz na Figura 2.5.

			A	C	C
		0	1	2	3
	0	**0**	← **-2**	← **-4**	← **-6**
A	1	↑ **-2**	↖ **1**	← **-1**	← **-3**
C	2	↑ **-4**	↑ **-1**	↖ **2**	↖← **0**
G	3	↑ **-6**	↑ **-3**	↑ **0**	↖ **1**
T	4	↑ **-8**	↑ **-5**	↑ **-2**	↖↑ **-1**

Figura 2.5: *Matriz de alinhamento da Figura 2.4 com ponteiros.*

Com a matriz de ponteiros P calculada, um alinhamento ótimo pode ser construído simplesmente seguindo qualquer caminho de ponteiros da célula $P[n][m]$ até a célula $P[0][0]$. Dessa forma, se há uma seta diagonal partindo de uma célula $P[i][j]$ no caminho, então a coluna do alinhamento associada inclui o caractere $s_1[i]$ alinhado com o caractere $s_2[j]$. Em contrapartida, se existe uma seta para cima partindo de uma célula $P[i][j]$ no caminho, então a coluna do alinhamento associada

Algoritmo 2 Calcula_Similaridade_Ponteiros(s_1, s_2, ω)

Entrada: Duas sequências s_1 e s_2 e uma função de pontuação ω.
Saída: O valor da similaridade $sim_\omega(s_1, s_2)$ de s_1 e s_2.

1: //Inicialização
2: $n \leftarrow |s_1|$;
3: $m \leftarrow |s_2|$;

4: //Condições base
5: $M[0][0] \leftarrow 0$;
6: $P[0][0] \leftarrow NULO$;
7: **para** $i \leftarrow 1$ até n **faça**
8: $M[i][0] \leftarrow M[i-1][0] + \omega(s_1[i], -)$;
9: $P[i][0] \leftarrow$ "↑";
10: **fim para**
11: **para** $j \leftarrow 1$ até m **faça**
12: $M[0][j] \leftarrow M[0][j-1] + \omega(-, s_2[j])$;
13: $P[0][j] \leftarrow$ "←";
14: **fim para**

15: //Recorrência 2.4
16: **para** $i \leftarrow 1$ até n **faça**
17: **para** $j \leftarrow 1$ até m **faça**
18: $M[i][j] \leftarrow max(M[i-1][j-1] + \omega(s_1[i], s_2[j]),\ M[i][j-1] + \omega(-, s_2[j]),\ M[i-1][j] + \omega(s_1[i], -))$;
19: **se** $M[i][j] = M[i-1][j-1] + \omega(s_1[i], s_2[j])$ **então**
20: $P[i][j] \leftarrow$ "↖";
21: **fim se**
22: **se** $M[i][j] = M[i-1][j] + \omega(s_1[i], -)$ **então**
23: $P[i][j] \leftarrow$ "↑";
24: **fim se**
25: **se** $M[i][j] = M[i][j-1] + \omega(-, s_2[j])$ **então**
26: $P[i][j] \leftarrow$ "←";
27: **fim se**
28: **fim para**
29: **fim para**

30: **devolva** $M[n][m]$;

inclui o caractere $s_1[i]$ alinhado com um espaço. Por fim, se há uma seta para a esquerda partindo de uma célula $P[i][j]$ no caminho, então a coluna do alinhamento associada inclui um espaço alinhado com o caractere $s_2[j]$. É com esta ideia que construímos um alinhamento ótimo $A = (\overline{s_1}, \overline{s_2})$ de s_1 e s_2. O Algoritmo 3 especifica a sequência de passos para se construir A.

Tomando-se como exemplo a matriz da Figura 2.5, e lembrando que $s_1 = ACGT$ e $s_2 = ACC$, o Algoritmo 3 poderia determinar o alinhamento ótimo $A' = (ACGT, AC-C)$. O caminho percorrido para determinar A' começaria em $P[4][3]$ e passaria por $P[3][2]$, $P[2][2]$ e $P[1][1]$, nesta ordem, até finalmente chegar em $P[0][0]$. Contudo, note que temos outro caminho de $P[4][3]$ até $P[0][0]$, passando por $P[3][3]$, $P[2][2]$ e $P[1][1]$. Se o algoritmo optasse por este segundo caminho, então obteríamos outro alinhamento ótimo $A'' = (ACGT, ACC-)$. Isto significa que podemos ter vários alinhamentos ótimos de duas sequências, de acordo com quantos caminhos diferentes existirem de $P[n][m]$ até $P[0][0]$.

A complexidade de tempo do Algoritmo 3 pode ser determinada facilmente. A parte de inicialização das variáveis do algoritmo, que começa na linha 2 e vai até a linha 6, consome tempo

constante, com exceção das linhas 4, que consome tempo $O(n)$, e 5, que consome tempo $O(m)$. A cada iteração do laço **enquanto**, decrementamos sempre uma unidade de i e/ou de j. Assim, no pior caso, teremos $n + m$ iterações neste laço até que a posição $P[0][0]$ seja alcançada. Como cada iteração consome tempo constante, gastamos tempo $O(n + m)$ no laço **enquanto**. Por fim, para inverter as sequências $\overline{s_1}$ e $\overline{s_2}$ nas linhas 25 e 26, respectivamente, gastamos tempo $O(n + m)$ utilizando um algoritmo simples. Portanto, um alinhamento ótimo de duas sequências s_1 e s_2 pode ser construído pelo Algoritmo 3 em tempo $O(n + m)$. O PAS é resolvido invocando primeiramente o Algoritmo 2, para calcular a matriz de ponteiros P, e depois executando o Algoritmo 3, para encontrar um alinhamento ótimo das duas sequências passadas como entrada. Portanto, é necessário tempo $O(n * m) + O(n + m) = O(n * m)$ para resolver o PAS.

Algoritmo 3 Calcula_Alinhamento_Ótimo(s_1, s_2, P)

Entrada: Duas sequências s_1 e s_2 e uma matriz de ponteiros P.
Saída: Um alinhamento ótimo $(\overline{s_1}, \overline{s_2})$ de s_1 e s_2.

1: //Inicialização
2: Seja $\overline{s_1}$ uma sequência vazia;
3: Seja $\overline{s_2}$ uma sequência vazia;
4: $i \leftarrow |s_1|$;
5: $j \leftarrow |s_2|$;
6: $k \leftarrow 1$;

7: //Laço que calcula um alinhamento ótimo $(\overline{s_1}, \overline{s_2})$ de s_1 e s_2
8: **enquanto** $i \neq 0$ ou $j \neq 0$ **faça**
9: **se** $P[i][j] =$ "↖" **então**
10: $\overline{s_1}[k] \leftarrow s_1[i]$;
11: $\overline{s_2}[k] \leftarrow s_2[j]$;
12: $i \leftarrow i - 1$;
13: $j \leftarrow j - 1$;
14: **senão se** $P[i][j] =$ "↑" **então**
15: $\overline{s_1}[k] \leftarrow s_1[i]$;
16: $\overline{s_2}[k] \leftarrow -$;
17: $i \leftarrow i - 1$;
18: **senão**
19: $\overline{s_1}[k] \leftarrow -$;
20: $\overline{s_2}[k] \leftarrow s_2[j]$;
21: $j \leftarrow j - 1$;
22: **fim se**
23: $k \leftarrow k + 1$;
24: **fim enquanto**

25: Inverta a sequência $\overline{s_1}$;
26: Inverta a sequência $\overline{s_2}$;

27: **devolva** $(\overline{s_1}, \overline{s_2})$;

Capítulo 3

O Problema do Alinhamento de Segmentos

Neste capítulo abordamos os problemas que constituem o objeto principal de estudo deste trabalho: o Problema do Alinhamento de Segmentos e o Problema do Alinhamento de Segmentos Múltiplo. Além de apresentar a definição formal dos problemas, mostramos também um exemplo de instância para cada um deles no intuito de facilitar sua compreensão, e alguns trabalhos relacionados à esses problemas presentes na literatura.

3.1 O Problema do Alinhamento de Segmentos

O Problema do Alinhamento de Segmentos, ou simplesmente PASG, pode ser definido como segue:

Problema do Alinhamento de Segmentos (PASG): dados um alfabeto Σ, tal que $'-' \notin \Sigma$, duas sequências s_1 e s_2 sobre Σ, de tamanho n e m respectivamente, um conjunto ordenado $B = \{b_1, b_2, ..., b_u\}$ de segmentos de s_1, um conjunto ordenado $C = \{c_1, c_2, ..., c_v\}$ de segmentos de s_2, e uma função de pontuação ω sobre $\overline{\Sigma}$, determinar uma cadeia $\Gamma_B = \{b_p, b_q, ..., b_r\}$ de segmentos de B e uma cadeia $\Gamma_C = \{c_w, c_x, ..., c_y\}$ de segmentos de C, tal que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C.

Note que os segmentos que compõem os conjuntos B e C podem se sobrepor. Contudo, os conjuntos de segmentos que constituem a solução do problema, ou seja, os conjuntos Γ_B e Γ_C, não podem incluir segmentos sobrepostos (pela própria definição de cadeia de segmentos). Observe também que, para qualquer instância do PASG, temos que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet) \geq 0$, já que $\Gamma_B = \emptyset$ e $\Gamma_C = \emptyset$ correspondem a uma solução válida, com $\Gamma_B^\bullet = \Gamma_C^\bullet = \epsilon$, que possui valor 0.

Informalmente, no PASG estamos interessados em encontrar subconjuntos ordenados e sem sobreposição de B e de C tal que as sequências resultantes da concatenação dos elementos desses subconjuntos são muito parecidas. Ou seja, buscamos por um subconjunto $\Gamma_B = \{b_p, b_q, ..., b_r\}$ de B e um subconjunto $\Gamma_C = \{c_w, c_x, ..., c_y\}$ de C, onde $b_p \prec b_q \prec ... \prec b_r$ e $c_w \prec c_x \prec ... \prec c_y$, tal que a pontuação de um alinhamento ótimo A de $b_p \bullet b_q \bullet ... \bullet b_r$ e $c_w \bullet c_x \bullet ... \bullet c_y$ é a maior entre todas as cadeias de segmentos de B e de C.

A Figura 3.1 ilustra uma instância do PASG para duas sequências s_1 e s_2 construídas sobre o alfabeto $\Sigma = \{A, ..., Z, \#\}$ e considerando uma função de pontuação $\omega(a, b)$ como definida a seguir:

$$\omega(a,b) = \begin{cases} 1, & \text{se } a = b \\ -1, & \text{se } a \neq b \\ -2, & \text{se } a = - \text{ ou } b = - \end{cases}.$$

Para a instância da Figura 3.1, as cadeias de segmentos $\Gamma_B = \{PROFESS, NAO, DESATENCIOSAMENTE\}$ e $\Gamma_C = \{PROFISSAO, DE, ATENCIOSO, MENTE\}$ são

aquelas cuja $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C. Um alinhamento ótimo de $\Gamma_B^\bullet$ e $\Gamma_C^\bullet$ com respeito a ω está representado na Figura 3.2.

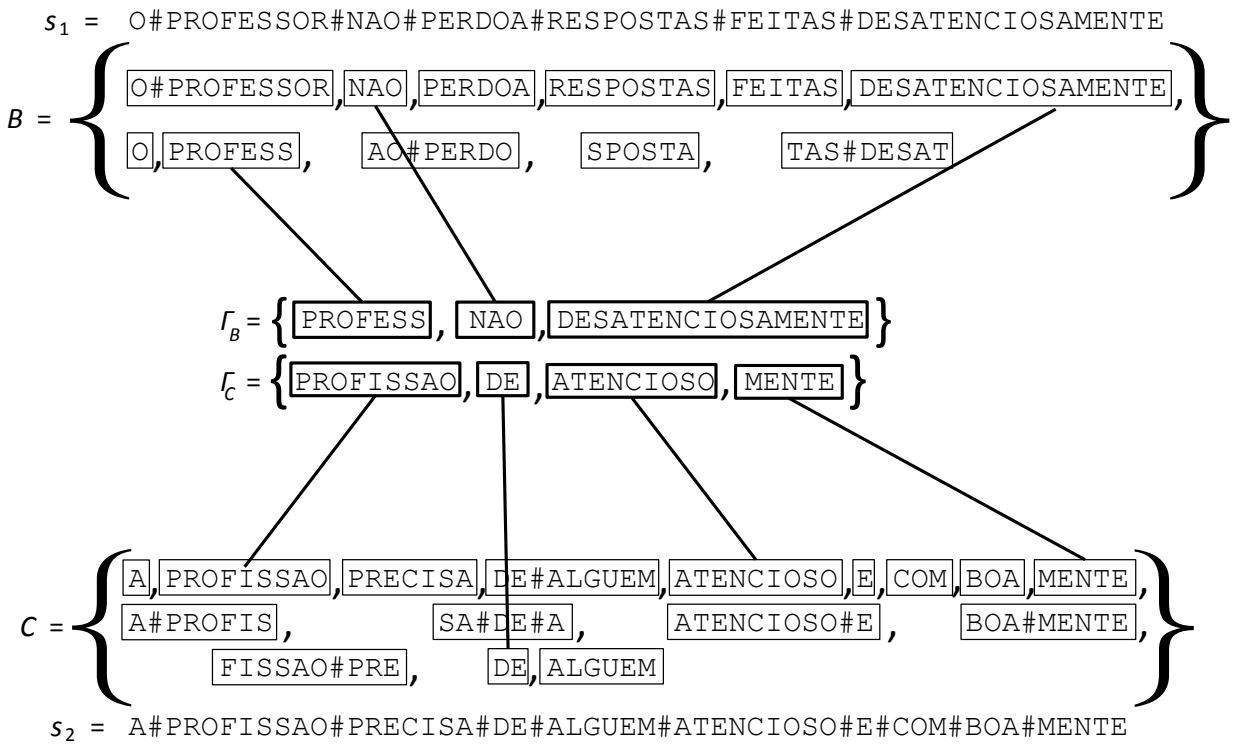


Figura 3.1: *Uma instância do PASG.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
$\overline{\Gamma_B^\bullet}$ =	P	R	O	F	E	S	S	N	A	O	D	E	S	A	T	E	N	C	I	O	S	A	M	E	N	T	E	
$\overline{\Gamma_C^\bullet}$ =	P	R	O	F	I	S	S	-	A	O	D	E	-	A	T	E	N	C	I	O	S	O	M	E	N	T	E	
Score =	+1	+1	+1	+1	-1	+1	+1	-2	+1	+1	+1	+1	-2	+1	+1	+1	+1	+1	+1	+1	+1	-1	+1	+1	+1	+1	+1	=17

Figura 3.2: *Um alinhamento ótimo de $\Gamma_B^\bullet$ e $\Gamma_C^\bullet$, que são as sequências resultantes da concatenação dos elementos das cadeias de segmentos Γ_B e Γ_C determinadas para o exemplo da Figura 3.1, com 23 matches, 2 mismatches, 2 spaces e uma pontuação igual a* 17.

3.2 O Problema do Alinhamento de Segmentos Múltiplo

O Problema do Alinhamento de Segmentos Múltiplo, ou simplesmente PASGM, pode ser visto como uma extensão do PASG. Esse problema encontra-se definido a seguir:

Problema do Alinhamento de Segmentos Múltiplo (PASGM): dados um alfabeto Σ, tal que $'-' \notin \Sigma$, $n > 2$ sequências $s_1, s_2, ..., s_n$ sobre Σ, n conjuntos ordenados de segmentos $B_1, B_2, ..., B_n$, onde B_i é um conjunto ordenado de segmentos de s_i, e uma função de pontuação ω sobre $\overline{\Sigma}$, determinar n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i, tal que $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet)$ é máxima.

A principal diferença entre o PASG e o PASGM está na quantidade de sequências e de conjuntos ordenados de segmentos dados como entrada. Conforme já foi visto, no PASG recebemos duas sequências s_1 e s_2 e seus respectivos conjuntos ordenados de segmentos, B de s_1 e C de s_2, e temos como objetivo determinar duas cadeias de segmentos Γ_B e Γ_C tal que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C. Em contraste, no PASGM recebemos como entrada um

número variável $n > 2$ de sequências $s_1, s_2, ..., s_n$ e seus respectivos conjuntos ordenados de segmentos, B_1 de s_1, B_2 de s_2, ..., e B_n de s_n, e temos como objetivo determinar n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$ (uma para cada conjunto ordenado de segmentos), cujas concatenações, quando comparadas entre si, são bem parecidas. Em termos formais, queremos que $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet)$ seja máxima.

A Figura 3.3 ilustra uma instância do PASGM para quatro sequências s_1, s_2, s_3 e s_4, construídas sobre o alfabeto $\Sigma = \{A, C, G, T, \#\}$, e considerando a função de pontuação $\omega(a, b)$ definida a seguir:

$$\omega(a,b) = \begin{cases} 1, & \text{se } a = b \\ -1, & \text{se } a \neq b \\ -2, & \text{se } a = - \text{ ou } b = - \end{cases} .$$

Vale observar que no exemplo da Figura 3.3 não temos segmentos sobrepostos em B_1, B_2, B_3 e B_4, mas poderíamos tê-los em alguns ou em todos conjuntos ordenados de segmentos da entrada. As cadeias de segmentos $\Gamma_1 = \{CGTAA, CTG\}$, $\Gamma_2 = \{CT, AA, CT\}$, $\Gamma_3 = \{CCAGCTG\}$ e $\Gamma_4 = \{CGTA, ACAA\}$ são aquelas tal que $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet)$ é máxima para a instância da Figura 3.3. Os alinhamentos ótimos de $\Gamma_1^\bullet, \Gamma_2^\bullet, \Gamma_3^\bullet$ e $\Gamma_4^\bullet$, assim como o cálculo de $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet)$ para a instância da Figura 3.3, estão representados na Figura 3.4.

Por fim, note que para qualquer instância do PASGM, $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) \geq 0$, já que n cadeias de segmentos vazias correspondem a uma solução válida que possui valor 0.

Figura 3.3: *Uma instância do PASGM para quatro sequências s_1, s_2, s_3 e s_4.*

$\overline{\Gamma_1^\bullet}$ =	C	G	T	A	A	C	T	G
$\overline{\Gamma_2^\bullet}$ =	C	-	T	A	A	C	T	-
Score =	+1	-2	+1	+1	+1	+1	+1	-2

$\overline{\Gamma_1^\bullet}$ =	C	G	T	A	A	C	T	G
$\overline{\Gamma_3^\bullet}$ =	C	C	-	A	G	C	T	G
Score =	+1	-1	-2	+1	-1	+1	+1	+1

$\overline{\Gamma_1^\bullet}$ =	C	G	T	A	A	C	T	G
$\overline{\Gamma_4^\bullet}$ =	C	G	T	A	A	C	A	A
Score =	+1	+1	+1	+1	+1	+1	-1	-1

$\overline{\Gamma_2^\bullet}$ =	C	T	A	A	C	T	-
$\overline{\Gamma_3^\bullet}$ =	C	C	A	G	C	T	G
Score =	+1	-1	+1	-1	+1	+1	-2

$\overline{\Gamma_2^\bullet}$ =	C	-	T	A	A	C	T	-
$\overline{\Gamma_4^\bullet}$ =	C	G	T	A	A	C	A	A
Score =	+1	-2	+1	+1	+1	+1	-1	-2

$\overline{\Gamma_3^\bullet}$ =	C	C	-	A	G	C	T	G
$\overline{\Gamma_4^\bullet}$ =	C	G	T	A	A	C	A	A
Score =	+1	-1	-2	+1	-1	+1	-1	-1

$$\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) =$$

$$sim_\omega(\Gamma_1^\bullet, \Gamma_2^\bullet) + sim_\omega(\Gamma_1^\bullet, \Gamma_3^\bullet) + sim_\omega(\Gamma_1^\bullet, \Gamma_4^\bullet) + sim_\omega(\Gamma_2^\bullet, \Gamma_3^\bullet) + sim_\omega(\Gamma_2^\bullet, \Gamma_4^\bullet) + sim_\omega(\Gamma_3^\bullet, \Gamma_4^\bullet) =$$

$$2 + 1 + 4 + 0 + 0 + (-3) = 4.$$

Figura 3.4: *Os alinhamentos ótimos de* $\Gamma_1^\bullet$, $\Gamma_2^\bullet$, $\Gamma_3^\bullet$ *e* $\Gamma_4^\bullet$ *e o cálculo de* $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet)$ *para a instância da Figura 3.3.*

3.3 Trabalhos relacionados

Até onde sabemos, não existem algoritmos eficientes para o PASG e nem para o PASGM. Mais do que isso, durante nossas pesquisas não encontramos trabalho algum na literatura que os definem e os abordam formalmente. O que de fato encontramos foram alguns trabalhos que tratam de problemas semelhantes. Esses trabalhos serão vistos a seguir.

Um problema similar ao PASG é o Problema do Alinhamento *Spliced* (*Spliced Alignment Problem*, do inglês), proposto por Gelfand *et al.* em [17]. O Problema do Alinhamento *Spliced*, ou simplesmente PA*Sp*, encontra-se definido a seguir:

Problema do Alinhamento *Spliced* (PA*Sp*): dados um alfabeto Σ, tal que $'-' \notin \Sigma$, duas sequências s_1 e s_2 sobre Σ, de tamanho n e m respectivamente, um conjunto ordenado $B = \{b_1, b_2, ..., b_u\}$ de segmentos de s_1 e uma função de pontuação ω sobre $\overline{\Sigma}$, determinar uma cadeia $\Gamma = \{b_p, b_q, ..., b_r\}$ de segmentos de B tal que $sim_\omega(\Gamma^\bullet, s_2)$ é máxima entre todas as cadeias de segmentos de B.

A definição do PA*Sp* deixa clara a diferença entre esse problema e o PASG. O PASG recebe, além de um conjunto ordenado de segmentos de s_1, um conjunto ordenado de segmentos de s_2, e tem por objetivo encontrar duas cadeias de segmentos, uma para cada sequência de entrada, cujas sequências resultantes da concatenação dos elementos dessas cadeias tenham a maior similaridade possível. Por outro lado, o PA*Sp* não recebe um conjunto ordenado de segmentos de s_2 e busca uma cadeia de segmentos Γ de B tal que a similaridade da sequência resultante da concatenação dos elementos de Γ e s_2 seja a maior possível.

Gelfand *et al.* em [17] apresentaram um algoritmo polinomial que resolve o PA*Sp* baseado na técnica de programação dinâmica. Para compreender a recorrência que o algoritmo implementa, é necessário primeiramente definir alguns termos.

Seja $b_k = s_1[l..i..m]$ um segmento de s_1 que contém uma posição i de s_1, com $l \leq i \leq m$. O i-prefixo de b_k, denotado por $b_k(i)$, é o segmento $b_k(i) = s_1[l..i]$. Para um segmento $b_k = s_1[l..m]$, os autores definem $first(k) = l$, $last(k) = m$ e $size(k) = m - l + 1$. Seja $\Gamma = \{b_p, b_q, ..., b_k, ..., b_r\}$ uma cadeia de segmentos de B tal que o segmento b_k contém a posição i de s_1. Os autores definem $\Gamma^\bullet(i)$ como a sequência $\Gamma^\bullet(i) = b_p \bullet b_q \bullet ... \bullet b_k(i)$. Dadas essas definições, considere a função S:

$$S(i,j,k) = \max_{\text{todas as cadeias } \Gamma \text{ que contêm o segmento } b_k} sim_\omega(\Gamma^\bullet(i), s_2[1..j]),$$

que está definida para $1 \leq i \leq n$, $1 \leq j \leq m$ e $1 \leq k \leq u$. Informalmente, $S(i,j,k)$ denota o valor da solução ótima do subproblema cuja instância é composta pela sequência s_1, por um conjunto ordenado $B' = \{b \in B : b \prec b_k\} \bigcup \{b_k[1..i]\}$ de segmentos de s_1 e pela sequência $s_2[1..j]$, e cujo objetivo é encontrar uma cadeia $\Gamma' = \{b_p, b_q, ..., b_k[1..i]\}$ de segmentos de B' tal que $sim_\omega(\Gamma'^\bullet, s_2[1..j])$ é máxima entre todas as cadeias de segmentos de B' cujo último segmento é $b_k[1..i]$.

$S(i,j,k)$ pode ser eficientemente calculada utilizando a técnica de programação dinâmica através da seguinte recorrência:

$$S(i,j,k) = \max \begin{cases} S(i-1,j-1,k) + \omega(s_1[i], s_2[j]), & \text{se } i \neq first(k) \\ S(i-1,j,k) + \omega(s_1[i], -), & \text{se } i \neq first(k) \\ max_{l \in B(first(k))} S(last(l), j-1, l) + \omega(s_1[i], s_2[j]), & \text{se } i = first(k) \\ max_{l \in B(first(k))} S(last(l), j, l) + \omega(s_1[i], -), & \text{se } i = first(k) \\ S(i, j-1, k) + \omega(-, s_2[j]). \end{cases}, \quad (3.1)$$

onde $B(i) = \{k : last(k) < i\}$ é o conjunto de segmentos que terminam estritamente antes da posição i em s_1.

Depois de calcular todos os valores de $S(i,j,k)$, com $1 \leq i \leq n$, $1 \leq j \leq m$ e $1 \leq k \leq u$, a pontuação de um alinhamento *spliced* ótimo (isto é, o valor $sim_\omega(\Gamma^\bullet, s_2)$) é determinada por:

$$\max_k S(last(k), m, k). \quad (3.2)$$

Por fim, para determinar a cadeia de segmentos Γ, basta refazer, em ordem inversa, a sequência de escolhas realizadas pelo algoritmo para calcular $\max_k S(last(k), m, k)$.

Um algoritmo polinomial que resolve o PA*Sp* e que faz uso da Recorrência 3.1 e da Equação 3.2 pode ser desenvolvido através da técnica de programação dinâmica, onde associamos a função S a uma matriz tridimensional M tal que $M[i][j][k]$ armazene o valor de $S(i,j,k)$. Para determinar a complexidade de tempo deste algoritmo, observe que a quantidade de entradas de M que devem ser calculadas é $m * \sum_{k=1}^{u} size(k)$. Considerando $c = \frac{1}{n} * \sum_{k=1}^{u} size(k)$, esse algoritmo possui complexidade de tempo $O(m*n*c + m*u^2)$. Para mais detalhes sobre o Problema do Alinhamento *Spliced*, incluindo otimizações de tempo e de espaço no algoritmo proposto pelos autores, além de resultados da sua aplicação em um problema prático, sugerimos o artigo [17].

O PASG também pode ser visto como uma generalização do Problema do Encadeamento Bidimensional (*Two-dimensional Chain Problem*, do inglês) [19]. Informalmente, o **Problema do Encadeamento Bidimensional** recebe como entrada duas sequências s_1 e s_2, um conjunto B de segmentos de s_1, um conjunto C de segmentos de s_2 e um conjunto S que inclui os valores das similaridades globais, com respeito a uma função de pontuação ω, de alguns (ou eventualmente todos) pares de segmentos de B e de C, e busca determinar uma cadeia $\Gamma_B = \{b_1, b_2, ..., b_q\}$ de segmentos de B e uma cadeia $\Gamma_C = \{c_1, c_2, ..., c_q\}$ de segmentos de C, onde ambas as cadeias incluem a mesma quantidade de segmentos (q segmentos), tal que os segmentos que ocupam as mesmas posições nessas cadeias sejam bem parecidos. Em outras palavras, a similaridade de b_1 e c_1 deve ser alta, assim como a similaridade de b_2 e c_2, e assim por diante. Em termos formais, o objetivo do Problema do Encadeamento Bidimensional é encontrar duas cadeias de segmentos $\Gamma_B = \{b_1, b_2, ..., b_q\}$ de B e $\Gamma_C = \{c_1, c_2, ..., c_q\}$ de C tal que $\sum_{i=1}^{q} sim_\omega(b_i, c_i)$ é máxima entre todas as cadeias de segmentos de B e de C que possuem a mesma quantidade de segmentos. É importante notar que o valor de $sim_\omega(b_i, c_i)$, com $1 \leq i \leq q$, deve estar em S. Caso contrário, as cadeias de segmentos da resposta não devem incluir o par de segmentos (b_i, c_i). Para mais detalhes sobre o Problema do Encadeamento Bidimensional e de algoritmos eficientes que o resolve, além de outras variantes deste problema, sugerimos [19].

Um outro problema similar ao PASG foi pesquisado por Novichkov *et al.* em [29]. Nesse artigo,

os autores descrevem e resolvem um problema biológico que pode ser visto como uma variante do Problema do Alinhamento *Spliced*. Abstraindo-se os termos e conceitos biológicos, podemos associar o problema biológico resolvido em [29] ao seguinte problema computacional: dadas como entrada duas sequências s e t, determinar duas cadeias de segmentos Γ_B e Γ_C, onde Γ_B é uma cadeia de segmentos de s e Γ_C é uma cadeia de segmentos de t, tal que Γ_B e Γ_C sejam o mais similar possível. Apesar dos autores não definirem formalmente como verificar se duas cadeias de segmentos são o mais similar possível, assumiremos que isto significa que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C. Novichkov *et al.* em [29] apresentam um algoritmo para resolver esse problema. Uma observação importante desse algoritmo é que os conjuntos de segmentos de s e t, que incluirão, entre outros, os segmentos que irão compor Γ_B e Γ_C, não são passados como parte da entrada, mas são construídos no primeiro passo executado pelo algoritmo. A Recorrência 3.3 é uma equação adaptada de [29] para a resolução do problema computacional associado ao problema biológico resolvido no artigo. Na Recorrência 3.3, considere que $Acceptor(B) = \{first(b) - 1 : b \in B\}$, $Acceptor(C) = \{first(c) - 1 : c \in C\}$, $Donor(B) = \{last(b) + 1 : b \in B\}$, $Donor(C) = \{last(c) + 1 : c \in C\}$ e que ω é uma função de pontuação qualquer.

$$M[m][n] = max \begin{cases} M[m-1][n-1] + \omega(s[m], t[n]) \\ M[m][n-1] + \omega(-, t[n]) \\ M[m-1][n] + \omega(s[m], -) \\ max_{i<m \text{ e } i \in Donor(B)} \left\{ \begin{array}{l} M[i][n-1] \\ M[i][n] + \omega(s[m], -) \end{array} \right\} se\ m \in Acceptor(B) \\ max_{j<n \text{ e } j \in Donor(C)} \left\{ \begin{array}{l} M[m-1][j] \\ M[m][j] + \omega(-, t[n]) \end{array} \right\} se\ n \in Acceptor(C) \\ max_{i<m \text{ e } i \in Donor(B),\ j<n \text{ e } j \in Donor(C)}\{M[i][j]\},\ se\ m \in Acceptor(B)\ e \\ n \in Acceptor(C) \end{cases} \tag{3.3}$$

A Recorrência 3.3 deveria então determinar, dadas como entrada duas sequências s e t, um conjunto B de segmentos de s e um conjunto C de segmentos de t, duas cadeias de segmentos Γ_B e Γ_C, onde Γ_B é uma cadeia de segmentos de B e Γ_C é uma cadeia de segmentos de C, tal que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C. Observe então que a Recorrência 3.3 parece resolver o PASG, mas de fato isso não acontece, como mostramos a seguir. Podemos ver, na Recorrência 3.3, que as três primeiras linhas são consideradas opções válidas a todo momento no cálculo de $M[m, n]$ e elas correspondem à recorrência utilizada pelo algoritmo de Needleman-Wunsch para o cálculo do alinhamento ótimo de duas sequências, introduzido na Seção 2.5.1. Explorando esse fato, podemos identificar uma classe de instâncias nas quais a Recorrência 3.3 não devolve uma resposta correta. Essa classe consiste de instâncias em que as duas sequências s_1 e s_2 são idênticas, mas que não existem duas cadeias de segmentos Γ_B e Γ_C, de B e C respectivamente, tal que $\Gamma_B^\bullet = s_1$ e/ou $\Gamma_C^\bullet = s_2$. Considere, por exemplo, a instância da Figura 3.5 e uma função de pontuação $\omega(a, b)$ como definida a seguir:

$$\omega(a, b) = \begin{cases} 1, & \text{se } a = b \\ -1, & \text{se } a \neq b \\ -2, & \text{se } a = - \text{ ou } b = - \end{cases}.$$

Se aplicarmos a Recorrência 3.3 nesse exemplo, o valor 4 será calculado, mas não existe uma cadeia Γ_B de segmentos de B e uma cadeia Γ_C de segmentos de C tal que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet) = 4$. A solução ótima para esse exemplo, cujo valor é 1, consiste em Γ_B incluir um segmento qualquer de B e Γ_C incluir o único segmento de C.

Outro problema similar ao PASG, chamado de **SEA** (um acrônimo para *Segment Alignment*), foi definido e solucionado em [43]. Informalmente, o problema resolvido em [43] consiste em, dadas como

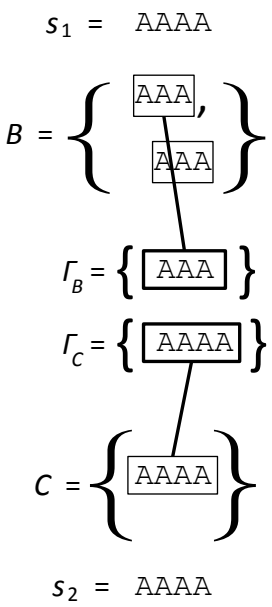


Figura 3.5: *Uma instância que mostra que a Recorrência 3.3 não resolve o PASG.*

entrada duas sequências s_1 e s_2, um conjunto B de segmentos de s_1 e um conjunto C de segmentos de s_2, determinar um alinhamento de s_1 e s_2 através da comparação de todas as combinações possíveis de segmentos. Os autores ainda consideram que deve existir ao menos um segmento contendo qualquer caractere das duas sequências de entrada. Para isso, segmentos virtuais são criados para conter os caracteres de uma sequência que não são cobertos pelos segmentos dados na entrada.

Sucintamente, a solução apresentada para o SEA em [43] busca por um alinhamento de s_1 e s_2 formado inteiramente por segmentos de B e de C, além de segmentos virtuais para os caracteres das sequências não cobertos pelos segmentos incluídos na solução. Dessa forma, se existe um trecho de uma das sequências de entrada que é coberto por apenas um segmento, esse segmento será incondicionalmente incluído na solução. A estratégia do algoritmo apresentado em [43] é determinar vários pares de cadeias de segmentos cujas sequências correspondentes sejam bem parecidas, e assim ir construindo o alinhamento de s_1 e s_2. Nesse sentido, apesar de apresentarem pontos em comuns, a formulação do SEA e do PASG são diferentes.

Capítulo 4

Solução algorítmica para o PASG

É importante observar que uma solução trivial para o PASG, baseada no método da força bruta, é computacionalmente inviável. Esta solução consistiria em comparar cada uma das possíveis cadeias de segmentos de B com todas as possíveis cadeias de segmentos de C, escolhendo, no final, aquele par de cadeias cujas sequências correspondentes apresentaram a maior similaridade. Um algoritmo baseado nessa ideia consome tempo $\Omega(2^u * 2^v)$ no pior caso, já que temos 2^u possíveis cadeias de segmentos de B e 2^v possíveis cadeias de segmentos de C, que precisam ser comparadas. Portanto, um algoritmo baseado no método da força bruta é exponencial e, consequentemente, ineficiente.

Neste capítulo apresentamos uma solução eficiente para o PASG, baseada na técnica de programação dinâmica. Descreveremos um algoritmo polinomial para o PASG seguindo os quatro passos para o desenvolvimento de um algoritmo de programação dinâmica propostos por Cormen *et al.* em [11], e apresentados na Seção 2.4.

4.1 Preliminares

Antes de apresentarmos um algoritmo eficiente para o PASG, introduziremos mais algumas definições e notações necessárias à sua compreensão, além daquelas apresentadas na Seção 2.1. Seja b_k um segmento qualquer de um conjunto ordenado B de segmentos de uma sequência s_1 e c_l um segmento qualquer de um conjunto ordenado C de segmentos de uma sequência s_2. Denotaremos por $\Gamma^{\star}_{b_k}$ e por $\Gamma^{\star}_{c_l}$ duas cadeias de segmentos, de $\Delta(b_k)$ e $\Delta(c_l)$ respectivamente, tal que $sim_{\omega}(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j]) \geq sim_{\omega}(\Gamma^{\bullet}_{b_k} \bullet b_k[1..i], \Gamma^{\bullet}_{c_l} \bullet c_l[1..j])$, para quaisquer outras cadeias de segmentos $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$. Ou seja, $\Gamma^{\star}_{b_k}$ e $\Gamma^{\star}_{c_l}$ correspondem a cadeias de segmentos de $\Delta(b_k)$ e $\Delta(c_l)$ cujas sequências correspondentes, quando concatenadas com $b_k[1..i]$ e $c_l[1..j]$ respectivamente, possuem a maior similaridade entre todos os possíveis pares de cadeias de segmentos de $\Delta(b_k)$ e $\Delta(c_l)$. Por questões de simplicidade, denotaremos $\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i]$ simplesmente por $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j]$ simplesmente por $\Gamma^{\star\bullet}_{c_l}(j)$. Finalmente, definimos $\Gamma^{\star\bullet}_{b_0}(0) = \epsilon$ e $\Gamma^{\star\bullet}_{c_0}(0) = \epsilon$.

Com base nas definições e notações acima, denotamos por $Opt(i, j, k, l)$, com $1 \leq i \leq |b_k|$, $1 \leq j \leq |c_l|$, $0 \leq k \leq u$ e $0 \leq l \leq v$, a similaridade de $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_l}(j)$. Informalmente, $Opt(i, j, k, l)$ corresponde à similaridade das sequências correspondentes a duas cadeias de segmentos, uma de B e outra de C, tal que o último elemento da cadeia de B é o segmento b_k, até a posição i, e o último elemento da cadeia de C é o segmento c_l, até a posição j. Além disso, esta similaridade é máxima entre todas as possíveis cadeias de segmentos de B e de C, que terminam com o segmento b_k até a posição i, e com o segmento c_l até a posição j, respectivamente.

É importante observar que de acordo com a definição de $Opt()$, podemos ter $Opt(i, j, k, l)$, onde $k = 0$ e/ou $l = 0$. $Opt(0, 0, 0, 0)$ denota a similaridade de $\Gamma^{\star\bullet}_{b_0}(0)$ e $\Gamma^{\star\bullet}_{c_0}(0)$. Logo, $Opt(0, 0, 0, 0)$ define a similaridade de ϵ e ϵ, o que chamamos de **solução vazia** e que possui valor 0. Lembramos que a solução vazia sempre está disponível para qualquer instância do PASG, o que significa que qualquer solução para esse problema possuirá valor maior ou igual a 0. Nos casos em que $k = 0$ e $l > 0$, $Opt(0, j, 0, l)$, com $1 \leq j \leq |c_l|$, denota a similaridade de $\Gamma^{\star\bullet}_{b_0}(0)$ e $\Gamma^{\star\bullet}_{c_l}(j)$, ou seja, a similaridade de ϵ e $\Gamma^{\star\bullet}_{c_l}(j)$. Os casos em que $k > 0$ e $l = 0$, $Opt(i, 0, k, 0)$, com $1 \leq i \leq |b_k|$, denota a similaridade de

$\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_0}(0)$, ou seja, a similaridade de $\Gamma^{\star\bullet}_{b_k}(i)$ e ϵ. As outras combinações de $Opt(i,j,k,l)$, com $k=0$ e/ou $l=0$, não estão definidas.

Com base na definição de $Opt()$, se conseguirmos calcular $Opt(i,j,k,l)$, para todos os valores possíveis de i, j, k e l, somos capazes de determinar uma cadeia $\Gamma_B = \{b_p, b_q, ..., b_r\}$ de segmentos de B e uma cadeia $\Gamma_C = \{c_w, c_x, ..., c_y\}$ de segmentos de C tal que $sim_\omega(\Gamma^\bullet_B, \Gamma^\bullet_C)$ é máxima entre todas as cadeias de segmentos de B e de C, resolvendo assim o PASG. A construção de Γ_B e Γ_C é feita no quarto passo da programação dinâmica. Nos três primeiros passos, focamos nossa atenção em como determinar $Opt(i,j,k,l)$ de forma eficiente, para todos os valores possíveis de i, j, k e l.

4.2 Subestrutura ótima do PASG

O primeiro passo para determinar $Opt(i,j,k,l)$ de forma eficiente utilizando programação dinâmica é identificar a subestrutura ótima do PASG. Essa subestrutura é dada pelo Lema 4.2.1 a seguir[1].

Lema 4.2.1 *Sejam s_1 e s_2 duas sequências sobre um alfabeto Σ qualquer, tal que $'-' \notin \Sigma$. Seja $B = \{b_1, b_2, ..., b_u\}$ um conjunto ordenado de segmentos de s_1, $C = \{c_1, c_2, ..., c_v\}$ um conjunto ordenado de segmentos de s_2, e ω uma função de pontuação sobre $\overline{\Sigma}$. Seja $\Gamma^\star_{b_k} \in \Delta(b_k)$ e $\Gamma^\star_{c_l} \in \Delta(c_l)$ um par de cadeias de segmentos tal que $sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j]) \geq sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i], \Gamma^\bullet_{c_l} \bullet c_l[1..j])$, para quaisquer outros pares $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$, onde $1 \leq i \leq |b_k|$, $1 \leq j \leq |c_l|$, $1 \leq k \leq u$ e $1 \leq l \leq v$. Considere $\Gamma^{\star\bullet}_{b_k}(i) = \Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i]$ e $\Gamma^{\star\bullet}_{c_l}(j) = \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j]$. Seja $A = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}, \overline{\Gamma^{\star\bullet}_{c_l}(j)})$ um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_l}(j)$. Definimos $\overline{n} = |\overline{\Gamma^{\star\bullet}_{b_k}(i)}|$ e $\overline{m} = |\overline{\Gamma^{\star\bullet}_{c_l}(j)}|$.*

1. *Se $i > 1$ e $j > 1$:*
   - (a) *se $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] \neq '-'$ e $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] \neq '-'$, então*
     - i. *não existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i-1], \Gamma^\bullet_{c_l} \bullet c_l[1..j-1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i-1], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j-1])$;*
     - ii. *$A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ é um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i-1)$ e $\Gamma^{\star\bullet}_{c_l}(j-1)$.*
   - (b) *se $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] = '-'$ e $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] \neq '-'$, então*
     - i. *não existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i], \Gamma^\bullet_{c_l} \bullet c_l[1..j-1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j-1])$;*
     - ii. *$A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ é um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_l}(j-1)$.*
   - (c) *se $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] \neq '-'$ e $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] = '-'$, então*
     - i. *não existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i-1], \Gamma^\bullet_{c_l} \bullet c_l[1..j]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i-1], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j])$;*
     - ii. *$A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ é um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i-1)$ e $\Gamma^{\star\bullet}_{c_l}(j)$.*
2. *Se $i = 1$ e $j = 1$:*
   - (a) *se $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] \neq '-'$ e $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] \neq '-'$, então*
     - i. *não existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^\bullet_{b_k}, \Gamma^\bullet_{c_l}) > sim_\omega(\Gamma^{\star\bullet}_{b_k}, \Gamma^{\star\bullet}_{c_l})$;*
     - ii. *$A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ é um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}$ e $\Gamma^{\star\bullet}_{c_l}$.*
   - (b) *se $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] = '-'$ e $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] \neq '-'$, então*

[1] O Lema 4.2.1 poderia ser reescrito de maneira mais sucinta, sem perda de informações, mas ele foi propositalmente escrito da forma como se encontra no texto para facilitar sua tradução em uma recorrência posteriormente.

   i. *não existe* $\Gamma_{b_k} \in \Delta(b_k)$ *e* $\Gamma_{c_l} \in \Delta(c_l)$ *tal que* $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1], \Gamma^\bullet_{c_l}) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1], \Gamma^{\star\bullet}_{c_l})$;
   ii. $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $\Gamma^{\star\bullet}_{b_k}(1)$ *e* $\Gamma^{\star\bullet}_{c_l}$.

(c) *se* $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] \neq\ '-'$ *e* $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] =\ '-'$, *então*
   i. *não existe* $\Gamma_{b_k} \in \Delta(b_k)$ *e* $\Gamma_{c_l} \in \Delta(c_l)$ *tal que* $sim_\omega(\Gamma^\bullet_{b_k}, \Gamma^\bullet_{c_l} \bullet c_l[1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k}, \Gamma^{\star\bullet}_{c_l} \bullet c_l[1])$;
   ii. $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $\Gamma^{\star\bullet}_{b_k}$ *e* $\Gamma^{\star\bullet}_{c_l}(1)$.

3. *Se* $i = 1$ *e* $j > 1$:

(a) *se* $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] \neq\ '-'$ *e* $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] \neq\ '-'$, *então*
   i. *não existe* $\Gamma_{b_k} \in \Delta(b_k)$ *e* $\Gamma_{c_l} \in \Delta(c_l)$ *tal que* $sim_\omega(\Gamma^\bullet_{b_k}, \Gamma^\bullet_{c_l} \bullet c_l[1..j-1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k}, \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j-1])$;
   ii. $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $\Gamma^{\star\bullet}_{b_k}$ *e* $\Gamma^{\star\bullet}_{c_l}(j-1)$.

(b) *se* $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] =\ '-'$ *e* $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] \neq\ '-'$, *então*
   i. *não existe* $\Gamma_{b_k} \in \Delta(b_k)$ *e* $\Gamma_{c_l} \in \Delta(c_l)$ *tal que* $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1], \Gamma^\bullet_{c_l} \bullet c_l[1..j-1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j-1])$
   ii. $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $\Gamma^{\star\bullet}_{b_k}(1)$ *e* $\Gamma^{\star\bullet}_{c_l}(j-1)$.

(c) *se* $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] \neq\ '-'$ *e* $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] =\ '-'$, *então*
   i. *não existe* $\Gamma_{b_k} \in \Delta(b_k)$ *e* $\Gamma_{c_l} \in \Delta(c_l)$ *tal que* $sim_\omega(\Gamma^\bullet_{b_k}, \Gamma^\bullet_{c_l} \bullet c_l[1..j]) > sim_\omega(\Gamma^{\star\bullet}_{b_k}, \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j])$;
   ii. $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $\Gamma^{\star\bullet}_{b_k}$ *e* $\Gamma^{\star\bullet}_{c_l}(j)$.

4. *Se* $i > 1$ *e* $j = 1$:

(a) *se* $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] \neq\ '-'$ *e* $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] \neq\ '-'$, *então*
   i. *não existe* $\Gamma_{b_k} \in \Delta(b_k)$ *e* $\Gamma_{c_l} \in \Delta(c_l)$ *tal que* $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i-1], \Gamma^\bullet_{c_l}) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i-1], \Gamma^{\star\bullet}_{c_l})$;
   ii. $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $\Gamma^{\star\bullet}_{b_k}(i-1)$ *e* $\Gamma^{\star\bullet}_{c_l}$.

(b) *se* $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] =\ '-'$ *e* $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] \neq\ '-'$, *então*
   i. *não existe* $\Gamma_{b_k} \in \Delta(b_k)$ *e* $\Gamma_{c_l} \in \Delta(c_l)$ *tal que* $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i], \Gamma^\bullet_{c_l}) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i], \Gamma^{\star\bullet}_{c_l})$;
   ii. $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $\Gamma^{\star\bullet}_{b_k}(i)$ *e* $\Gamma^{\star\bullet}_{c_l}$.

(c) *se* $\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}] \neq\ '-'$ *e* $\overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}] =\ '-'$, *então*
   i. *não existe* $\Gamma_{b_k} \in \Delta(b_k)$ *e* $\Gamma_{c_l} \in \Delta(c_l)$ *tal que* $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i-1], \Gamma^\bullet_{c_l} \bullet c_l[1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i-1], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1])$;
   ii. $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ *é um alinhamento ótimo de* $\Gamma^{\star\bullet}_{b_k}(i-1)$ *e* $\Gamma^{\star\bullet}_{c_l}(1)$.

Prova (1.a) (i) Suponha, por contradição, que existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i-1], \Gamma^\bullet_{c_l} \bullet c_l[1..j-1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i-1], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j-1])$. Seja $A'' = (\overline{\Gamma^\bullet_{b_k}(i-1)}, \overline{\Gamma^\bullet_{c_l}(j-1)})$ um alinhamento ótimo de $\Gamma^\bullet_{b_k} \bullet b_k[1..i-1]$ e $\Gamma^\bullet_{c_l} \bullet c_l[1..j-1]$, ou seja, $Score(A'') = sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i-1], \Gamma^\bullet_{c_l} \bullet c_l[1..j-1])$. Se adicionarmos $b_k[i]$ a $\overline{\Gamma^\bullet_{b_k}(i-1)}$ e $c_l[j]$ a $\overline{\Gamma^\bullet_{c_l}(j-1)}$, obteremos um alinhamento $A''' = (\overline{\Gamma^\bullet_{b_k}(i-1)} \bullet b_k[i], \overline{\Gamma^\bullet_{c_l}(j-1)} \bullet c_l[j])$, que é um alinhamento de $\Gamma^\bullet_{b_k} \bullet b_k[1..i]$ e $\Gamma^\bullet_{c_l} \bullet c_l[1..j]$, tal que $Score_\omega(A''') = sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i-1], \Gamma^\bullet_{c_l} \bullet c_l[1..j-1]) + \omega(b_k[i], c_l[j]) > Score_\omega(A) = sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i-1], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j-1]) + \omega(b_k[i], c_l[j])$. Ou seja, $sim_\omega(\Gamma^\bullet_{b_k} \bullet b_k[1..i], \Gamma^\bullet_{c_l} \bullet c_l[1..j]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j])$, o que é uma contradição da nossa hipótese.

(ii) Suponha, por contradição, que $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ não é um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i-1)$ e $\Gamma^{\star\bullet}_{c_l}(j-1)$. Logo, existe um alinhamento $A'' = (\overline{b}, \overline{c})$ de $\Gamma^{\star\bullet}_{b_k}(i-1)$ e $\Gamma^{\star\bullet}_{c_l}(j-1)$ tal que $Score_\omega(A'') > Score_\omega(A')$. Se adicionarmos $b_k[i]$ a $\overline{b}$ e $c_l[j]$ a $\overline{c}$, obteremos o alinhamento $A''' = (\overline{b} \bullet b_k[i], \overline{c} \bullet c_l[j])$, que é um alinhamento de $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_l}(j)$, tal que $Score_\omega(A''') = Score_\omega(A'') + \omega(b_k[i], c_l[j]) > Score_\omega(A) = Score_\omega(A') + \omega(b_k[i], c_l[j])$, o que é uma contradição da nossa hipótese de que A é um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_l}(j)$.

As provas de todos os outros itens do Lema 4.2.1 são similares às duas acima.

■

De acordo com o Lema 4.2.1, um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_l}(j)$ contém, em seu interior, alinhamentos ótimos de $\Gamma^{\star\bullet}_{b_k}$ concatenado com um prefixo de $b_k[1..i]$ e $\Gamma^{\star\bullet}_{c_l}$ concatenado com um prefixo de $c_l[1..j]$. Consequentemente, isto mostra que uma solução ótima para uma instância qualquer do PASG contém, em seu interior, soluções ótimas para subproblemas desta instância, demonstrando a propriedade de subestrutura ótima.

4.3 Solução recursiva para o PASG

Neste segundo passo, iremos propor uma solução recursiva, na forma de uma recorrência, para o PASG. Essa recorrência calcula $Opt(i, j, k, l)$ para todos os valores possíveis de i, j, k e l, e estabelece um relacionamento recursivo entre o valor de $Opt(i, j, k, l)$ e o valor de $Opt(i', j', k', l')$, tal que o cálculo de $Opt(i', j', k', l')$ é um subproblema que deve ser resolvido para calcularmos $Opt(i, j, k, l)$.

Começaremos a descrição da solução recursiva para o PASG identificando suas condições base e, para isso, estamos supondo que $\omega(a, b) < 0$, se $a = -$ ou $b = -$. Para o PASG, as condições base são as seguintes:

$$Opt(0, 0, 0, 0) = 0. \tag{4.1}$$

$$Opt(0, j, 0, l) = \sum_{t=1}^{j} \omega(-, c_l[t]), \text{com } 1 \leq j \leq |c_l| \text{ e } 1 \leq l \leq v. \tag{4.2}$$

$$Opt(i, 0, k, 0) = \sum_{t=1}^{i} \omega(b_k[t], -), \text{com } 1 \leq i \leq |b_k| \text{ e } 1 \leq k \leq u. \tag{4.3}$$

Para facilitar a compreensão das condições base, suponha a existência de dois segmentos virtuais $b_0 \in B$ e $c_0 \in C$, com $b_0 = c_0 = \epsilon$ e $|b_0| = |c_0| = 0$, tal que para todo segmento $b \in B$ e $c \in C$, $b_0 \prec b$ e $c_0 \prec c$. A similaridade de $\Gamma^{\star\bullet}_{b_0}(0)$ e $\Gamma^{\star\bullet}_{c_0}(0)$, ou da solução vazia, denotada por $Opt(0, 0, 0, 0)$ (condição base 4.1) é igual a 0 (a similaridade de ϵ e ϵ). A similaridade de $\Gamma^{\star\bullet}_{b_0}(0)$ e $\Gamma^{\star\bullet}_{c_l}(j)$, com $1 \leq j \leq |c_l|$ e $1 \leq l \leq v$, denotada por $Opt(0, j, 0, l)$ (condição base 4.2), é igual a $\sum_{t=1}^{j} \omega(-, c_l[t])$. A intuição por trás dessa igualdade está no fato de que a melhor maneira de alinhar $\Gamma^{\star\bullet}_{b_0}(0) = \epsilon$ com $\Gamma^{\star\bullet}_{c_l}(j)$ é alinhar ϵ somente com $c_l[1..j]$ (a adição de qualquer outro segmento $c_{l'} \prec c_l$ a $\Gamma^{\star}_{c_l}(j)$ irá aumentar a quantidade de espaços no alinhamento e, consequentemente, irá diminuir sua pontuação). A similaridade de $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_0}(0)$, com $1 \leq i \leq |b_k|$ e $1 \leq k \leq u$, denotada por $Opt(i, 0, k, 0)$ (condição base 4.3), é igual a $\sum_{t=1}^{i} \omega(b_k[t], -)$. A intuição por trás dessa igualdade é análoga à anterior.

Para transformar a subestrutura ótima identificada no Lema 4.2.1 em uma recorrência, considere os Teoremas 4.3.1, 4.3.2, 4.3.3, 4.3.4 e o Corolário 4.3.5.

Teorema 4.3.1 *Se $i > 1$ e $j > 1$, $Opt(i, j, k, l)$ é igual a um dos seguintes valores:*

1. $Opt(i-1, j-1, k, l) + \omega(b_k[i], c_l[j])$;
2. $Opt(i, j-1, k, l) + \omega(-, c_l[j])$;
3. $Opt(i-1, j, k, l) + \omega(b_k[i], -)$.

Prova Considerando o Lema 4.2.1, a última coluna do alinhamento $A = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}, \overline{\Gamma^{\star\bullet}_{c_l}(j)})$, denotada por $(\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}])$, deve ser uma das três seguintes opções: (1) o caractere $b_k[i]$ alinhado com o caractere $c_l[j]$, ou (2) um espaço alinhado com o caractere $c_l[j]$ ou (3) o caractere $b_k[i]$ alinhado com um espaço.

1. Na primeira opção, $b_k[i]$ está alinhado com $c_l[j]$. Pelo Lema 4.2.1, temos que $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ é um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i-1)$ e $\Gamma^{\star\bullet}_{c_l}(j-1)$. Como não existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^{\bullet}_{b_k} \bullet b_k[1..i-1], \Gamma^{\bullet}_{c_l} \bullet c_l[1..j-1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i-1], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j-1])$, temos que a primeira opção possui valor $Opt(i,j,k,l) = Score(A) = Score(A') + \omega(b_k[i], c_l[j]) = Opt(i-1, j-1, k, l) + \omega(b_k[i], c_l[j])$;

2. Na segunda opção, um espaço está alinhado com $c_l[j]$. Pelo Lema 4.2.1, temos que $A' = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}[1..\overline{n}-1], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[1..\overline{m}-1])$ é um alinhamento ótimo de $\Gamma^{\star\bullet}_{b_k}(i)$ e $\Gamma^{\star\bullet}_{c_l}(j-1)$. Como não existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^{\bullet}_{b_k} \bullet b_k[1..i], \Gamma^{\bullet}_{c_l} \bullet c_l[1..j-1]) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1..i], \Gamma^{\star\bullet}_{c_l} \bullet c_l[1..j-1])$, temos que a primeira opção possui valor $Opt(i,j,k,l) = Score(A) = Score(A') + \omega(-, c_l[j]) = Opt(i, j-1, k, l) + \omega(-, c_l[j])$;

3. A terceira opção é análoga à anterior e possui valor $Opt(i,j,k,l) = Opt(i-1, j, k, l) + \omega(b_k[i], -)$.

Como, por definição, dois espaços não podem ser alinhados, nós cobrimos todas as opções no caso em que $i > 1$ e $j > 1$.

■

Teorema 4.3.2 *Se $i = 1$ e $j = 1$, $Opt(i,j,k,l)$ é igual a um dos seguintes valores:*

1. $max_{b_{k'} \prec b_k,\ c_{l'} \prec c_l}\{Opt(|b_{k'}|, |c_{l'}|, k', l') + \omega(b_k[1], c_l[1])\}$;
2. $max_{c_{l'} \prec c_l}\{Opt(1, |c_{l'}|, k, l') + \omega(-, c_l[1])\}$;
3. $max_{b_{k'} \prec b_k}\{Opt(|b_{k'}|, 1, k', l) + \omega(b_k[1], -)\}$.

Prova Considerando o Lema 4.2.1, a última coluna do alinhamento $A = (\overline{\Gamma^{\star\bullet}_{b_k}(i)}, \overline{\Gamma^{\star\bullet}_{c_l}(j)})$, denotada por $(\overline{\Gamma^{\star\bullet}_{b_k}(i)}[\overline{n}], \overline{\Gamma^{\star\bullet}_{c_l}(j)}[\overline{m}])$, deve ser uma das três seguintes opções: (1) o caractere $b_k[1]$ alinhado com o caractere $c_l[1]$, ou (2) um espaço alinhado com o caractere $c_l[1]$ ou (3) o caractere $b_k[1]$ alinhado com um espaço.

1. Na primeira opção, $b_k[1]$ está alinhado com $c_l[1]$. Pelo Lema 4.2.1, temos que não existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^{\bullet}_{b_k}, \Gamma^{\bullet}_{c_l}) > sim_\omega(\Gamma^{\star\bullet}_{b_k}, \Gamma^{\star\bullet}_{c_l})$. Note que o par de cadeias de segmentos $(\Gamma^{\star}_{b_k}, \Gamma^{\star}_{c_l})$ é uma solução para o subproblema onde $B = before(b_k)$ e $C = before(c_l)$, ou seja, $\Gamma^{\star}_{b_k}$ é uma cadeia de segmentos de $before(b_k)$ e $\Gamma^{\star}_{c_l}$ é uma cadeia de segmentos de $before(c_l)$ tal que $sim_\omega(\Gamma^{\star\bullet}_{b_k}, \Gamma^{\star\bullet}_{c_l})$ é máxima entre todas as cadeias de segmentos de $before(b_k)$ e de $before(c_l)$. Considerando que o último segmento de $\Gamma^{\star}_{b_k}$ é $b_{k'}$ e o último segmento de $\Gamma^{\star}_{c_l}$ é $c_{l'}$, podemos calcular $sim_\omega(\Gamma^{\star\bullet}_{b_k}, \Gamma^{\star\bullet}_{c_l})$ tomando todos os pares de segmentos $(b_{k'} \in before(b_k), c_{l'} \in before(c_l))$ e calculando $max\{Opt(|b_{k'}|, |c_{l'}|, k', l')\}$. Observe que $b_{k'}$ pode ser b_0, assim como $c_{l'}$ pode ser c_0. Portanto, a primeira opção possui valor $Opt(i,j,k,l) = max_{b_{k'} \prec b_k,\ c_{l'} \prec c_l}\{Opt(|b_{k'}|, |c_{l'}|, k', l') + \omega(b_k[1], c_l[1])\}$.

2. Na segunda opção, um espaço está alinhado com $c_l[1]$. Pelo Lema 4.2.1, temos que não existe $\Gamma_{b_k} \in \Delta(b_k)$ e $\Gamma_{c_l} \in \Delta(c_l)$ tal que $sim_\omega(\Gamma^{\bullet}_{b_k} \bullet b_k[1], \Gamma^{\bullet}_{c_l}) > sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1], \Gamma^{\star\bullet}_{c_l})$. Considerando que o último segmento de $\Gamma^{\star}_{c_l}$ é $c_{l'}$, podemos calcular $sim_\omega(\Gamma^{\star\bullet}_{b_k} \bullet b_k[1], \Gamma^{\star\bullet}_{c_l})$ tomando todos os segmentos $c_{l'} \in before(c_l)$ e calculando $max\{Opt(1, |c_{l'}|, k, l')\}$. Observe que $c_{l'}$ pode ser c_0. Portanto, a segunda opção possui valor $Opt(i,j,k,l) = max_{c_{l'} \prec c_l}\{\ Opt(1, |c_{l'}|, k, l') + \omega(-, c_l[1])\}$.

3. A terceira opção é análoga à anterior e possui valor $Opt(i,j,k,l) = max_{b_{k'} \prec b_k}\{Opt(|b_{k'}|, 1, k', l) + \omega(b_k[1], -)\}$.

Como, por definição, dois espaços não podem ser alinhados, nós cobrimos todas as opções no caso em que $i = 1$ e $j = 1$.

■

Teorema 4.3.3 *Se $i = 1$ e $j > 1$, $Opt(i,j,k,l)$ é igual a um dos seguintes valores:*

1. $max_{b_{k'} \prec b_k}\{Opt(|b_{k'}|, j-1, k', l) + \omega(b_k[1], c_l[j])\}$;
2. $Opt(1, j-1, k, l) + \omega(-, c_l[j])$;
3. $max_{b_{k'} \prec b_k}\{Opt(|b_{k'}|, j, k', l) + \omega(b_k[1], -)\}$.

Prova As provas da primeira e da terceira opções são análogas à prova da terceira opção do Teorema 4.3.2 e a prova da segunda opção é análoga à prova da segunda opção do Teorema 4.3.1.

■

Teorema 4.3.4 *Se $i > 1$ e $j = 1$, $Opt(i,j,k,l)$ é igual a um dos seguintes valores:*

1. $max_{c_{l'} \prec c_l}\{Opt(i-1, |c_{l'}|, k, l') + \omega(b_k[i], c_l[1])\}$;
2. $max_{c_{l'} \prec c_l}\{Opt(i, |c_{l'}|, k, l') + \omega(-, c_l[1])\}$;
3. $Opt(i-1, 1, k, l) + \omega(b_k[i], -)$.

Prova As provas da primeira e da segunda opções são análogas à prova da segunda opção do Teorema 4.3.2 e a prova da terceira opção é análoga à prova da terceira opção do Teorema 4.3.1.

■

Corolário 4.3.5 *$Opt(i,j,k,l)$ deve ser igual ao maior dos valores disponíveis.*

Baseando-se nos Teoremas 4.3.1, 4.3.2, 4.3.3, 4.3.4 e no Corolário 4.3.5, podemos derivar a recorrência que calcula $Opt(i,j,k,l)$, para $1 \leq i \leq |b_k|$, $1 \leq j \leq |c_l|$, $1 \leq k \leq u$ e $1 \leq l \leq v$, apresentada a seguir:

$$
Opt(i,j,k,l) = \begin{cases}
max \left\{ \begin{array}{l} Opt(i-1, j-1, k, l) + \omega(b_k[i], c_l[j]) \\ Opt(i, j-1, k, l) + \omega(-, c_l[j]) \\ Opt(i-1, j, k, l) + \omega(b_k[i], -) \end{array} \right\} se\ i > 1\ e\ j > 1 \\
max \left\{ \begin{array}{l} max_{b_{k'} \prec b_k,\ c_{l'} \prec c_l}\{Opt(|b_{k'}|, |c_{l'}|, k', l') + \omega(b_k[1], c_l[1])\} \\ max_{c_{l'} \prec c_l}\{Opt(1, |c_{l'}|, k, l') + \omega(-, c_l[1])\} \\ max_{b_{k'} \prec b_k}\{Opt(|b_{k'}|, 1, k', l) + \omega(b_k[1], -)\} \end{array} \right\} se\ i = 1\ e\ j = 1 \\
max \left\{ \begin{array}{l} Opt(1, j-1, k, l) + \omega(-, c_l[j]) \\ max_{b_{k'} \prec b_k} \left\{ \begin{array}{l} Opt(|b_{k'}|, j-1, k', l) + \omega(b_k[1], c_l[j]) \\ Opt(|b_{k'}|, j, k', l) + \omega(b_k[1], -) \end{array} \right. \end{array} \right\} se\ i = 1\ e\ j > 1 \\
max \left\{ \begin{array}{l} Opt(i-1, 1, k, l) + \omega(b_k[i], -) \\ max_{c_{l'} \prec c_l} \left\{ \begin{array}{l} Opt(i-1, |c_{l'}|, k, l') + \omega(b_k[i], c_l[1]) \\ Opt(i, |c_{l'}|, k, l') + \omega(-, c_l[1]) \end{array} \right. \end{array} \right\} se\ i > 1\ e\ j = 1
\end{cases}
\tag{4.4}
$$

Por fim, após calcularmos $Opt(i,j,k,l)$ para todos os valores possíveis de i, j, k e l utilizando a Recorrência 4.4, é possível calcular o valor de $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$, onde Γ_B é uma cadeia de segmentos

de B e Γ_C é uma cadeia de segmentos de C, tal que essa similaridade é máxima entre todas as cadeias de segmentos de B e de C. Isto é feito observando que o último segmento de Γ_B é algum segmento b_k e o último segmento de Γ_C é algum segmento c_l e então consideramos todos os pares de segmentos $(b_k \in B, c_l \in C)$ e calculamos $max\{Opt(|b_k|, |c_l|, k, l)\}$. Observe que b_k pode ser b_0, assim como c_l pode ser c_0. Portanto, o valor de $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é:

$$max_{b_k \in B,\ c_l \in C}\{Opt(|b_k|, |c_l|, k, l)\}. \tag{4.5}$$

4.4 Calculando $Opt(i, j, k, l)$ eficientemente

Com base na Recorrência 4.4, na Equação 4.5 e nas condições base 4.1, 4.2 e 4.3, é possível desenvolver facilmente um algoritmo recursivo para calcular o valor de $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$, onde Γ_B é uma cadeia de segmentos de B e Γ_C é uma cadeia de segmentos de C, tal que esta similaridade é máxima entre todas as cadeias de segmentos de B e de C. Contudo, o número total de chamadas recursivas feitas durante a execução desse algoritmo cresce exponencialmente com u, v e o tamanho dos segmentos de B e C. Portanto, esse algoritmo recursivo seria ineficiente.

Se observarmos melhor o espaço de subproblemas gerados durante a execução do algoritmo recursivo, podemos perceber que existem exatamente $1+\sum_{b\in B}|b|+\sum_{c\in C}|c|$ valores de $Opt(i, j, k, l)$ que devem ser calculados, quando $k = 0$ e/ou $l = 0$ (condições base), e $\sum_{b\in B,\ c\in C}|b| * |c|$, quando $k > 0$ e $l > 0$. De modo grosseiro, podemos dizer que existem $O(u * v * bMax * cMax)$ valores de $Opt(i, j, k, l)$ que devem ser calculados, onde $bMax = max\{|b| : b \in B\}$ e $cMax = max\{|c| : c \in C\}$. Isto significa que o número total de subproblemas distintos gerados durante a execução do algoritmo recursivo é limitado por um polinômio no tamanho da entrada. Sendo assim, e observando o fato de que a quantidade de subproblemas resolvidos pelo algoritmo recursivo é exponencial, podemos concluir que ele resolve os mesmos subproblemas repetidas vezes. Isso mostra que o PASG apresenta a característica de subproblemas sobrepostos. Essa é a segunda característica fundamental de um problema em que a programação dinâmica pode ser aplicada.

Neste terceiro passo, apresentamos um algoritmo de programação dinâmica que resolve cada subproblema apenas na primeira vez que o encontra, e então guarda o valor da solução em uma matriz, que será consultada nos próximos momentos que o subproblema for encontrado novamente. Fazendo isso, obtemos um algoritmo polinomial para o problema.

Organizaremos os cálculos de $Opt(i, j, k, l)$ em uma matriz quadridimensional M, que chamaremos de matriz de alinhamento de segmentos M, de tal forma que o valor $Opt(i, j, k, l)$ esteja armazenado em $M[i][j][k][l]$. Uma forma de visualizar M é imaginá-la como sendo composta por $(u+1)*(v+1)$ submatrizes de alinhamento. Cada submatriz de alinhamento é uma matriz bidimensional que armazena os valores de $Opt(i, j, k', l')$ para um dado k' e um dado l'. Ou seja, a submatriz de alinhamento localizada na linha k' e na coluna l' de M (ou simplesmente a submatriz de alinhamento $[k'][l']$) armazenará as células $M[i][j][k'][l']$, com $1 \leq i \leq |b_{k'}|$ e $1 \leq j \leq |c_{l'}|$. A Figura 4.1 ilustra esta forma de visualizar a matriz de alinhamento de segmentos M. O espaço consumido por M é $O(u * v * bMax * cMax)$, onde $bMax = max\{|b| : b \in B\}$ e $cMax = max\{|c| : c \in C\}$[2].

Adotando-se a estratégia de baixo para cima, primeiramente devemos calcular $Opt(i, j, k, l)$ para os menores valores possíveis de k e l e então calcular $Opt(i, j, k, l)$ para valores crescentes de k e l. Isto significa que particionaremos o cálculo de $Opt(i, j, k, l)$, calculando uma submatriz de alinhamento de M por vez. Esse processamento se dará preenchendo todas as $(u + 1) * (v + 1)$ submatrizes de alinhamento uma linha por vez, em ordem crescente. Para cada linha, as submatrizes são preenchidas em ordem crescente das colunas de M. Dentro de cada submatriz, a estratégia é similar. As células de cada submatriz de alinhamento são preenchidas linha a linha, crescentemente. Em cada linha, as células são preenchidas em ordem crescente das colunas. Assim, os valores necessários para preencher $M[i][j][k][l]$ já estarão disponíveis quando esta célula for alcançada. Perceba que as submatrizes

[2] O espaço exato consumido por M é $1 + \sum_{b\in B}|b| + \sum_{c\in C}|c| + \sum_{b\in B,\ c\in C}|b| * |c|$ células (a mesma quantidade de valores de $Opt(i, j, k, l)$ que devem ser calculados).

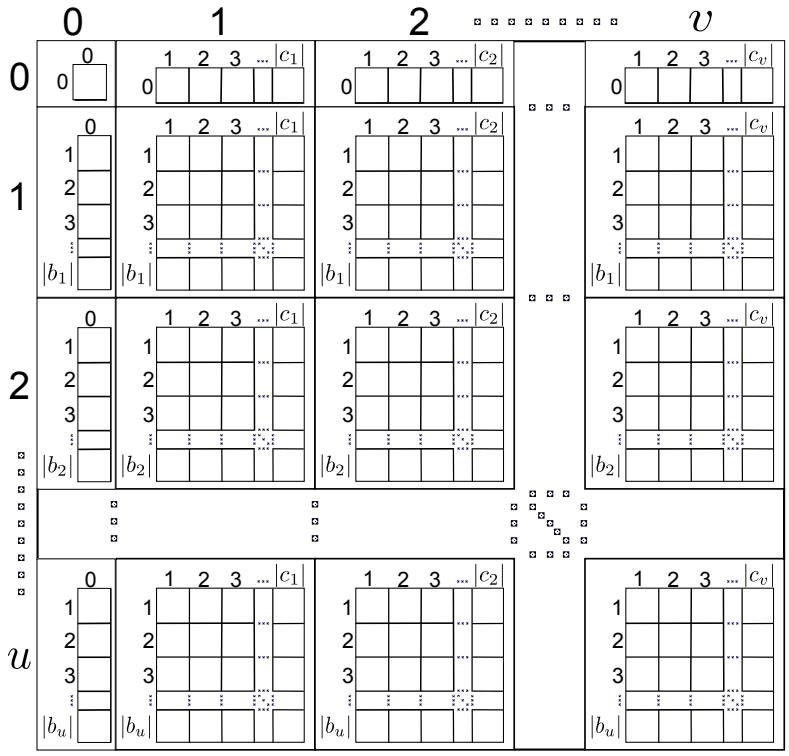


Figura 4.1: *Representação da matriz de alinhamento de segmentos quadridimensional M utilizando uma combinação de submatrizes de alinhamento bidimensionais.*

de alinhamento da linha 0 e/ou da coluna 0 de M representam as condições base e podem ser preenchidas diretamente.

Um algoritmo que segue essa estratégia de baixo para cima e que calcula o valor de $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ baseando-se nas condições base 4.1, 4.2 e 4.3 e nas Equações 4.4 e 4.5 está especificado no Algoritmo 4. Observe que no Algoritmo 4 utilizamos adicionalmente uma matriz de apontadores P com as mesmas dimensões da matriz de alinhamento de segmentos M. Dadas duas células $M[i][j][k][l]$ e $M[i'][j'][k'][l']$ de M, se $M[i'][j'][k'][l']$ foi selecionada e usada para calcular $M[i][j][k][l]$, a célula $P[i][j][k][l]$ de P conterá o endereço da célula $P[i'][j'][k'][l']$. A matriz de apontadores P irá facilitar o processo de determinar uma cadeia $\Gamma_B = \{b_p, b_q, ..., b_r\}$ de segmentos de B e uma cadeia $\Gamma_C = \{c_w, c_x, ..., c_y\}$ de segmentos de C, tal que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C. Esse processo será descrito na próxima seção.

Analisando o Algoritmo 4, podemos ver que é necessário um tempo constante para preencher cada célula referente às condições base. Para preencher as células que não pertencem às condições base, ou seja, as células $M[i][j][k][l]$, com $1 \leq k \leq u$ e $1 \leq l \leq v$, é necessário: 1) tempo constante, se $i > 1$ e $j > 1$; 2) tempo $O(u * v)$, se $i = 1$ e $j = 1$; 3) tempo $O(u)$, se $i = 1$ e $j > 1$; 4) tempo $O(v)$, se $i > 1$ e $j = 1$. A Tabela 4.1 mostra a complexidade de tempo necessário para preencher todas as células da matriz M.

Assumindo que, dado um conjunto de segmentos, a cardinalidade do conjunto é maior ou igual ao tamanho do maior segmento deste conjunto, isto é, $u \geq bMax$ e $v \geq cMax$, com $bMax = max\{|b| : b \in B\}$ e $cMax = max\{|c| : c \in C\}$, podemos concluir que o tempo necessário para preencher M é $O(u^2 * v^2)$. Para preencher cada célula de P basta obter o endereço de uma célula de P, que é uma operação que consome tempo constante. Dessa maneira, o tempo necessário para preencher P é simplesmente a quantidade de células de P, e podemos dizer que também é $O(u^2 * v^2)$. Com a matriz M preenchida, o último passo do Algoritmo 4 é determinar e devolver o valor de $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ (linha 43), o que consome tempo $O(u * v)$. Com essas análises, podemos concluir que a complexidade de tempo total do Algoritmo 4 é $O(u^2 * v^2)$.

Algoritmo 4 PASGSim(s_1, s_2, B, C, ω)

Entrada: Duas sequências s_1 e s_2, de tamanho n e m respectivamente, um conjunto ordenado $B = \{b_1, b_2, ..., b_u\}$ de segmentos de s_1, um conjunto ordenado $C = \{c_1, c_2, ..., c_v\}$ de segmentos de s_2 e uma função de pontuação ω.

Saída: O valor de $sim_\omega(\Gamma^\bullet_B, \Gamma^\bullet_C)$, tal que essa similaridade é máxima entre todas as cadeias de segmentos de B e de C.

1: //Condições base
2: $M[0][0][0][0] \leftarrow 0$;
3: $P[0][0][0][0] \leftarrow NULO$;

4: **para** $l \leftarrow 1$ até v **faça**
5: $M[0][1][0][l] \leftarrow M[0][0][0][0] + \omega(-, c_l[1])$;
6: $P[0][1][0][l] \leftarrow$ endereço de $P[0][0][0][0]$;
7: **para** $j \leftarrow 2$ até $|c_l|$ **faça**
8: $M[0][j][0][l] \leftarrow M[0][j-1][0][l] + \omega(-, c_l[j])$;
9: $P[0][j][0][l] \leftarrow$ endereço de $P[0][j-1][0][l]$;
10: **fim para**
11: **fim para**

12: **para** $k \leftarrow 1$ até u **faça**
13: $M[1][0][k][0] \leftarrow M[0][0][0][0] + \omega(b_k[1], -)$;
14: $P[1][0][k][0] \leftarrow$ endereço de $P[0][0][0][0]$;
15: **para** $i \leftarrow 2$ até $|b_k|$ **faça**
16: $M[i][0][k][0] \leftarrow M[i-1][0][k][0] + \omega(b_k[i], -)$;
17: $P[i][0][k][0] \leftarrow$ endereço de $P[i-1][0][k][0]$;
18: **fim para**
19: **fim para**

20: //Recorrência 4.4
21: **para** $k \leftarrow 1$ até u **faça**
22: **para** $l \leftarrow 1$ até v **faça**
23: **para** $i \leftarrow 1$ até $|b_k|$ **faça**
24: **para** $j \leftarrow 1$ até $|c_l|$ **faça**
25: **se** $i > 1$ e $j > 1$ **então**
26: $$M[i][j][k][l] \leftarrow max \begin{cases} M[i-1][j-1][k][l] + \omega(b_k[i], c_l[j]) \\ M[i][j-1][k][l] + \omega(-, c_l[j]) \\ M[i-1][j][k][l] + \omega(b_k[i], -) \end{cases}$$
27: **fim se**

28: **se** $i = 1$ e $j = 1$ **então**
29: $$M[i][j][k][l] \leftarrow max \begin{cases} max_{b_{k'} \prec b_k,\ c_{l'} \prec c_l}\{M[|b_{k'}|][|c_{l'}|][k'][l'] + \omega(b_k[1], c_l[1])\} \\ max_{c_{l'} \prec c_l}\{M[1][|c_{l'}|][k][l'] + \omega(-, c_l[1])\} \\ max_{b_{k'} \prec b_k}\{M[|b_{k'}|][1][k'][l] + \omega(b_k[1], -)\} \end{cases}$$
30: **fim se**

31: **se** $i = 1$ e $j > 1$ **então**
32: $$M[i][j][k][l] \leftarrow max \begin{cases} M[1][j-1][k][l] + \omega(-, c_l[j]) \\ max_{b_{k'} \prec b_k} \begin{cases} M[|b_{k'}|][j-1][k'][l] + \omega(b_k[1], c_l[j]) \\ M[|b_{k'}|][j][k'][l] + \omega(b_k[1], -) \end{cases} \end{cases}$$
33: **fim se**

Algoritmo 4 PASGSim(s_1, s_2, B, C, ω) - continuação

34: **se** $i > 1$ e $j = 1$ **então**

35: $$M[i][j][k][l] \leftarrow max \begin{cases} M[i-1][1][k][l] + \omega(b_k[i], -) \\ max_{c_{l'} \prec c_l} \begin{cases} M[i-1][|c_{l'}|][k][l'] + \omega(b_k[i], c_l[1]) \\ M[i][|c_{l'}|][k][l'] + \omega(-, c_l[1]) \end{cases} \end{cases}$$

36: **fim se**

37: Seja $M[i'][j'][k'][l']$ a célula selecionada nas condições acima;

38: $P[i][j][k][l] \leftarrow$ endereço de $P[i'][j'][k'][l']$;

39: **fim para**

40: **fim para**

41: **fim para**

42: **fim para**

43: **devolva** $max_{b_k \in B,\ c_l \in C}\{M[|b_k|][|c_l|][k][l]\}$;

Célula $M[i][j][k][l]$	Tempo para preencher uma célula	Quantidade de células	Tempo total
Com $k = 0$ e/ou $l = 0$	$O(1)$	$O(u*bMax+v*cMax)$	$O(u*bMax+v*cMax)$
Com $i > 1$, $j > 1$, $k \geq 1$ e $l \geq 1$	$O(1)$	$O(u*v*bMax*cMax)$	$O(u*v*bMax*cMax)$
Com $i = 1$, $j = 1$, $k \geq 1$ e $l \geq 1$	$O(u*v)$	$O(u*v)$	$O(u^2*v^2)$
Com $i = 1$, $j > 1$, $k \geq 1$ e $l \geq 1$	$O(u)$	$O(u*v*cMax)$	$O(u^2*v*cMax)$
Com $i > 1$, $j = 1$, $k \geq 1$ e $l \geq 1$	$O(v)$	$O(u*v*bMax)$	$O(u*v^2*bMax)$

Tabela 4.1: *Complexidade de tempo necessário para preencher todas as células da matriz M, onde $bMax = max\{|b| : b \in B\}$ e $cMax = max\{|c| : c \in C\}$.*

4.5 Determinando Γ_B e Γ_C

Para resolver o PASG, nos resta determinar uma cadeia de segmentos Γ_B de B e uma cadeia de segmentos Γ_C de C tal que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C.

Podemos determinar Γ_B e Γ_C facilmente através da matriz de alinhamento de segmentos M e da matriz de apontadores P, preenchidas durante a execução do Algoritmo 4. Sejam $b_{k'}$ e $c_{l'}$ dois segmentos de B e de C, respectivamente, tal que $M[|b_{k'}|][|c_{l'}|][k'][l'] = max_{b_k \in B,\ c_l \in C}\{M[|b_k|][|c_l|][k][l]\}$. Determinamos Γ_B e Γ_C seguindo qualquer caminho de apontadores de $P[|b_{k'}|][|c_{l'}|][k'][l']$ até $P[0][0][0][0]$ e gravando todos os segmentos de B e de C pelos quais passamos durante esse caminho (com exceção de b_0 e c_0). O Algoritmo 5 implementa essa ideia.

Analisando o Algoritmo 5, a linha 3 é executada em tempo $O(u * v)$ enquanto que as outras instruções da parte de inicialização (linhas 2, 4 e 5) são executadas em tempo constante. Sobre a quantidade de tempo necessário para executar o laço **enquanto**, observe que se a variável *atual* está apontando para a célula $P[i][j][k][l]$ da matriz P em uma dada iteração, na iteração seguinte ela passa a apontar para uma das seguintes células: $P[i-1][j][k][l]$, ou $P[i][j-1][k][l]$, ou $P[i-1][j-1][k][l]$ ou $P[i'][j'][k'][l']$, com $1 \leq i' \leq |b_{k'}|$, $1 \leq j' \leq |c_{l'}|$, $b_{k'} \prec b_k$ e/ou $c_{l'} \prec c_l$. Informalmente, a cada iteração do laço **enquanto**, a variável *atual* passa a apontar para uma célula vizinha ou para uma submatriz mais próxima de $P[0][0][0][0]$. Assim, no pior caso, teremos $1 + \sum_{b \in B} |b| + \sum_{c \in C} |c|$ iterações nesse laço até que a variável *atual* atinja a posição $P[0][0][0][0]$ e aponte para $NULO$,

Algoritmo 5 PASG(M, P)

Entrada: Uma matriz de alinhamento de segmentos M e uma matriz de apontadores P.
Saída: Uma cadeia de segmentos Γ_B de B e uma cadeia de segmentos Γ_C de C tal que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C.

1: //Inicialização
2: Seja *atual* uma variável apontadora;
3: Faça *atual* apontar para a célula $P[|b_{k'}|][|c_{l'}|][k'][l']$ de P tal que $M[|b_{k'}|][|c_{l'}|][k'][l'] = max_{b_k \in B,\ c_l \in C}\{M[|b_k|][|c_l|][k][l]\}$;
4: $\Gamma_B = \emptyset$;
5: $\Gamma_C = \emptyset$;

6: //Laço principal - constrói Γ_B e Γ_C
7: **enquanto** $atual \neq NULO$ **faça**
8: Seja $P[i][j][k][l]$ a célula para a qual *atual* aponta;
9: **se** $b_k \neq b_0$ e $b_k \notin \Gamma_B$ **então**
10: Adicione b_k no começo de Γ_B;
11: **fim se**
12: **se** $c_l \neq c_0$ e $c_l \notin \Gamma_C$ **então**
13: Adicione c_l no começo de Γ_C;
14: **fim se**
15: $atual \leftarrow P[i][j][k][l]$;
16: **fim enquanto**

17: **devolva** Γ_B e Γ_C;

determinando a saída do laço. Cada iteração consumirá tempo constante se implementarmos Γ_B e Γ_C como duas listas tais que tanto a recuperação do elemento no início da lista como a adição de um elemento no início dela consuma tempo constante. Dessa forma, a complexidade de tempo do Algoritmo 5 é $O(u*bMax+v*cMax)$, com $bMax = max\{|b| : b \in B\}$ e $cMax = max\{|c| : c \in C\}$.

Finalmente, podemos resolver o PASG invocando o Algoritmo 4 e, depois, o Algoritmo 5. Através das análises de tempo destes algoritmos, concluímos que é possível resolver o PASG em tempo $O(u^2*v^2)$. Aqui finalizamos a descrição de todos os quatro passos necessários para o desenvolvimento de um algoritmo de programação dinâmica para o PASG.

A solução aqui descrita para o PASG foi implementada no decorrer deste trabalho na linguagem C. Para mais informações sobre essa implementação, veja o Apêndice A.

Capítulo 5

Sobre a complexidade e inaproximabilidade do PASGM

Antes de discutirmos possíveis abordagens para tratar o PASGM, precisamos definir sua complexidade computacional e analisar questões relacionadas à sua aproximabilidade. Neste capítulo, provamos que o PASGM é NP-Completo e que é muito improvável que exista um algoritmo de aproximação com uma boa razão para ele.

5.1 NP-Completude do PASGM

Para provar que o PASGM é NP-Completo, devemos primeiramente formular uma versão de decisão para ele. Essa versão está definida a seguir:

Problema do Alinhamento de Segmentos Múltiplo - versão de decisão (PASGMD): dados um alfabeto Σ, tal que $'-' \notin \Sigma$, $n > 2$ sequências $s_1, s_2, ..., s_n$ sobre Σ, n conjuntos ordenados de segmentos $B_1, B_2, ..., B_n$, onde B_i é um conjunto ordenado de segmentos de s_i, uma função de pontuação ω sobre $\overline{\Sigma}$ e um inteiro positivo l, determinar se existem n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i, tal que $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = l$.

Consideraremos a versão de decisão do PASGM, PASGMD, para provar que ele é NP-Completo. Apesar de serem problemas diferentes, note que se conseguirmos resolver a versão de otimização do problema, podemos resolver a versão de decisão simplesmente invocando o algoritmo que resolve a versão de otimização e avaliando se $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = l$. Desta forma, o PASGM é, no mínimo, tão difícil quanto o PASGMD.

Para a prova da NP-Completude do PASGMD, referenciaremos a versão de decisão do Problema da Subsequência Comum Mais Longa Múltipla introduzida na Seção 2.3.1, e descrita abaixo novamente:

Problema da Subsequência Comum Mais Longa Múltipla - versão de decisão (LCSMD): dadas $n > 2$ sequências $s_1, s_2, ..., s_n$ construídas sobre um mesmo alfabeto finito Σ, e um inteiro positivo k, existe uma sequência $t \in \Sigma^*$, com $|t| \geq k$, tal que t é subsequência de $s_1, s_2, ...$ e s_n?

O LCSMD é um problema NP-Completo e uma prova da sua NP-Completude pode ser vista em [25]. Com isto, poderíamos utilizá-lo na prova da NP-Completude do PASGMD, mas, para simplificá-la, usaremos outra versão de decisão do Problema da Subsequência Comum Mais Longa Múltipla descrita a seguir:

Problema da Subsequência Comum Mais Longa Múltipla - versão de decisão 2 (LCSMD2): dadas $n > 2$ sequências $s_1, s_2, ..., s_n$ construídas sobre um mesmo alfabeto finito

Σ, e um inteiro positivo k, existe uma sequência $t \in \Sigma^*$, com $|t| = k$, tal que t é subsequência de $s_1, s_2, ...$ e s_n?

A diferença entre as duas versões de decisão do Problema da Subsequência Comum Mais Longa Múltipla é que, enquanto o LCSMD procura por uma sequência $t \in \Sigma^*$, com $|t| \geq k$, tal que t é subsequência de $s_1, s_2, ...$ e s_n, o LCSMD2 procura por uma sequência $t \in \Sigma^*$, com $|t| = k$, tal que t é subsequência de $s_1, s_2, ...$ e s_n.

Antes de utilizar o LCSMD2 em nossa prova da NP-Completude do PASGMD, precisamos mostrar que ele é NP-Completo.

Lema 5.1.1 *O LCSMD2 é NP-Completo.*

Prova Com base na Seção 2.3, sabemos que para provar que o LCSMD2 é NP-Completo é necessário primeiro mostrar que o LCSMD2 pertence à classe NP e, depois, fazer uma redução polinomial de um problema NP-Completo a ele.

Para mostrar que o LCSMD2 $\in$ NP, é necessário apresentar um algoritmo verificador polinomial para ele. Um algoritmo verificador polinomial para o LCSMD2 receberia como entrada $n > 2$ sequências $s_1, s_2, ..., s_n$, um inteiro positivo k (instância do problema), e uma sequência $t \in \Sigma^*$ (certificado). Com a entrada, o algoritmo verificaria se $|t| = k$ e se t é subsequência de $s_1, s_2, ...$ e s_n. Em caso afirmativo, o algoritmo devolveria SIM. Caso contrário, ele devolveria NÃO. Considerando $m = max\{|s_1|, |s_2|, ..., |s_n|\}$, é possível construir um algoritmo verificador trivial para o LCSMD2 cuja complexidade de tempo é $O(n * (m + k))$. Desta forma, LCSMD2 $\in$ NP.

Agora, faremos uma redução polinomial do LCSMD ao LCSMD2. Para isso, suponha que temos um algoritmo que resolve o LCSMD2 em tempo polinomial (que será invocado através de $LCSMD2()$). Desenvolveremos então um algoritmo polinomial, chamado $RedLCSMD$, que resolve o LCSMD utilizando um número polinomial de chamadas a $LCSMD2()$. O algoritmo $RedLCSMD$ recebe como entrada $n > 2$ sequências $s_1, s_2, ..., s_n$ construídas sobre um mesmo alfabeto finito Σ, e um inteiro positivo k. No seu primeiro passo, o algoritmo determina a sequência s_{min}, que é a menor sequência da entrada. Posteriormente, ele invoca $LCSMD2(\{s_1, s_2, ..., s_n\}, i)$, para $k \leq i \leq |s_{min}|$, e verifica se alguma destas chamadas devolve SIM. Em caso afirmativo, podemos concluir que existe uma sequência $t \in \Sigma^*$, com $|t| \geq k$, tal que t é subsequência de $s_1, s_2, ...$ e s_n, e então o algoritmo $RedLCSMD$ devolve SIM. Em caso negativo, o algoritmo devolve NÃO.

Desta forma, o LCSMD2 é NP-Completo.

■

Utilizando-se do Lema 5.1.1, o Teorema 5.1.2 prova que o PASGMD é NP-Completo.

Teorema 5.1.2 *O PASGMD é NP-Completo.*

Prova Com base na Seção 2.3, sabemos que para provar que o PASGMD é NP-Completo é necessário primeiro mostrar que o PASGMD pertence à classe NP e, depois, fazer uma redução polinomial de um problema NP-Completo a ele.

Para mostrar que o PASGMD $\in$ NP, é necessário apresentar um algoritmo verificador polinomial para ele. Um algoritmo verificador polinomial para o PASGMD é trivial e recebe como entrada n sequências $s_1, s_2, ...$ e s_n, n conjuntos de segmentos $B_1, B_2, ...$ e B_n, uma função de pontuação ω, um inteiro positivo l (instância do problema), e n cadeias de segmentos $\Gamma_1, \Gamma_2, ...$ e Γ_n (certificado), onde Γ_i é uma cadeia de segmentos de B_i. Com esses dados, a única coisa que o algoritmo deve fazer é verificar se $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = l$. Em caso afirmativo, o algoritmo verificador devolve SIM. Caso contrário, ele devolve NÃO. Considerando $m = max\{|\Gamma_1^\bullet|, |\Gamma_2^\bullet|, ..., |\Gamma_n^\bullet|\}$, é possível construir um algoritmo verificador para o PASGMD de complexidade de tempo $O(n^2 * m^2)$. Desta forma, o PASGMD $\in$ NP.

Para o segundo passo da prova da NP-Completude do PASGMD, mostraremos uma redução polinomial do LCSMD2 ao PASGMD. Faremos isto transformando uma instância arbitrária

$I = (\{s_1, s_2, ..., s_n\}, k)$ do LCSMD2 em uma instância $I' = (\{s'_1, s'_2, ..., s'_n\}, \{B_1, B_1, ..., B_n\}, \omega, l)$ do PASGMD, de tal forma que I é positiva se e somente se I' é positiva. Para isso, primeiramente atribuímos o conjunto $\{s_1, s_2, ..., s_n\}$ ao conjunto $\{s'_1, s'_2, ..., s'_n\}$. Ou seja, o conjunto de sequências recebidas é o mesmo para ambos os problemas. Em segundo lugar, definimos os n conjuntos ordenados de segmentos $B_1, B_2, ..., B_n$. A definição do conjunto ordenado de segmentos B_i consiste em considerar cada caractere da sequência correspondente s_i como um segmento, ou seja, cada segmento de B_i corresponde a um caractere de s_i na ordem em que eles aparecem nessa sequência.

Sobre a função de pontuação ω, a definimos como:

$$\omega(a,b) = \begin{cases} 1, & \text{se } a = b \\ -(Opt^+ + 1), & \text{se } a \neq b \\ -(Opt^+ + 1), & \text{se } a = - \text{ ou } b = - \end{cases},$$

onde $Opt^+ = \frac{n*(n-1)}{2} * |s_{max}|$, tal que s_{max} é a maior sequência da entrada. Em outras palavras, Opt^+ é um limitante superior para o valor $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet)$, para quaisquer n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$. Ou seja, $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) \leq Opt^+$ e, disso, temos que $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) < Opt^+ + 1$, para quaisquer n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$.

Por fim, definimos l em função de k da seguinte forma:

$$l = \frac{n * (n-1) * k}{2}.$$

Utilizando as transformações supracitadas, convertemos uma instância arbitrária I do LCSMD2 em uma instância I' do PASGMD. Suponha, agora, que temos um algoritmo que resolve o PASGMD em tempo polinomial (que será invocado através de $PASGMD()$). Podemos desenvolver então um algoritmo polinomial, que chamaremos de $RedLCSMD2$, que resolve o LCSMD2. O algoritmo $RedLCSMD2$ recebe como entrada uma instância $I = (\{s_1, s_2, ..., s_n\}, k)$ do LCSMD2 e transforma-a em uma instância I' do PASGMD, conforme descrito anteriormente. Por fim, o algoritmo $RedLCSMD2$ invoca o $PASGMD()$ passando I' como entrada e devolve SIM, se esta chamada ao $PASGMD()$ devolveu SIM, e NÃO caso contrário.

Para concluir a prova da NP-Completude do PASGMD, é necessário mostrar que a instância I do LCSMD2 é positiva se e somente se a instância I' do PASGMD é positiva. Para isso, suponha inicialmente que I é positiva, ou seja, existe uma sequência $t \in \Sigma^*$, com $|t| = k$, tal que t é subsequência de $s_1, s_2, ...$ e s_n. Mostraremos que I' também é positiva. Durante a execução do algoritmo $RedLCSMD2$, quando ele invocar o algoritmo $PASGMD()$ passando a instância I' como entrada, em algum momento o $PASGMD()$ conseguirá determinar n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i, tal que $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = l$, já que essas n cadeias de fato existem. Como a instância I é positiva, o algoritmo $PASGMD()$ sempre terá a opção de determinar essas n cadeias tais que $\Gamma_1^\bullet = \Gamma_2^\bullet = ... = \Gamma_n^\bullet = t$. Ou seja, $\Gamma_1 = \Gamma_2 = ... = \Gamma_n$ e cada segmento de Γ_i, com $1 \leq i \leq n$, é composto por um caractere de t, na ordem em que eles aparecem nessa sequência. Desta forma, temos que:

$$\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = \sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(t,t).$$

Como $|t| = k$, temos:

$$\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(t,t) = \sum_{i=2}^{n}\sum_{j=1}^{i-1} k = \frac{n * (n-1) * k}{2}.$$

Como $l = \frac{n*(n-1)*k}{2}$, o $PASGMD()$ sempre terá a opção de escolher as n cadeias de segmentos desta maneira e I' será positiva, se I for positiva.

Agora, suponha que I' é positiva, ou seja, existem n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i, tal que $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = l$. Mostraremos que I

também é positiva. Como $l = \frac{n*(n-1)*k}{2}$, precisamos mostrar que existe uma sequência $t \in \Sigma^*$, com $|t| = k = \frac{2*l}{n*(n-1)}$, tal que t é subsequência de $s_1, s_2, ...$ e s_n.

Se existem n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i, tal que $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = l$, então $\Gamma_1 = \Gamma_2 = ... = \Gamma_n$. Suponha, por contradição, que $\Gamma_1 = \Gamma_2 = ... = \Gamma_n$ não é verdade, ou seja, existe pelo menos um par de cadeia de segmentos (Γ_i, Γ_j) tal que $\Gamma_i \neq \Gamma_j$. Dessa forma, em pelo menos um dos $\frac{n*(n-1)}{2}$ alinhamentos de $\Gamma_1^\bullet, \Gamma_2^\bullet, ..., \Gamma_n^\bullet$ haverá alguma coluna que será um *mismatch* ou um *space*. Como a pontuação de um *mismatch* ou de um *space* é $-(Opt^+ + 1)$, então teremos que $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) < 0$, o que é uma contradição de nossa hipótese de que $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = l$, pois l é um inteiro positivo.

Assim, temos que existem n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i, tal que $\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = l$ e $\Gamma_1 = \Gamma_2 = ... = \Gamma_n$. Podemos notar que $\Gamma_1^\bullet = \Gamma_2^\bullet = ... = \Gamma_n^\bullet$ e que qualquer $\Gamma_i^\bullet$, com $1 \leq i \leq n$, é subsequência de $s_1, s_2, ...$ e s_n. Por fim, observe que:

$$\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) = \sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_{k'}^\bullet, \Gamma_{k'}^\bullet) = l,$$

para um inteiro k' qualquer entre 1 e n. Dessa forma, temos que:

$$\sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_{k'}^\bullet, \Gamma_{k'}^\bullet) = \sum_{i=2}^{n}\sum_{j=1}^{i-1} |\Gamma_{k'}^\bullet| = \frac{n*(n-1)*|\Gamma_{k'}^\bullet|}{2} = l,$$

e, isolando $|\Gamma_{k'}^\bullet|$:

$$|\Gamma_{k'}^\bullet| = \frac{2*l}{n*(n-1)}.$$

Logo, se escolhermos $t = \Gamma_{k'}^\bullet$, para um inteiro k' qualquer entre 1 e n, temos que $t \in \Sigma^*$, com $|t| = k = \frac{2*l}{n*(n-1)}$, tal que t é subsequência de $s_1, s_2, ...$ e s_n e, por consequência, que I é positiva.

Utilizando os métodos descritos, conseguimos então desenvolver o algoritmo $RedLCSMD2$ que resolve o LCSMD2, se tivermos em mãos um algoritmo que resolve o PASGMD em tempo polinomial.

Sobre a complexidade do algoritmo $RedLCSMD2$, a atribuição do conjunto $\{s_1, s_2, ..., s_n\}$ ao conjunto $\{s'_1, s'_2, ..., s'_n\}$ consome tempo $O(1)$, assim como a definição de l. Para definir os conjuntos de segmentos $B_1, B_2, ..., B_n$, basta percorrer cada caractere de cada sequência s_i e ir adicionando-os em um conjunto B_i. Este mecanismo gasta tempo $O(n * m)$, onde $m = max\{|s_1|, |s_2|, ..., |s_n|\}$. Para definir ω, além de operações que consomem tempo constante, é necessário determinar $Opt^+ = \frac{n*(n-1)}{2} * |s_{max}|$, que pode ser feito em tempo $O(n * m)$. Por fim, nossa redução tem complexidade de tempo $O(n * m)$.

De todas as observações discutidas aqui, concluímos que o PASGMD é NP-Completo.

■

5.2 Inaproximabilidade do PASGM

Outro resultado interessante sobre o PASGM diz respeito à possibilidade de se desenvolver algoritmos de aproximação com uma boa razão para esse problema. Em caso afirmativo, o ideal seria concentrar esforços para desenvolver tais algoritmos e melhorá-los continuamente. Contudo, os Lemas 5.2.1 e 5.2.2 e o Teorema 5.2.3 a seguir mostram que é muito improvável existir um algoritmo de aproximação com uma boa razão para o PASGM.

Lema 5.2.1 *Se existe uma constante $\delta > 0$ tal que o PASGM possui um algoritmo de aproximação de razão $\frac{1}{n^\delta}$, então o LCSM também possui um algoritmo de aproximação de razão $\frac{1}{n^\delta}$.*

Prova Suponha que o PASGM possua um algoritmo de aproximação, que chamaremos de Aprox-PASGM, de razão $\frac{1}{n^\delta}$, com $\delta > 0$. Com essa hipótese, podemos desenvolver um algoritmo de aproximação de razão $\frac{1}{n^\delta}$ para o LCSM, que chamaremos de AproxLCSM.

A ideia do algoritmo AproxLCSM é, primeiramente, transformar uma instância arbitrária I do LCSM em uma instância I' do PASGM seguindo o mesmo mecanismo mostrado no Teorema 5.1.2. Contudo, como neste caso estamos falando do problema de otimização, e não de decisão, a instância I não define o inteiro k e a instância I' também não define o inteiro l. Dessa forma, nessa transformação recebemos a instância $I = (\{s_1, s_2, ..., s_n\})$ do LCSM e construímos a instância $I' = (\{s'_1, s'_2, ..., s'_n\}, \{B_1, B_1, ..., B_n\}, \omega)$ do PASGM, a partir de I, utilizando a mesma sequência de passos explicada no Teorema 5.1.2, com exceção da definição de l em função de k. Construída a instância I', o próximo passo do algoritmo AproxLCSM é invocar o AproxPASGM passando como entrada I' e, ao obter as n cadeias de segmentos $\Gamma'_1, \Gamma'_2, ..., \Gamma'_n$, devolver uma sequência $t' = \Gamma'^{\bullet}_i$, para um i qualquer entre 1 e n, como resposta. Nessa prova, usaremos a notação $\Gamma'_1, \Gamma'_2, ..., \Gamma'_n$ para representar as cadeias de segmentos devolvidas pelo algoritmo AproxPASGM e $\Gamma_1, \Gamma_2, ..., \Gamma_n$ para representar as cadeias de segmentos de uma solução ótima de uma instância do PASGM.

É trivial mostrar que o algoritmo AproxLCSM possui complexidade de tempo polinomial, já que tanto a transformação de I em I' como o algoritmo AproxPASGM possuem complexidade de tempo polinomial[1]. No restante dessa prova, iremos mostrar que o algoritmo AproxLCSM é um algoritmo de aproximação para o LCSM de razão $\frac{1}{n^\delta}$.

Sejam t a maior subsequência comum a $s_1, s_2, ..., s_n$, $OptLCSM(I)$ o valor de uma solução ótima de I, ou seja, o tamanho de t, $OptPASGM(I')$ o valor de uma solução ótima de I' e $AproxPASGM(I')$ o valor da solução devolvida pelo algoritmo AproxPASGM quando recebe a instância I' como entrada. Temos então que:

$$OptPASGM(I') = \sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet).$$

Mostraremos que $\Gamma_1 = \Gamma_2 = ... = \Gamma_n$. Suponha, por contradição, que isso não é verdade, ou seja, existe pelo menos um par de cadeia de segmentos (Γ_i, Γ_j) tal que $\Gamma_i \neq \Gamma_j$. Dessa forma, em pelo menos um dos $\frac{n*(n-1)}{2}$ alinhamentos de $\Gamma_1^\bullet, \Gamma_2^\bullet, ..., \Gamma_n^\bullet$ haverá alguma coluna que será um *mismatch* ou um *space*. Como a pontuação de um *mismatch* ou de um *space* é $-(Opt^+ + 1)$, então teremos que $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) < 0$, o que é uma contradição de nossa hipótese de que $\Gamma_1, \Gamma_2, ..., \Gamma_n$ é uma solução ótima de I', já que $\Gamma''_1 = \Gamma''_2 = ... = \Gamma''_n = \emptyset$ é uma solução com valor maior.

Podemos mostrar também que $|\Gamma_k| = OptLCSM(I)$, para um k qualquer entre 1 e n. Lembremo-nos que $OptLCSM(I) = |t|$. Como $\Gamma_1 = \Gamma_2 = ... = \Gamma_n$, temos que $\Gamma_1^\bullet = \Gamma_2^\bullet = ... = \Gamma_n^\bullet$. Com esta última igualdade, juntamente com o fato de que $\Gamma_k^\bullet$ é subsequência de s_k, podemos concluir que qualquer $\Gamma_k^\bullet$, com $1 \leq k \leq n$, é subsequência de $s_1, s_2, ...$ e s_n. Agora suponha, por contradição, que $|\Gamma_k| < OptLCSM(I)$. Nessa situação, podemos escolher n cadeias de segmentos $\Gamma''_1, \Gamma''_2, ..., \Gamma''_n$ de tal forma que $\Gamma''^{\bullet}_1 = \Gamma''^{\bullet}_2 = ... = \Gamma''^{\bullet}_n = t$. Assim, $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma''^{\bullet}_i, \Gamma''^{\bullet}_j) > \sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet)$, o que é uma contradição de nossa hipótese de que $\Gamma_1, \Gamma_2, ..., \Gamma_n$ é uma solução ótima de I'. Por outro lado, suponha, por contradição, que $|\Gamma_k| > OptLCSM(I)$. Temos então que $\Gamma_k^\bullet$ é uma subsequência comum a $s_1, s_2, ...$ e s_n tal que $|\Gamma_k^\bullet| > OptLCSM(I) = |t|$, o que é uma contradição de nossa hipótese de que t é a maior subsequência comum a $s_1, s_2, ..., s_n$. Dessa maneira, temos que $|\Gamma_k| = OptLCSM(I)$.

Das observações acima, temos que:

$$\begin{aligned}
OptPASGM(I') &= \sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet) \\
&= \sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_k^\bullet, \Gamma_k^\bullet) \\
&= \sum_{i=2}^{n} \sum_{j=1}^{i-1} |\Gamma_k| \qquad (5.1) \\
&= \sum_{i=2}^{n} \sum_{j=1}^{i-1} OptLCSM(I) \\
&= \frac{n*(n-1)*OptLCSM(I)}{2}.
\end{aligned}$$

[1] Lembremo-nos que, por definição, um algoritmo de aproximação é necessariamente um algoritmo polinomial.

Como AproxPASGM é um um algoritmo de aproximação para o PASGM de razão $\frac{1}{n^\delta}$, temos que:

$$AproxPASGM(I') \geq \frac{1}{n^\delta} * OptPASGM(I'). \tag{5.2}$$

Das Equações 5.1 e 5.2, podemos verificar que:

$$AproxPASGM(I') \geq \frac{1}{n^\delta} * \frac{n*(n-1)*OptLCSM(I)}{2}. \tag{5.3}$$

Se executarmos o algoritmo AproxLCSM, conforme descrito, passando como entrada uma instância I, conseguimos achar uma sequência $t' = \Gamma_i'^{\bullet}$, para um i qualquer entre 1 e n, que é subsequência de todas as sequências em I. Como $t' = \Gamma_1'^{\bullet} = \Gamma_2'^{\bullet} = ... = \Gamma_n'^{\bullet}$, temos que:

$$\begin{aligned} AproxPASGM(I') &= \sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(\Gamma_i'^{\bullet}, \Gamma_j'^{\bullet}) \\ &= \sum_{i=2}^{n}\sum_{j=1}^{i-1} sim_\omega(t', t') \\ &= \sum_{i=2}^{n}\sum_{j=1}^{i-1} |t'| \\ &= \frac{n*(n-1)*|t'|}{2}. \end{aligned} \tag{5.4}$$

Logo, pelas Equações 5.3 e 5.4:

$$\frac{n*(n-1)*|t'|}{2} \geq \frac{1}{n^\delta} * \frac{n*(n-1)*OptLCSM(I)}{2};$$

$$|t'| \geq \frac{1}{n^\delta} * OptLCSM(I),$$

o que significa que o algoritmo AproxLCSM sempre devolverá uma sequência t' tal que t' é subsequência de todas as sequências em I e $|t'| \geq \frac{1}{n^\delta} * OptLCSM(I)$. Logo, o AproxLCSM é um algoritmo de aproximação para o LCSM de razão $\frac{1}{n^\delta}$.

■

Lema 5.2.2 *Se existe uma constante $\delta > 0$ tal que o LCSM possui um algoritmo de aproximação de razão $\frac{1}{n^\delta}$, onde n é a quantidade de sequências da entrada, então P=NP.*

Prova A prova pode ser encontrada em [41].

■

Teorema 5.2.3 *Se existe uma constante $\delta > 0$ tal que o PASGM possui um algoritmo de aproximação de razão $\frac{1}{n^\delta}$, onde n é a quantidade de sequências da entrada, então P=NP.*

Prova A prova é uma consequência direta dos Lemas 5.2.1 e 5.2.2.

■

O Teorema 5.2.3 mostra que, a não ser que P=NP, não conseguimos desenvolver um algoritmo de aproximação com uma boa razão para o PASGM.

Capítulo 6

Heurísticas para o Problema do Alinhamento de Segmentos Múltiplo

Conforme visto no capítulo anterior, o Problema do Alinhamento de Segmentos Múltiplo é NP-Completo e é muito improvável que exista um algoritmo de aproximação com uma boa razão para ele. Sendo assim, decidimos abordá-lo por meio de heurísticas. Neste capítulo detalhamos três heurísticas desenvolvidas para o PASGM e analisamos os resultados de uma avaliação experimental, utilizando testes artificiais, envolvendo cada uma delas.

6.1 Preliminares

Para uma melhor compreensão das heurísticas desenvolvidas neste trabalho, assuma que a entrada para o PASGM é dada pela tripla $(\{s_1, s_2, ..., s_n\}, \{B_1, B_2, ..., B_n\}, \omega)$, onde $\{s_1, s_2, ..., s_n\}$ é um conjunto de n sequências construídas sobre um alfabeto Σ tal que $'-' \notin \Sigma$, $\{B_1, B_2, ..., B_n\}$ são n conjuntos ordenados de segmentos de $s_1, s_2, ..., s_n$, respectivamente, e ω é uma função de pontuação sobre $\overline{\Sigma}$. Por questões de simplicidade, chamemos de $\mathcal{S}$ o conjunto $\{s_1, s_2, ..., s_n\}$ e de $\mathcal{B}$ o conjunto $\{B_1, B_2, ..., B_n\}$. Nas heurísticas a serem descritas utilizamos o algoritmo que resolve o PASG como uma função auxiliar e o invocamos da seguinte forma: $PASG(s_1, s_2, B, C, \omega)$. Essa chamada devolve um par de cadeias de segmentos (Γ_B, Γ_C) tal que $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ é máxima entre todas as cadeias de segmentos de B e de C. Definimos $PASGSim$ como um algoritmo que resolve o PASG, mas que devolve o valor $sim_\omega(\Gamma_B^\bullet, \Gamma_C^\bullet)$ ao invés do par de cadeias de segmentos (Γ_B, Γ_C). Invocamos o algoritmo $PASGSim$ da seguinte forma: $PASGSim(s_1, s_2, B, C, \omega)$.

Para simplificar a análise da complexidade de tempo das heurísticas propostas, assumiremos que, dado um conjunto de segmentos, a cardinalidade do conjunto é maior ou igual ao tamanho do maior segmento desse conjunto. Em outras palavras, assumiremos que $|B_i| \geq B_{i_{max}}$, com $B_{i_{max}} = max\{|b| : b \in B_i\}$, para todo $1 \leq i \leq n$. Finalmente, considere $\mathcal{B}_{max} = max\{|B_i|\}$, ou seja, $\mathcal{B}_{max}$ denota a cardinalidade do maior conjunto em $\mathcal{B}$.

6.2 Heurística da cadeia de segmentos consenso (Heurística 1)

A **cadeia de segmentos consenso** Γ_i de um conjunto ordenado de segmentos B_i de uma sequência s_i é uma cadeia de segmentos obtida da seguinte forma. Primeiramente, invocamos $PASG(s_i, s_j, B_i, B_j, \omega)$ para todo $s_j \in \mathcal{S} \setminus \{s_i\}$ e seu respectivo B_j. Depois disso, realizamos uma contagem para verificar quantas vezes cada segmento de B_i apareceu nas soluções devolvidas pelas $n-1$ chamadas a $PASG()$. A cadeia de segmentos consenso Γ_i é então construída reunindo todos os segmentos que foram incluídos em mais da metade das soluções. Observe que escolhendo somente os segmentos que participam de mais da metade das soluções, não é possível escolher um par de segmentos sobrepostos.

Esta primeira heurística consiste em determinar uma cadeia de segmentos consenso para cada

sequência s_i de $\mathcal{S}$, tendo como resultado final o conjunto $\{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$, que constitui a solução devolvida por ela. O Algoritmo 6 detalha os passos dessa heurística.

Algoritmo 6 Heurística_Cadeia_Segmentos_Consenso($\{s_1, s_2, ..., s_n\}$, $\{B_1, B_2, ..., B_n\}$, ω)

Entrada: n sequências $\{s_1, s_2, ..., s_n\}$, n conjuntos ordenados de segmentos $\{B_1, B_2, ..., B_n\}$ e uma função de pontuação ω.
Saída: n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i.

1: //Inicialização
2: Seja $contador_i$ um vetor de inteiros indexado pelos segmentos do conjunto ordenado de segmentos B_i;
3: Inicialize todas as posições de $contador_i$ com 0, para $1 \leq i \leq n$;
4: $\Gamma_1 \leftarrow \Gamma_2 \leftarrow ... \leftarrow \Gamma_n \leftarrow \emptyset$;

5: //Invoca $PASG(s_i, s_j, B_i, B_j, \omega)$, para todo $s_j \in S \setminus \{s_i\}$, com $1 \leq i \leq n$ e conta os segmentos presentes nas respostas
6: **para** i de 1 até n **faça**
7: **para** j de 1 até n **faça**
8: **se** $i \neq j$ **então**
9: $(\Gamma_B, \Gamma_C) \leftarrow PASG(s_i, s_j, B_i, B_j, \omega)$;
10: **para cada** segmento s de Γ_B **faça**
11: $contador_i[s] = contador_i[s] + 1$;
12: **fim para**
13: **fim se**
14: **fim para**
15: **fim para**

16: //Determina a cadeia de segmentos de consenso Γ_i de B_i, para $1 \leq i \leq n$
17: **para** i de 1 até n **faça**
18: **para cada** segmento s de B_i **faça**
19: **se** $contador_i[s] > (n-1)/2$ **então**
20: adicione s à Γ_i;
21: **fim se**
22: **fim para**
23: **fim para**

24: **devolva** $\Gamma_1, \Gamma_2, ..., \Gamma_n$;

A complexidade de tempo das linhas 2-4 da Heurística 1 é $O(n * \mathcal{B}_{max})$. Para determinar a complexidade de tempo do laço **para** das linhas 6-15, primeiramente determinaremos a complexidade de tempo das operações das linhas 9-12. A complexidade de tempo da linha 9 é $O((\mathcal{B}_{max})^4)$. A complexidade de tempo das linhas 10-12 é $O(\mathcal{B}_{max})$. Logo, a complexidade de tempo das linhas 9-12 é $O((\mathcal{B}_{max})^4) + O(\mathcal{B}_{max}) = O((\mathcal{B}_{max})^4)$. Como as operações nas linhas 9-12 são executadas $n * (n-1)$ vezes, a complexidade de tempo do laço **para** das linhas 6-15 é $O(n * (n-1) * (\mathcal{B}_{max})^4) = O(n^2 * (\mathcal{B}_{max})^4)$. A complexidade de tempo do laço **para** das linhas 17-23 é $O(n * \mathcal{B}_{max})$. Portanto, a complexidade de tempo do Algoritmo 6 é $O(n * \mathcal{B}_{max}) + O(n^2 * (\mathcal{B}_{max})^4) + O(n * \mathcal{B}_{max}) = O(n^2 * (\mathcal{B}_{max})^4)$.

6.3 Heurística gulosa (Heurística 2)

A ideia da heurística gulosa para determinar uma cadeia de segmentos Γ_i é, primeiramente, selecionar uma sequência $s_k \neq s_i$ tal que $PASGSim(s_i, s_k, B_i, B_k, \omega) \geq PASGSim(s_i, s_j, B_i, B_j, \omega)$, para todo $s_j \in \mathcal{S} \setminus \{s_i\}$ e seu respectivo B_j. Depois disso, invocamos $PASG(s_i, s_k, B_i, B_k, \omega)$ e atribuímos a primeira cadeia de segmentos devolvida por essa chamada à Γ_i. O procedimento descrito

é repetido para todas as sequências s_i de $\mathcal{S}$, tendo como resultado final o conjunto $\{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$ que constitui a solução devolvida pela heurística gulosa. O Algoritmo 7 detalha os passos dessa heurística.

Algoritmo 7 Heurística_Gulosa($\{s_1, s_2, ..., s_n\}$, $\{B_1, B_2, ..., B_n\}$, ω)

Entrada: n sequências $\{s_1, s_2, ..., s_n\}$, n conjuntos ordenados de segmentos $\{B_1, B_2, ..., B_n\}$ e uma função de pontuação ω.
Saída: n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i.

```
1:  para i de 1 até n faça
2:    max ← −∞;
3:    k ← −1;
4:    para j de 1 até n faça
5:      se i ≠ j então
6:        se PASGSim(s_i, s_j, B_i, B_j, ω) > max então
7:          max ← PASGSim(s_i, s_j, B_i, B_j, ω);
8:          k ← j;
9:        fim se
10:     fim se
11:   fim para
12:   (Γ_i, x) ← PASG(s_i, s_k, B_i, B_k, ω);
13: fim para

14: devolva Γ_1, Γ_2, ..., Γ_n;
```

A complexidade de tempo do Algoritmo 7 é determinada pela complexidade de tempo do laço **para** das linhas 1-13. Para determiná-la, primeiramente analisaremos a complexidade de tempo das operações dentro deste laço. A complexidade de tempo das operações das linhas 2 e 3 é $O(1)$. A complexidade de tempo das operações das linhas 5-10 é limitada pela complexidade de tempo da chamada à função $PASGSim()$. Portanto, a complexidade de tempo dessas operações é $O((\mathcal{B}_{max})^4)$. O laço **para** das linhas 4-11 executa essas operações $n-1$ vezes. Portanto, a complexidade de tempo do laço **para** das linhas 4-11 é $O((n-1) * (\mathcal{B}_{max})^4) = O(n * (\mathcal{B}_{max})^4)$. A complexidade de tempo da linha 12 é $O((\mathcal{B}_{max})^4)$. Portanto, a complexidade de tempo das operações das linhas 2-12 é $O(1) + O(n * (\mathcal{B}_{max})^4) + O((\mathcal{B}_{max})^4) = O(n * (\mathcal{B}_{max})^4)$. O laço **para** das linhas 1-13 executa as operações das linhas 2-12 n vezes. Logo, a complexidade de tempo desse laço mais externo e consequentemente do Algoritmo 7 é $O(n^2 * (\mathcal{B}_{max})^4)$.

6.4 Heurística da cadeia de segmentos central (Heurística 3)

A **cadeia de segmentos central** Γ_i de um conjunto ordenado de segmentos B_i de uma sequência s_i é a cadeia de segmentos obtida da seguinte forma. Primeiramente, para cada inteiro j, com $1 \leq j \leq n$ e $j \neq i$, invocamos $PASG(s_i, s_j, B_i, B_j, \omega)$ e chamamos a primeira cadeia de segmentos, do par de cadeias de segmentos (Γ_B, Γ_C) devolvido por esta chamada a $PASG()$, de $\Gamma_i(j)$. Dessa forma, ao final de todas as invocações a $PASG()$, teremos $n-1$ cadeias de segmentos $\Gamma_i(j)$. Elas são as $n-1$ possíveis candidatas à cadeia de segmentos Γ_i, que fará parte da resposta final. Para definir Γ_i, utilizamos o algoritmo proposto por Gelfand *et al.* em [17] para a resolução do Problema do Alinhamento *Spliced*, introduzido na Seção 3.3. Mais especificamente, selecionamos, dentre as $n-1$ cadeias $\Gamma_i(j)$ criadas anteriormente, aquela tal que

$$\sum_{k=1,\ k\neq i}^{n} GelfandSim(s_k, B_k, \Gamma_i(j)^{\bullet}, \omega)$$

é máximo, onde $GelfandSim(s_1, B, s_2, \omega)$ denota uma invocação ao algoritmo definido em [17] para resolução do Problema do Alinhamento *Spliced*, mas que devolve apenas o valor da solução ótima e não a solução ótima em si. Informalmente, buscamos por aquele $\Gamma_i(j)^\bullet$ que seja o mais parecido possível com as concatenações das cadeias de segmentos dos conjuntos $B_{k\neq i}$, cadeias essas encontradas pelo algoritmo definido por Gelfand *et al.*

Dada a definição de cadeia de segmentos central, o que a terceira heurística faz é determinar a cadeia de segmentos central Γ_i para cada sequência s_i de $\mathcal{S}$, usando o procedimento descrito, e assim definir o conjunto $\{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$. O Algoritmo 8 especifica esta heurística.

Algoritmo 8 Heurística_Cadeia_Segmentos_Central($\{s_1, s_2, ..., s_n\}$, $\{B_1, B_2, ..., B_n\}$, ω)

Entrada: n sequências $\{s_1, s_2, ..., s_n\}$, n conjuntos ordenados de segmentos $\{B_1, B_2, ..., B_n\}$ e uma função de pontuação ω.
Saída: n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$, onde Γ_i é uma cadeia de segmentos de B_i.

```
1:  para i de 1 até n faça
2:    //Calcula cada Γ_i(j)
3:    para j de 1 até n faça
4:      se j ≠ i então
5:        (Γ_i(j), x) ← PASG(s_i, s_j, B_i, B_j, ω);
6:      fim se
7:    fim para

8:    //Seleciona aquele Γ_i(j) tal que Σ_{k=1, k≠i}^{n} GelfandSim(s_k, B_k, Γ_i(j)•, ω) é máximo.
9:    max ← −∞;
10:   para j de 1 até n faça
11:     se j ≠ i então
12:       soma ← 0;
13:       para k de 1 até n faça
14:         se k ≠ i então
15:           soma ← soma + GelfandSim(s_k, B_k, Γ_i(j)•, ω);
16:         fim se
17:       fim para
18:       se soma > max então
19:         max ← soma;
20:         Γ_i ← Γ_i(j);
21:       fim se
22:     fim se
23:   fim para
24: fim para

25: devolva Γ_1, Γ_2, ..., Γ_n;
```

Para simplificar a análise da complexidade de tempo do Algoritmo 8, assumiremos que o tempo necessário para executar o algoritmo $GelfandSim()$ é $\Theta(\mathcal{G})$. Primeiramente, a complexidade de tempo do laço **para** das linhas 3-7 é $O((n-1) * (\mathcal{B}_{max})^4) = O(n * (\mathcal{B}_{max})^4)$. Para determinar o tempo necessário para executar o laço **para** das linhas 10-23, observe que a complexidade de tempo da linha 15 é $O(\mathcal{G})$ e do laço **para** das linhas 13-17 é $O((n-1) * \mathcal{G}) = O(n * \mathcal{G})$. Podemos concluir então que o tempo necessário para executar as linhas 12-21, que inclui operações de tempo constante e o laço **para** das linhas 13-17, também é $O(n * \mathcal{G})$ e, sendo assim, a complexidade de tempo do laço **para** das linhas 10-23 é $O((n-1) * (n * \mathcal{G})) = O(n^2 * \mathcal{G})$. Com essas informações, podemos dizer que o tempo necessário para executar as linhas 3-23 é $O(n * (\mathcal{B}_{max})^4 + n^2 * \mathcal{G})$. Logo, a complexidade de tempo do Algoritmo 8 é $O(n * (n * (\mathcal{B}_{max})^4 + n^2 * \mathcal{G})) = O(n^2 * (\mathcal{B}_{max})^4 + n^3 * \mathcal{G})$.

6.5 Avaliação das heurísticas com instâncias de testes artificiais

As três heurísticas propostas foram implementadas na linguagem C[1] e avaliadas comparando-se o resultado devolvido por cada uma delas ao serem executadas em uma grande quantidade de instâncias de testes artificiais. O conjunto de instâncias de testes inclui 31894 instâncias criadas artificialmente. Cada instância de teste inclui de 3 a 5 sequências, cada uma possuindo de 10 a 20 caracteres do alfabeto $\Sigma = \{A, C, G, T\}$[2]. O conjunto ordenado de segmentos de uma sequência possui de 5 a 15 segmentos aleatórios dessa sequência. Na execução de cada heurística, a função de pontuação ω utilizada foi:

$$\omega(a,b) = \begin{cases} 1, & \text{se } a = b \\ -1, & \text{se } a \neq b \\ -2, & \text{se } a = - \text{ ou } b = - \end{cases} .$$

As informações de saída devolvidas pelas heurísticas foram levemente alteradas para que não determinem apenas as n cadeias de segmentos $\Gamma_1, \Gamma_2, ..., \Gamma_n$ que compõem a solução, mas também o valor da função objetivo $\sum_{i=2}^{n} \sum_{j=1}^{i-1} sim_\omega(\Gamma_i^\bullet, \Gamma_j^\bullet)$. As comparações entre as heurísticas levam em conta o valor da função objetivo obtido por cada uma, sendo que quanto maior este valor, melhor é o resultado obtido pela heurística.

Com o objetivo de verificar o quanto as heurísticas estão próximas do valor de uma solução ótima, comparamos os resultados devolvidos pelas execuções delas, sobre cada um dos casos de teste, com o valor de uma solução ótima correspondente encontrada utilizando um algoritmo força-bruta para o PASGM. Fazemos isso calculando o valor $\alpha = \frac{|f(s(I)^\star) - f(H_x(I))|}{f(s(I)^\star)}$, onde $f(s(I)^\star)$ é o valor de uma solução ótima de uma instância I e $f(H_x(I))$ é o valor da solução devolvida pela Heurística x quando recebe a instância I como entrada [30]. Quanto menor for o valor α, melhor é a solução devolvida pela heurística.

Os testes foram realizados em um computador com um processador Intel Pentium Dual CPU E2220, com dois núcleos de 2,4 GHz cada e 3,25 GB de RAM, no sistema operacional Windows XP PRO SP3. A Tabela 6.1 resume os resultados da execução das heurísticas e do programa força-bruta sobre o conjunto de instâncias de testes.

Programa	Valor médio (d. p.)	α médio (d. p.)	α máx	Ótimas	Positivas	Tempo médio
Força-bruta	10,85 (8,09)	0,00 (0,00)	0,00	31894	31894	107 *s*
Heurística 1	-38,72 (34,72)	6,53 (8,90)	143,00	535	1207	7,8 *ms*
Heurística 2	-11,23 (19,45)	3,92 (6,72)	104,00	2187	9053	7,7 *ms*
Heurística 3	-8,80 (22,94)	4,18 (8,81)	148,00	4489	12257	23,6 *ms*

Tabela 6.1: *Tabela que resume os resultados da execução do programa de força-bruta e das heurísticas sobre as 31894 instâncias de testes artificiais. A coluna* ***Valor médio (d. p.)*** *indica a média aritmética dos valores da função objetivo obtida por cada programa, juntamente com o desvio padrão (entre parênteses). A coluna* α ***médio (d. p.)*** *indica a média aritmética dos valores* α *obtida por cada programa, juntamente com o desvio padrão (entre parênteses). A coluna* α ***máx*** *indica o pior valor* α *obtido por cada programa. As colunas* ***Ótimas*** *e* ***Positivas*** *indicam a quantidade de instâncias sobre as quais cada programa obteve uma solução ótima e uma solução cujo valor é positivo, respectivamente. Por fim, a coluna* ***Tempo médio*** *contém a média aritmética dos tempos de execução de cada programa.*

Considerando os valores da Tabela 6.1, podemos ver que a Heurística 1 foi a que apresentou os piores resultados. Mais especificamente, essa heurística obteve o pior valor médio, o pior α médio e o segundo pior α máximo, o que significa que as soluções encontradas por ela foram muito ruins. Consequentemente, ela também foi a heurística que encontrou o menor número de soluções ótimas

[1] Observações sobre a implementação e a execução das heurísticas podem ser encontradas no Apêndice A.

[2] As sequências de uma instância de teste são todas geradas a partir de algumas modificações (inserção, remoção e substituição de caracteres) em uma sequência base. Portanto, elas possuem um certo grau de similaridade.

e positivas.

Sobre a Heurística 2, ela obteve o segundo melhor valor médio, perdendo para o valor médio da Heurística 3. Contudo, a segunda heurística apresentou um desvio padrão menor que o da terceira heurística nessa medida, o que indica que os valores de suas soluções tendem a ser menos dispersos do que os valores das soluções da Heurística 3. É devido a esse fato que acreditamos que, apesar de apresentar um valor médio menor, a Heurística 2 apresentou um α médio melhor do que o da Heurística 3. Outro número que sustenta essa crença é que a Heurística 2 exibiu o melhor α máximo enquanto a Heurística 3 exibiu o pior α máximo (pior até mesmo que o da Heurística 1). Apesar da Heurística 2 se sobressair nas medidas que consideram o valor α, ela encontrou uma quantidade menor de soluções ótimas e positivas quando comparada com a Heurística 3.

Com a análise supracitada, podemos concluir que a Heurística 1 foi a que obteve o pior desempenho nessa rodada de testes, enquanto que as Heurísticas 2 e 3 foram melhores. Os valores das soluções encontradas pela Heurística 2 estão, em média, mais próximos dos valores das soluções ótimas, já que ela obteve os melhores números nas colunas α médio e α máx. Apesar disso, a Heurística 3 foi a que apresentou o melhor valor médio e a que encontrou a maior quantidade de soluções ótimas e positivas. Devido a esse último fato, acreditamos que a Heurística 3 se sobressaiu nesses testes.

Com relação ao tempo médio consumido por cada heurística para processar uma instância de teste, as Heurísticas 1 e 2 apresentaram um tempo médio praticamente equivalente, enquanto que a Heurística 3 mostrou-se cerca de 3 vezes mais lenta do que as duas primeiras. Isso se deve principalmente às $n*(n-1)*(n-1)$ chamadas ao algoritmo que resolve o Problema do Alinhamento *Spliced*.

Capítulo 7

Identificação de genes por comparação de DNAs

Neste capítulo detalharemos a aplicação dos programas desenvolvidos na tarefa de identificação de genes. Essa tarefa possui grande importância prática no contexto da Biologia Molecular, já que o estudo de soluções para o problema computacional relacionado pode ser útil no controle de pragas, na prevenção de doenças, no tratamento de endemias, etc. Neste contexto, a primeira seção deste capítulo apresenta alguns dos principais conceitos ligados à Biologia Molecular necessários para uma melhor compreensão da aplicação dos programas desenvolvidos nesta área. Na segunda seção, introduzimos formalmente o problema de identificação de genes, a motivação para o seu estudo e os principais métodos disponíveis na literatura para atacá-lo. Na terceira seção, discutimos a aplicação dos programas desenvolvidos neste trabalho na tarefa de identificação de genes e apresentamos os resultados de uma avaliação experimental dos nossos programas com sequências de DNAs reais.

7.1 Conceitos básicos da Biologia Molecular

A **Biologia Molecular** é o ramo da Biologia que estuda a formação, estrutura e função de moléculas essenciais à vida, tais como ácidos nucleicos e proteínas, incluindo seus papéis nos processos de replicação de células e transmissão de informação genética. As próximas subseções introduzem conceitos básicos da Biologia Molecular, retirados de [2, 4, 21, 24, 36, 38].

7.1.1 A célula

As **células** são as unidades estruturais e funcionais de todos os organismos vivos, com capacidade de obter nutrientes, convertê-los em energia, desempenhar funções especializadas e se reproduzir. As células são organizadas em duas categorias: procariotas e eucariotas. Células procariotas não apresentam membrana nuclear, uma membrana que envolve o núcleo e o separa do citoplasma (um fluído que contém as outras estruturas celulares). A maioria dos organismos procariotos são compostos por apenas uma célula (unicelulares) procariota. As bactérias e as algas são os organismos procariotos mais estudados e conhecidos. Já nas células eucariotas, o núcleo é envolvido e separado do citoplasma pela membrana nuclear e armazena moléculas importantes para o desenvolvimento da célula. É dentro dele que ocorre boa parte dos processos de replicação celular. As células eucariotas compõem a maioria dos organismos vivos, incluindo animais, plantas e fungos, e é nela que focaremos as aplicações deste trabalho.

7.1.2 Ácidos nucleicos

Nas células, encontramos dois tipos de ácidos nucleicos: o DNA e o RNA. O **DNA** (*deoxyribonucleic acid*) ou **ADN** (ácido desoxirribonucleico, em português) é um longo polímero (macromoléculas formadas por unidades menores) de moléculas mais simples, denominadas nucleotídeos.

Um **nucleotídeo** consiste de uma pentose, um resíduo de fosfato e uma base nitrogenada (ou simplesmente base). A pentose corresponde a uma molécula de açúcar formada por cinco carbonos numerados de 1' a 5'. O resíduo de fosfato nada mais é do que uma molécula com um átomo de fósforo cercado por quatro oxigênios. Finalmente, sobre as **bases nitrogenadas**, no DNA elas são quatro: adenina (A), citosina (C), guanina (G) e timina (T). Uma representação de um nucleotídeo de uma molécula de DNA pode ser vista na Figura 7.1.

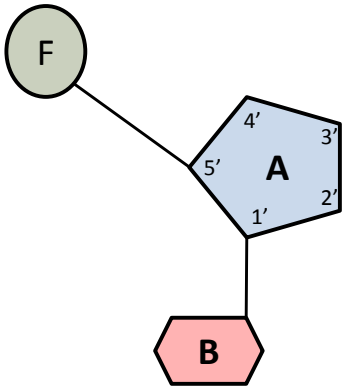


Figura 7.1: *Uma representação de um nucleotídeo de uma molécula de DNA, onde A é a pentose (açúcar), F é o resíduo de fosfato e B é a base nitrogenada.*

Estruturalmente, podemos ver o DNA como uma cadeia dupla de nucleotídeos, onde cada cadeia constitui uma fita. Uma **fita** é uma sequência de nucleotídeos unidos quimicamente. Dois nucleotídeos de uma fita estão ligados pelos carbonos 3' e 5' da pentose, através do resíduo de fosfato. Se a ligação é feita entre o carbono 5' de um nucleotídeo e o carbono 3' do próximo nucleotídeo na fita, dizemos que esta fita possui orientação 5' $\rightarrow$ 3'. Na Figura 7.2 pode ser vista uma fita com orientação 5' $\rightarrow$ 3'. Analogamente, se a ligação é feita entre o carbono 3' de um nucleotídeo e o carbono 5' do próximo nucleotídeo na fita, dizemos que esta fita possui orientação 3' $\rightarrow$ 5'. A extremidade da fita que possui um nucleotídeo cujo carbono 3' está livre é denominada extremidade 3'. Similarmente, a extremidade da fita que possui um nucleotídeo cujo carbono 5' está livre é denominada extremidade 5'.

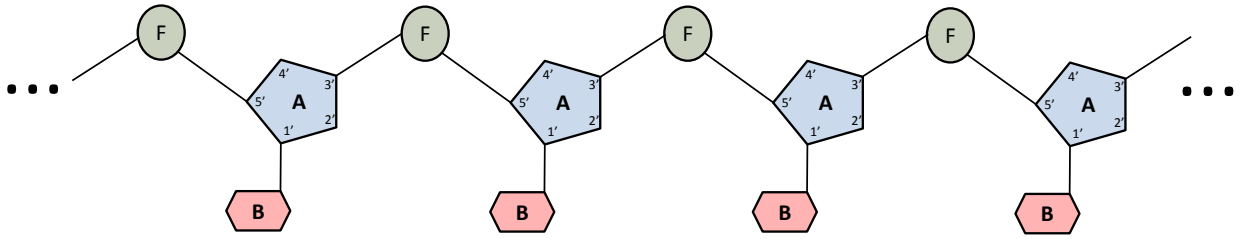


Figura 7.2: *Exemplo de uma fita com orientação 5' $\rightarrow$ 3'. Observe o resíduo de fosfato ligando os carbonos 5' e 3' das pentoses dos nucleotídeos.*

Ainda sobre a estrutura do DNA, suas duas fitas possuem orientações opostas e elas se encontram dispostas de forma helicoidal, em torno de um eixo imaginário. As duas fitas são ligadas através de pontes de hidrogênio, formadas entre os seguintes pares de bases: A-T, T-A, C-G e G-C. Nesse sentido, dizemos que A e T são **bases complementares**, assim como C e G. Veja a Figura 7.3 para um exemplo da disposição do DNA no espaço. Como consequência disto, dada uma fita de DNA e sua orientação, podemos deduzir a orientação e os nucleotídeos que formam a outra fita do DNA, também denominada de complemento reverso. Dentre os vários papéis desempenhados pelo DNA, um dos mais importantes é o de armazenar as informações necessárias para a síntese das proteínas de um organismo. Finalmente, para realizar alguma tarefa, o mecanismo celular pode fazer o uso das **regiões funcionais** do DNA, que são trechos que possuem alguma função dentro do DNA. Por

outro lado, o DNA também inclui as **regiões não funcionais**, que são trechos que não possuem nenhuma função aparente dentro do DNA.

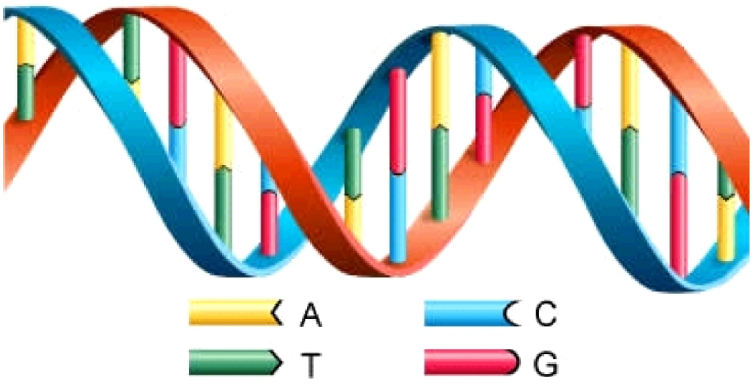


Figura 7.3: *As duas fitas dispostas helicoidalmente, em torno de um eixo imaginário, e ligadas através de pontes de hidrogênios formadas entre pares de bases complementares. Figura adaptada de http://www.astrochem.org/sci/Nucleobases.php.*

Sobre o **RNA** (*ribonucleic acid*) ou **ARN** (ácido ribonucleico, em português), ele é um ácido nucleico similar ao DNA, mas com três diferenças notáveis. Em primeiro lugar, a pentose do RNA possui uma molécula de oxigênio adicional ligada ao carbono 2'. No RNA, a base timina (T) é substituída pela base uracila (U). Desta forma, os seguintes pares de bases são complementares no RNA: A-U, U-A, C-G e G-C. A terceira diferença é que o RNA é formado por apenas uma fita. Vale também salientar que existem diferentes tipos de RNA dentro da célula, cada um deles desempenhando uma função específica e bem definida. Exemplos incluem RNA polimerases I, II e III, RNA ribossomal e RNA transportador.

7.1.3 Síntese de proteínas

Proteínas são cadeias de moléculas mais simples, denominadas aminoácidos, que desempenham inúmeras funções vitais em um organismo. Os aminoácidos que compõem uma proteína são ligados através de ligações peptídicas e, dessa forma, proteínas também são conhecidas como cadeias polipeptídicas. Existem 20 aminoácidos diferentes e eles podem ser combinados de várias maneiras distintas dando origem às mais diversas proteínas.

Os **genes** são trechos do DNA que codificam as informações necessárias para a síntese de proteínas[1]. Simplificadamente, podemos dizer que os genes armazenam a informação sobre quais aminoácidos são necessários para a produção de uma determinada proteína. Uma sequência de DNA é composta por vários genes separados por regiões denominadas **regiões intergênicas**. Os mecanismos celulares conseguem reconhecer o começo de um gene graças a uma região funcional chamada de **promotor**. A região funcional que indica o término de um gene é conhecida como **terminador**. Nos genes eucariotos, a fração que se encontra entre o promotor e o terminador geralmente é composta por partes alternantes, chamadas íntrons e éxons. Os **íntrons** são regiões não codificantes do gene, ou seja, são regiões que aparentemente não armazenam informações para a produção de proteínas e também não possuem nenhuma função dentro da célula. Os **éxons** são regiões codificantes, armazenando as informações sobre quais aminoácidos são necessários para a produção de uma proteína. Os éxons de um gene podem ser classificados de várias formas:

- éxon único: se um gene é composto por apenas um éxon, este éxon é denominado éxon único;
- éxon inicial: o primeiro éxon de um gene composto por mais de um éxon;

[1] De fato, um gene corresponde a qualquer região do DNA passível de ser transcrita em uma molécula de RNA. Contudo, neste trabalho, estamos chamando de genes apenas aquelas regiões que podem ser transcritas em RNAs mensageiros.

- éxon final: o último éxon de um gene composto por mais de um éxon;
- éxons internos: os éxons que se encontram entre o primeiro e o último éxon de um gene composto por mais de um éxon.

O mecanismo celular consegue identificar o início de um éxon (ou fim de um íntron) e fim de um éxon (ou início de um íntron) através de regiões especiais denominadas **sítios de aceitação e doação**, respectivamente. Com raríssimas exceções, os sítios de aceitação são representados pelos dinucleotídeos AG, que aparecem antes do primeiro nucleotídeo do éxon, e os de doação são representados pelos dinucleotídeos GT, que aparecem depois do último nucleotídeo do éxon. Ou seja, os sítios de aceitação e doação não fazem parte do éxon, mas sim dos íntrons. Além disso, é importante ressaltar que os éxons iniciais e terminais não apresentam sítios de aceitação e de doação, mas sim códons de iniciação e de parada, respectivamente. Os **códons de iniciação e de parada** são regiões funcionais que indicam onde começar e onde terminar o estágio de tradução (detalhado posteriormente), respectivamente. Normalmente, um códon de iniciação é composto pela tripla de nucleotídeos ATG, enquanto que um códon de parada é composto por uma das seguintes triplas de nucleotídeos: TAA, TAG e TGA. Por fim, ainda temos duas regiões funcionais, denominadas 5'-UTR e 3'-UTR, que constituem o início do primeiro éxon e o fim do último éxon, respectivamente. A 5'-UTR aparece depois do promotor, mas antes do códon de iniciação e a 3'-UTR aparece depois do códon de parada, mas antes do terminador.

O **processo de síntese de proteínas** em eucariotos transforma as informações contidas nos genes em proteínas e possui três fases: **transcrição**, ***splicing*** e **tradução**. Esse processo está descrito simplificadamente a seguir.

Primeiramente, o mecanismo celular reconhece o início de um gene com base na localização do seu promotor e então começa a fase de transcrição. Nessa fase, é sintetizada uma molécula especial de RNA, conhecida como **pré-RNA mensageiro** (abreviado para **pré-RNAm**), a partir das informações presentes entre o promotor e o terminador do gene em questão. O pré-RNAm consistirá de uma cadeia de nucleotídeos complementar ao gene, com a base correspondente à adenina sendo a uracila (e não a timina). Para isso, as fitas do DNA se separam por meio da quebra das pontes de hidrogênio existentes entre elas e o pré-RNAm começa a ser produzido. Quando a transcrição termina (ao localizar o terminador), a molécula de pré-RNAm se separa da cadeia de DNA e são reestabelecidas as pontes de hidrogênio das duas fitas e a dupla hélice do DNA.

Tendo uma cópia do gene inteiro no pré-RNAm (excetuando-se as regiões que definem o promotor e o terminador), é executada a segunda fase, chamada fase de *splicing*. Nessa fase, o mecanismo celular descarta os íntrons, mas mantém os éxons, que irão compor o **RNA mensageiro maduro** (**RNAm**). Essa separação é feita com base na identificação dos sítios de doação e de aceitação do gene em questão, onde o mecanismo celular consegue delimitar onde começa e termina cada éxon e íntron.

O terceiro estágio, de tradução, é o processo de construção da cadeia de aminoácidos que forma a proteína sendo sintetizada. Cada um dos 20 aminoácidos existentes está associado a uma sequência de três nucleotídeos. Esta tripla de nucleotídeos é denominada **códon**. Cada códon está associado a um aminoácido conforme a Tabela 7.1, chamada de **Tabela do Código Genético**. O mecanismo celular começa, a partir do códon de iniciação, a tradução de cada códon do RNAm em seu aminoácido correspondente. Essa tradução termina quando é localizado um dos três códons de parada. Observe que alguns códons estão associados ao mesmo aminoácido. Este códons recebem o nome de **códons sinônimos**. Por exemplo, os códons GCA, GCC, GCG e GCT codificam o aminoácido Alanina. A Figura 7.4 resume o processo de síntese de proteínas descrito.

É importante comentar que as células eucariotas ainda admitem um outro processo chamado ***splicing* alternativo**, em que alguns éxons podem ser eliminados da molécula de pré-RNAm durante a fase de *splicing*. Isto permite que várias proteínas diferentes sejam sintetizadas com base em um único gene.

Primeira base	Segunda base T	C	A	G	Terceira base
T	Fenilalanina-F	Serina	Tirosina-Y	Cisteina-C	T
	Fenilalanina	Serina	Tirosina	Cisteina	C
	Leucina-L	Serina	Stop	Stop	A
	Leucina	Serina	Stop	Triptofano-W	G
C	Leucina	Prolina-P	Histidina-H	Arginina-R	T
	Leucina	Prolina	Histidina	Arginina	C
	Leucina	Prolina	Glutamina-Q	Arginina	A
	Leucina	Prolina	Glutamina	Arginina	G
A	Isoleucina-I	Treonina-T	Asparagina-N	Serina-S	T
	Isoleucina	Treonina	Asparagina	Serina	C
	Isoleucina	Treonina	Lisina-K	Arginina	A
	Metionina-M	Treonina	Lisina	Arginina	G
G	Valina-V	Alanina-A	Aspartato-D	Glicina-G	T
	Valina	Alanina	Aspartato	Glicina	C
	Valina	Alanina	Glutamato-E	Glicina	A
	Valina	Alanina	Glutamato	Glicina	G

Tabela 7.1: *Tabela do Código Genético.*

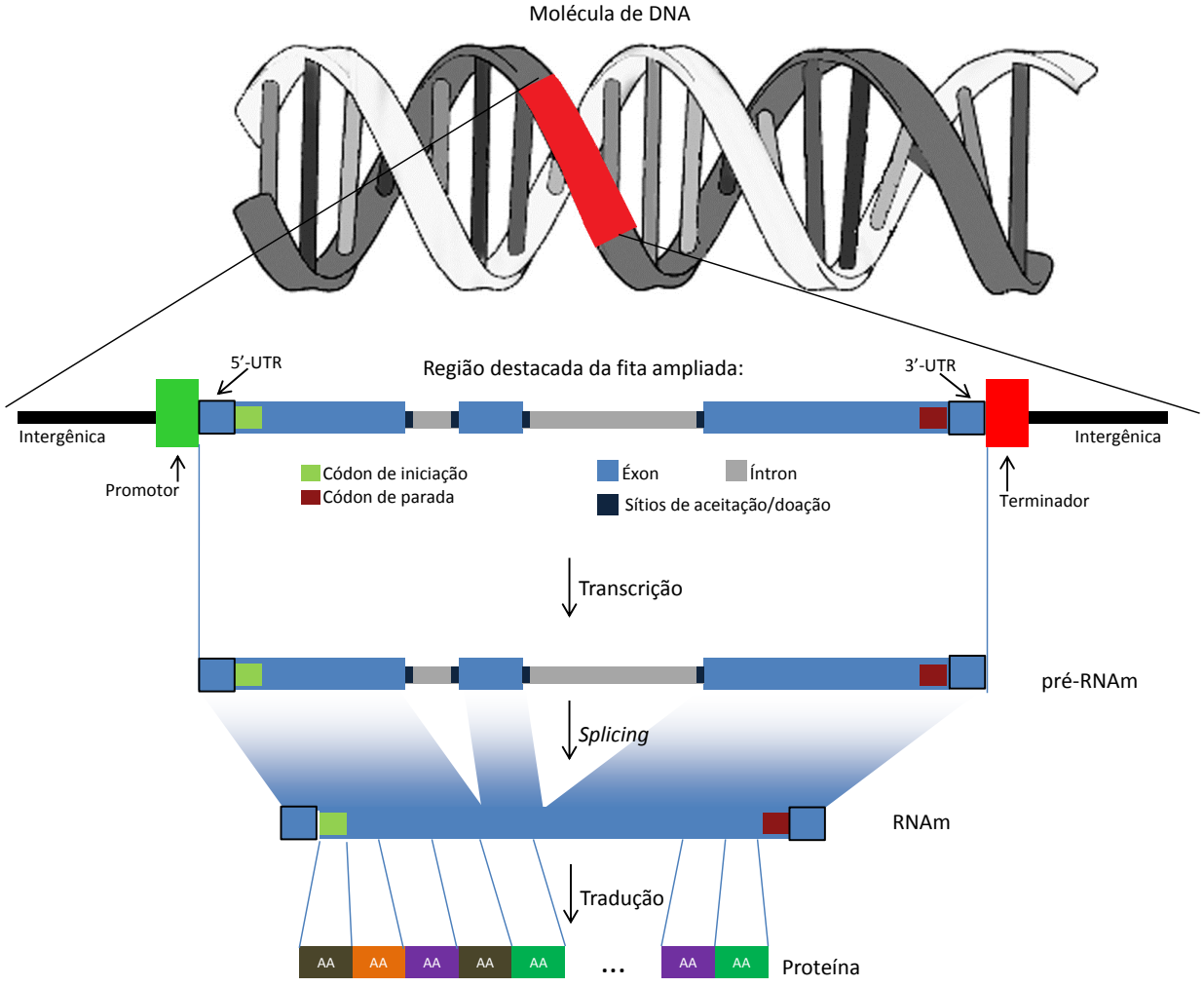


Figura 7.4: *Processo simplificado de síntese de proteínas em células eucariotas.*

7.1.4 Mutações

O DNA de um organismo pode sofrer várias alterações no decorrer do tempo. Essas mudanças, mais conhecidas como mutações, podem ser herdadas ou ocorrer durante a vida do organismo devido tanto a fatores externos (como exposição à radiação ultravioleta do sol), como a fatores internos (como falhas no processo de replicação do DNA de uma célula). Mais especificamente, uma

mutação é uma alteração permanente nos nucleotídeos que compõem um trecho de DNA. Essas alterações correspondem, basicamente, a substituições, inserções ou remoções de bases e podem envolver um único nucleotídeo assim como longos trechos de DNA. Quando uma ou mais mutações ocorrem em regiões funcionais que estão relacionadas com o processo de síntese de proteínas, elas podem interferir neste processo, fazendo com que determinadas proteínas sejam produzidas em uma quantidade diferente da ideal ou com que o mecanismo celular passe a produzir uma variante da proteína original, que pode desempenhar o mesmo papel ou não. Tal interferência pode ter tanto um impacto negativo como positivo para o organismo. Em casos extremos, o impacto negativo de uma ou mais mutações no DNA de um indivíduo pode levar à sua morte (**mutações deletérias**). Quando as mutações trazem algum benefício ao indivíduo (**mutações benéficas**) que as adquiriu, elas podem ser passadas para seus descendentes e, em determinados casos, dar origem a uma espécie distinta da espécie do organismo original. Neste contexto, dados dois organismos, sejam eles da mesma espécie ou não, se ambos possuem um gene derivado de um ancestral comum, este gene é chamado de **gene homólogo**.

É importante notar que as mutações são observadas com muito mais frequência nas regiões não funcionais, como os íntrons e as regiões intergênicas. Estas mutações aparentemente não alteram o processo de síntese de proteínas e, portanto, não apresentam impactos significativos no funcionamento celular. Essa tendência de se observar muito mais mutações nas regiões não funcionais do que nas regiões funcionais é chamada de **princípio de conservação das bases**.

7.2 O problema de identificação de genes

O **problema de identificação de genes** consiste em, dada uma sequência de DNA, determinar a estrutura de cada um de seus genes, ou seja, a posição inicial e final de cada éxon que os constituem[2]. Localizar os éxons que compõem os genes de um DNA é uma tarefa complicada devido a vários fatores. Como já mencionado, DNAs eucariotos podem possuir vários genes separados por regiões intergênicas, e eles são compostos por regiões codificantes intercaladas com regiões não codificantes. Ademais, os genes geralmente constituem uma parte muito pequena de um DNA. Estima-se, por exemplo, que os genes constituem apenas 3% do DNA humano [12], cujo tamanho é de cerca de 3.000.000.000 de nucleotídeos. Por fim, não existe nenhum padrão explícito e único de nucleotídeos que distingue precisamente os genes das regiões intergênicas. Estas mesmas dificuldades podem ser encontradas ao se tentar distinguir precisamente os éxons dos íntrons dentro de um gene. Normalmente, os íntrons são maiores do que os éxons, mas não existe nenhum padrão explícito e único de nucleotídeos que permita diferenciar estas regiões ou as fronteiras entre elas. Existe ainda um problema adicional que envolve os chamados pseudogenes. **Pseudogenes** são trechos de DNA muito parecidos com os genes, mas que não são expressos no processo de síntese de proteína.

7.2.1 Motivação

A resolução do problema de identificação de genes possui inúmeras implicações práticas, que vão do tratamento de doenças e manufaturação de novos medicamente a combates de pragas. Na medicina, por exemplo, identificar os genes do parasita *Trypanossoma cruzi* é um passo fundamental para se compreender os mecanismos que ele utiliza para se proteger do sistema imune do hospedeiro, auxiliando na busca pela cura da doença de Chagas [32]. Também pode-se prever um tumor de acordo com a presença ou não de mutações em determinados genes do DNA de um indivíduo. Na agricultura, é possível modificar geneticamente espécies de plantas vulneráveis a uma certa praga, se os genes do DNA de uma planta imune a ela tiverem sido identificados a priori [5].

Apesar da importância prática do problema, estudos mostram que a exatidão das ferramentas disponíveis hoje para a resolução do problema de identificação de genes está muito distante da ideal

[2]No problema de identificação de genes aqui abordado, consideramos que o primeiro éxon começa no códon de iniciação, e não na região 5'-UTR, e que o último éxon termina no códon de parada, e não na região 3'-UTR.

[3, 6]. Este fato, juntamente com a importância associada ao problema, motivam pesquisas adicionais neste campo, com o intuito de desenvolver novas metodologias e programas de identificação de genes.

7.2.2 Métodos para identificação de genes

Todas as características e dificuldades associadas ao problema de identificação de genes, juntamente com a motivação prática para o seu estudo, tornam este problema difícil e interessante. Como consequência disso, várias abordagens e metodologias foram desenvolvidas para atacar o problema. Nesta seção, são apresentados os principais métodos para a identificação de genes. Esses métodos costumam ser divididos em duas categorias principais: **métodos intrínsecos** e **métodos extrínsecos**.

7.2.3 Métodos intrínsecos

Ao se identificar os genes de uma sequência de DNA utilizando um método intrínseco, considera-se somente os dados presentes na sequência em questão. Em outros termos, estes métodos não fazem uso de informações contidas em outras sequências de DNA relacionadas àquela sendo processada. Os métodos intrínsecos podem ser classificados em dois grupos, dependendo da estratégia utilizada para identificação dos genes: **métodos estatísticos** e **métodos de busca por sinais**.

Métodos estatísticos

Os métodos estatísticos baseiam-se em informações estatísticas derivadas de sequências de DNA e de genes já conhecidos. A ideia geral é procurar por regiões em uma sequência de DNA cujas características estatísticas assemelham-se a de regiões funcionais já localizadas em outras sequências de DNA. Tomando-se o DNA humano como exemplo, sabe-se que, com poucas exceções, a quantidade de nucleotídeos Gs e Cs é maior nos éxons do que nos íntrons [27]. Esta informação estatística pode ser usada, juntamente com outras, para localizar os éxons presentes em uma sequência de DNA.

Uma ferramenta que utiliza um método estatístico baseia-se em uma função que, dado um trecho do DNA de entrada, calcula um número real relacionado à probabilidade desse trecho corresponder à região funcional procurada [18]. De forma mais detalhada, a utilização deste método requer inicialmente a escolha (ou o desenvolvimento) de uma ou mais métricas estatísticas. Em seguida, é definida uma janela de tamanho fixo que é deslizada pela sequência de DNA em que buscamos a região funcional de interesse. Para cada janela é atribuída uma pontuação, que é um valor que sumariza todas as métricas selecionadas em apenas uma. As janelas com as melhores pontuações são aquelas que possuem a maior probabilidade de incluir a região funcional buscada, de acordo com as métricas escolhidas.

Uma importante métrica estatística, denominada **uso de códons** (*codon usage*, do inglês), mensura a frequência com que os códons aparecem nas regiões funcionais. Esta medida é derivada da observação de que, em regiões funcionais, há um uso diferenciado dos códons pelo mecanismo celular. Esta métrica é aplicada comparando-se a frequência de cada códon, de uma região funcional procurada, com a frequência dos códons em uma região qualquer da sequência. Dessa forma, é possível estimar a probabilidade de um certo trecho da sequência corresponder à região funcional procurada. Trechos cujos códons aparecem com frequências similares às frequências de códons da região funcional procurada possuem maior chance de incluir tal região [18]. Com base na Tabela 7.2, que mostra a frequência com que os 64 códons aparecem nos éxons de genes humanos, podemos utilizar esta métrica para tentar localizar os éxons que constituem os genes de uma sequência de DNA humana.

Na prática, existem várias maneiras de calcular a probabilidade de um certo trecho de DNA humano incluir um éxon. Uma das mais utilizadas é chamada de ***log-likelihood ratio***. Seja $C = c_1, c_2, ..., c_n$ uma sequência de n códons de um DNA, $f(c)$ a frequência do códon c na região codificante procurada e $f_0(c)$ a frequência do códon c em uma região não codificante. Geralmente,

Códon	Frequência	Códon	Frequência	Códon	Frequência	Códon	Frequência
ATT	0,0160	CTT	0,0132	GTT	0,0110	TTT	0,0176
ACT	0,0131	CCT	0,0175	GCT	0,0184	TCT	0,0152
AAT	0,0170	CAT	0,0109	GAT	0,0218	TAT	0,0122
AGT	0,0121	CGT	0,0045	GGT	0,0108	TGT	0,0106
ATC	0,0208	CTC	0,0196	GTC	0,0145	TTC	0,0203
ACC	0,0189	CCC	0,0198	GCC	0,0277	TCC	0,0177
AAC	0,0191	CAC	0,0151	GAC	0,0251	TAC	0,0153
AGC	0,0195	CGC	0,0104	GGC	0,0222	TGC	0,0126
ATA	0,0075	CTA	0,0072	GTA	0,0071	TTA	0,0077
ACA	0,0151	CCA	0,0169	GCA	0,0158	TCA	0,0122
AAA	0,0244	CAA	0,0123	GAA	0,0290	TAA	0,0010
AGA	0,0122	CGA	0,0062	GGA	0,0165	TGA	0,0016
ATG	0,0220	CTG	0,0396	GTG	0,0281	TTG	0,0129
ACG	0,0061	CCG	0,0069	GCG	0,0074	TCG	0,0044
AAG	0,0319	CAG	0,0342	GAG	0,0396	TAG	0,0008
AGG	0,0120	CGG	0,0114	GGG	0,0165	TGG	0,0132

Tabela 7.2: *Frequência dos 64 códons nos éxons de genes humanos. Adaptada de www.kazusa.or.jp/codon/cgi-bin/showcodon.cgi?species=9606. Último acesso em 21/05/2013.*

adota-se um modelo aleatório para determinar a distribuição dos códons nas regiões não codificantes, ou seja, assumimos que $f_0(c) = \frac{1}{64}$, para qualquer códon c . A *log-likelihood ratio* de C, denotada por $P(C)$, é dada por:

$$P(C) = \sum_{i=1}^{n} log(f(c_i)) - log(f_0(c_i)).$$

A partir de uma janela $W = b_1, b_2, ..., b_m$ com m nucleotídeos, é possível identificar três sequências de códons diferentes para W. A primeira sequência, denotada por C^1, dita estar no *frame* 1, constitui-se de $\lfloor \frac{m}{3} \rfloor$ códons, onde o primeiro códon é formado por b_1, b_2 e b_3, o segundo códon é formado por b_4, b_5 e b_6 e assim por diante. A segunda sequência de códons, C^2, dita estar no *frame* 2, constitui-se de $\lfloor \frac{m-1}{3} \rfloor$ códons, onde o primeiro códon é formado por b_2, b_3 e b_4, o segundo códon é formado por b_5, b_6 e b_7 e assim por diante. Por fim, a terceira sequência, C^3, que está no *frame* 3, constitui-se de $\lfloor \frac{m-2}{3} \rfloor$ códons, onde o primeiro códon é formado por b_3, b_4 e b_5, o segundo códon é formado por b_6, b_7 e b_8 e assim por diante. Desta forma, para calcular o *log-likelihood ratio* de uma janela W, primeiramente calculamos três *log-likelihood ratio*, um para cada *frame* de W, e então selecionamos o maior destes três. Mais detalhes sobre o uso de códons para identificação de região funcionais podem ser encontrados em [18].

Existe uma série de outras métricas estatísticas que auxiliam na identificação de regiões funcionais ou codificantes. Um exemplo é a métrica chamada uso de aminoácidos, que está baseada na observação de que como os códons são utilizados de forma diferenciada nos éxons, os aminoácidos, por consequência, também acabam sendo usados de forma diferenciada nessas regiões. Outra métrica envolve calcular a frequência não de códons (triplas de nucleotídeos), mas sim de hexamers (sêxtupla de nucleotídeos) e pode ser mais apropriada que o uso de códons para calcular a probabilidade de um trecho de DNA corresponder a uma região codificante porque explora a dependência entre códons adjacentes. Uma descrição geral sobre métodos estatísticos é apresentada em [15].

Métodos de busca por sinais

Sinais são padrões em uma sequência de DNA que indicam o início ou o fim de uma região específica. Exemplos de sinais são os promotores, terminadores, sítios de aceitação e de doação,

códon de iniciação e de parada, etc. A identificação desses sinais é importante pois se eles pudessem ser encontrados facilmente, poderia-se localizar rapidamente os éxons presentes em uma sequência de DNA. Sucintamente, os métodos de busca por sinais procuram identificar sinais associados ao processo de expressão gênica a fim de determinar a presença de genes em uma sequência. A ideia é localizar, dentro de uma sequência, porções que se assemelham a um sinal específico. Por exemplo, poderia-se encontrar regiões do DNA que teriam a maior probabilidade de ser um promotor de acordo com um certo padrão e, então, presumir que um gene vem após a região encontrada.

Um dos métodos mais comuns de busca por sinais é a **matriz de peso** ou, sucintamente, **PWM** (*positional weight matrix*, do inglês), proposta por Staden em [40]. Uma matriz de peso armazena uma compilação da frequência de cada nucleotídeo presente em um sinal e em regiões que o cercam. Por exemplo, Salzberg em [35] define uma PWM para identificar códons de iniciação em vertebrados que considera, além dos nucleotídeos do sinal, 12 nucleotídeos antes e 4 nucleotídeos depois do códon. Esta PWM, M_1, está representada na Figura 7.5.

	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6
A	0,23	0,25	0,33	0,22	0,16	0,29	0,20	0,25	0,22	0,66	0,27	0,15	1,0	0,0	0,0	0,28	0,24	0,11	0,26
C	0,40	0,32	0,28	0,35	0,48	0,31	0,21	0,33	0,56	0,05	0,50	0,58	0,0	0,0	0,0	0,16	0,29	0,24	0,40
G	0,17	0,25	0,17	0,25	0,18	0,16	0,46	0,21	0,17	0,27	0,12	0,22	0,0	0,0	1,0	0,48	0,20	0,45	0,21
T	0,19	0,19	0,21	0,18	0,19	0,24	0,14	0,21	0,06	0,02	0,11	0,05	0,0	1,0	0,0	0,09	0,26	0,21	0,12

Figura 7.5: *PWM para identificar códons de iniciação em vertebrados retirada de [35].*

Para localizar sinais dentro de uma sequência de DNA utilizando uma PWM, geralmente definimos uma janela com um tamanho idêntico ao número de colunas da PWM. Tomando-se como exemplo a PWM M_1, devemos então definir uma janela de tamanho 19. Feito isso, deslizamos a janela pela sequência de DNA, atribuindo uma pontuação para cada trecho do DNA contido na janela. Considerando a PWM M_1, para calcular a probabilidade do códon C, que começa na posição k e se estende até a posição $k+2$, ser um códon de iniciação, definimos uma janela J que vai da posição $k-12$ até a posição $k+6$ e calculamos a pontuação $S(J)$ de J, que corresponde à probabilidade de C ser um códon de iniciação, da seguinte forma:

$$S(J) = \sum_{i=k-12}^{k+6} log(M_1[J[i]][i]).$$

Matrizes de peso geralmente são calculadas considerando uma independência entre os nucleotídeos analisados. Um segundo método, conhecido como **WAM** (*weight array matrix*, do inglês), calcula, para cada posição i, a probabilidade de ocorrer um certo nucleotídeo n_1 dado que o nucleotídeo n_2 aparece na posição $i-1$. Esta abordagem foi introduzida por Zhang e Marr em [33] e é baseada na observação de que a probabilidade de um nucleotídeo estar em uma certa posição depende do nucleotídeo que ocorre na posição anterior. Salzberg em [35] define uma WAM, M_2, representada na Figura 7.6, para identificar códons de iniciação em vertebrados. Assim como a PWM M_1, a WAM M_2 também possui tamanho 19 e para calcular a probabilidade do códon C, que começa na posição k e se estende até a posição $k+2$, ser um códon de iniciação, definimos uma janela J que vai da posição $k-12$ até a posição $k+6$. Para calcular a pontuação $S(J)$ de J, utiliza-se uma fórmula similar àquela usada no contexto das PWMs:

$$S(J) = \sum_{i=k-12}^{k+6} log(M_2[J[i]|J[i-1]][i]).$$

Uma descrição geral sobre métodos de busca por sinais é apresentada em [16].

	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4	+5	+6
$A_i\|A_{i-1}$	0,23	0,24	0,42	0,27	0,16	0,30	0,16	0,20	0,16	0,44	0,28	0,29	1,0	0,0	0,0	0,0	0,38	0,11	0,37
$C_i\|A_{i-1}$	0,23	0,27	0,24	0,32	0,57	0,29	0,08	0,22	0,67	0,06	0,45	0,17	0,0	0,0	0,0	0,0	0,14	0,19	0,27
$G_i\|A_{i-1}$	0,23	0,45	0,24	0,28	0,23	0,30	0,68	0,45	0,14	0,47	0,15	0,50	0,0	0,0	0,0	0,0	0,40	0,59	0,27
$T_i\|A_{i-1}$	0,23	0,05	0,10	0,14	0,04	0,11	0,08	0,13	0,03	0,03	0,13	0,05	0,0	1,0	0,0	0,0	0,08	0,11	0,08
$A_i\|C_{i-1}$	0,40	0,35	0,30	0,25	0,15	0,33	0,29	0,08	0,32	0,78	0,48	0,08	1,0	0,0	0,0	0,0	0,32	0,18	0,38
$C_i\|C_{i-1}$	0,40	0,26	0,33	0,26	0,47	0,29	0,28	0,47	0,46	0,04	0,41	0,80	0,0	0,0	0,0	0,0	0,29	0,29	0,28
$G_i\|C_{i-1}$	0,40	0,09	0,11	0,20	0,10	0,07	0,21	0,05	0,13	0,17	0,10	0,05	0,0	0,0	0,0	0,0	0,01	0,17	0,13
$T_i\|C_{i-1}$	0,40	0,30	0,26	0,29	0,28	0,31	0,21	0,40	0,10	0,01	0,0	0,07	0,0	0,0	0,0	0,0	0,38	0,37	0,22
$A_i\|G_{i-1}$	0,17	0,17	0,45	0,22	0,24	0,29	0,29	0,41	0,21	0,59	0,19	0,19	1,0	0,0	0,0	0,28	0,17	0,09	0,22
$C_i\|G_{i-1}$	0,17	0,35	0,19	0,37	0,40	0,36	0,33	0,30	0,55	0,03	0,67	0,35	0,0	0,0	0,0	0,15	0,35	0,28	0,47
$G_i\|G_{i-1}$	0,17	0,33	0,15	0,30	0,21	0,17	0,29	0,16	0,21	0,34	0,06	0,44	0,0	0,0	0,0	0,48	0,14	0,39	0,23
$T_i\|G_{i-1}$	0,17	0,15	0,21	0,11	0,16	0,17	0,09	0,14	0,03	0,03	0,07	0,01	0,0	0,0	0,0	0,09	0,34	0,21	0,07
$A_i\|T_{i-1}$	0,19	0,10	0,11	0,11	0,07	0,20	0,05	0,06	0,14	0,47	0,30	0,11	1,0	0,0	0,0	0,0	0,04	0,03	0,13
$C_i\|T_{i-1}$	0,19	0,47	0,37	0,51	0,48	0,32	0,20	0,40	0,59	0,12	0,20	0,82	0,0	0,0	0,0	0,0	0,44	0,17	0,46
$G_i\|T_{i-1}$	0,19	0,26	0,24	0,22	0,23	0,24	0,60	0,27	0,20	0,38	0,10	0,03	0,0	0,0	1,0	0,0	0,30	0,69	0,25
$T_i\|T_{i-1}$	0,19	0,17	0,28	0,16	0,23	0,25	0,15	0,27	0,06	0,03	0,40	0,03	0,0	0,0	0,0	0,0	0,22	0,12	0,16

Figura 7.6: *WAM para identificar códons de iniciação em vertebrados retirada de [35].*

7.2.4 Métodos extrínsecos

Os métodos extrínsecos utilizam-se de informações presentes em um conjunto Λ de sequências de DNA para identificar uma determinada região funcional na sequência analisada. Na grande maioria dos casos, espera-se que estas sequências sejam evolutivamente relacionadas, e pode-se até mesmo ter sequências em Λ cuja região funcional em questão também ainda não foi identificada. A ideia geral dos métodos extrínsecos é localizar trechos da sequência analisada significativamente parecidos com sequências representativas da região funcional sendo buscada. Este método está baseado no princípio de conservação das bases, que pressupõe que as regiões funcionais estão menos suscetíveis a mutações aleatórias que aquelas sem função evidente.

Ferramentas baseadas nos métodos extrínsecos geralmente fazem uso de variantes dos algoritmos de alinhamento para comparar sequências. Um bom exemplo de uma variante que pode ser aplicada no problema de identificação de genes é o algoritmo que determina o melhor alinhamento global generalizado de duas sequências, descrito em [20]. Um **alinhamento global generalizado** é um alinhamento de duas sequências A e B, dadas como entrada, que consiste de substituições, buracos e blocos de diferença, e é apropriado para comparar pares de sequências que possuem trechos conservados (similares) separados por trechos não conservados. Uma substituição associa um caractere de A com um caractere de B. Um **buraco** consiste apenas de caracteres de uma sequência associados com o símbolo $-$ (espaço) na outra sequência. Um **bloco de diferença** consiste de caracteres de uma ou das duas sequências associados com o símbolo $+$. Existem três tipos de blocos de diferença. Um bloco de diferença do tipo 1 consiste apenas de caracteres de A, enquanto que um bloco de diferença do tipo 2 consiste apenas de caracteres de B e um bloco de diferença do tipo 3 consiste de caracteres de ambas as sequências. A Figura 7.7 mostra um exemplo de um alinhamento global generalizado. Como podemos ver na Figura 7.7, blocos de diferença são utilizados para representar grandes diferenças entre as sequências, enquanto *mismatches* e buracos representam diferenças menores.

Huang e Chao descrevem em [20] um algoritmo baseado em programação dinâmica para achar um alinhamento global generalizado ótimo de duas sequências, de acordo com uma função de pontuação (a função de pontuação, neste caso, deve considerar não apenas *matches*, *mismatches* e *spaces*, mas também buracos e blocos de diferença). Este algoritmo está implementado em um programa chamado *Global Alignment Program Version 3*, ou simplesmente **GAP3**. O GAP3 pode ser aplicado com sucesso ao tentar localizar regiões funcionais em sequências homólogas. Sequências homólogas,

```
GCGCTCCGGGACGCCTTCCGCCGTCGGGAGCCCTACAACTACCTGCAGAGGGCCTATTAC
++++++++++++++++++++++++++|||||||  ||||||||||||||||||||||||  |||
                          GGGAGCCTTACAACTACCTGCAGAGGGCCTACTAC

CAGGTGGGGAGCGGGCCGGGCAG                                        TAG
||||||  ||---|||||||  |||++++++++++++++++++++++++++++++++++++++++
CAGGTGCGG    GGGCCGGCCAGGGTGCTACCCCAAGCCTACTGACTGTCTTACTGG

CCTTCCCCAGAGCCCCCTAGCCGCAGGCACCAGAGGGTCCAAGACAAGACTGGAAGGGCA
++++++++++++++++++++++||  ||   |||  |  |||||  ||  ||   ||||  |  |  |
                       CAAGCTTCAGCGAGTCCAGGAGAAAGCTGGGAAGCCC

CCTCGGGTTCGG        GAGGAGCTGTGAGTGGCT
|   |||||  |||------|||||  ||||||  |||||++++++++++++++++++++++
CGCCGGGTCCGGGTCCGAGAGGAACTGTGAATGGCTGAGCCTGCTTCTCGAGGATCAGGC
```

Figura 7.7: *Um exemplo de um alinhamento global generalizado de duas sequências, retirado de [20]. Este alinhamento possui três blocos de diferença (cada um indicado por uma sequência de símbolos +), dois buracos (cada um indicado por uma sequência de símbolos −) e quatro trechos conservados (cada um indicado por uma sequência de símbolos | e espaços em branco).*

podendo ser até mesmo de organismos diferentes, geralmente possuem trechos que são bem parecidos, onde provavelmente se localizam as regiões funcionais, e trechos diferentes, onde provavelmente se localizam as regiões não funcionais [20]. O algoritmo de alinhamento global clássico visto no Capítulo 2 não é apropriado neste caso, pois mesmo que as duas sequências possuam regiões funcionais muito parecidas, geralmente as regiões não funcionais são muito maiores, o que pode diminuir consideravelmente a pontuação de um alinhamento global das duas sequências e ocultar a informação de que elas possuem regiões bem parecidas. A Figura 7.8 mostra uma representação em mais alto nível do alinhamento buscado pelo GAP3.

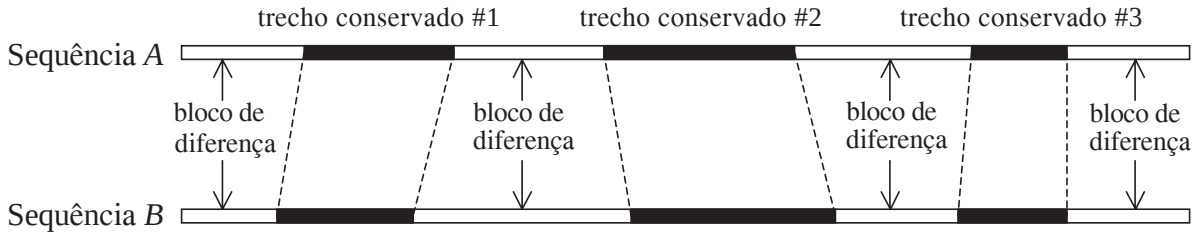


Figura 7.8: *Uma representação de um alinhamento global generalizado, onde trechos conservados das duas sequências (em preto) são alinhados e intercalados com blocos de diferença (em branco). Adaptada de [20].*

Apesar do GAP3 não tentar propriamente identificar um gene em um trecho de uma sequência de DNA, podemos utilizá-lo para encontrar os éxons (ou os possíveis éxons) de um gene. Para isto, basta executá-lo passando com entrada a sequência que codifica o gene em questão e uma sequência homóloga. Desta forma, espera-se que os trechos conservados identificados pelo GAP3 na primeira sequência sejam (ou incluem) os éxons do gene buscado.

A principal vantagem dos métodos extrínsecos fica evidente quando encontramos uma alta similaridade entre uma região da sequência sendo analisada e uma (ou mais) sequência(s) que codifica(m) um gene já descoberto. Isto nos dá boas indicações da localização do gene que a sequência analisada codifica e, além disso, informações relacionadas ao gene já conhecido podem auxiliar na determinação da função do gene na sequência analisada. No mesmo contexto, a desvantagem deste método surge quando não existe nenhum gene parecido com aquele que se quer localizar na sequência ana-

lisada. Contudo, devido ao desenvolvimento e aperfeiçoamento dos métodos de sequenciamento de DNA, atualmente temos à disposição um elevado número de DNAs total ou parcialmente sequenciados. Com esta grande base de dados e apoiados no princípio da conservação das bases, alguns estudos como [26] concluíram que o método de predição de genes por comparação de DNAs é o mais promissor para obtenção dos melhores resultados.

Nos métodos extrínsecos também temos a possibilidade de comparar uma sequência de DNA com uma sequência de cDNA (superficialmente, um cDNA é um trecho de um DNA composto apenas por éxons), RNA ou proteína. Por exemplo, podemos encontrar, dadas uma sequência de DNA e um cDNA que contém os éxons de um determinado gene G, quais trechos da sequência de DNA que mais se assemelham com o cDNA. Com isto, fica mais simples verificar se a sequência de DNA codifica um gene homólogo à G. Um exemplo de problema que pode ser aplicado na comparação de uma sequência de DNA com uma sequência de cDNA é o Problema do Alinhamento *Spliced*, introduzido na Seção 3.3.

7.3 Aplicação dos programas desenvolvidos

Assim como outros problemas da Biologia, a tarefa de identificação de genes pode ser modelada através de problemas da otimização combinatória, que por sua vez podem ser tratados utilizando conceitos e técnicas da Ciência da Computação. Neste trabalho, modelamos esta tarefa através do PASG e do PASGM. Neste sentido, os programas que implementam a solução do PASG e as heurísticas do PASGM podem ser vistos como ferramentas de identificação de genes baseadas em métodos extrínsecos, pois objetivam localizar os éxons que compõem um gene de uma sequência de DNA de interesse através da sua comparação com uma ou várias outras sequências de DNA.

Com o PASG, mais especificamente, estamos interessados em resolver o seguinte problema biológico: dadas duas sequências (ou trechos de duas sequências) de DNA que codificam dois genes homólogos e um conjunto dos prováveis éxons de cada gene, identificar quais destes éxons compõem os genes codificados em cada sequência, utilizando-se do princípio de conservação das bases. Para isso, consideramos $\Sigma = \{A, C, G, T\}$, s_1 e s_2 as duas sequências (ou trechos de duas sequências) de DNA da entrada, B e C os conjuntos ordenados dos prováveis éxons de cada gene e adotamos

$$\omega(a,b) = \begin{cases} 1, & \text{se } a = b \\ -1, & \text{se } a \neq b \\ -2, & \text{se } a = - \text{ ou } b = - \end{cases} .$$

Desta forma, podemos aplicar o Algoritmo 5 e obter como resposta duas cadeias de éxons Γ_B e Γ_C. Apoiando-se no princípio de conservação das bases, podemos supor que os genes homólogos codificados em cada sequência são muito parecidos, mesmo que eles tenham sido afetados por eventuais mutações genéticas. Desta forma, a probabilidade dos éxons que compõem Γ_B e Γ_C serem os éxons procurados (ou éxons reais) dos genes homólogos presentes em cada sequência é alta.

No problema de identificação de genes, assim como em vários outros, se aumentarmos a quantidade de evidências podemos obter melhores respostas. É com esta filosofia que propomos também utilizar o PASGM para resolver este problema biológico. Neste caso, ao invés de receber apenas duas sequências e seus respectivos conjuntos de possíveis éxons, recebemos $n > 2$ sequências $s_1, s_2, ..., s_n$ (contendo cada uma um gene homólogo) e seus respectivos conjuntos de possíveis éxons, B_1 de s_1, B_2 de s_2,..., e B_n de s_n, e aplicamos as heurísticas propostas no Capítulo 6 para determinar n cadeias de éxons $\Gamma_1, \Gamma_2, ..., \Gamma_n$. Apoiando-se novamente no princípio de conservação das bases, a probabilidade dos éxons que compõem $\Gamma_1, \Gamma_2, ..., \Gamma_n$ serem os éxons procurados dos genes homólogos presentes em cada sequência é alta. Com um maior número de sequências na entrada, esperamos obter resultados melhores do que aqueles obtidos com a aplicação do PASG no problema de identificação de genes.

Na primeira subseção desta seção descrevemos como foram obtidas as sequências de DNAs reais que compõem o conjunto de testes utilizado em nossa avaliação experimental. Na segunda subseção,

apresentamos o conjunto de medidas de avaliação de predições de genes utilizadas neste trabalho. Tais medidas possibilitam compararmos os programas de identificação de genes aqui desenvolvidos. Os resultados dessas comparações são mostrados na terceira subseção. No que segue, utilizaremos o termo PASG para referenciar tanto o problema como o programa desenvolvido baseado no algoritmo proposto.

7.3.1 Conjunto de testes

O conjunto de testes de nossa avaliação experimental foi baseado no conjunto de testes de [21, 36]. Em [21, 36], os autores utilizaram dados do projeto ENCODE [10] na geração do conjunto de testes para avaliação das ferramentas de identificação de genes propostas por eles. O ENCODE é um projeto que visa identificar todos os elementos funcionais e estruturais do DNA humano (*Homo Sapiens*). Para gerar o conjunto de testes, os autores filtraram todos os genes das 44 regiões analisadas na fase piloto do projeto ENCODE, escolhendo apenas os que codificassem uma e somente uma proteína (descartando pseudogenes e genes com *splicing* alternativo) e que obedecessem à estrutura de éxons e íntrons descrita na Seção 7.1.

Para cada gene G do *Homo Sapiens* escolhido em [21, 36], buscamos sequências de vários outros organismos que codificam um gene homólogo à G. Dentre esses organismos estão: *Anopheles gambiae, Arabidopsis thaliana, Ashbya gossypii, Bos taurus, Caenorhabditis elegans, Canis lupus, Danio rerio, Drosophila melanogaster, Gallus gallus, Macaca mulatta, Magnaporthe oryzae, Mus musculus, Neurospora crassa, Pan troglodytes, Rattus norvegicus, Saccharomyces cerevisiae* e *Schizosaccharomyces pombe*. As sequências com genes homólogos à G foram recuperadas da base de dados HomoloGene, que permite a detecção automática de genes homólogos anotados de vários DNAs eucariotos completamente sequenciados [37]. Quando buscamos por um determinado gene G na HomoloGene, obtemos como resultado um conjunto de sequências onde cada uma inclui um gene homólogo à G. Algumas filtragens foram então realizadas nas sequências retornadas por nossas buscas na HomoloGene. Foram eliminadas do conjunto de testes as sequências que: 1) não codificam nenhuma proteína; 2) codificam proteínas hipotéticas; 3) os éxons procurados de seu gene não estão completamente identificados; 4) não obedecem à estrutura de éxons e íntrons descrita na Seção 7.1; 5) possuem éxons compostos por outras bases além de A, C, G e T; 6) está na fita cuja orientação é 3' $\rightarrow$ 5'. Nos casos em que não foi encontrado nenhum homólogo para o gene G que atende às condições postas anteriores, G foi eliminado do conjunto de testes. Eventualmente, uma busca na HomoloGene identifica que um determinado organismo possui duas ou mais sequências que codificam genes homólogos à G. Neste caso, selecionamos apenas uma destas sequências arbitrariamente.

Na extração das sequências da HomoloGene, consideramos 1000 bases antes do início do gene e 1000 bases depois do fim do gene. As sequências extraídas desta forma irão compor o conjunto de testes, juntamente com o conjunto de prováveis éxons de cada gene de cada sequência.

Uma ferramenta simples, que chamamos de **Geradora de Éxons**, foi implementada para construir o conjunto de prováveis éxons que irão compor a entrada do PASG. Ela identifica trechos de uma sequência de DNA que possuem grande probabilidade de ser um dos éxons do gene codificado nessa sequência. Para gerar o conjunto de prováveis éxons B_1 de uma sequência s_1, precisamos de outra sequência homóloga de referência s_2. Dada essa sequência, primeiramente executamos o GAP3 passando como entrada as sequências s_1 e s_2, localizando assim os trechos conservados dessas sequências. Qualquer provável éxon identificado por nossa ferramenta está contido em algum trecho conservado identificado pelo GAP3 e possui tamanho de, no máximo, 300 nucleotídeos (este limite foi escolhido a partir de um estudo de [34]). Desta forma, teremos muitos prováveis éxons iniciais, internos e terminais. Num segundo passo, nós pontuamos cada um destes éxons criando, no final, um conjunto que inclui no máximo 50 éxons prováveis, onde 5 são iniciais, 40 são internos e 5 são terminais.

A pontuação P de um éxon interno é um valor que corresponde à combinação da pontuação de seu sítio de aceitação (P_1) e de doação (P_2), da pontuação dos códons que o compõem (P_3) e da pontuação ponderada do alinhamento global generalizado gerado pelo GAP3 (P_4). A pontuação dos

sítios de aceitação (P_1) e de doação (P_2) é dada através da pontuação da janela do sítio em questão calculada utilizando as WAMs definidas em [35]. A pontuação dos códons que compõem o éxon (P_3) é baseada no uso de códons. P_3 é o maior valor entre os *log-likelihood ratios* calculados para os três *frames* do éxon, onde utilizamos a Tabela 7.2 como referência para determinar a frequência de cada códon. Por fim, pontuamos o trecho em que está o éxon no alinhamento global generalizado gerado pelo GAP3 utilizando a função de pontuação

$$\omega(a,b) = \begin{cases} 1, & \text{se } a = b \\ -1, & \text{se } a \neq b \\ -2, & \text{se } a = - \text{ ou } b = - \end{cases},$$

e dividimos o valor obtido pelo tamanho do éxon, calculando assim P_4.

Depois de determinar P_1, P_2, P_3 e P_4 para todos os éxons internos contidos nos trechos conservados de s_1, normalizamos cada um desses valores no intervalo $[0,1]$, onde 0 denota a menor pontuação e 1 denota a maior pontuação. A pontuação P de um éxon interno é calculada então como $P = 0,3 * P_1 + 0,3 * P_2 + 0,1 * P_3 + 0,3 * P_4$. Por fim, são escolhidos os 40 éxons internos com maiores pontuações.

O processo é muito similar para éxons iniciais e terminais, mas utilizamos códons de iniciação e sítios de doação para os éxons iniciais e sítios de aceitação e códons de parada para os éxons terminais. Devemos observar que [35] não apresenta WAMs para códons de parada. Neste caso, utilizamos três PWMs, uma para o códon TAA, outra para o códon TAG e outra para o códon TGA, para organismos vertebrados apresentadas por Cavener e Ray em [9].

Seja G um gene do organismo *Homo Sapiens* que estamos considerando em nossos testes, s_{hu} a sequência de DNA humana que codifica G e suponha que encontramos n genes homólogos à G, que obedecem às condições citadas anteriormente para serem inclusos no conjunto de testes. Chamaremos de s_{ho_m}, onde $1 \leq m \leq n$, a sequência de DNA que codifica o m-ésimo gene homólogo à G e de B_{hu} e B_{ho_m} os conjuntos de prováveis éxons de s_{hu} e de s_{ho_m}, respectivamente. Para o gene G, geraremos n instâncias de teste, onde cada instância de teste será composta por $\{s_{hu}, B_{hu}, s_{ho_m}, B_{ho_m}\}$, com $1 \leq m \leq n$. Estabelecemos que se em uma dada instância de teste $B_{hu} = \emptyset$ ou $B_{ho_m} = \emptyset$, então esta instância é desconsiderada. Seguindo os mecanismos supracitados para geração do conjunto de testes, obtivemos 1068 instâncias de teste. Vale observar que nestas instâncias, temos uma média de 6,82 éxons procurados por sequência e a ferramenta Geradora de Éxons determina, em média, um conjunto com 48,88 prováveis éxons para cada sequência, onde apenas 46,7% de todos éxons procurados do conjunto de testes estão nos conjuntos de prováveis éxons gerados.

Da maneira como está definida cada instância de teste, podemos aplicar o PASG para tentar identificar os éxons procurados que compõem o gene humano e o gene homólogo. Por hora ainda não detalharemos como foi gerado o conjunto de testes para a avaliação do PASGM.

7.3.2 Medidas de avaliação

Para avaliarmos a qualidade de predição dos nossos algoritmos, utilizaremos as medidas de especificidade e sensibilidade propostas por Burset e Guigó em [6]. Estas duas medidas são extensivamente utilizadas na literatura para este fim e podem ser calculadas em dois níveis: o de nucleotídeos e o de éxons.

Para compreender melhor como a especificidade e a sensibilidade no nível de nucleotídeos e de éxons são calculadas, considere as seguintes definições. Todos os éxons que compõem um gene são chamados de **éxons procurados** ou **éxons anotados**. Éxons que foram preditos por uma ferramenta de identificação de genes são chamados de **éxons preditos**. Se as posições de início e fim de um éxon predito são as mesmas de um éxon procurado, dizemos que o éxon procurado foi **predito corretamente**. Nucleotídeos que compõem éxons procurados de um gene são chamados de **nucleotídeos realmente codificantes**. Caso contrário, eles são chamados de **nucleotídeos realmente não-codificantes**. Nucleotídeos que compõem éxons preditos por uma ferramenta são chamados de **nucleotídeos preditos como codificantes**. Caso contrário, eles são chamados de

nucleotídeos preditos como não-codificantes. Sejam **FN** a quantidade de nucleotídeos incorretamente preditos como não-codificantes, **FP** a quantidade de nucleotídeos incorretamente preditos como codificantes, **VN** a quantidade de nucleotídeos corretamente preditos como não-codificantes e **VP** a quantidade de nucleotídeos corretamente preditos como codificantes. A Figura 7.9, adaptada de [6], representa estas quatro medidas em um exemplo.

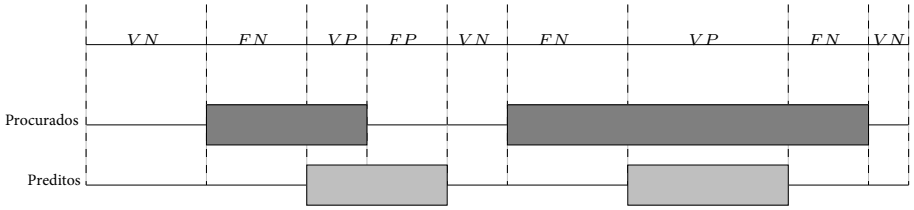


Figura 7.9: *Exemplo que representa as medidas FN, FP, VN e VP.*

A **especificidade no nível de nucleotídeos**, denotada por Sp_n, é a proporção de nucleotídeos realmente não-codificantes corretamente identificados como não-codificantes por uma ferramenta e pode ser calculada através da seguinte fórmula:

$$Sp_n = \frac{VN}{VN + FP}. \tag{7.1}$$

Um problema na definição do cálculo de Sp_n é que VN geralmente é muito maior que FP, produzindo valores altos e que não nos dá nenhuma informação sobre a qualidade da ferramenta. Desta forma, Sp_n é normalmente calculada através da seguinte fórmula alternativa, que é a que adotaremos neste texto:

$$Sp_n = \frac{VP}{VP + FP}. \tag{7.2}$$

Por outro lado, a **sensibilidade no nível de nucleotídeos**, denotada por Sn_n, é a proporção de nucleotídeos realmente codificantes corretamente identificados como codificantes por uma ferramenta e pode ser calculada através da seguinte fórmula:

$$Sn_n = \frac{VP}{VP + FN}. \tag{7.3}$$

Finalmente, uma medida conhecida como **Correlação Aproximada**, denotada por AC, é capaz de resumir as medidas Sp_n e Sn_n. A AC pode ser calculada através da seguinte fórmula:

$$AC = \frac{1}{2} * (\frac{VP}{VP + FN} + \frac{VP}{VP + FP} + \frac{VN}{VN + FP} + \frac{VN}{VN + FN}) - 1. \tag{7.4}$$

Para calcular a **especificidade e sensibilidade no nível de éxons**, denotadas por Sp_e e Sn_e respectivamente, considere que NEA é a quantidade de éxons procurados, NEP é a quantidade de éxons preditos por uma ferramenta e NEC é a quantidade de éxons procurados que foram preditos corretamente pela ferramenta. Sp_e mede a proporção de éxons procurados corretamente preditos pela ferramenta, com relação à quantidade de éxons preditos e pode ser calculada através da seguinte fórmula:

$$Sp_e = \frac{NEC}{NEP}. \tag{7.5}$$

Sn_e mede a proporção de éxons procurados corretamente preditos pela ferramenta, com relação à quantidade de éxons procurados e pode ser calculada por:

$$Sn_e = \frac{NEC}{NEA}. \tag{7.6}$$

Por fim, podemos resumir Sp_e e Sn_e em uma única medida conhecida por Av_e, onde:

$$Av_e = \frac{Sp_e + Sn_e}{2}. \tag{7.7}$$

O conceito de especificidade e sensibilidade pode ser estendido para o nível de bordas de éxons. As **bordas de um éxon** denotam as posições de início e fim do éxon. Neste nível, o objetivo é mensurar a quantidade de bordas preditas corretamente. Para calcular a **especificidade e sensibilidade no nível de bordas**, denotadas por Sp_b e Sn_b respectivamente, considere que NEA é a quantidade de éxons procurados, NEP é a quantidade de éxons preditos por uma ferramenta e NBC é a quantidade de bordas de éxons procurados que foram preditas corretamente pela ferramenta. As duas fórmulas seguintes calculam Sp_b e Sn_b, respectivamente:

$$Sp_b = \frac{NBC}{2 * NEP}; \tag{7.8}$$

$$Sn_b = \frac{NBC}{2 * NEA}. \tag{7.9}$$

Podemos sumarizar Sp_b e Sn_b na medida Av_b, onde:

$$Av_b = \frac{Sp_b + Sn_b}{2}. \tag{7.10}$$

Além das medidas descritas, calcularemos outros valores relacionados às predições dos éxons iniciais, internos e terminais. Para os éxons internos, contaremos quantos deles foram preditos corretamente pela ferramenta (**EC**) e quantos tiveram apenas uma de suas bordas preditas corretamente (**EB**). Para o caso dos éxons internos que não possuíram nenhuma borda predita corretamente, contaremos quantos deles estão contidos totalmente em um éxon predito (**ETP**), quantos estão contidos parcialmente em um éxon predito (**EPP**) e quantos não possuem nenhuma interseção com nenhum éxon predito (**ENP**). Estas medidas também serão aplicadas aos éxons iniciais e terminais.

7.3.3 Resultados obtidos

Nesta seção descrevemos os resultados obtidos em três comparações realizadas envolvendo o PASG e sua aplicação no problema de identificação de genes. Na primeira delas, comparamos o PASG com um programa que resolve um problema chamado PAPS, implementado em [36], com base no conjunto de testes descrito na Seção 7.3.1. O **PAPS** é um problema muito similar ao Problema do Encadeamento Bidimensional introduzido na Seção 3.3, com a diferença que o PAPS recebe uma função de pontuação ω a mais na entrada e não recebe um conjunto que inclui os valores das similaridades globais de alguns pares de segmentos, mas calcula estes valores *on-the-fly*. É importante notar que o PAPS também modela o problema de identificação de genes e é possível então construir uma ferramenta de predição de genes baseada em um algoritmo que o resolve. A segunda comparação é parecida com a primeira, mas os conjuntos de prováveis éxons das sequências incluem, obrigatoriamente, todos os éxons procurados[3]. A terceira comparação avalia o desempenho das três heurísticas propostas no Capítulo 6 entre si e com o PASG. Essas comparações foram realizadas em um computador com um processador Intel Core i3-2350M, com quatro núcleos de 2,3 GHz cada e 4 GB de RAM, no sistema operacional Windows 7. No que segue, utilizaremos o termo PAPS para referenciar tanto o problema como o programa que o resolve.

Primeira comparação

As medidas de especificidade e sensibilidade obtidas pelo PASG e pelo PAPS no conjunto de testes na primeira comparação podem ser vistas na Tabela 7.3.

Pela Tabela 7.3, podemos perceber que o PASG foi levemente superior que o PAPS no nível de nucleotídeos, obtendo um valor maior na medida AC, mas perdeu para o PAPS no nível de

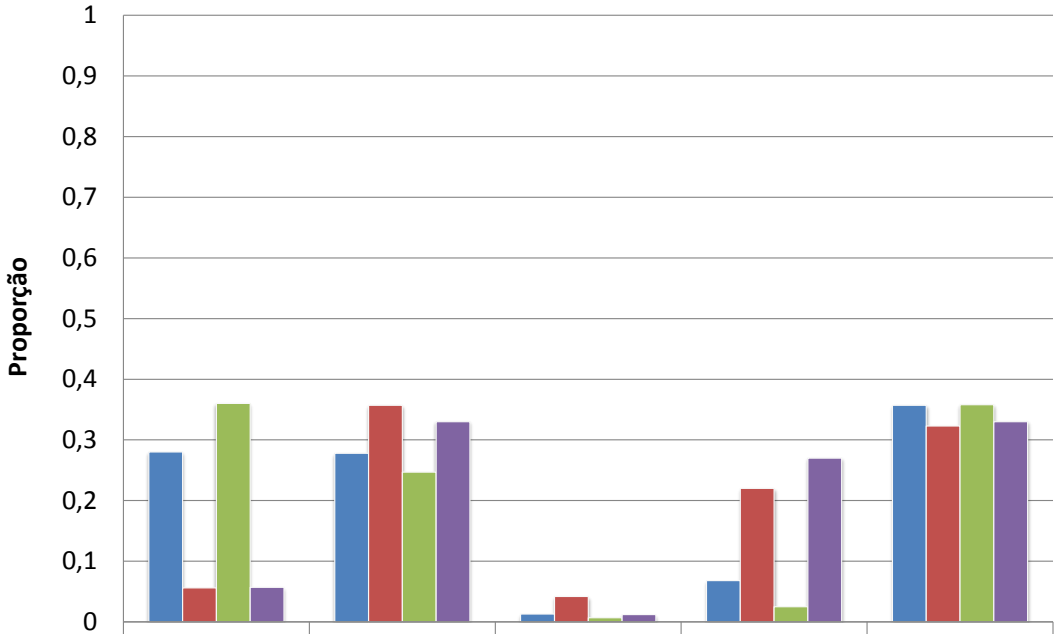
[3]Lembremo-nos que as instâncias de teste da primeira comparação incluem apenas 46,7% dos éxons procurados.

Ferramenta	Nucleotídeos			Éxons			Bordas		
	Sp_n	Sn_n	AC	Sp_e	Sn_e	Av_e	Sp_b	Sn_b	Av_b
PASG	0,723	**0,601**	**0,604**	0,215	0,187	0,201	0,363	**0,362**	0,362
PAPS	**0,734**	0,586	0,602	**0,231**	**0,192**	**0,212**	**0,381**	0,361	**0,371**

Tabela 7.3: *Resultado da comparação entre o PASG e o PAPS no conjunto de testes descrito na Seção 7.3.1.*

éxons e de bordas, obtendo valores menores nas medidas Av_e e Av_b. Acreditamos que este resultado insatisfatório de ambas as ferramentas deve-se, em parte, ao baixo desempenho de nossa ferramenta Geradora de Éxons. Como veremos mais adiante, quando os éxons procurados estão nos conjuntos de prováveis éxons das sequências, o PASG apresenta resultados melhores que o PAPS.

Para compreender melhor o comportamento do PASG nesta rodada de testes, a Figura 7.10 mostra a proporção dos éxons procurados que foram preditos corretamente pela ferramenta (EC), que tiveram apenas uma de suas bordas preditas corretamente (EB), que estão contidos totalmente em um éxon predito (ETP), que estão contidos parcialmente em um éxon predito (EPP) e que não possuem nenhuma interseção com algum éxon predito (ENP). Estes valores também foram calculados separadamente para os éxons iniciais, internos e terminais.



	EC	EB	ETP	EPP	ENP
Éxons	0,28	0,278	0,013	0,068	0,357
Iniciais	0,056	0,357	0,042	0,22	0,323
Internos	0,36	0,247	0,007	0,025	0,358
Terminais	0,057	0,33	0,012	0,27	0,33

Figura 7.10: *Gráfico que mostra a proporção dos éxons procurados que foram preditos corretamente pela ferramenta (EC), que tiveram apenas uma de suas bordas preditas corretamente (EB), que estão contidos totalmente em um éxon predito (ETP), que estão contidos parcialmente em um éxon predito (EPP) e que não possuem nenhuma interseção com algum éxon predito (ENP) pelo PASG na primeira comparação.*

Pela Figura 7.10, vemos que o PASG identificou corretamente apenas 28% dos éxons procurados, o que não é um número tão ruim se lembrarmos que somente 46,7% de todos éxons procurados estão nos conjuntos de prováveis éxons gerados. Pela mesma figura, vemos também que foi possível predizer corretamente um bom número de éxons internos (36%), enquanto que conseguimos predizer corretamente apenas 5,6% dos éxons iniciais e 5,7% dos éxons terminais. Nossa ferramenta conseguiu identificar corretamente uma das bordas de 35,7% dos éxons iniciais, de 33% dos éxons terminais,

de 24,7% dos éxons internos e de 27,8% de todos os éxons procurados. A predição dos éxons internos pela nossa ferramenta pode ser considerada boa, já que 60,7% (36% + 24,7%) dos éxons internos tiveram pelo menos uma de suas bordas preditas corretamente. Apesar da predição correta dos éxons iniciais e terminais ser muito aquém do esperado, o desempenho da ferramenta foi razoável na predição de uma das bordas destes éxons, tanto que em 41,3% (5,6% + 35,7%) dos éxons iniciais e em 38,7% (5,7% + 33%) dos éxons terminais foi possível predizer pelo menos uma de suas bordas. A medida ETP não nos dá muita informação e a medida EPP diz que 22% dos éxons iniciais e 27% dos terminais apresentaram uma interseção com algum éxon predito. Por fim, a medida ENP diz que não conseguimos predizer nenhum nucleotídeo de 35,7% dos éxons procurados, o que é um resultado insatisfatório. Finalmente, é importante observar que a ferramenta identificou um total de 16041 éxons, dos quais 6676 (41,6%) são éxons que não possuem nenhuma interseção com nenhum éxon procurado. Este é outro resultado ruim da ferramenta nessa rodada de testes, onde quase metade dos éxons preditos não possuem nenhum nucleotídeo em comum com um éxon procurado.

Segunda comparação

As medidas de especificidade e sensibilidade obtidas pelo PASG e pelo PAPS na segunda rodada de testes são apresentadas na Tabela 7.4. Nesta comparação, os conjuntos de prováveis éxons das sequências incluem todos os éxons procurados.

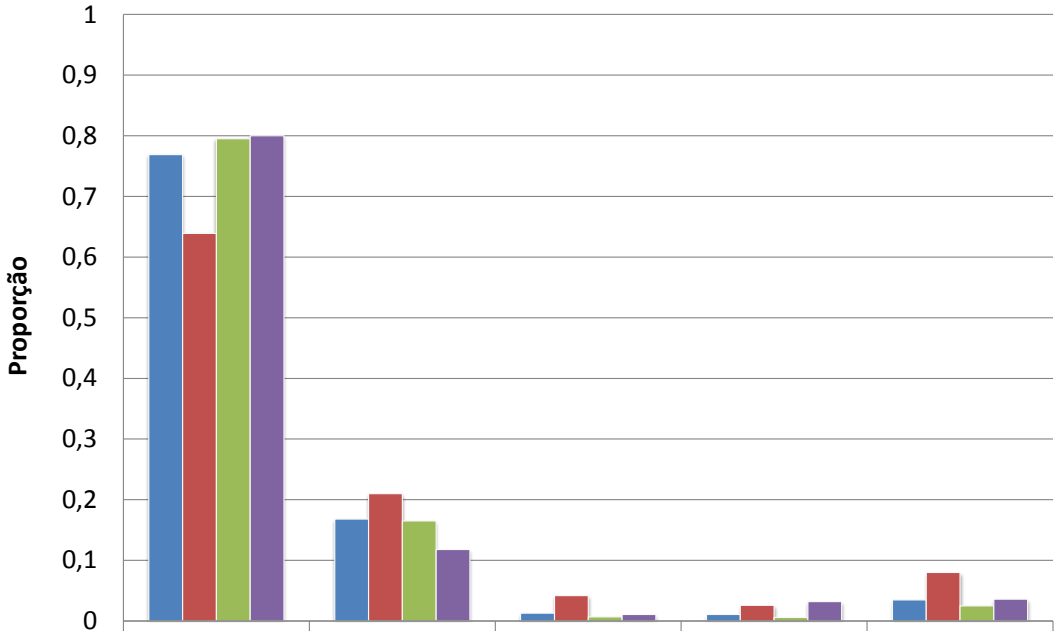
Ferramenta	Nucleotídeos			Éxons			Bordas		
	Sp_n	Sn_n	AC	Sp_e	Sn_e	Av_e	Sp_b	Sn_b	Av_b
PASG	0,785	**0,958**	**0,843**	**0,588**	**0,728**	**0,658**	**0,658**	**0,825**	**0,742**
PAPS	**0,794**	0,927	0,830	0,574	0,698	0,636	0,649	0,794	0,721

Tabela 7.4: *Resultado da comparação entre o PASG e o PAPS no conjunto de testes descrito na Seção 7.3.1, onde os conjuntos de prováveis éxons das sequências incluem todos os éxons procurados.*

Nesta segunda comparação, o PASG foi superior ao PAPS nos três níveis de avaliação, obtendo valores maiores para AC, Ave e Avb (observando a tabela, o PAPS foi superior ao PASG somente na medida Sp_n).

Para compreender melhor o comportamento do PASG nesta segunda rodada de testes, a Figura 7.11 mostra os valores de EC, EB, ETP, EPP e ENP obtidos pela ferramenta na segunda comparação. O PASG foi capaz de identificar corretamente 76,9% dos éxons procurados, o que é um bom valor. Mais especificamente, o PASG conseguiu identificar corretamente 80% dos éxons terminais, 79,5% dos éxons internos e 63,9% dos éxons iniciais. Como mostra a medida EC, a ferramenta apresentou melhores resultados na identificação correta dos éxons terminais e internos. Ademais, o PASG identificou corretamente uma das bordas de 16,8% de todos os éxons procurados, 21% dos éxons iniciais, 16,5% dos éxons internos e 11,8% dos éxons terminais. Apesar da proporção dos éxons iniciais corretamente identificados pela ferramenta ter sido mais baixa que a proporção dos éxons terminais e internos, o PASG identificou corretamente uma boa quantidade de uma das bordas dos éxons iniciais. Em números, nossa ferramenta conseguiu predizer pelo menos uma das bordas de 84,9% (63,9% + 21%) dos éxons iniciais, de 96% (79,5% + 16,5%) dos éxons internos e de 91,8% (80% + 11,8%) dos éxons terminais. No geral, 93,7% (76,9% + 16,8%) de todos os éxons procurados tiveram pelo menos uma de suas bordas identificadas corretamente. Consideramos este valor um resultado excelente obtido pela ferramenta. Comparando os valores obtidos nas medidas ETP e EPP, com os das medidas EC e EB, os primeiros não nos dão muitas informações, já que uma minoria dos éxons procurados foram categorizados nestas medidas. É importante observar que, considerando a medida ENP, apenas 3,5% dos éxons procurados não possuem nenhuma interseção com algum éxon predito, uma melhora significativa em relação à medida ENP da primeira comparação. Pelos dados apresentados na Figura 7.11, vemos que, em geral, a qualidade da predição do PASG em relação aos éxons iniciais foi inferior à qualidade apresentada pela ferramenta em relação aos éxons internos e terminais, tanto que 8% dos éxons anotados inicias não possuíam nenhuma

interseção com nenhum éxon predito. Na Seção 7.3.4 discutimos como podemos melhorar este aspecto em específico. Por fim, a ferramenta identificou um total de 16972 éxons, onde 2911 (17,1%) são éxons que não possuem nenhuma interseção com nenhum éxon procurado. Consideramos esse fato um bom resultado, que atesta, junto com os outros, um bom desempenho de nossa ferramenta nessa rodada de testes.



	EC	EB	ETP	EPP	ENP
Éxons	0,769	0,168	0,013	0,011	0,035
Iniciais	0,639	0,21	0,042	0,026	0,08
Internos	0,795	0,165	0,007	0,006	0,025
Terminais	0,8	0,118	0,011	0,032	0,036

Figura 7.11: *Gráfico que mostra a proporção dos éxons procurados que foram preditos corretamente pela ferramenta (EC), que tiveram apenas uma de suas bordas preditas corretamente (EB), que estão contidos totalmente em um éxon predito (ETP), que estão contidos parcialmente em um éxon predito (EPP) e que não possuem nenhuma interseção com nenhum éxon predito (ENP) pelo PASG na segunda comparação.*

Com uma análise mais detalhada do desempenho das ferramentas nesta rodada de testes, foi possível chegar a um resultado interessante. Para isto, dividimos o conjunto de testes em duas partes: na primeira, deixamos apenas as 161 instâncias cuja quantidade de éxons procurados na sequência humana é diferente da quantidade de éxons procurados na sequência homóloga e, na segunda, as 907 instâncias cuja quantidade de éxons procurados na sequência humana é igual à quantidade de éxons procurados na sequência homóloga. O resultado da comparação entre o PASG e o PAPS utilizando a primeira parte do conjunto de testes está descrito na Tabela 7.5 e, utilizando a segunda parte, na Tabela 7.6.

Ferramenta	Nucleotídeos			Éxons			Bordas		
	Sp_n	Sn_n	AC	Sp_e	Sn_e	Av_e	Sp_b	Sn_b	Av_b
PASG	0,896	**0,874**	**0,871**	**0,752**	**0,721**	**0,737**	**0,803**	**0,779**	**0,791**
PAPS	**0,898**	0,708	0,772	0,563	0,518	0,540	0,645	0,591	0,618

Tabela 7.5: *Resultado da comparação entre o PASG e o PAPS utilizando a primeira parte do conjunto de testes.*

Na Tabela 7.5, podemos ver que o PASG apresentou resultados bem superiores aos do PAPS, tanto que a diferença entre os valores obtidos pelas duas ferramentas nas medidas AC (9,9% de

Ferramenta	Nucleotídeos			Éxons			Bordas		
	Sp_n	Sn_n	AC	Sp_e	Sn_e	Av_e	Sp_b	Sn_b	Av_b
PASG	0,765	**0,973**	0,838	0,558	0,729	0,644	0,632	**0,834**	0,733
PAPS	**0,776**	0,966	**0,840**	**0,576**	**0,730**	**0,653**	**0,649**	0,830	**0,739**

Tabela 7.6: *Resultado da comparação entre o PASG e o PAPS utilizando a segunda parte do conjunto de testes.*

diferença), Av_e (19,7% de diferença) e Av_b (17,3% de diferença) é muito significativa em relação às duas comparações anteriores. Um fato que nos chamou a atenção é que o PASG se mostrou muito mais sensível que o PAPS, obtendo 16,6% a mais que o PAPS na medida Sn_n, 20,3% a mais na medida Sn_e e 18,8% a mais na medida Sn_b. Este resultado está diretamente relacionado com a formulação de ambos os problemas. Enquanto o PASG busca por duas cadeias de segmentos onde cada uma pode incluir uma quantidade diferente de segmentos, o PAPS busca por duas cadeias de segmentos que incluem, necessariamente, a mesma quantidade de segmentos. Esta diferença na formulação teórica dos problemas refletiu nos testes da primeira parte, onde o PASG apresentou resultados significativamente melhores que o PAPS quando o conjunto de testes inclui somente as instâncias cuja quantidade de éxons procurados na sequência humana é diferente da quantidade de éxons procurados na sequência homóloga.

A Tabela 7.6 mostra que o PAPS foi melhor que o PASG na segunda parte do conjunto de testes, que inclui somente as instâncias cuja quantidade de éxons procurados na sequência humana é igual à quantidade de éxons procurados na sequência homóloga. Novamente, isto era esperado, devido à formulação de ambos os problemas. Observamos porém que as diferenças entre os valores obtidos pelas duas ferramentas nas medidas AC (0,2% de diferença), Av_e (0,9% de diferença) e Av_b (0,6% de diferença) não são tão significativas, como nos testes da primeira parte. Mesmo com o PASG apresentando resultados piores do que os do PAPS na segunda parte do conjunto de testes, nossa ferramenta ainda se demonstrou levemente mais sensível. Este resultado nos permite especular que o PASG é muito mais apropriado que o PAPS quando as duas sequências possuem uma quantidade diferente de éxons procurados, mas ainda assim ele possui um desempenho próximo do PAPS quando as sequências possuem a mesma quantidade de éxons procurados.

Terceira comparação

Na terceira rodada de testes, o objetivo foi verificar o desempenho das três heurísticas propostas no Capítulo 6, quando aplicadas ao problema de identificação de genes. Para isto, primeiramente modificamos o conjunto de testes descrito na Seção 7.3.1 da seguinte forma. Seja G um gene do organismo *Homo Sapiens*, presente em nossos testes, e suponha que temos n genes homólogos à G. Desta forma, no conjunto de testes que usamos para testar o PASG e o PAPS, tínhamos n pares de sequências, cada par composto por uma sequência humana e uma sequência homóloga (e seus respectivos conjuntos de prováveis éxons). Para testar as heurísticas propostas para o PASGM nesta terceira comparação, nós consideramos a sequência humana e suas n sequências homólogas como uma única instância I do conjunto de testes. Nesta instância I também temos, além das sequências, os conjuntos de seus prováveis éxons, que são os mesmos do conjunto de testes do PASG e do PAPS. É importante observar que, para a sequência humana, temos n conjuntos de prováveis éxons, conforme o mecanismo que utilizamos para gerá-los. Para escolher um único conjunto de prováveis éxons da sequência humana, olhamos para os resultados obtidos pelo PASG na segunda comparação (Tabela 7.4) e selecionamos a instância I' que obteve o maior Av_e na sequência humana. Assim, o conjunto de prováveis éxons da sequência humana da instância I' é o conjunto de prováveis éxons da sequência humana da instância I, nessa última rodada de testes. Vale notar que nesta terceira comparação, assim como na segunda, os conjuntos de prováveis éxons das sequências obrigatoriamente incluem todos os éxons procurados.

Utilizando o mecanismo supracitado, geramos um conjunto de instâncias para testar as heurísticas propostas para o PASGM no problema de identificação de genes. Neste conjunto, excluímos

cinco instâncias que possuíam apenas um homólogo, já que nesses casos não faz sentido executar nenhuma das heurísticas, obtendo 206 instâncias de teste.

Nesta última rodada de testes, comparamos não só as heurísticas entre si, mas também com o PASG. Com isto, buscamos uma resposta para as seguintes perguntas: dadas uma sequência que contém um gene G do organismo *Homo Sapiens* e $n > 1$ sequências que contêm genes homólogos à G, qual das três heurísticas é a melhor para identificar os éxons presentes na sequência humana? Ou será que nós conseguimos uma melhor qualidade na predição se executarmos n vezes o programa que resolve o PASG (uma vez para cada sequência homóloga) e selecionar o melhor resultado destas execuções? Para responder a essas perguntas, nós calculamos as medidas de especificidade e sensibilidade no nível de nucleotídeos, éxons e bordas levando em conta apenas os resultados obtidos pelas ferramentas na sequência humana. Para comparar os resultados das heurísticas com o PASG, selecionamos, dentre as n execuções do PASG, aquela que obteve o maior valor de Av_e. O resultado da comparação entre as três heurísticas propostas para o PASGM e o PASG, conforme descrita, é apresentado na Tabela 7.7.

Ferramenta	Nucleotídeos			Éxons			Bordas		
	Sp_n	Sn_n	AC	Sp_e	Sn_e	Av_e	Sp_b	Sn_b	Av_b
Heurística 1	**0,945**	0,935	0,923	**0,897**	0,881	**0,889**	**0,919**	0,901	0,910
Heurística 2	0,910	**0,994**	0,940	0,803	0,880	0,841	0,838	0,925	0,882
Heurística 3	0,942	0,988	**0,958**	0,865	**0,909**	0,887	0,894	**0,943**	**0,919**
PASG	0,916	0,969	0,933	0,826	0,881	0,854	0,856	0,916	0,886

Tabela 7.7: *Resultado da comparação entre as três heurísticas propostas para o PASGM e o PASG.*

Como podemos ver na Tabela 7.7, o PASG não foi a melhor ferramenta em nenhuma medida. Considerando as medidas que sumarizam os resultados, seu desempenho só foi melhor que o da Heurística 1 na medida AC e que o da Heurística 2 nas medidas Av_e e Av_b. O desempenho da Heurística 2 também foi ruim, já que obteve os piores valores nas medidas Av_e e Av_b, apesar de ter obtido o melhor valor na medida Sn_n e o segundo melhor valor na medida AC. Contudo, as Heurísticas 1 e 3 foram superiores ao PASG e à Heurística 2. Devido a esse fato, acreditamos que, no problema de identificação de genes, se aumentarmos a quantidade de evidências podemos obter melhores respostas. Neste caso específico, obter várias sequências homólogas a uma determinada sequência humana e tratá-las em conjunto, como uma única instância, através das heurísticas propostas para o PASGM produziu resultados melhores do que tratar estas sequências homólogas individualmente através de várias execuções do programa que implementa o PASG.

Voltando a nossa atenção para o desempenho das ferramentas, concluímos que a Heurística 3 foi a mais apropriada nesta rodada de testes. Nossa conclusão está baseada no fato dela ter obtido o melhor valor nas medidas AC e Av_b e perder na medida Av_e para Heurística 1 por apenas 0,2%. A Heurística 3 também se demonstrou bastante sensível, obtendo os melhores valores nas medidas Sn_e e Sn_b e o segundo melhor valor na medida Sn_n. A Heurística 1 ficou em segundo lugar por obter o melhor valor na medida Av_e e se demonstrar bastante específica, superando as outras ferramentas em todas as medidas de especificidade.

7.3.4 Discussão

O PASG e o PASGM são problemas teóricos com aplicação na tarefa prática de identificar os éxons de um gene codificado em uma sequência de DNA. Baseado nisso, é importante notar que os resultados obtidos na Seção 7.3.3 podem ser melhorados significativamente se incorporarmos informações biológicas e químicas sobre os genes na formulação do PASG e do PASGM. Por exemplo, uma restrição que podemos adicionar em nossos programas para obtermos uma predição com maior qualidade, principalmente em relação aos éxons iniciais e terminais, é considerar que uma cadeia é válida somente se o seu primeiro éxon começa com o códon ATG e seu último éxon termina com os códons TAA, TAG ou TGA.

Outra melhoria em nossos programas seria a consideração de uma pontuação diferenciada para buracos no processo de alinhamento. Quando duas sequências de DNA são comparadas, e assumimos a possibilidade da ocorrência de mutações, a existência de um buraco com k *spaces* é mais provável do que a existência de k *spaces* isolados. Isto acontece porque um buraco é devido a um único evento de mutação que modificou um trecho de nucleotídeos consecutivos de uma sequência, enquanto que *spaces* isolados representam vários eventos distintos de mutações, e a ocorrência de um evento de mutação é mais comum do que a ocorrência de vários [38]. Diferenciar *spaces* e buracos no processo de alinhamento, atribuindo pesos diferentes a eles através de funções gerais e afins de penalização, fará com que nossos programas busquem alinhamentos que favoreçam *spaces* consecutivos a *spaces* isolados e, desta forma, que possuam mais coerência em termos biológicos.

Quando buscamos por genes que codificam proteínas em uma sequência de DNA, funções de pontuações simples, como as que temos utilizado em nossos programas, às vezes não são as melhores opções. Proteínas correspondem a cadeias de aminoácidos e, como já vimos, os trechos do DNA que as codificam estão menos suscetíveis à mutações aleatórias. Contudo, se mutações ocorrerem em trechos que codificam proteínas, as propriedades bioquímicas dos aminoácidos que compõem a proteína em questão influenciam na mutação. Por exemplo, é mais comum que um aminoácido seja substituído por outro que possui um tamanho similar ao primeiro, do que por outro com um tamanho muito maior ou muito menor. A tendência de determinados aminoácidos se ligarem à moléculas de água também pode influenciar na mutação [38]. Além disso, existem vários outros fatores químicos e biológicos que podemos incluir em nossas ferramentas para melhorar a qualidade da predição dos éxons procurados de um gene. Na literatura, isto é feito geralmente através de funções de pontuações conhecidas por **matrizes PAM**, que relacionam um par de aminoácidos com um valor real. As matrizes PAM tentam representar a probabilidade de substituir um determinado aminoácido por outro, considerando diversos fatores químicos e biológicos, como os dois citados anteriormente. Geralmente, estas matrizes são construídas através da contagem e do cálculo da taxa de substituição de um aminoácido por outro, observando uma grande quantidade de mutações em um conjunto de proteínas relacionadas. Para mais detalhes sobre matrizes PAM e como elas são calculadas, sugerimos [38].

Utilizando matrizes PAM, podemos considerar o alinhamento dos aminoácidos de sequências de DNA, ao invés do alinhamento de seus nucleotídeos. Note que, desta forma, temos três cadeias diferentes de aminoácidos codificadas por uma sequência de DNA, uma cadeia para cada *frame* da sequência. Uma vantagem de considerar o alinhamento dos aminoácidos de sequências de DNA é que certas mutações não provocam nenhum efeito no processo de síntese de proteínas. Por exemplo, se em algum trecho do DNA temos o códon TTG, que codifica para o aminoácido Leucina, e este trecho sofrer uma ou mais mutações que alteram o códon TTG para CTT, CTC, CTA ou CTG, nenhuma modificação será percebida pelo mecanismo celular. Isto porque estes cinco códons são sinônimos, ou seja, codificam para o mesmo aminoácido. Logo, o processo de alinhamento de aminoácidos de sequências de DNA trataria estes casos corretamente, enquanto que, em um alinhamento de nucleotídeos, códons sinônimos são diferenciados, o que não é apropriado considerando o problema de identificação de genes.

Por fim, o objetivo deste capítulo não foi mostrar que nossos programas são boas ferramentas para a predição dos éxons de um gene codificado em uma sequência de DNA. Para isto, seria necessário elencar as melhores ferramentas presentes na literatura para este fim e criar conjuntos de testes para então comparar e averiguar, de fato, a qualidade de nossos programas. Esta comparação poderá ser realizada futuramente, mas nosso objetivo aqui foi apenas mostrar a aplicabilidade de nossa modelagem. Isto significa que é válido utilizar o PASG e o PASGM para modelar problemas práticos e que eles apresentaram resultados apropriados quando os aplicamos no problema de identificação de genes, mesmo sem incorporarmos nenhuma informação biológica ou química para melhorar a qualidade da predição.

Capítulo 8

Conclusão

Neste trabalho formulamos dois novos problemas de otimização combinatória: o Problema do Alinhamento de Segmentos (PASG) e sua versão múltipla, o Problema do Alinhamento de Segmentos Múltiplo (PASGM). Desenvolvemos então um algoritmo eficiente baseado na técnica de programação dinâmica que soluciona o PASG e verificamos que não existe um algoritmo polinomial para o PASGM, através de uma prova de sua NP-Completude. Além disso, concluímos também que não é possível desenvolver um algoritmo de aproximação para esse último problema, a não ser que P seja igual a NP. Devido a essas duas propriedades do PASGM, propomos três heurísticas para tratá-lo, que foram avaliadas experimentalmente comparando os valores das soluções devolvidas por elas em instâncias de testes criadas artificialmente e aleatoriamente.

Aplicamos as soluções desenvolvidas neste trabalho na tarefa prática de identificação de genes, modelando-a através do PASG e do PASGM. Utilizando as medidas de especificidade e sensibilidade de Burset e Guigó [6] avaliamos as ferramentas desenvolvidas em instâncias de testes reais atestando sua aplicabilidade na tarefa de identificação de genes. Mais especificamente, vimos que o PASG pode ser usado para identificar os éxons de dois genes homólogos codificados em duas sequências, sendo bastante apropriado nos casos em que esses genes possuem estruturas diferentes. No fim, foi possível verificar, através de testes com instâncias reais, que a Heurística da cadeia de segmentos central foi a ferramenta mais apropriada para a tarefa de identificação de genes considerando todos os programas desenvolvidos neste trabalho. Um fato interessante foi que as heurísticas apresentaram resultados melhores que o PASG nessa tarefa, atestando que, em nossos experimentos, se aumentarmos a quantidade de evidências obtemos melhores respostas.

Não podemos dizer que o estudo do PASG e do PASGM encerra-se aqui. Algo que pode ser desafiador ao aplicar nossas soluções na prática é o alto custo de processamento e de espaço ao tratarmos de sequências longas e/ou um número expressivo de segmentos. Nesse sentido, seria válido um estudo mais aprofundado do PASG na busca de algoritmos mais eficientes ou de uma prova de que nosso algoritmo é ótimo. Uma grande questão em aberto seria se podemos modelar outras tarefas e problemas práticos através do PASG. Nesse contexto, dependendo do campo de aplicação, se o custo de processamento e de espaço do algoritmo aqui proposto for proibitivo, seria interessante também o estudo de algoritmos mais rápidos e que devolvam soluções sub-ótimas.

Nesta dissertação apenas iniciamos o estudo do PASGM, demonstrando que ele é NP-Completo e que é muito improvável existir um algoritmo de aproximação com uma boa razão para ele. Contudo, nossos resultados são válidos apenas para determinadas funções de pontuação. Explorar a complexidade desse problema com diferentes classes de funções de pontuações (ou até mesmo com funções de pontuações genéricas) ainda é um desafio que pode revelar vários resultados interessantes. Em especial, acreditamos que é possível encontrar resultados relevantes ao estudar a complexidade do PASGM com funções de pontuação que induzem métricas, como a distância de edição, por exemplo. Seguindo uma outra linha, outros trabalhos futuros que podemos citar acerca do PASGM é o estudo de algoritmos probabilísticos, modelos de programação linear e metaheurísticas, como algoritmos genéticos e a técnica GRASP [13], na tentativa de tratá-lo. Também seria interessante uma avaliação experimental mais rigorosa do que a realizada neste trabalho, que capture melhor os pontos

fracos e fortes de cada heurística.

Num contexto mais prático, também é pertinente explorar melhor a aplicação de nossas ferramentas na tarefa de identificação de genes. Essa exploração envolve, primeiramente, melhorar a qualidade da ferramenta Geradora de Éxons, cujo objetivo é construir o conjunto de prováveis éxons de uma sequência de DNA. Isso pode ser feito incorporando outras métricas estatísticas em nosso programa, como o uso de aminoácidos e hexamers, e considerando outros métodos de busca de sinais, como os modelos de máxima entropia [44], redes Bayseanas [8] e modelos ocultos de Markov [23]. Em segundo lugar, seria também interessante verificar se e o quanto a qualidade dos resultados do PASGM aplicado à tarefa de identificação de genes melhora com o aumento da quantidade de sequências evolutivamente relacionadas na entrada. Ademais, também seria pertinente implementar e testar as melhorias discutidas na Seção 7.3.4, como a incorporação de informações biológicas e químicas em nossos programas e a consideração de uma pontuação diferenciada para buracos e do alinhamento de aminoácidos de sequências de DNA utilizando matrizes PAM.

Os resultados obtidos neste trabalho mostram que cumprimos o objetivo de estudar detalhadamente o Problema do Alinhamento de Segmentos e sua versão múltipla, o Problema do Alinhamento de Segmentos Múltiplo. Atestamos a aplicabilidade desses problemas na tarefa prática de identificação de genes e mostramos trabalhos futuros que podem estender este estudo, tanto em seu contexto teórico quanto prático.

Apêndice A

Observações sobre a implementação dos algoritmos propostos

Neste apêndice discutimos algumas observações sobre a implementação do algoritmo que propusemos para o PASG e das heurísticas desenvolvidas para tratar o PASGM. Todas as nossas soluções foram implementadas na linguagem de programação C e seus códigos-fonte encontram-se disponíveis publicamente no sítio http://www.facom.ufms.br/~said/pasg/implementacao.zip.

A.1 Função de pontuação

Na implementação dos algoritmos propostos nesse trabalho, fixamos a função de pontuação ω em:

$$\omega(a,b) = \begin{cases} 1, & \text{se } a = b \\ -1, & \text{se } a \neq b \\ -2, & \text{se } a = - \text{ ou } b = - \end{cases} .$$

A.2 Execução dos programas

Para a correta execução dos nossos programas, é necessário especificar na entrada as sequências e seus respectivos conjuntos ordenados de segmentos. No caso do PASG, basta especificar apenas duas sequências s_1 e s_2, um conjunto ordenado $B = \{b_1, b_2, ..., b_u\}$ de segmentos de s_1 e um conjunto ordenado $C = \{c_1, c_2, ..., c_v\}$ de segmentos de s_2. Isso é feito informando dois argumentos por linha de comando. Cada um desses dois argumentos denotará o caminho de um arquivo que guarda uma das sequências de entrada, que deve estar no formato FASTA [1]. O arquivo que armazena o conjunto ordenado de segmentos de uma determinada sequência é deduzido a partir do caminho do arquivo que contém essa sequência. De forma geral, o sufixo .PREDEX é adicionado ao caminho do arquivo que contém a sequência s_i para deduzir o caminho do arquivo que contém o conjunto ordenado de segmentos B_i. Um arquivo que armazena um conjunto ordenado de n segmentos obedece ao seguinte formato: $i_1..f_1, i_2..f_2, ..., i_n..f_n$, onde i_k e f_k representam, respectivamente, o início e o fim do segmento k. Além disso, assume-se que esses segmentos estão ordenados.

Sobre as três heurísticas propostas para o PASGM, elas foram implementadas em um único programa. Para a execução de uma heurística que recebe $n > 2$ sequências $s_1, s_2, ..., s_n$ e n conjuntos ordenados de segmentos $B_1, B_2, ..., B_n$, é necessário invocar esse programa e especificar vários argumentos por linha de comando. O primeiro argumento é um inteiro entre 1 e 3 e indica qual das três heurísticas deve ser executada, onde 1 especifica a Heurística da cadeia de segmentos consenso, 2 especifica a Heurística gulosa e 3 especifica a Heurística da cadeia de segmentos central. Após esse primeiro argumento, é necessário especificar n argumentos que denotarão os caminhos dos arquivos que incluem as n sequências de entrada. Cada um desses argumentos segue a mesma ideia dos dois argumentos que devem ser passados para executar o programa que resolve o PASG. Dessa forma,

os caminhos dos arquivos que contêm cada conjunto ordenado de segmentos são deduzidos dos caminhos dos arquivos que contêm as sequências.

É importante notar que, de acordo como está implementada nossas soluções, cada sequência da entrada é construída sobre o alfabeto $\Sigma = \{A, C, G, T\}$ e as cadeias de segmentos que compõem a resposta encontrada pelos programas são escritas na saída padrão.

A.3 Uma otimização do algoritmo que resolve o PASG

A implementação do algoritmo que resolve o PASG segue estritamente os Algoritmos 4 e 5, com exceção de uma otimização, detalhada a seguir. Em alguns experimentos, percebemos que não conseguíamos executar nosso programa porque ele precisava de uma quantidade de memória maior do que o limite imposto pelo sistema operacional. Dessa forma, tivemos que otimizar a quantidade de memória utilizada pelo programa. Fizemos isso modificando os dados armazenados na matriz de alinhamento de segmentos M e na matriz de apontadores P. Conforme está descrito no Capítulo 4, cada uma destas matrizes armazenam $1 + \sum_{b \in B} |b| + \sum_{c \in C} |c| + \sum_{b \in B,\ c \in C} |b| * |c|$ células. Podemos melhorar essa complexidade de espaço consideravelmente e assintoticamente armazenando, em M e P, somente a última linha e a última coluna de cada submatriz de alinhamento. Contudo, ainda é necessário calcular todas as células de cada submatriz de alinhamento, o que fazemos em duas novas matrizes auxiliares M_A e P_A, ambas de dimensões $bMax \times cMax$, onde $bMax = max\{|b| : b \in B\}$ e $cMax = max\{|c| : c \in C\}$. Após calcular todas as células de uma determinada submatriz de alinhamento em M_A e P_A, copiamos para M e P somente a última linha e coluna de M_A e P_A, respectivamente. Para recuperar quais segmentos de B e de C fazem parte da solução ótima com essa otimização, cada célula c de uma submatriz p de P (lembrando que c é uma célula da última linha e/ou da última coluna de p) deverá então apontar para a célula c' da submatriz p' de P, onde c' é a primeira célula que aparecia no caminho de c a $P[0][0][0][0]$ (antes da otimização) tal que $p' \neq p$. Note que c' é necessariamente uma célula da última linha e/ou coluna de p'.

Com a otimização detalhada no parágrafo anterior, é possível resolver o PASG com a matriz M e P armazenando, cada uma, $1 + \sum_{b \in B} |b| + \sum_{c \in C} |c| + \sum_{b \in B,\ c \in C} (|b| + |c| - 1)$ células. Além disso, é necessário alocar as matrizes auxiliares M_A e P_A, onde cada uma requer $bMax * cMax$ células, tal que $bMax = max\{|b| : b \in B\}$ e $cMax = max\{|c| : c \in C\}$. Essa otimização melhora a complexidade de espaço do algoritmo que resolve o PASG assintoticamente. Sem a otimização descrita, o algoritmo utiliza $O(u * v * bMax * cMax)$ células, enquanto que com a otimização ele utiliza $O(u * v * (bMax + cMax) + bMax * cMax)$ células.

A.4 Observações sobre a implementação das heurísticas para o PASGM

É importante notar que as implementações das heurísticas seguem como base os Algoritmos 6, 7 e 8, mas possuem algumas otimizações que diminuem seus tempos de execução, apesar de não melhorar suas complexidades de tempo assintoticamente.

Na terceira heurística, o algoritmo definido por Gelfand *et al.* em [17] para a resolução do Problema do Alinhamento *Spliced* é utilizado como uma função auxiliar. Segundo [17], esse algoritmo encontra-se implementado no programa PROCRUSTES 2.0, disponível publicamente no sítio http://www_hto.usc.edu/software/procrustes/. Contudo, algumas tentativas sem sucesso foram realizadas para obtenção deste programa no sítio especificado (última tentativa de acesso data de 15 de Abril de 2013). Devido a esse fato, decidimos utilizar uma implementação desse algoritmo na linguagem C desenvolvida por Kishi em [21], disponível publicamente no sítio https://code.google.com/p/projetomestrado/ (último acesso em 15 de Abril de 2013) sob o nome de PROCRUSTES. Os seguintes arquivos de código-fonte estão associados a essa implementação: `alglib.h`, `alglib.c`, `biolib.h`, `biolib.c`, `iolib.h`, `iolib.c`, `procrustes.h` e `procrustes.c`.

Referências Bibliográficas

[1] What is FASTA format? http://zhanglab.ccmb.med.umich.edu/FASTA/. Último acesso em 03/05/2012. 77

[2] S. S. Adi. *Identificação de Genes por Comparação de Sequências*. PhD thesis, IME-USP, São Paulo, Brasil, 2005. 53

[3] S. S. Adi and C. E. Ferreira. Uma avaliação de ferramentas para a predição de genes. *Anais do XXII Congresso da Sociedade Brasileira de Computação*, 3:133–143, 2002. 59

[4] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Science, 5th edition, 2007. 53

[5] A. F. Bent. Plant disease resistance genes: Function meets structure. *The Plant Cell*, 8: 1757–1771, 1996. 58

[6] M. Burset and R. Guigó. Evaluation of gene structure prediction programs. *Genomics*, 34: 353–367, 1996. 2, 59, 66, 67, 75

[7] M. H. Carvalho, M. Cerioli, R. Dahab, P. Feofiloff, C. G. Fernandes, C. E. Ferreira, K. S. Guimarães, F. K. Miyazawa, J. C. Pina Jr., and J. A. Soares. *Uma introdução sucinta a algoritmos de aproximação*. Editora do IMPA, 1st edition, 2001. 3, 7, 8

[8] R. Castelo and R. Guigó. Splice site identification by idlBNs. *Bioinformatics (Oxford, England)*, 20 Suppl 1, 2004. 76

[9] D. R. Cavener and S. C. Ray. Eukaryotic start and stop translation sites. *Nucleic Acids Research*, 19(12):3185–92, 1991. 66

[10] The ENCODE Project Consortium. The ENCODE (ENCyclopedia Of DNA Elements) Project. *Science*, 306(5696):636–640, 2004. 65

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Science / Engineering / Math, 3rd edition, 2009. 3, 4, 5, 6, 8, 9, 12, 29

[12] L. Duret, D. Mouchiroud, and C. Gautier. Statistical analysis of vertebrate sequences reveals that long genes are scarce in gc-rich isochores. *Journal of Molecular Evolution*, 40:308–317, 1995. 58

[13] T. A. Feo and M. G. C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8(2):67 – 71, 1989. 75

[14] P. Feofiloff, A. Carvalho, and T. Kowaltowski. Minicurso de Análise de Algoritmos. http://www.ime.usp.br/~pf/livrinho-AA/. Último acesso em 15/02/2012. 3, 4, 5

[15] J. W. Fickett and C.-S. Tung. Assessment of protein coding measures. *Nucleic Acids Research*, 20(24):6441–6450, 1992. 60

[16] M. S. Gelfand. Prediction of function in DNA sequence analysis. *Journal of computational biology*, 2(1):87–115, 1995. 61

[17] M. S. Gelfand, A. A. Mironov, and P. A. Pevzner. Gene Recognition Via Spliced Sequence Alignment. *Proceedings of the National Academy of Sciences of the United States of America*, 93:9061–9066, 1996. 1, 24, 25, 49, 50, 78

[18] R. Guigo. *GENETIC DATABASES: DNA Composition, Codon Usage and Exon Prediction*, chapter DNA Composition, Codon Usage and Exon Prediction (chapter 4), pages 53–80. Academic Press, 1999. 59, 60

[19] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997. 1, 3, 12, 25

[20] X. Huang and K.-M. Chao. A generalized global alignment algorithm. *Bioinformatics*, 19(2): 228–233, 2003. ix, x, 62, 63

[21] R. M. Kishi. Identificação de Genes e o Problema do Alinhamento Spliced Múltiplo. Master's thesis, FACOM-UFMS, Campo Grande, Brasil, 2010. 53, 65, 78

[22] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2005. 3, 4, 5, 6, 7

[23] A. S. Krogh. *An introduction to hidden Markov models for biological sequences*, pages 45–63. Elsevier, 1998. 76

[24] B. Lewin. *Genes VII*. Oxford, 1st edition, 2001. 53

[25] D. Maier. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM*, 1978. 41

[26] C. Mathé, M. F. Sagot, T. Schiex, and P. Rouzé. Current methods of gene prediction, their strengths and weaknesses. *Nucleic acids research*, 30(19):4103–4117, 2002. 10, 64

[27] D. Mouchiroud, G. D'Onofrio, B. Aïssani, G. Macaya, C. Gautier, and G. Bernardi. The distribution of genes in the human genome. *Gene*, 100:181–187, 1991. 59

[28] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48:443–453, 1970. 3, 10, 12, 16

[29] P. S. Novichkov, M. S. Gelfand, and A. A. Mironov. Gene recognition in eukaryotic DNA by comparison of genomic sequences. *Bioinformatics*, 17:1011–1018, 2001. 25, 26

[30] J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley, 1st edition, 1984. 3, 8, 51

[31] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98:9748–9753, 2001. 10

[32] B. M. Porcel, A. Tran, M. Tammi, Z. Nyarady, M. Rydåker, T. P. Urmenyi, E. Rondinelli, U. Pettersson, B. Andersson, and L. Åslund. Gene survey of the pathogenic protozoan trypanosoma cruzi. *Genome Research*, 10:1103–1107, 2000. 58

[33] Zhang M. Q. and Marr T. G. A weight array model for splicing signal analysis. *Computer Applications in the Biosciences*, 9:499–509, 1993. 61

[34] M. K. Sakharkar, V. T. Chow, and P. Kangueane. Distributions of exons and introns in the human genome. *In Silico Biol.*, 4(4):387–393, 2004. 65

[35] S. Salzberg. A method for identifying splice sites and translational start sites in eukaryotic mRNA. *Bioinformatics*, 13(4):365–376, 1997. ix, 61, 62, 66

[36] R. F. Santos. Identificação de Genes por comparação de DNAs. Master's thesis, FACOM-UFMS, Campo Grande, Brasil, 2010. 53, 65, 68

[37] Eric W. Sayers, Tanya Barrett, Dennis A. Benson, Evan Bolton, Stephen H. Bryant, Kathi Canese, Vyacheslav Chetvernin, Deanna M. Church, Michael DiCuccio, Scott Federhen, Michael Feolo, Lewis Y. Geer, Wolfgang Helmberg, Yuri Kapustin, David Landsman, David J. Lipman, Zhiyong Lu, Thomas L. Madden, Tom Madej, Donna R. Maglott, Aron Marchler-Bauer, Vadim Miller, Ilene Mizrachi, James Ostell, Anna R. Panchenko, Kim D. Pruitt, Gregory D. Schuler, Edwin Sequeira, Stephen T. Sherry, Martin Shumway, Karl Sirotkin, Douglas J. Slotta, Alexandre Souvorov, Grigory Starchenko, Tatiana A. Tatusova, Lukas Wagner, Yanli Wang, W. John Wilbur, Eugene Yaschenko, and Jian Ye. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, 38(Database-Issue):5–16, 2010. 65

[38] J. C. Setubal and J. Meidanis. *Introduction to computational molecular biology.* PWS Publishing Company, 1997. 3, 12, 53, 74

[39] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147:195–197, 1981. 10

[40] R. Staden. Computer methods to locate signals in nucleic acid sequences. *Nucleic Acids Research*, (12):505–519, 1984. 61

[41] J. Tao and L. Ming. On the approximation of shortest common supersequences and longest common subsequences. In *Automata, Languages and Programming*, volume 820, pages 191–202. Springer Berlin Heidelberg, 1994. 46

[42] V. V. Vazirani. *Approximation algorithms.* Springer-Verlag New York, Inc., 2001. 3, 8

[43] Y. Ye, L. Jaroszewski, W. Li, and A. Godzik. A segment alignment approach to protein comparison. *Bioinformatics*, 19:742–749, 2003. 26, 27

[44] G. W. Yeo and C. B. Burge. Maximum entropy modeling of short sequence motifs with applications to RNA splicing signals. *Journal of computational biology*, 11(2-3):377–394, 2004. 76

[45] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18:821–829, 2008. 10