UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL - UFRGS
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

EDUARDO ISAIA FILHO

Uma Metodologia para Computação com DNA

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profª. Drª. Laira Vieira Toscani
Orientadora

Porto Alegre, setembro de 2004.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Isaia Filho, Eduardo

Uma Metodologia para Computação com DNA / Eduardo Isaia Filho – Porto Alegre: Programa de Pós-Graduação em Computação, 2004.

80 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2004. Orientadora: Laira Vieira Toscani.

1. Computação com DNA. 2. Metodologias de programação com DNA. 3. Método de filtragem. I. Toscani, Laira Vieira. II. Título.

AGRADECIMENTOS

Em primeiro lugar gostaria de agradecer aos meus pais Eduardo e Magali e aos meus irmãos Tatiane e André, que estiveram sempre do meu lado me incentivando e me apoiando durante todas as minhas dificuldades, nunca me deixando desistir.

À minha amiga e companheira Sirlei, pelos estímulos e paciência para a conclusão deste trabalho.

À minha filha Emily, pela compreensão de ter o pai longe durante a realização deste trabalho.

À minha orientadora Laira Vieira Toscani, por me auxiliar em todos os passos deste trabalho, me incentivando e resolvendo pacientemente todas as minhas dúvidas.

Ao pessoal das horas de lazer (escalada), pelos momentos de descontração, em especial ao Duca, Leo, Guili, Naoki e Bruce;

Ao professor Luciano P. Gaspary, pelo incentivo e auxílio para o ingresso ao mestrado.

A todas as pessoas que contribuíram, direta ou indiretamente, para a realização deste trabalho, e a Deus, por dar-me forças para continuar.

SUMÁRIO

LISTA DE ABREVIATURAS

A	Adenosina
C	Citosina
DES	*Data Encryption Standard*
DNA	*Deoxiribonucleic Acid* (Ácido Desoxirribonucléico)
G	Guanina
GB	*Gygabits*
IBM	*International Business Machines*
MIPS	Milhões de instruções por segundo
PCR	*Polymerase Chain Reaction*
RSA	Rivest, Shamir e Adleman
SAT	*Satisfiability*
T	Timina

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A computação com DNA é um campo da Bioinformática que, através da manipulação de seqüências de DNA, busca a solução de problemas. Em 1994, o matemático Leonard Adleman, utilizando operações biológicas e manipulação de seqüências de DNA, solucionou uma instância de um problema intratável pela computação convencional, estabelecendo assim, o início da computação com DNA. Desde então, uma série de problemas combinatoriais vem sendo solucionada através deste modelo de programação.

Este trabalho analisa a computação com DNA, com o objetivo de traçar algumas linhas básicas para quem deseja programar nesse ambiente. Para isso, são apresentadas algumas vantagens e desvantagens da computação com DNA e, também, alguns de seus métodos de programação encontrados na literatura.

Dentre os métodos estudados, o método de filtragem parece ser o mais promissor e, por isso, uma metodologia de programação, através deste método, é estabelecida. Para ilustrar o método de Filtragem Seqüencial, são mostrados alguns exemplos de problemas solucionados a partir deste método.

Palavras-chave: computação com DNA, metodologias de programação com DNA, método de filtragem.

A DNA Computing methodology

ABSTRACT

DNA computing is a field of Bioinformatics that, through the manipulation of DNA sequences, looks for the solution of problems. In 1994 the mathematician Leonard Adleman, using biological operations and DNA sequences manipulation, solved an instance of a problem considered as intractable by the conventional computation, thus establishing the beginning of the DNA computing. Since then, a series of combinatorial problems were solved through this model of programming.

This work studies the DNA computing, aiming to present some basic guide lines for those people interested in this field. Advantages and disadvantages of the DNA computing are contrasted and some methods of programming found in literature are presented.

Amongst the studied methods, the filtering method appears to be the most promising and for this reason it was chosen to establish a programming methodology. To illustrate the sequential filtering method, some examples of problems solved by this method are shown.

Keywords: DNA computing, DNA programming methodologies, filtering method.

1 INTRODUÇÃO

Há, pelo menos, 15 anos [EIS89] a Biologia sentiu a necessidade de utilizar ferramentas computacionais para análise de dados bioquímicos e de biologia molecular. Da mesma forma, os cientistas da computação viram nos experimentos da biologia molecular, uma forma de resolver certos problemas intratáveis pela computação tradicional. Surge então uma nova ciência multidisciplinar, denominada Bioinformática, resultante da união de diversas linhas de conhecimento como: a ciência da computação, a engenharia de softwares, a matemática e a biologia molecular. Seus principais objetivos são: alcançar novas descobertas biológicas a partir da utilização de técnicas matemáticas e desvendar a grande quantidade de dados que vem sendo obtida, a partir de dados de seqüências de DNA e de proteínas.

Desde que foram definidos, na década de 70, existe um grande interesse no meio científico em encontrar uma solução eficiente para uma determinada classe de problemas matemáticos chamados problemas NP-Completo. Sabe-se que técnicas atuais da computação convencional têm sido ineficientes na solução dos mesmos. Em 1994, o matemático Leonard Adleman, utilizando operações biológicas e manipulação de seqüências de DNA, solucionou uma instância do problema do caminho Hamiltoniano, um problema NP-Completo. Adleman assim, estabeleceu o início da computação com DNA, criando o primeiro computador de DNA. Desde então, alguns outros problemas NP-Completos vem sendo solucionados através da manipulação de DNA.

A computação com DNA é um campo da Bioinformática que, através da manipulação de seqüências de DNA, busca a solução de problemas. Uma motivação é os problemas intratáveis pela computação convencional. O fato de ser possível descobrir algoritmos de DNA eficientes, que solucionem problemas NP-Completo, indica que a computação com DNA é quantitativamente superior à computação eletrônica [GIF94].

O grande poder da computação com DNA é que com um pequeno volume de solução (DNA), pode-se ter um vasto número de combinações em um curto período de tempo [FIT2000].

Para desenvolver algoritmos para a programação com DNA, faz-se necessária uma outra maneira de pensar e de construir os algoritmos, uma vez que, a programação com DNA possui um maciço paralelismo, ou seja, a mesma é não-determinística, diferente da programação tradicional determinística.

Ao contrário da computação convencional, que possui metodologias para construção de algoritmos como os métodos *top-down*, *botton-up*, modular, entre outros, a computação com DNA ainda não possui metodologias bem definidas. Tais metodologias facilitariam a construção e a compreensão dos algoritmos descritos para a computação com DNA.

1.1 Motivação

Este trabalho é motivado pela necessidade cada vez maior de se definir uma metodologia clara para a programação com DNA. Uma das barreiras encontradas nesta área é a atual carência de padrões que permitam biólogos e cientistas da computação desenvolverem uma linguagem comum, a qual resulte numa metodologia de programação dos algoritmos desenvolvidos.

1.2 Objetivos

O objetivo inicial deste trabalho foi definir uma metodologia para desenvolvimento de algoritmos para computação com DNA.

Esta metodologia deve auxiliar os cientistas da computação a projetarem seus algoritmos para computação com DNA e facilitar a compreensão dos biólogos ao simularem estes algoritmos através de experimentos em laboratório.

1.3 Estrutura do texto

O texto aborda temas que abrangem: vantagens e desvantagens da programação com DNA, metodologias para desenvolvimento de algoritmos e soluções de problemas NP-Completo, através de algoritmos moleculares.

Após este capítulo introdutório, esta dissertação está organizada em cinco capítulos intitulados: Pesquisas na área, Métodos para computação com DNA, Metodologia, Ilustração da definição do método de Filtragem Seqüencial e Conclusões.

O capítulo 2 apresenta um estudo introdutório na computação com DNA, nele é feita uma breve descrição da estrutura do DNA e como este pode ser manipulado em laboratório. Além disto, são relatadas algumas vantagens e desvantagem desta nova forma de computação em relação à computação convencional. Também descreve-se as atuais áreas de atuação da computação com DNA.

No capítulo 3 são descritos os principais métodos já propostos para a computação com DNA, destacando as suas vantagens e desvantagens; estes métodos são: o método de *splicing*, o método DNA-Pascal, o método de construção e o método de filtragem.

No capítulo 4, o método de filtragem é aprofundado e, através dele, é proposta uma metodologia de programação com DNA, onde suas características são analisadas e justificadas.

No capítulo 5 serão apresentados exemplos de problemas solucionados através do método de Filtragem Seqüencial. Os problemas serão representados utilizando a notação proposta por Garey e Johnson [GAR79].

Por fim, o capítulo 6 apresenta as conclusões deste trabalho, bem como os aspectos que ficaram em aberto como sugestões e perspectivas.

2 PESQUISAS NA ÁREA

A computação com DNA é um novo desafio para a Ciência da Computação. Esta ciência está baseada na manipulação genética de fitas de DNA e através de um conjunto de ações microbiológicas, que codificam os dados estruturados de um problema, em fitas de DNA (seqüências de nucleotídeos) e computando uma solução através da manipulação destas fitas.

Esta técnica surgiu como uma proposta para utilizar o grande poder do paralelismo das reações moleculares na solução de problemas ainda não resolvidos satisfatoriamente pela computação convencional. A idéia não é substituir os computadores atuais por computadores de DNA, e sim, utilizá-los no futuro em casos onde nem os supercomputadores são eficientes.

Utilizar DNA para realizar computação não é o mesmo que utilizar a computação convencional, uma vez que a computação com DNA representa informações através de nucleotídeos (Adenosina, Timina, Guanina, Citosina) e utiliza reações moleculares para processar suas informações. A computação convencional representa informações através de bits (0's e 1's) e utiliza impulsos eletrônicos para processar informações. No processamento convencional os efeitos são bem conhecidos e na computação com DNA existe uma forte componente probabilística.

Um computador de DNA é basicamente um conjunto de fitas de DNA, especialmente selecionadas, e um conjunto de processos biológicos que manipulam estas fitas. Desta manipulação resulta a solução do problema proposto. Na computação convencional uma solução é construída, enquanto que na computação com DNA, devido ao fator probabilístico, muitas soluções precisam ser construídas para o resultado ser confiável.

Na seção seguinte será realizada uma breve descrição da estrutura das seqüências de DNA e como estas seqüências podem ser manipuladas em laboratório. Um estudo mais detalhado da manipulação das moléculas de DNA pode ser obtido em [ALB94].

2.1 Manipulação de seqüências de DNA

O DNA é uma molécula formada pela ligação de outras pequenas moléculas, chamadas nucleotídeos. Existem quatro diferentes tipos de nucleotídeos: adenosina, guanina, timina e citosina, respectivamente denominadas A, G, T e C.

Na computação com DNA trabalha-se com seqüências (pedaços) de uma molécula de DNA e estas seqüências podem ser simples ou duplas. As seqüências simples (FIGURA 2.1) são formadas pela concatenação (síntese) dos nucleotídeos, podendo ser construída qualquer seqüência desejada.

ATGCTAGCT

Figura 2.1: Seqüência simples de DNA

Já as seqüências duplas (FIGURA 2.2) - forma natural da molécula de DNA *in vivo* - podem ser obtidas através de fragmentos de uma molécula de DNA ou pela ligação de fitas simples, porém existem algumas restrições entre estas ligações, que são: A pode ligar-se somente a T, T pode ligar-se somente a A, G pode ligar-se somente a C e C pode ligar-se somente a G, nenhuma outra combinação é possível. Esta restrição é chamada de complementaridade de Watson-Crick.

A T G C T A G C T
T A C G A T C G A

Figura 2.2: Seqüência dupla de DNA

A estrutura da seqüência de DNA pode possuir duas direções que são: 5’ – 3’ ou 3’ – 5’. Essa direção se dá através da ligação de um nucleotídeo do grupo fosfato (5’) com o grupo hidroxila (3’) de outro nucleotídeo. A direção das seqüências é importante para definir a ligação entre duas seqüências simples e, além de satisfazer a restrição de Watson-Crick, as seqüências devem ser de direções opostas (FIGURA 2.3).

5’ - A T G C T A G C T - 3’
3’ - T A C G A T C G A - 5’

Figura 2.3: Direção das seqüências

A manipulação destas seqüências de DNA (simples ou duplas), para computar alguma informação, dá-se através de uma série de operações biológicas as quais são responsáveis por: formar seqüências de nucleotídeos (sintetização), ligar seqüências simples em duplas (hibridização), separar seqüências duplas em simples (desnaturação), adicionar nucleotídeos a seqüências incompletas (polimerização), cortar seqüências (através de enzimas de restrição), multiplicar seqüências (através da reação em cadeia da *Polimerase* ou PCR), separar seqüências (através da eletroforese em gel), entre outras. A seguir estas operações serão comentadas.

- ***Sintetise de seqüências***. Uma fita de DNA pode ser criada em laboratório através de uma máquina, denominada sintetizador de DNA. Esta fita pode ter qualquer seqüência de nucleotídeos desejada.
- ***Hibridização de seqüências***. Em condições de temperatura ideal, fitas simples de DNA complementares irão se ligar, formando uma única fita dupla (FIGURA 2.4).

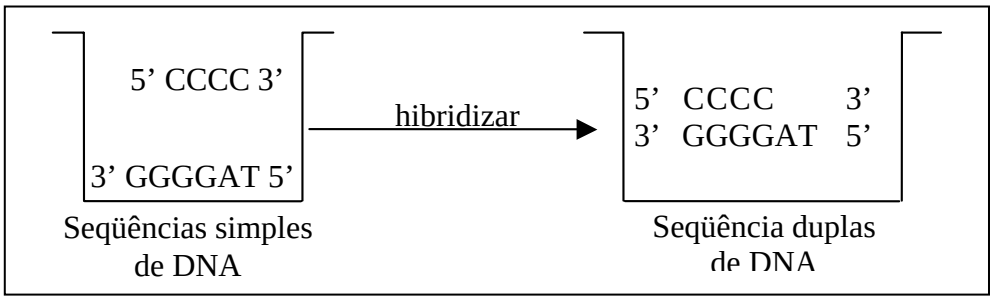


Figura 2.4: Operação de Hibridização

- ***Desnaturação de seqüências***. Moléculas de DNA podem ser separadas em fitas simples, sem que haja a quebra das ligações entre os nucleotídeos de uma mesma fita. Aquecendo a solução, em um intervalo de temperaturas entre 85° Celsius a 95° Celsius, as fitas duplas de DNA se separam formando fitas simples (FIGURA 2.5).

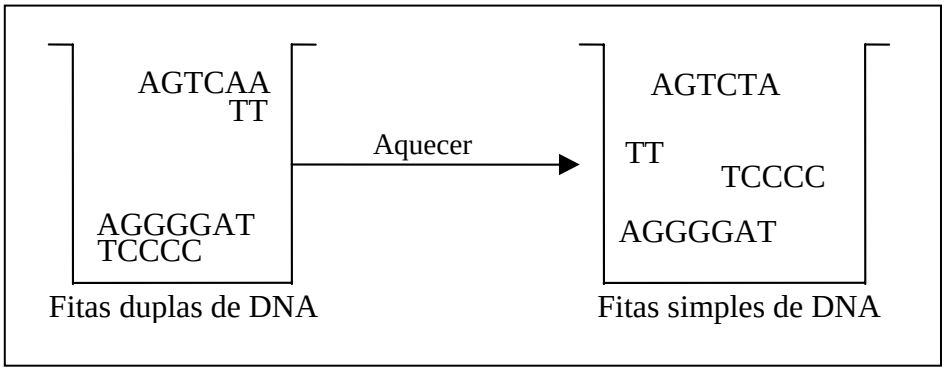


Figura 2.5: Operação de desnaturação

- ***Polimerização em cadeia***. A reação em cadeia da *Polimerase* ou PCR (do inglês *Polimerase chain reaction*) é uma operação para duplicar seqüências de DNA. Esta operação utiliza a enzima *polimerase*, a qual é responsável por adicionar nucleotídeos ausentes em uma seqüência de DNA já existente (FIGURA 2.6).

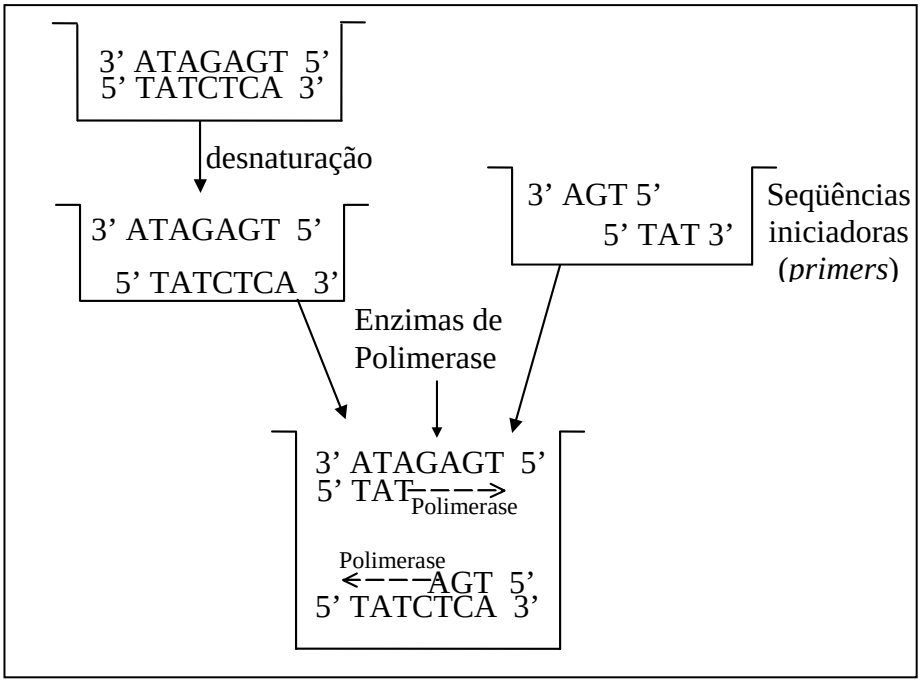


Figura 2.6: Operação de PCR

Na FIGURA 2.6 esta demonstrada uma fita dupla que pode ser duplicada; para isso, a solução deve ser aquecida para que a fita dupla torne-se simples (desnaturação). Após a fita dupla ser separada, são acrescidos à solução oligonucleotídeos[1] (seqüências iniciadoras ou *primers*) complementares das seqüências que se deseja duplicar. Estes oligonucleotídeos irão se ligar às fitas desejadas (hibridização). Adicionam-se então, a enzima *polimerase* para que elas completem os nucleotídeos ausentes, formando assim duas novas fitas duplas.

- ***Restrição das seqüências***. Para restringir (cortar) seqüências de DNA são utilizadas as enzimas de restrição, as quais são responsáveis por localizar seqüências especificas de DNA e corta-la (FIGURA 2.7).

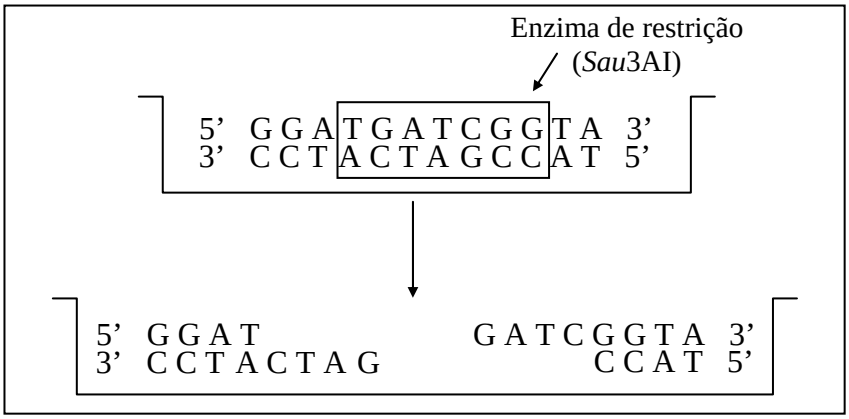


Figura 2.7: Operação de restrição utilizando a enzima *Sau*3AI

[1] Pequena seqüência de DNA formada por 10 a 50 nucleotídeos.

- ***Ligação de seqüências***. Após uma operação de hibridização, alguns nucleotídeos não são ligados ao seu vizinho, isto é, a fita possui uma descontinuidade; para ser realizada esta ligação é utilizada a enzima *ligase*. Esta operação permite obter uma fita unificada com todas as ligações realizadas com seus respectivos complementos (FIGURA 2.8).

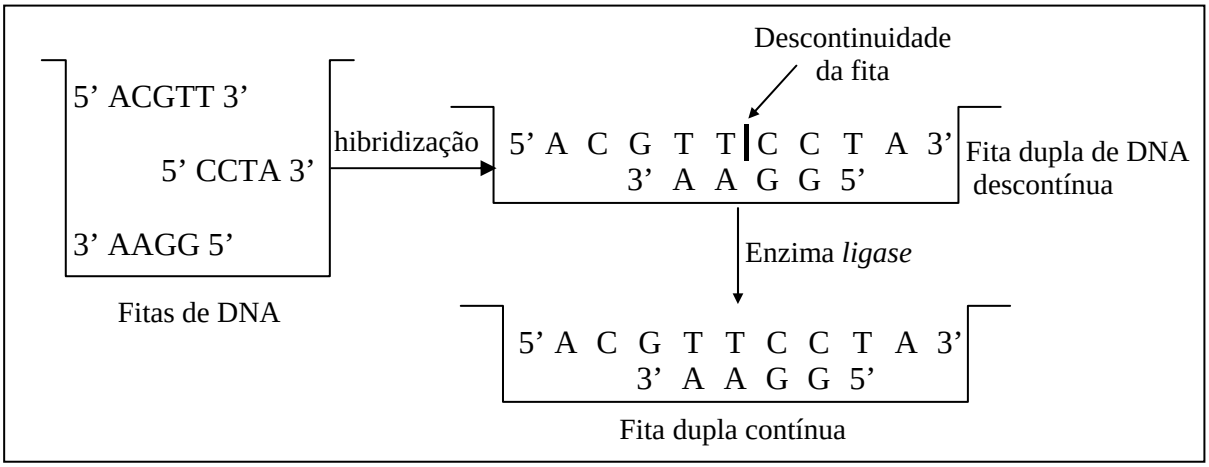


Figura 2.8: Operação de ligação

- ***Separação de seqüências por tamanho***. A separação dos diferentes fragmentos de DNA pode ser realizada através da operação de eletroforese me gel. Eletroforese é o movimento de moléculas ordenadas num campo magnético. O DNA possui uma carga negativa, então quando ele é submetido a um campo magnético, ele tende a migrar em direção ao pólo positivo. A taxa de migração de uma molécula, numa solução aquosa, depende da sua forma e da sua carga elétrica. Como o DNA tem a mesma carga por unidade de tamanho, ele migra a uma mesma velocidade numa solução aquosa. Entretanto, se a eletroforese for realizada em um gel (geralmente agarose e poliacrilamida ou uma combinação dos dois), a taxa de migração de uma molécula também passa a ser afetada pelo seu tamanho. Isto se deve ao fato de o gel ser uma “rede” densa de poros através da qual as moléculas podem passar. Portanto, pequenas moléculas migram mais rápido através do gel, o que torna possível a separação por tamanho (FIGURA 2.9).

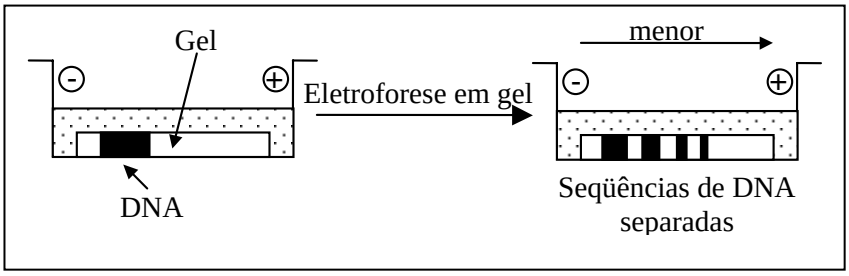


Figura 2.9: Operação de eletroforese em gel

- ***Separação de seqüências por afinidade***. Esta operação é utilizada para separar, em uma solução, seqüências conhecidas. A separação é realizada através da complementaridade de Watson-Crick. Para separar uma seqüência alvo, é criada a sua seqüência complementar acrescida de um elemento magnético. Esta seqüência complementar é adicionada à solução que deve estar desnaturada. Através da hibridização a seqüência complementar irá se ligar a seqüência alvo e então, coloca-se um imã próximo à solução que atrairá a seqüência, separando-a das demais.

A motivação para o estudo da utilização de DNA para realizar computações, vem das vantagens que a mesma apresenta sobre a computação eletrônica. A seguir serão descritas algumas das vantagens da computação com DNA, em relação à computação eletrônica.

2.2 Vantagens da computação com DNA

As principais vantagens da computação com DNA são:

- *Maciço paralelismo.* Uma computação com DNA pode executar 10^{14} MIPS (Milhões de Instruções por Segundo), enquanto que um supercomputador executa somente 10^{9} MIPS [ROO96]. Na computação com DNA as reações químicas acontecem muito rapidamente (se for levada em consideração a quantidade de informação tratada) e em paralelo. As moléculas de DNA são sintetizadas com uma estrutura química, que representa uma informação numérica, com isso uma vasta possibilidade de combinações é possível.
- *Quantidade de informação.* Em uma pequena massa de DNA é possível sintetizar uma grande quantidade de informação. Em um laboratório é possível abrigar 10^{22} moléculas de DNA, cada uma codificando potencialmente 400 bits de informações, o que é mais informação do que pode ser armazenada em um disco rígido de 20 GB.
- *Resolve problemas complexos rapidamente*. Através da computação com DNA foi possível resolver o problema do caminho Hamiltoniano, em questão de semanas. Apesar do problema proposto por Adleman apresentar uma instância pequena - 7 nodos - e poder ser resolvido em milisegundos por um computador convencional, uma instância maior poderia ser resolvida em alguns meses, através da computação com DNA, enquanto que poderia levar alguns milhões de anos para ser resolvida por computadores eletrônicos.

No entanto, existem alguns problemas e obstáculos que ainda limitam a computação com DNA, os quais são descritos a seguir.

2.3 Desvantagens da computação com DNA

As desvantagens da computação com DNA são:

- *Lentidão*. Problemas simples são solucionados de forma rápida, através da computação convencional. As reações biológicas na computação com DNA são executadas de forma rápida, porém suas implementações podem ser muito demoradas, dependendo da complexidade do problema. Por exemplo, localizar uma seqüência específica na solução, pode levar de 15 minutos a 3 horas [AMO97].
- *Ineficiência.* A computação com DNA soluciona problemas combinatoriais de forma eficiente e rápida. Contudo, seu uso torna-se desvantajoso quando deseja-se simular, por exemplo, uma máquina de estados finitos ou mesmo uma máquina de Turing. Não que isso não seja possível, mas o tempo gasto é muito grande, sendo mais eficiente a utilização da computação tradicional [GRA98].
- *Inconfiabilidade*. Computadores convencionais são confiáveis, pois os elétrons não mudam. Porém, mutações aleatórias na estrutura do DNA, podem acontecer naturalmente. Outro problema é que as moléculas de DNA podem sofrer deterioração (hidrólise) [DEA96] durante o processo de execução de uma computação, o que seria identificado somente no final do experimento.
- *Dificuldade de comunicação.* A falta de interação entre biólogos e matemáticos limita a evolução da computação com DNA; por exemplo, para solucionar os problemas combinatoriais fazem-se necessários conhecimentos: matemáticos (definição do problema) e biológicos (operações utilizadas para manipular as seqüências de DNA).
- *Tolerância a erros*. As margens de erro aceitáveis na matemática e na biologia são diferentes, por exemplo, a biologia molecular tolera uma margem de erro de 5%, já para os matemáticos esta margem de erro é grande.
- *Troca de informações entre moléculas*. Na computação com DNA, cada molécula de DNA é vista como sendo um processador independente trabalhando em paralelo. Na computação tradicional os processadores, em paralelo, podem trocar informações através de barramentos, o que não acontece na computação com DNA [DAS98].
- *Codificação padrão*. Ao contrário da programação tradicional que possui suas estruturas de dados bem definidas, a computação com DNA ainda não possui padrões de codificações de dados; por exemplo, a simulação dos números reais através de seqüências de DNA ainda não foi possível.

A tabela 2.1 apresenta, de forma resumida, as vantagens e desvantagens da computação com DNA em relação à computação tradicional.

Tabela 2.1: Um paralelo entre a computação com DNA e a computação convencional.

Computação baseada em DNA	Computação baseada em *Microchip*
A aplicação de operações individuais é lenta.	A aplicação de operações individuais é rápida.
Podem ser aplicadas bilhões de operações simultaneamente.	O número de operações aplicadas simultaneamente é significativamente menor.
Pode prover uma grande capacidade de memória em pequeno espaço.	Capacidade de memória menor.
Para computar alguma informação, faz-se necessário uma considerável preparação dos dados de entrada.	Para computar alguma informação, faz-se necessário apenas um teclado para informar os dados de entrada.
DNA é sensível a deteriorações químicas.	Dados eletrônicos são vulneráveis, mas são facilmente recuperados.
Não existem barramentos entre moléculas que permitam a troca de informações entre moléculas.	A troca de informações entre processadores é relativamente simples.
Não existem metodologias de programação bem estabelecidas.	As metodologias de programação são eficientes e estabelecidas.
Não possui representação das estruturas de dados bem definida.	Estruturas de dados bem definidas.

2.4 Aplicações da computação com DNA

Atualmente a computação com DNA está sendo aplicada nas seguintes áreas:

- *Implementação de memórias associativas*. Uma das propriedades das moléculas de DNA é a possibilidade de codificar uma grande densidade de informação em uma pequena quantidade de DNA - 1 bit por nanômetro cúbico [ADL95]. Outra característica da computação com DNA é que uma grande variedade de operações biológicas pode ser aplicada em todas as moléculas do tubo-teste, simultaneamente. Estas características conferem à computação molecular um grande potencial e, por estas razões, Reif [REI95] teve a idéia de implementar memórias usando moléculas de DNA.

- *Proposição de soluções para problemas combinatoriais*. Seqüências de DNA estão sendo utilizadas para armazenar e manipular informações. Devido ao maciço paralelismo, as seqüências de DNA estão sendo utilizadas para gerar soluções para problemas combinatoriais, onde faz-se necessário a representação e o teste de um grande número de soluções possíveis para uma dada instância do problema.

- *Criptografia*. A computação com DNA despertou interesse nos estudiosos da área de criptografia, após Beaver [BEA94] sugerir um algoritmo molecular para solucionar um problema de fatoração, de um número grande em um tempo polinomial de passos. A fatoração de números grandes é um problema relevante nas aplicações de criptografia, uma vez que através da fatoração de números grandes são criadas as chaves que são a base do sistema de criptografia, como por exemplo no sistema RSA. Em [BON95] é proposto

> um algoritmo para quebrar o DES (*Data Encryption Standard*) que é um procedimento de encriptação de dados da IBM, que é largamente utilizado. O DES utiliza chaves de 56 bits para encriptar mensagens de 64 bits. Segundo o autor, o DES poderia ser quebrado em aproximadamente 4 meses através da utilização de moléculas de DNA.

Como a computação com DNA é uma ciência nova e em crescimento, ainda existem muitas pesquisas a serem realizadas para que esta venha a se desenvolver. Como pode ser visto, uma nova maneira de pensar e de construir algoritmos faz-se necessária para desenvolver o grande potencial desta nova forma de computação. Pesquisar e desenvolver metodologias que sejam compreendidas tanto por matemáticos, cientistas da computação e biólogos para descrever algoritmos será com certeza um grande passo para o desenvolvimento deste novo método de computação.

3 MÉTODOS PARA COMPUTAÇÃO COM DNA

Neste capítulo serão descritos alguns dos métodos propostos para computação com DNA. Estes métodos são baseados na representação formal de DNA, a qual é uma representação matemática baseada em um alfabeto (conjunto de símbolos, neste caso {A, T, G, C}), em regras de composição de palavras e em algumas operações de controle (as operações típicas de DNA, descritas na seção 2.1, são: hibridização, desnaturação, entre outras).

3.1 Método de *Splicing*

Este método foi introduzido por Head [HEA87], mas somente depois de quase 10 anos ele foi utilizado para computação com DNA. É um método abstrato que constrói linguagens a partir de fitas duplas de DNA, sob a aplicação de enzimas de restrição, seguidas por aplicações de enzimas de *ligase*.

O método consiste basicamente em 2 passos: cortar seqüências de DNA em locais específicos e colar os fragmentos destas seqüências, com finais complementares e de mesma direção.

A notação utilizada nesta seção segue as noções básicas da teoria das linguagens formais. Para uma obtenção detalhada dos conceitos da teoria das linguagens formais aqui utilizados, recomenda-se [SAL73] e [MEN97]. A teoria das linguagens formais estuda as linguagens (conjuntos de palavras), as quais são geradas sobre um alfabeto Σ (conjunto finito não vazio de símbolos), por exemplo, {0, 1} ou {A, T, G, C}. As palavras sobre esse alfabeto são formadas pela aplicação da operação de concatenação de símbolos (normalmente representada pelo "$\bullet$", mas que, muitas vezes, é omitida para simplificar a notação). Σ^* é a linguagem de todas as palavras sobre o alfabeto Σ, inclusive a palavra vazia. Por exemplo, TAG $\in \Sigma^*$ para Σ = { A, T, G, C } pois TAG = T$\bullet$A$\bullet$G e T, A, G são símbolos do alfabeto Σ.

O método de *splicing* possui como ponto de partida, um multiconjunto de axiomas (fitas duplas de DNA) e um conjunto de regras de *splicing* (enzimas de restrição). Multiconjuntos são conjuntos que possuem elementos repetidos. Axiomas são cadeias iniciais de { A, T, G, C }. As enzimas são responsáveis por cortar as fitas duplas de DNA em dois pedaços (fragmentos). Estes pedaços, mais tarde, poderão ser recombinados entre si ou poderão ser recombinados com outras fitas duplas cortadas, desde que os pedaços de fitas sejam complementares e de mesma direção. Este processo permite criar novas fitas duplas de DNA. O multiconjunto resultante deste processo é visto como sendo uma linguagem, isto é, um multiconjunto de palavras [DAL98].

A operação de *splicing* resulta na concatenação de um prefixo de uma palavra com um sufixo de outra palavra. Por exemplo, a aplicação da operação de *splicing* nas palavras “video” e “cada” pode gerar a palavra “vida”.

Para compreender como o método de *splicing* pode ser utilizado com fitas duplas de DNA, vê-se o exemplo a seguir: considere as duas seqüências de DNA, representadas na FIGURA 3.1.

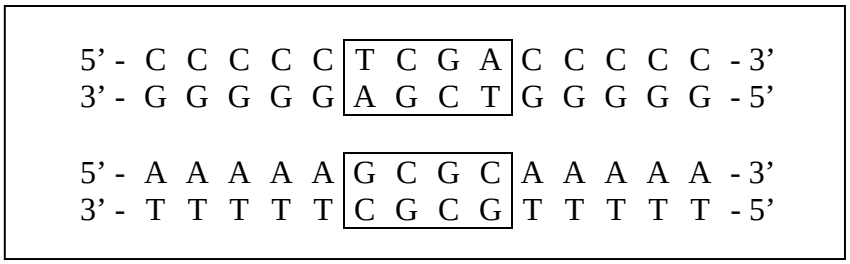


Figura 3.1: Seqüências duplas de DNA

As enzimas de restrição *Taq*I e *Sci*NI são caracterizadas por reconhecerem subseqüências específicas de uma fita dupla de DNA e cortarem estas subseqüências, conforme demonstrado na FIGURA 3.2.

Figura 3.2: Formato do corte das enzimas *Taq*I e *Sci*NI

Assim, estas enzimas cortam as seqüências da FIGURA 3.1, nas posições demonstradas na FIGURA 3.2, resultando quatro fragmentos de seqüência (FIGURA 3.3).

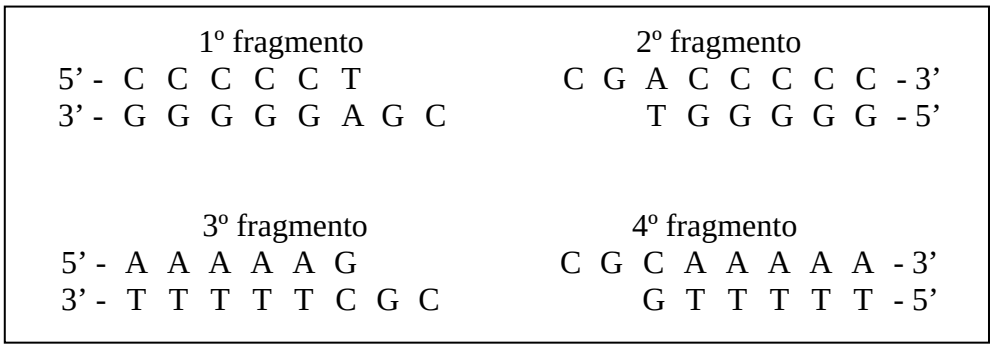


Figura 3.3: Fragmentos resultantes da aplicação das enzimas de restrição

Nota-se que o primeiro e o quarto fragmento são anti-paralelos e complementares e possuem a mesma direção, da mesma forma que o segundo e o terceiro fragmento

também são complementares e de mesma direção. Então, através da recombinação, é possível que as seguintes duas novas seqüências sejam formadas (FIGURA 3.4).

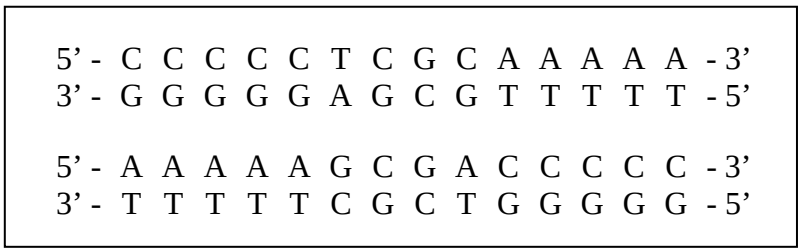


Figura 3.4: Seqüências resultantes da recombinação de fragmentos

Quando são usadas as enzimas de restrição, deve-se observar se as mesmas pertencem à mesma classe. Conforme a classe pertencente, a enzima pode gerar diferentes tipos de corte. Por exemplo, as enzimas *Taq*I e *Sci*NI pertencem à mesma classe e cortam suas respectivas seqüências, preservando a mesma direção dos fragmentos de seqüência, o que permite que estes fragmentos venham a ligar-se entre si. Se, ao invés de usar-se a enzima *Sci*NI, nas seqüências iniciais (FIGURA 3.1), fosse usada a enzima *Hha*I, que reconhece a mesma subseqüência de *Sci*NI, porém pertencente a uma outra classe, o corte gerado na seqüência inicial seria diferente, formando assim, os seguintes novos fragmentos da segunda seqüência da FIGURA 3.1:

3º fragmento	4º fragmento
5’ - A A A A A G C G	C A A A A A - 3’
3’ - T T T T T C	G C G T T T T T - 5’

Note que apesar de ainda existirem nucleotídeos complementares ao final de cada fragmento (1º com 3º e 2º com 4º) a ligação entre eles não é possível devido a direção dos fragmentos não ser a mesma. No decorrer deste texto serão referenciados os fragmentos 3º e 4º gerados a partir da enzima *Sci*NI (FIGURA 3.3).

Para facilitar a representação das palavras sobre o alfabeto { A, T, C, G }, serão consideradas as seqüências anteriores da FIGURA 3.1, como sendo fitas simples de DNA, porém, por convenção, estas fitas simples representam moléculas de DNA (fitas duplas), respeitando a complementaridade de Watson-Crick. Então as seqüências iniciais serão representadas, respectivamente, pelas palavras:

C C C C C T C G A C C C C C

A A A A A G C G C A A A A A

Para definir as enzimas de restrição, não se pode considerar apenas as informações sobre os nucleotídeos reconhecidos pelas mesmas, mas também o tipo de extremidades criadas, quando se corta as seqüências. As enzimas são definidas por uma tripla (u, x, v) de palavras sobre o alfabeto Σ = { A, T, C, G }, onde (u, v) é o local no qual o corte será

feito e x é a seqüência sobressalente. No caso das enzimas *Taq*I e *Sci*NI, as seguintes triplas são definidas:

$$(T , CG , A) \text{ e } (G, CG , C)$$

Formalmente, têm-se duas palavras w_1 e w_2 e um par de triplas $((u_1,x_1,v_1),(u_2,x_2,v_2))$, tal que:

$$w_1 = w_1' \, u_1 \, x_1 \, v_1 \, w_1''$$

w_1'					u_1	x_1		v_1	w_1''				
C	C	C	C	C	T	C	G	A	C	C	C	C	C

$$w_2 = w_2' \, u_2 \, x_2 \, v_2 \, w_2''$$

w_2'					u_2	x_2		v_2	w_2''				
A	A	A	A	A	G	C	G	C	A	A	A	A	A

Para a operação de *splicing* acontecer, é necessário que ambas as triplas sejam da mesma classe e que x_1 seja igual a x_2; satisfeitas estas condições, as palavras resultantes da operação de *splicing* são:

$$z_1 = w_1' \, u_1 \, x \, v_2 \, w_2''$$

$$z_2 = w_2' \, u_2 \, x \, v_1 \, w_1''$$

Em [PAU98], o autor demonstra que, ao invés de utilizarem-se um par de triplas, é possível serem definidas as enzimas com um par de tuplas $((u'_1,v_1),(u'_2,v_2))$, onde $u'_1 = u_1x$ e $u'_2 = u_2x$. Assim, pode-se definir uma regra de *splicing*, a partir de uma quádrupla $(u_1, u_2; u_3, u_4)$.

A quádrupla $(u_1, u_2; u_3, u_4)$ de palavras sobre o alfabeto Σ, determina os locais possíveis de *splicing* (pontos de corte) na primeira e na segunda palavra (primeira ; segunda) onde, u_1 e u_4 são recombinadas como resultado da operação de *splicing*, enquanto o subproduto de u_2 e u_3 não é utilizado, assumindo-se que a operação de *splicing* gera somente um resultado (alguns autores, como [PIS97] e [ZIN99], consideram valida a ligação de u_2 e u_3, sendo então, u_2 e u_3 mais um resultado da operação de *splicing*). No exemplo anterior:

$$u_1 = TCG, \; u_2 = A, \; u_3 = GCG \text{ e } u_4 = C.$$

Apesar de saber-se que uma regra de *splicing* tem como parâmetro quatro palavras separadas, para o tratamento formal é mais fácil definir-se uma regra de *splicing* como uma única palavra. Formalmente, uma regra de *splicing* (sobre um alfabeto Σ) é uma palavra com o seguinte formato:

$$r = u_1 \# u_2 \$ u_3 \# u_4$$

onde # e $ são dois símbolos especiais, não pertencentes a Σ, usados como separadores, e $u_i \in \Sigma^*$ $(1 \leq i \leq 4)$.

Tal regra r, aplicada em duas palavras x e y, resulta em uma palavra z ($x, y, z \in \Sigma^*$) como segue:

$(x,y) \rightarrow_r z$ se e somente se $x = x_1u_1u_2x_2$,
$y = y_1u_3u_4y_2$ e
$z = x_1u_1u_4y_2$,
para qualquer $x_1, x_2, y_1, y_2 \in \Sigma^*$

No exemplo anterior:

$x_1 = x_2 =$ C C C C C e $y_1 = y_2 =$ A A A A A

O método de *splicing* é interessante como modelo abstrato, fundamentado nos conceitos da teoria das linguagens formais, entretanto, está-se interessado no método pela sua proposta inicial: modelar as linguagens de fitas duplas de DNA, sob a influência de enzimas de restrição e de enzimas *ligase*.

Há aspectos nos quais o método de *splicing* apresenta algumas limitações bioquímicas, as quais devem ser consideradas ao implementá-lo na prática. Estas limitações são:

- Em um modelo prático, a quantidade de fitas iniciais e o número de enzimas de restrição diferentes são finitos, sendo assim, o multiconjunto de axiomas e o conjunto de regras de *splicing* do modelo serão finitos. A máquina de Turing (computação universal) possui uma fita infinita. O computador convencional apesar de não possuir uma capacidade de memória infinita pode simular uma máquina de Turing, usando a memória auxiliar, que pode ser expandida facilmente, tornando-se, desta forma, uma computação universal. Mas para determinados modelos de *splicing*, que utilizam um grande número de axiomas e de regras de *splicing*, a massa de DNA necessária será muito grande para representar as informações, tornando-se inviável. Por isso nem sempre o método de *splicing* permite uma simulação da computação com máquina de Turing [DAS96].
- Na prática, as fitas de DNA são consumidas durante o *splicing*: quando a fita w é gerada a partir das fitas x e y, as fitas x e y deixam de existir.
- O comprimento do sítio reconhecido pelas enzimas de restrição é limitado de 6 a 8 bases, por isso as enzimas de restrição não podem reconhecer longas seqüências.

O método de *splicing* não é muito utilizado nos experimentos atuais em computação com DNA e, portanto, sua utilização prática ainda requer maiores estudos [DAS96]. A motivação para este estudo é o fato deste método ser muito bem fundamentado na teoria das linguagens formais.

3.2 Método DNA-Pascal

Em [ROO95] é apresentado um método de programação molecular, denominado DNA-Pascal. O método utiliza tipos de dados particulares e algumas das instruções usuais da linguagem Pascal, adicionando novas operações e testes que correspondem às operações biológicas executadas nas fitas de DNA.

Nesta abordagem, os números naturais são codificados utilizando-se palavras de um alfabeto finito Σ. No contexto da genética e seguindo-se as idéias de Adleman e de Lipton, $\Sigma_{DNA} = \{A, C, G, T\}$ deve ser o alfabeto usado, porém, para melhor compreensão, o autor [ROO95] utiliza o alfabeto $\Sigma = \{0,1\}$. Entretanto, será visto no decorrer desta seção que os resultados não dependem da escolha do alfabeto.

O DNA-Pascal é uma Pascal reduzido, que trabalha somente com variáveis e *arrays* (vetores) de inteiros não negativos. Neste método é definido um novo tipo de variável, denominado multiconjunto finito de palavras. Para manipular estes multiconjuntos, são acrescentados a um conjunto reduzido de operações do Pascal, como por exemplo, os comandos *if*, *for*, *while*, atribuição, entre outros, algumas novas operações especiais e testes. Na tabela 3.1, as novas operações são definidas, onde *m* é uma variável que representa um número natural, *x* é uma palavra, *a* é uma subpalavra e T, T_1 e T_2 são multiconjuntos finitos de palavras. Na tabela 3.2 são definidas as operações de teste.

Tabela 3.1: Conjunto de operações especiais em DNA-Pascal.

Abreviação	Nome	Instrução
(UN)	***Union***	$T := T_1 \cup T_2$
(IN)	***Initialization***	$T := IN(m)$ $IN(m) = \{0,1\}^m$ com $m \geq 0$
(AS)	***Assigment***	$T := T_1$
(EX)	***Extraction***	$T := EX(T_1,m,a)$ $EX(T_1,m,a) = T_1 \cap (\{0,1\}^{m-1} a\{0,1\}^*)$, com $a \in \{0,1\}$ com $m \geq 1$
(SX)	***Subword Extraction***	$T := Sx(T_1,x)$ $SX(T_1,x) = T_1 \cap (\{0,1\}^* x \{0,1\}^*)$ com $x \in \{0,1\}^*$
(EW)	***Empty Word***	$T := \{\varepsilon\}$
(RA)	***Right Adding***	$T := T_1 . a$ $T_1 . a = \{za \mid z \in T_1\}$
(LA)	***Left Adding***	$T := a . T_1$ $a . T_1 = \{az \mid z \in T_1\}$
(CO)	***Concatenation***	$T := T_1 . T_2$

		$T_1 . T_2 = \{xy \mid x \in T_1, y \in T_2\}$
(RC)	***Right Cut***	$T := T_1/$ $T_1/ = \{z/ \mid z \in T_1\}$ com $za/ = z$ para $a \in \{0,1\}$ e $\varepsilon/ = \varepsilon$
(LC)	***Left Cut***	$T := \backslash T_1$ $\backslash T_1 = \{\backslash z \mid z \in T_1\}$ com $\backslash az = z$ para $a \in \{0,1\}$ e $\backslash \varepsilon = \varepsilon$
(IS)	***Intersection***	$T := T_1 \cap T_2$

Tabela 3.2: Conjunto de operações de teste em DNA-Pascal.

Abreviação	Nome	Instrução	Instrução é interpretada como "verdade" se:
(SU)	***Subset***	$T_1 \subseteq T_2$	O multiconjunto T_1 contêm as palavras de T_2.
(EM)	***Emptiness***	$T = \varnothing$	O multiconjunto T for vazio.
(ME)	***Membership***	$x \in T$	A palavra *x* pertencer ao multiconjunto T.

A entrada de um programa, em DNA-Pascal, é uma palavra ou um multiconjunto de palavras de {0, 1}*, que são dadas com o valor inicial de uma variável palavra (ou várias variáveis palavras). As demais variáveis usadas no programa são inicializadas vazias, isto é, variáveis palavras recebem o valor ε e as demais variáveis (multiconjuntos) recebem valor ∅. A computação termina por uma instrução "aceita" ou "rejeita".

Para ilustrar o uso das operações de DNA-Pascal, considera-se o seguinte exemplo. Dado Σ = {A, C, G, T}, tem-se o seguinte programa:

```
begin
  T0 := IN(n)                    {inicializar T0 com todas palavras de tamanho n}
  T1 := SX(T0, C);  {extrai de T0 todas as palavras que possuem C como subpalavra}
  T2 := SX(T0, G);  {extrai de T0 todas as palavras que possuem G como subpalavra}
  T0 := T1 ∪ T2;                 {une em T0, T1 e T2, criando um único conjunto}
  if T0 = ∅ then rejeita else aceita
end
```

Este é um programa simples, que retorna "aceita" se o T_0 contém pelo menos uma palavra que contenha um ou mais A's e/ou T's e, caso contrário, retorna "rejeita". Em [ROO95] encontra-se a definição de um programa em DNA-Pascal para solucionar o problema *Satisfability*, um conhecido problema NP-Completo.

A utilidade do DNA-Pascal é que esta metodologia permite provar os resultados sobre a capacidade computacional, dos diferentes modelos de computação de DNA, que podem ser expressos utilizando-se diferentes conjuntos de operações e testes. Por

exemplo, em [ROO95], o autor prova que o modelo de Lipton resolve – em um tempo polinomial – exatamente os problemas pertencentes à classe de complexidade NP. Neste modelo, Lipton propõe cinco operações: *extract*, *delect*, *amplify*, *union* e *initialization*. Utilizando estas operações, Lipton apresenta uma solução para o problema NP-Completo 3-SAT, em um tempo polinomial. Este modelo está bem definido em [LIP94].

Já em [ROO96], caracteriza-se a capacidade computacional do conjunto de operações, definindo-se algumas propriedades cruciais alcançadas, se algumas operações (e/ou testes) são usados. **N**a tabela 3.3 são apresentadas as propriedades que algumas operações possuem. Estas propriedades são:

- **Propriedade de bloco**. A operação preserva a propriedade dos multiconjuntos conterem, para todo $i \in$ IN, todas as palavras de tamanho i ou nenhuma palavra de tamanho i.
- **Propriedade de seleção**. A operação pode excluir palavras de um determinado multiconjunto, sendo as palavras restantes modificadas ou não.
- **Propriedade de identificação**. A operação pode identificar diferentes palavras. Em particular, as operações que podem substituir ou cortar símbolos ou subpalavras que têm essa propriedade.

Tabela 3.3: Propriedades de algumas operações.

Operações / **Propriedades**	UM IN CO	LC RC	LA RA	EX
Bloco	✓	✓	✗	✗
Seleção	✗	✗	✗	✓
Identificação	✗	✓	✗	✗

Pode-se dizer que DNA-Pascal é uma meta-linguagem utilizada para estudar a computabilidade de modelos da computação com DNA.

3.3 Método de Construção

A proposta deste método é isolar a solução do problema de um grande conjunto inicial, ou seja, os algoritmos gradualmente construirão as soluções para o problema dado. Este método foi usado por Ogihara [OGI96], Baum [BAU96] e Guarnieri [GAU96].

Ogihara e Ray [OGI96] utilizaram o método de construção para implementar uma simulação de circuitos booleanos que usam portas *and* e *or*. Neste modelo os circuitos lógicos são divididos em níveis, como na FIGURA 3.5, os quais são computados em paralelo, onde a saída de um nível inferior será a entrada de um nível superior. Ao final da computação o circuito lógico estará resolvido.

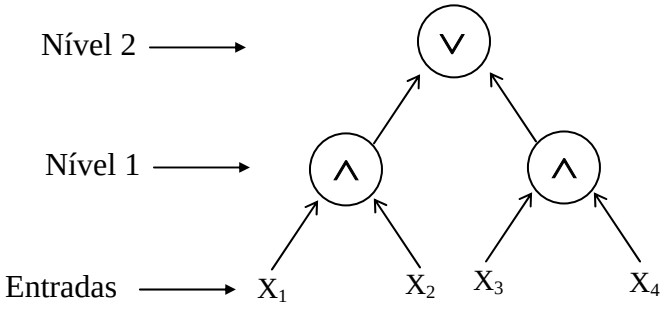


Figura 3.5: Circuito lógico de 2 níveis

Baum e Boneh [BAU96] propuseram, através do método de construção, uma nova forma de obter a solução de um problema. Ao invés de usar a "força bruta", como por exemplo no algoritmo de Adleman [ADL94], onde a solução do problema deve estar presente no conjunto de entrada do algoritmo, Baum e Boneh utilizaram as técnicas da programação dinâmica para desenvolver seus algoritmos moleculares. A idéia básica da programação dinâmica consiste em dividir o problema inicial em subproblemas menores e mais simples de serem solucionados. Assim, os subproblemas são resolvidos separadamente obtendo-se soluções parciais. A cada passo, as soluções parciais são combinadas, dando origem a soluções para subproblemas maiores, até que se obtenha uma solução para o problema original. Este método é apropriado para a computação com DNA, pois os subproblemas podem ser resolvidos paralelamente. Os autores, através da programação dinâmica, desenvolveram um algoritmo molecular que resolve determinadas instâncias do problema da mochila. O algoritmo pode ser encontrado em [BAU96].

Guarnieri, Fliss e Bancroft [GAU96] utilizaram o método de construção para definir uma forma de somar dois números binários não negativos de dois dígitos. Na adição binária, a soma de dois números pode gerar, além do bit de saída, um bit carregador, por exemplo, 1 + 1 = 10. A soma binária dos dígitos 1 e 1 resulta em 0 e é incluído à próxima posição de bit o dígito 1 (bit carregador). Este é adicionado aos dígitos da posição, no caso do exemplo anterior, a próxima posição de bit possui os dígitos 0's, 1 + 0 = 1 (FIGURA 3.6).

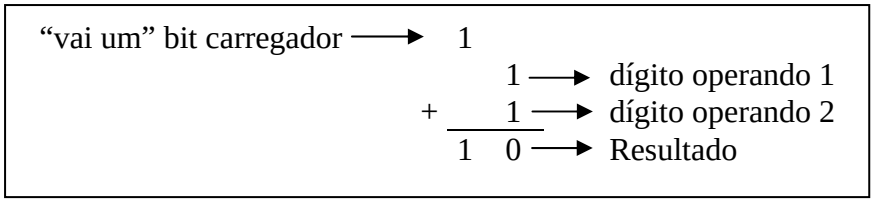


Figura 3.6: Soma binária

Os mesmos autores propuseram uma forma de simular esta adição, utilizando seqüências de DNA. A solução da soma de dois números binários de dois dígitos é construída a partir de uma codificação inicial padrão, onde cada dígito é representado pela concatenação de algumas das subseqüências ($0_{(0)}$, $0_{(1)}$, $1_{(0)}$, $1_{(1)}$, $1_{(2)}$, $X_{(0)}$, $X_{(1,0)}$, $X_{(1)}$, $X_{(2,1)}$, $Y_{(0)}$, $Y_{(1,0)}$, $Y_{(1)}$, $Y_{(2,1)}$, $C_{(1)}$ e seus complementos, por exemplo $\overline{X}_{(0)}$ e a

subseqüência complementar de $X_{(0)}$), formando uma fita simples única e não complementar (FIGURA 3.7).

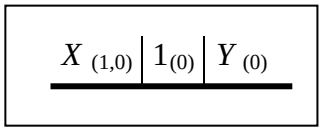


Figura 3.7: Seqüência representativa de um dígito

Na FIGURA 3.7, as subseqüências X e Y representam o operador de posição e a subseqüência 1 (podendo também ser a 0) indica o valor do dígito. Os números entre parênteses, quando únicos, representam a posição do dígito e quando em par, a transição da posição, por exemplo, (1,0) representa a transição da posição 0 para a posição 1, neste caso, o operador é denominado operador de transição da posição. A subseqüência $C_{(1)}$ representa o bit carregador.

Cada dígito tem uma codificação específica dependendo da posição que este se encontra. As codificações serão as seguintes:

Para os dígitos na posição 0, as seqüências têm o seguinte formato:

- **Operando 1**: Cada dígito é codificado por duas fitas distintas tendo a direção 5'- 3'. Cada fita é formada pelo operador de transição da posição ($\overline{X}_{(1,0)}$ ou $\overline{Y}_{(1,0)}$), seguido do elemento representativo do valor do dígito na posição 0 ($\bar{0}_{(0)}$ ou $\bar{1}_{(0)}$) e o operador de posição ($\overline{X}_{(0)}$ ou $\overline{Y}_{(0)}$). A codificação dos números 0 e 1 é demonstrada a seguir:

$$0 = \left[\begin{array}{l} 5' \ \overline{X}_{(1,0)} \mid \bar{0}_{(0)} \mid \overline{X}_{(0)} \ 3' \\ 5' \ \overline{X}_{(1,0)} \mid \bar{1}_{(0)} \mid \overline{Y}_{(0)} \ 3' \end{array}\right] \qquad 1 = \left[\begin{array}{l} 5' \ \overline{X}_{(1,0)} \mid \bar{1}_{(0)} \mid \overline{X}_{(0)} \ 3' \\ 5' \ \overline{Y}_{(1,0)} \mid \bar{0}_{(0)} \mid \overline{Y}_{(0)} \ 3' \end{array}\right]$$

- **Operando 2**: Cada dígito é codificado por uma fita específica de direção 3' – 5', podendo ser $X_{(0)}$ ou $Y_{(0)}$ se o dígito for 0 ou 1, respectivamente. Esta fita servirá como um *primer*, que desencadeará a reação para efetuar a soma. A codificação dos números 0 e 1 é demonstrada a seguir:

$$0 = \left[3' \ \underline{\quad X_{(0)} \quad} \ 5' \right] \qquad 1 = \left[3' \ \underline{\quad Y_{(0)} \quad} \ 5' \right]$$

Para os dígitos na posição 1, as seqüências têm o seguinte formato:

- **Operando 1**: Cada dígito é representado por três fitas distintas, tendo a direção 5'- 3'. Cada fita é formada pelo operador de transição da posição ($\overline{X}_{(2,1)}$ ou $\overline{Y}_{(2,1)}$), seguido do elemento representativo do valor do dígito na posição 0 ($\overline{0}_{(1)}$ ou $\overline{1}_{(1)}$) e o operador de posição ($\overline{X}_{(1)}$ ou $\overline{Y}_{(1)}$) ou o carregador ($\overline{C}_{(1)}$). A codificação dos números 0 e 1 é demonstrada a seguir:

$$0 = \left[\begin{array}{l} 5' \ \overline{X}_{(2,1)} \mid \overline{0}_{(1)} \mid \overline{X}_{(1)} \ 3' \\ 5' \ \overline{X}_{(2,1)} \mid \overline{1}_{(1)} \mid \overline{Y}_{(1)} \ 3' \\ 5' \ \overline{Y}_{(2,1)} \mid \overline{0}_{(1)} \mid \overline{C}_{(1)} \ 3' \end{array}\right] \qquad 1 = \left[\begin{array}{l} 5' \ \overline{X}_{(2,1)} \mid \overline{1}_{(1)} \mid \overline{X}_{(1)} \ 3' \\ 5' \ \overline{Y}_{(2,1)} \mid \overline{0}_{(1)} \mid \overline{Y}_{(1)} \ 3' \\ 5' \ \overline{Y}_{(2,1)} \mid \overline{1}_{(1)} \mid \overline{C}_{(1)} \ 3' \end{array}\right]$$

- **Operando 2**: Cada dígito é codificado por duas fitas distintas de direção 5' – 3' e cada uma contém o operador de posição ($\overline{X}_{(1)}$ ou $\overline{Y}_{(1)}$) ou o carregador ($\overline{C}_{(1)}$), seguido do operador de transição da posição ($\overline{X}_{(1,0)}$ ou $\overline{Y}_{(1,0)}$). A codificação dos números 0 e 1 é demonstrada a seguir:

$$0 = \left[\begin{array}{l} 5' \ \overline{X}_{(1)} \mid \overline{X}_{(1,0)} \ 3' \\ 5' \ \overline{Y}_{(1)} \mid \overline{Y}_{(1,0)} \ 3' \end{array}\right] \qquad 1 = \left[\begin{array}{l} 5' \ \overline{Y}_{(1)} \mid \overline{X}_{(1,0)} \ 3' \\ 5' \ \overline{C}_{(1)} \mid \overline{Y}_{(1,0)} \ 3' \end{array}\right]$$

Além de codificar as seqüências de cada dígito, também é criada uma seqüência para representar o bit carregador-final, que será ligado à seqüência resultado, caso a soma dos dígitos da posição 1 resultem um bit carregador. Esta seqüência será codificada da seguinte forma:

$$\text{Carregador-final} = \left[5' \ \overline{1}_{(2)} \mid \overline{Y}_{(2,1)} \ 3' \right]$$

Após todos os dígitos serem codificados em suas respectivas posições, a entrada da computação de uma adição é a seguinte: todas as seqüências que representam o operando 1 e o operando 2, das posições 0 e 1, e a seqüência do carregador-final. A seguir, será exemplificada a soma entre dois números binários: 11 e 01.

Passo 1: *Efetuar a soma entre os dígitos da posição 0*. As seqüências que representam o dígito 1 (operando 1) e o dígito 1 (operando 2); através da hibridização,

as seqüências complementares irão se ligar. A seguir, é aplicada na solução, a enzima *polimerase* para completar o restante da fita (FIGURA 3.8). A solução é então aquecida, para que a fita dupla torne-se novamente duas fitas simples (desnaturação), onde uma delas será utilizada para continuar a computação.

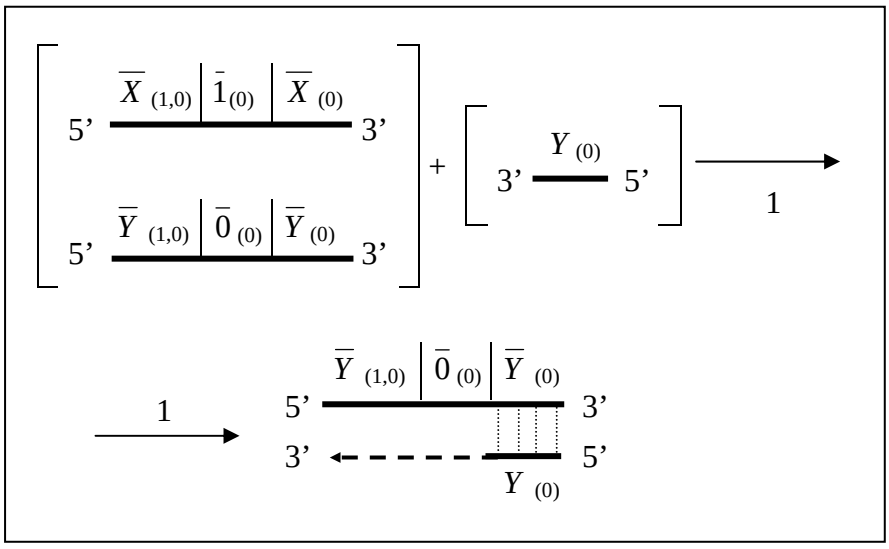


Figura 3.8: Soma entre os dígitos da posição 0

Passo 2: *Somar o dígito da posição 1, do segundo operando, com a seqüência resultante do passo 1*. Para isso, são acrescidas na solução, as seqüências que representam o dígito 1 (operando 2). Através da hibridização, as seqüências complementares vão se ligar, formando uma nova fita dupla. Novamente é aplicada na solução, a enzima *polimerase* para completar a fita (FIGURA 3.9). Novamente a solução é aquecida para que a fita dupla torne-se duas fitas simples, onde uma delas dará continuidade à computação.

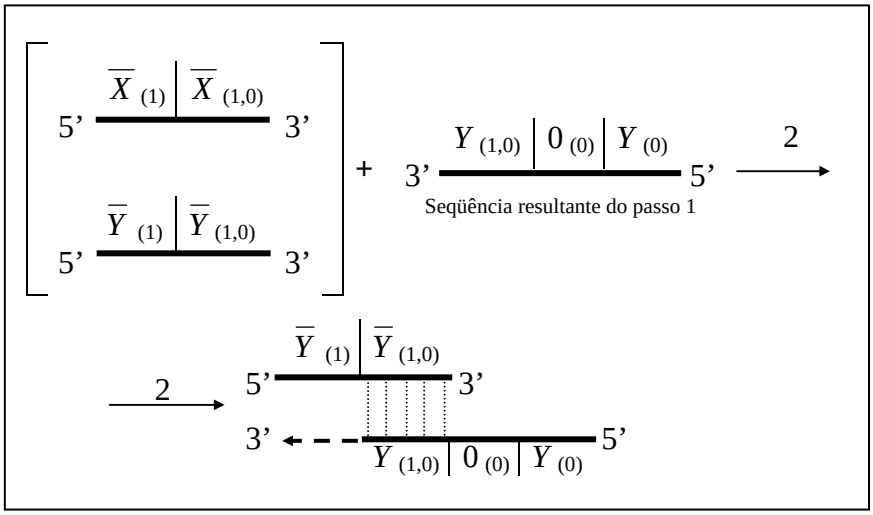


Figura 3.9: Soma do operando 2, da posição 1, com o resultado do passo 1

Passo 3: *Somar o dígito da posição 1, do primeiro operando, com a seqüência resultante do passo 2*. Acrescenta-se na solução, as seqüências que representam o dígito

1 (operando 1). As operações de hibridização, polimerização e desnaturação irão se repetir neste passo (FIGURA 3.10).

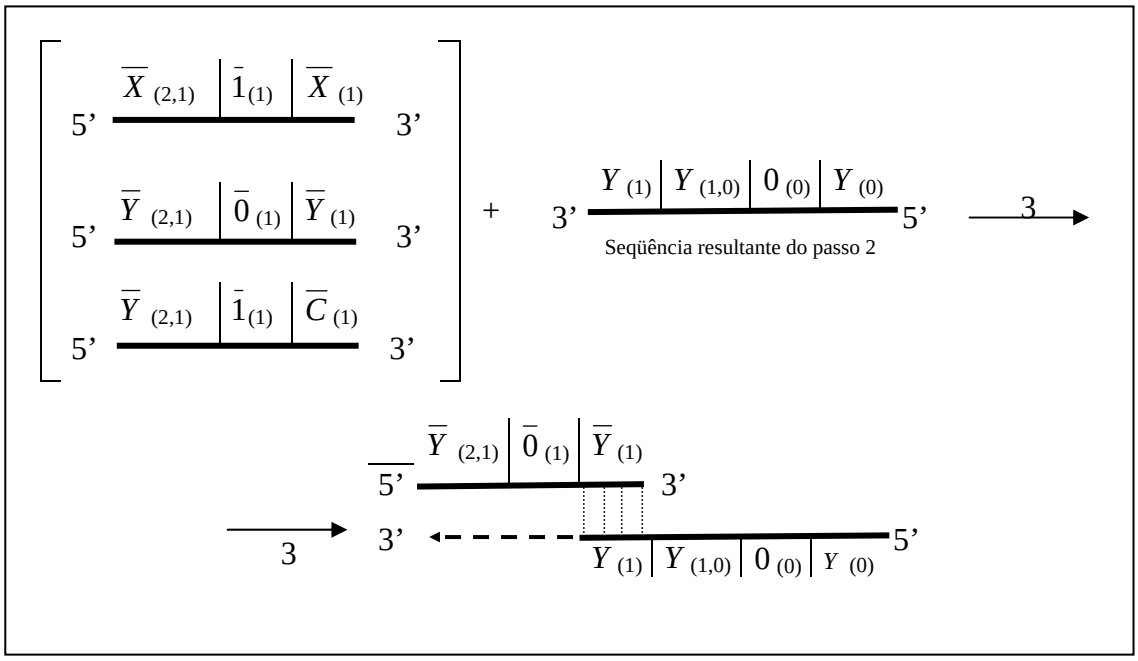


Figura 3.10: Soma do operando 1, da posição 1, com o resultado do passo 2

Passo 4: *Adicionar o carregador-final*. Ao final, é colocada na solução a seqüência que representa o carregador-final (FIGURA 3.11). Esta seqüência, que representa o carregador-final, estenderá a seqüência resultante do passo 3, somente sobre condições apropriadas, ou seja, se for necessária a adição de um carregador-final. Neste passo as operações de hibridização, polimerização e desnaturação irão se repetir como nos passos anteriores.

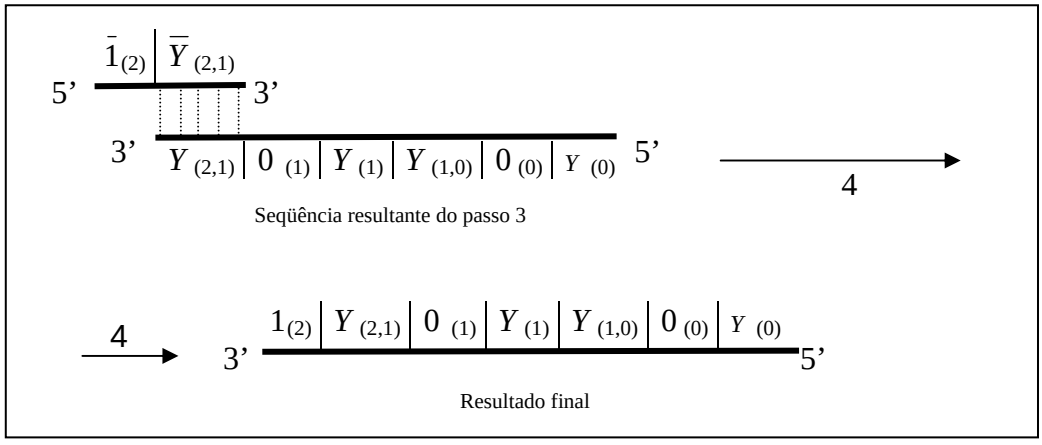


Figura 3.11: Adicionando o carregador-final

A seqüência resultante do passo 4 é o resultado final, da soma entre os números binários. Esta é separada através da operação de eletroforese em gel de agarose e pode

ser interpretada, pois contém todas as informações que codificam o número binário. No exemplo anterior o resultado da soma é 100.

Porém, como os autores (Guarnieri, Fliss e Bancroft [GAU96]) previram, uma limitação desta abordagem é que a fita de saída da operação de adição, pode não servir como fita de entrada para um novo nível de adição, sendo possível somente adicionar dois números de dois dígitos, pois a seqüência de DNA, gerada como saída da adição, terá uma codificação (formato) diferente das seqüências de entrada do método.

A desvantagem do método de construção é que a complexidade das operações cresce rapidamente, conforme o tamanho dos operandos aumenta. Isto limitará o tamanho da entrada e, conseqüentemente, o tamanho do problema que se pode resolver.

3.4 Método de Filtragem

O método de filtragem consiste de uma seqüência de operações, em um multiconjunto finito de palavras. Isto é, o método tem como entrada um multiconjunto e, através de operações de filtragem, chega à solução do problema proposto, se é que ela existe. Este método pode ser dividido em duas partes: Uma pré-computação, a qual é a construção do multiconjunto e a filtragem propriamente dita para a obtenção da solução.

Usaram o método de filtragem, entre outros, Adleman [ADL95], Lipton [LIP95], Amos [AMO97], Beigel [BEI97] e Liu [LIU96]. Cada um deles usou o método de forma ligeiramente diferente e propôs uma série de operações de filtragem, conforme a necessidade do seu problema.

A explicação biológica de como as operações propostas por cada autor são implementadas, assim como a viabilidade das mesmas, serão comentadas no capítulo 4. Já os algoritmos exemplificando o método serão comentados no capítulo 5.

3.4.1 Solução de Adleman e Lipton

Inicialmente, Adleman propôs algumas operações de filtragem e, posteriormente, Lipton utilizou-as para executar seus experimentos. As operações, propostas por Adleman, são executadas sobre um conjunto inicial de palavras sobre um alfabeto. As operações utilizadas são:

- *separate*(T,S). Dado um multiconjunto T e uma subpalavra S, criam-se 2 novos conjuntos: +(T,S) e –(T,S), onde +(T,S) são todas as palavras de T que contém S e -(T,S) são todas palavras de T que não contém S.
- *merge*(T_1, T_2, …, T_n). Dado os multiconjuntos T_1, T_2, …, T_n, cria-se U(T_1, T_2, …, T_n) = $T_1 U\ T_2 U\ \ldots U T_n$.
- *detect*(T). Dado um multiconjunto T, retorna-se “verdadeiro” se T é não vazio, senão retorna-se “falso”.

Por exemplo: Dado Σ= {A, B, C}, considera-se o seguinte algoritmo:

(1) Input (T) {entrada do multiconjunto}
(2) T ← -(T,B) {retira de T todas as palavras que contém B}
(3) T ← -(T,C) {retira de T todas as palavras que contém C}
(4) Output (*detect*(T)) {testa se T está vazio}

Este algoritmo somente retorna "verdadeiro" se o conjunto inicial contiver, pelo menos, uma palavra contendo 'A', caso contrário retorna "falso".

Adleman utilizou suas operações para solucionar uma instância do problema do caminho Hamiltoniano e uma instância do problema 3-vértice-colorível, ambos sendo problemas NP-Completo. Lipton as utilizou, propondo a solução de outro problema NP-Completo, o SAT (*Satisfiability*). Os algoritmos propostos serão apresentados no capítulo 5.

3.4.2 Solução de Amos

Amos [AMO96] apresentou a descrição de um método de filtragem paralelo. Este método foi o primeiro a prover um padrão formal para a descrição dos algoritmos de DNA para qualquer problema na classe NP.

Neste método, todas as computações começam com a construção do multiconjunto inicial de palavras T. E consistirá em retirar do conjunto de soluções possíveis um subconjunto com um certo padrão determinado.

A principal diferença do método de filtragem paralelo está na proposta da remoção das palavras. Todos os outros métodos propõem passos de separação, onde as palavras são conservadas e podem ser utilizadas posteriormente na computação. No método de filtragem paralelo, no entanto, as palavras que são removidas, são descartadas e não participam de computações futuras. O conjunto básico de operações é:

- *remove*(T,{S_i}). A operação *remove* é executada a partir da composição das duas operações:
    - *mark*(T, {S_i}). Esta operação marca todas palavras do conjunto T, que contêm pelo menos uma ocorrência de uma subpalavra definida em {S_i}. Nota-se que {S_i} é um conjunto e S_i é o seu elemento genérico, podendo ser uma subpalavra ou uma condição que define uma subpalavra.
    - *destroy*(T). Esta operação remove todas as palavras marcadas em T.
- *union*({T_i},T). Esta operação cria o conjunto T, que é o conjunto união dos conjuntos T_i's de {T_i}.
- *copy*(T,{T_i}). Esta operação produz T_i's cópias de T.
- *select*(T). Esta operação seleciona um elemento aleatório de T, se o conjunto T é vazio então é retornado *empty*.

Amos, utilizando o seu método de filtragem, definiu a solução para o problema das permutações e para o problema 3-vértice-colorível.

3.4.3 Solução de Beigel e Fu

O método de filtragem foi usado por Beigel e Fu, e eles propuseram novas operações de filtragem, diferentes das de Adleman e Amos. Como nos exemplos anteriores, a computação tem início através da construção do conjunto inicial, contendo a palavra $x = x_1 \ldots x_n$, que é binária n-bit. A palavra x é representada pela seguinte seqüência de DNA: $B_1(x_1)B_2(x_2) \ldots B_n(x_n)$, onde $B_i(x_j)$ é o código que representa o valor x_j ($x_j = 0$ ou 1) para o $i^{ésimo}$ bit.

Beigel e Fu propõem o seguinte conjunto básico de operações:

- *sep*(T_1, c, T_2, T_3). Dados um multiconjunto T_1 e um caracter c, são gerados dois novos multiconjuntos: T_2, contendo todas as palavras de T_1 que contém c, em qualquer lugar da palavra, e T_3, contendo todas palavras de T_1 que não contém c. No final desta operação, o multiconjunto T_1 fica vazio.
- *separate*(T_1, c, i, T_2, T_3). Dados um multiconjunto T_1, um caracter c e um número i, são gerados dois novos multiconjuntos: T_2, contendo todas as palavras de T_1, cujo $i^{ésimo}$ caracter é c e T_3, contendo todas palavras de T_1, cujo $i^{ésimo}$ caracter não é c. No final desta operação, o multiconjunto T_1 fica vazio.
- *append*(T, c). Dado um multiconjunto T e um caracter c, resultará a concatenação de c ao final de cada palavra de T.
- *merge*(T_1, T_2, T_3). Dados os multiconjuntos T_1 e T_2, produz-se a união dos multiconjuntos T_1 e T_2 em T_3. No final desta operação, os multiconjuntos T_1 e T_2 ficam vazios.
- *amplify*(T_1, T_2, T_3). Dado um multiconjunto T_1, produz-se T_2 e T_3 idênticos a T_1. No final desta operação, o multiconjunto T_1 fica vazio.
- *test*(T). Dado um multiconjunto T, retorna-se “verdadeiro” se T é não vazio, senão retorna-se “falso”.

Além deste conjunto de operações básicas citadas em [BEI97], [BEI98a], [BEI98b], [FU98a], [FU98b], outras operações foram apresentadas pelos autores, porém elas se aplicam a casos muito específicos. Estas operações são: *split* [BEI98a], *rightcut* [BEI98a], *pour* [BEI98a], *cut* [FU98a], *polymerase* [FU98b] e *sep12* [FU98b].

Beigel e Fu resolveram, através de suas operações, uma instância do problema SAT (*Satisfiability*), apresentado no capítulo 5.

3.4.4 Solução de Liu

Em [LIU96], Liu define um método denominado “*surface-based*”, a qual nada mais é do que um método de filtragem. Este método também possui um conjunto inicial, constituído de todas as soluções possíveis para o problema, o qual é submetido a

determinadas operações, esperando-se obter o conjunto de soluções viáveis para o problema.

Segundo Liu, suas operações são mais eficazes que as demais, pois reduzem significativamente a perda de moléculas de DNA durante a execução dos passos de seus algoritmos, uma vez que, suas operações são de simples implementação biológica. O conjunto básico de operações é:

- *mark*(i, b). Esta operação marca todas as palavras de T, em que o $i^{ésimo}$ bit possui o valor b.
- *mark*((i_1, b_1), (i_2, b_2), . . . , (i_k, b_k)). Esta operação é uma extensão de *mark*(i, b) e nela, uma palavra é marcada, baseada no valor de muitos bits.
- *destroy-marked(*T). Esta operação elimina todas as palavras marcadas, na operação de *mark*, do conjunto T.
- *destroy-unmarked*(T). Esta operação elimina todas as palavras não marcadas, na operação de *mark*, do conjunto T.
- *unmark*(T). Esta operação desmarca todas as palavras marcadas do conjunto T.
- *test-if-empty*(T). Esta operação determina se o conjunto T é vazio ou não. Caso seja vazio, o resultado da operação é “falso”, caso contrário, é “verdadeiro”.

Liu demonstrou a eficácia de suas operações apresentando duas soluções para o problema SAT (*Satisfiability*), demonstradas no capítulo 5.

4 METODOLOGIA

Os métodos mencionados no capítulo 3, foram estudos com o objetivo de buscar aquele que apresentasse as melhores condições de programação com DNA. Considerando as vantagens e desvantagens de cada método, foi escolhido o método de filtragem, para aprofundar os estudos e ser o foco deste trabalho, já que este é um método natural de programação com DNA, e faz uso de operações simples de serem implementadas em laboratório. Além disto, o método de filtragem apresenta-se como o melhor método para resolução de problemas combinatoriais[2] da classe NP, pois faz uso do maciço paralelismo que caracteriza esse tipo de programação.

A seguir, será definido um método de programação com DNA, baseado no método de filtragem, denominado método de Filtragem Seqüencial. As idéias apresentadas no capítulo 3, para o método de filtragem, serão aqui estabelecidas de forma a auxiliar o programador a desenvolver cada fase de projeto de um algoritmo. Neste capítulo serão abordados conceitos da teoria da solução de problemas, utilizando como bibliografia básica [BOV93].

4.1 Definição do Método de Filtragem Seqüencial

O método de Filtragem Seqüencial consiste em, a partir de um conjunto inicial de soluções possíveis[3], aplicar uma seqüência de operações de "peneira", a fim de eliminar as possíveis soluções não desejadas. Este método pode ser dividido em três fases: uma pré-computação, que é a construção do conjunto inicial das soluções possíveis, uma peneira, que elimina do conjunto inicial as soluções não viáveis[4] e a recuperação da solução.

As três fases são bem distintas. A primeira fase é a construção do conjunto inicial - base da computação - que constitui o conjunto de soluções possíveis para o problema proposto. Essa fase é caracterizada pelo não-determinismo inerente a reações biológicas, resultando na construção, em paralelo, de todas as possíveis soluções. E é esta a característica que faz o método apropriado para resolver problemas combinatoriais da classe NP.

---

[2] Um problema é dito combinatorial quando o espaço de soluções é construído através de arranjos, agrupamentos, ordenações ou seleção de objetos discretos, usualmente finitos. [TOS 2000]

[3] Solução possível é qualquer solução dentro do espaço de soluções definida pelo problema. [BOV93]

[4] Solução não viável é uma solução possível que não satisfaz o predicado de solução do problema. [BOV93]

Na segunda fase é aplicada ao conjunto inicial, uma série de operações de peneira, para eliminar as palavras que fazem parte do conjunto de soluções possíveis, mas não são viáveis.

A terceira fase é caracterizada pela verificação da existência e pela eventual recuperação da solução para o problema.

Neste capítulo serão apresentadas as três fases que constituem o método de Filtragem Seqüencial. Em 4.2 descreve-se como proceder para obter o conjunto inicial do método. Em 4.3 são descritas e classificadas, conforme cada categoria, as operações que caracterizam a fase de peneira. Na seção 4.4 são apresentadas as operações de teste, que caracterizam a terceira fase do método. As operações biológicas citadas neste capítulo foram detalhadas no capítulo 2.

4.2 Construção do conjunto inicial

Um conjunto inicial, para um problema, consiste de um conjunto de palavras tipicamente de tamanho O(n) (onde n é o tamanho do problema [AMO97]), que inclui todas as soluções possíveis (cada uma codificada por uma palavra) do problema a ser resolvido. Na verdade, o conjunto inicial é um multiconjunto, pois este possui elementos repetidos. No decorrer do texto quando o conjunto inicial for referenciado, entenda-se este como sendo um multiconjunto.

O conjunto inicial é a representação das entradas do problema, por exemplo, se o problema é gerar uma permutação de um conjunto ordenado de n elementos, então o conjunto inicial pode ser todas as palavras na forma $p(i_1)p(i_2)....\ p(i_n)$, onde cada i_k pode ser qualquer um dos inteiros no intervalo [1...n], e $p(i_k)$ codifica a informação na "posição i_k", isto é, todas as permutações possíveis [AMO96]. Aqui o conjunto tem cardinalidade exponencial no tamanho do problema (O(n!)). Por outro lado, se o problema é colorir grafos, o conjunto inicial conterá todas as palavras, representando todas atribuições de cores possíveis aos vértices do grafo.

Para construir o conjunto inicial faz-se necessário pensar como estas informações serão processadas, pois a codificação deve facilitar o processamento. Por exemplo, na fase de codificação, deve-se pensar se é necessário localizar uma subseqüência no início das seqüências de DNA. Caso seja necessário, esta subseqüência deve ser codificada de forma que sua localização possa ser identificada. No decorrer do capítulo, esta explicação será melhor explanada.

Apesar da construção do conjunto inicial ser dependente do problema a ser resolvido, algumas considerações são gerais e úteis. Para obter o conjunto inicial pode-se considerar quatro etapas, que são as seguintes:

1º. *Representação*. Gerar as seqüências representativas dos elementos que constituem o problema. Nesta etapa deve-se conhecer muito bem o problema a ser solucionado e construir seqüências que permitam que, através de operações biológicas, seja possível encontrar a solução do problema.

2º. *Reprodução*. Reproduzir estas seqüências, da primeira etapa, em número suficiente para que todas ligações possíveis tenham chance de acontecer.

3º. *Ligação*. Gerar aleatoriamente as ligações entre as seqüências de DNA, produzidas na segunda etapa.

4º. *Eliminação*. Eliminar as seqüências que não representam soluções possíveis para o problema.

Na etapa de *representação* cada elemento que compõe o problema é representado por uma seqüência de DNA. Para gerar estas seqüências pode-se utilizar dois métodos. Um deles, é a retirada da seqüência desejada do DNA de um organismo qualquer, como por exemplo uma bactéria. Esta forma, apesar de ser simples, é a menos utilizada, já que seria necessário restringir a fita de DNA do organismo, para reduzí-la somente à seqüência desejada. Outra forma de obter as seqüências desejadas, é através da utilização da operação de sintetização de seqüência em um sintetizador de oligonucleotídeos.

Geradas as seqüências de DNA representativas dos elementos que constituem a solução, deve-se reproduzir estas seqüências em número significativo (etapa de *reprodução*). Para multiplicar estas seqüências, utiliza-se, repetitivamente, a operação denominada reação em cadeia da *polimerase* (PCR). Esta operação produz novas cópias da seqüência desejada em um curto espaço de tempo.

Depois de replicadas as seqüências, utiliza-se à operação de hibridização para que as seqüências sejam ligadas entre si, formando um conjunto de várias seqüências. Para tornar as seqüências contínuas, são usadas as enzimas *ligase* (etapa de *ligação*). Nota-se que este conjunto é formado por fitas duplas de DNA, obtidas pela união das seqüências que representam os elementos do problema.

A quarta etapa, a de eliminação, depende da codificação utilizada. Esta etapa deve eliminar as fitas geradas que, grosseiramente, sabe-se que não serão úteis, isto é, são soluções não possíveis. Ordinariamente, uma eliminação que se faz necessária diz respeito ao tamanho da fita. Neste caso, procede-se utilizando a operação eletroforese em gel. Esta operação separa seqüências de DNA, de acordo com o tamanho da seqüência, ficando assim, simples separar as seqüências com um tamanho específico.

Restam então, apenas as seqüências que formam o conjunto de soluções possíveis para o problema (possivelmente, tendo várias seqüências repetidas). Tem-se assim, o conjunto inicial.

Para melhor compreensão de cada fase do método, será mostrado como exemplo, a solução para o problema do Caminho Hamiltoniano que foi o primeiro problema a ser solucionado, utilizando-se fragmentos de DNA [ADL94]. A codificação das entradas do problema, conforme Adleman definiu, é mostrada com mais detalhe no Anexo A.

Problema do caminho Hamiltoniano de v a w

Entrada: Um grafo G (V, E) direcionado e v, w $\in$ V.

Pergunta: Existe um caminho Hamiltoniano de v para w para este grafo?

O problema do caminho Hamiltoniano consiste em determinar se num dado grafo é possível, a partir de um vértice inicial, chegar ao vértice final, passando por todos os outros vértices do grafo uma única vez.

Considera-se o grafo direcionado G da FIGURA 4.1 como sendo um grafo a ser avaliado. Os vértices 0 e 3 são o vértice inicial e o vértice final, respectivamente.

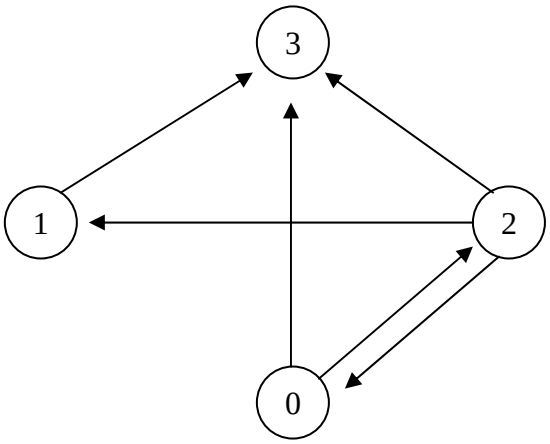


Figura 4.1: Grafo direcionado G

Adleman [ADL94] estabeleceu que cada vértice seria codificado por uma seqüência de DNA específica e cada aresta seria codificada, concatenando o complemento do final da seqüência de DNA do vértice de origem com o complemento do início do vértice de destino. As seqüências que codificam os vértices e as arestas podem ser vistas nas tabelas 4.1 e 4.2, respectivamente.

Tabela 4.1: Codificação das seqüências de DNA para cada vértice.

Vértice	Seqüência de DNA
0	CGTA
1	TGCA
2	ACTT
3	CCGT

Tabela 4.2: Codificação das seqüências de DNA para cada aresta possível.

Aresta	Seqüência de DNA
0 – 2	ATTG
0 – 3	ATGG
2 – 1	AAAC
2 – 3	AAGG
2 – 0	AAGC
1 – 3	GTGG

Após construir todas as seqüências que codificam os vértices e as arestas (etapa de *representação*), utiliza-se a operação de reação em cadeia da *polimerase* (PCR) para criar várias cópias destas seqüências (etapa de *reprodução*). Em seguida, colocam-se

estas seqüências em uma solução para que, através da hibridização, as seqüências complementares, na solução, formem um conjunto de seqüências que contenham o conjunto das representações das soluções possíveis (etapa de *ligação*), ou seja, formem todos os caminhos possíveis do grafo (FIGURA 4.2). O próximo passo é eliminar todas as seqüências que não possuem o tamanho desejado (etapa de *eliminação*). Neste caso, como cada vértice possui quatro nucleotídeos e cada aresta possui quatro nucleotídeos, o tamanho da fita que codifica uma solução possível será de 16 bases (nucleotídeos). Após eliminadas estas seqüências, o que resta é o conjunto das soluções possíveis para o problema.

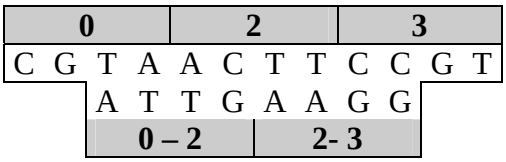


Figura 4.2: Possível seqüência do conjunto inicial

Observa-se que este exemplo é apenas acadêmico e para definir a melhor combinação para formar as seqüências de codificação desejada para um dado problema, deve-se observar algumas restrições que podem ser vistas em [BLU96]. Para codificar informações em seqüências de DNA, o ideal é que cada seqüência tenha, no mínimo, 20 pares de bases, ou seja, cada seqüência deve ser formada por 20 nucleotídeos. Destes nucleotídeos o ideal é que se tenha uma predominância, na medida do possível, de C e G, pois estes apresentam uma ligação de 3 pontes de hidrogênio. Enquanto que a ligação entre A e T possui somente 2 pontes de hidrogênio, sendo portanto, menos estável.

4.3 Peneira

Uma vez construído o conjunto inicial de palavras contendo todas as soluções possíveis do problema a ser resolvido, possivelmente com várias cópias repetidas, são aplicadas a ele as operações de peneira que vão, sucessivamente eliminando do conjunto inicial, as soluções não viáveis.

As operações de peneira são executadas sobre um conjunto de seqüências de DNA (que serão tratadas nesta seção como uma palavra sobre o alfabeto {A, T, C, G}) e têm como objetivo, selecionar seqüências com uma determinada característica.

As operações de peneira podem ser classificadas em duas categorias, como:

- **Operações de separação**: estas operações identificam, selecionam e descartam palavras com certas características como: iniciando, terminando, contendo ou não contendo uma determinada subpalavra.
- **Operações de unificação**: estas operações são caracterizadas por duplicar, criar nova cópia de uma palavra ou criar um multiconjunto de palavras, com a união de multiconjuntos dados.

Em ambas categorias existem operações específicas para serem utilizadas em diferentes contextos; na seção seguinte serão apresentadas estas operações.

4.3.1 Operações de Separação

As operações de separação são usadas no método de Filtragem Seqüencial para identificar, selecionar e descartar palavras com determinadas características. As operações de separação podem ser divididas em:

- **Identificação**: esta operação possui como parâmetros um multiconjunto T e uma palavra S. Sua função é identificar palavras do multiconjunto T que possuam S como subpalavra.
  Notação: identifica(T, S)
- **Localização**: esta operação possui como parâmetros um multiconjunto T, uma palavra S e uma posição i. Sua função é localizar todas as seqüências que possuem S como subpalavra, na posição i.
  Notação: localiza(T, S, { i })
- **Verificação**: esta operação possui como parâmetros um multiconjunto T e um inteiro positivo B. Sua função é gerar um novo multiconjunto, onde são armazenadas todas as palavras de T que contém comprimento menor ou igual a B.
  Notação:verifica(T, B)

A implementação biológica destas operações, com exceção da operação de verificação, dá-se, geralmente, através da operação de “separação por subseqüência”. Esta operação consiste em localizar e separar uma seqüência alvo, com determinadas características. Para localizar a seqüência alvo é necessário criar uma sonda que descreve a cadeia de nucleotídeos a ser localizada. Esta sonda é uma seqüência complementar da palavra que deseja-se identificar, ou seja, a seqüência complementar de S.

Para facilitar a localização de uma seqüência alvo pode-se utilizar, por exemplo, o sistema avidina/biotina, o qual é um kit contendo todas as substâncias necessárias para o funcionamento do sistema (FIGURA 4.3).

Figura 4.3: Kit do sistema Avidina/Biotina

Neste sistema faz-se necessário a criação de uma sonda, para localizar a seqüência alvo. Esta sonda deverá ser acrescido de biotina[5]. A biotina é um marcador não radiativo (vitamina H), que marca um dos carbonos que compõe um nucleotídeo. Após marcar a sonda, ele então é introduzido à solução e, através do processo de hibridização, ele ligar-se-á às seqüências alvo. Em seguida, deve ser adicionada à solução uma outra substância chamada, avidina[6], que é uma enzima que liga-se fortemente à biotina. A avidina irá ligar-se a todas seqüências que possuem biotina em sua composição. E por fim, para melhor visualizar as seqüências marcadas, já que a avidina não é um corante e não emite luz, é adicionado à solução um corante, por exemplo, Peroxido de Hidrogênio associado com Diamino Benzidina, que ao reagir com a avidina irá fazer com que as seqüências que contenham avidina, ganhem uma coloração diferente das demais, podendo assim, ser facilmente identificadas. Este processo está exemplificado na FIGURA 4.4, onde deseja-se identificar a seqüência que contenha a subseqüência GTTG; a sonda utilizado será a subseqüência complementar CAAC.

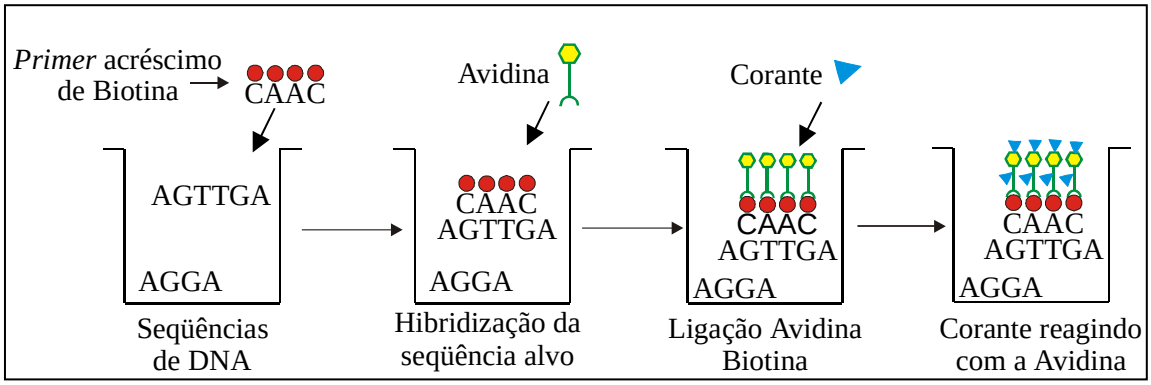


Figura 4.4: Sistema Avidina/Biotina

Quando deseja-se localizar uma subseqüência em um local específico, na seqüência de DNA, deve-se tomar cuidado na escolha da sonda, já que este vai garantir ou não, o sucesso da operação. Além do sistema avidina/biotina, existem outras formas de marcar as sondas, como por exemplo, o fósforo, a digoxigenina, entre outros.

Ao utilizar esta forma de implementação para as operações de separação (identificação e localização), pode-se, tanto aproveitar as seqüências que possuem coloração diferente, ou seja, as seqüências marcadas, como também, as que não possuem coloração diferenciada, formando assim, dois novos multiconjuntos: o multiconjunto das seqüências que possuem a subpalavra S e o multiconjunto das seqüências que não possuem a subpalavra S. O que vai determinar a utilização de qual dos multiconjuntos será usado é o algoritmo proposto para solucionar o problema.

Uma outra forma de trabalhar com as operações de separação (identificação e localização) é através de um multiconjunto único, ou seja, ao invés de separar as seqüências que contém uma determinada subseqüência daquelas que não a contém,

[5] Composto de baixo peso molecular utilizado como coenzima. Muito útil em técnicas de laboratório, como um marcador covalente para proteínas, permitindo sua detecção usando a avidina.
[6] Proteína encontrada na clara do ovo cru, que liga-se fortemente à biotina.

utiliza-se uma marcação e eliminam-se as seqüências marcadas ou as não marcadas, do multiconjunto.

Como o conjunto inicial possui apenas fitas simples de DNA, uma forma de marcar as seqüências alvo é torná-las fitas duplas. Para marcar determinadas seqüências, criam-se seus *primers*, que quando inseridos na solução irão ligar-se às seqüências alvo, tornando-as fitas duplas, ou seja, seqüências marcadas.

Para eliminar do multiconjunto as seqüências desejadas, que podem ser as marcadas ou as não marcadas, utilizam-se as enzimas exonucleases. Estas enzimas consomem nucleotídeos do fragmento de DNA. Se for desejado eliminar as seqüências marcadas, pode-se utilizar, por exemplo, as enzimas DNAse I que são responsáveis por destruírem fitas duplas de DNA. Já, se o desejo for eliminar as seqüências não marcadas, pode-se utilizar, por exemplo, as enzimas S1 nuclease que são responsáveis por destruírem fitas simples de DNA. Este processo é demonstrado na FIGURA 4.5.

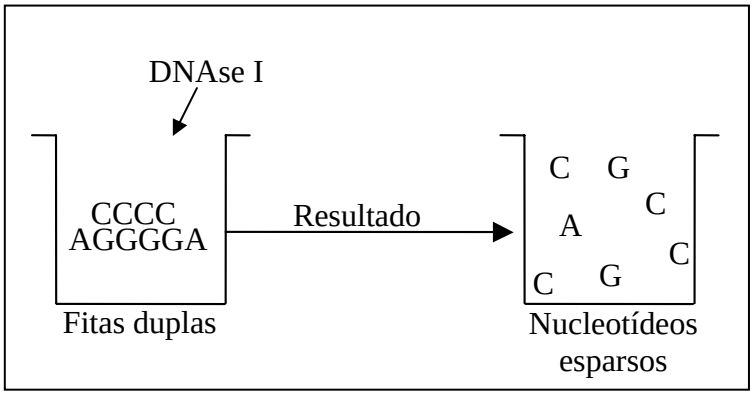


Figura 4.5: Eliminação de fragmentos de DNA

Caso, durante a simulação, for necessário desmarcar as seqüências basta aplicar a operação de desnaturação que separa as fitas duplas em fitas simples.

Ao contrário das operações de identificação e localização, a operação de verificação pode ser facilmente executada através da operação eletroforese em gel, onde as seqüências de DNA são separadas por tamanho, conforme FIGURA 4.6.

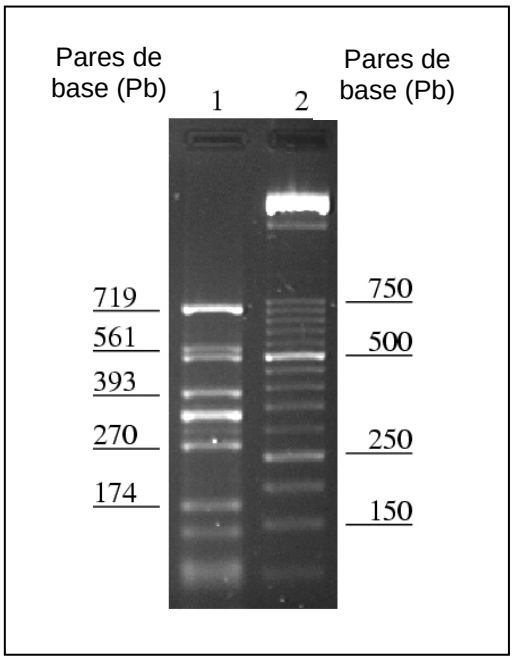


Figura 4.6: Fotografia de uma operação eletroforese em gel

Os números ao lado das bandas de DNA indicam o peso molecular (Pb) dos fragmentos de DNA.

4.3.2 Operações de Unificação

As operações de unificação são caracterizadas, no método de Filtragem Seqüencial, por duplicar, criar novas cópias de uma palavra ou criar um multiconjunto de palavras. As operações de unificação são:

- **União**: esta operação consiste em, dados os multiconjuntos T_1, T_2, …, T_n, criar a união destes multiconjuntos. Para implementar esta operação biologicamente, basta serem colocados os conteúdos dos multiconjuntos (dentro de um tubo chamado *eppendorf*) T_1, T_2, …, T_n em um único multiconjunto.
  Notação: uni(T_1, T_2, …, T_n)
- **Duplicação**: esta operação consiste em duplicar o conteúdo do multiconjunto T. Esta operação é implementada biologicamente, utilizando-se a técnica chamada de reação em cadeia da *polimerase* (PCR). Com esta técnica é possível produzir, em um curto espaço de tempo, muitas cópias de uma molécula de DNA.
  Notação: duplica(T)
- **Amplificação**: esta operação consiste em criar um novo conjunto T', que é uma cópia idêntica do conjunto T. A implementação biológica desta operação é bastante complicada e muito sujeita a erros [AMO96].
  Notação: amplifica(T, T')

Para exemplificar a fase de peneira, considera-se o conjunto inicial criado na fase anterior (construção do conjunto inicial), para o problema do caminho Hamiltoniano. Depois de construído o conjunto inicial, contendo todas as possíveis soluções do problema, faz-se necessário submeter este conjunto a uma seqüência de passos (algoritmo), que elimine do conjunto inicial, as soluções não viáveis ao problema.

Algoritmo para o problema do caminho Hamiltoniano

{ *Entrada*: Conjunto inicial T.

Saída: solução (solução = V existe o caminho Hamiltoniano, caso contrário solução = F).}

(1) T' ← localiza(T, v_0, { 0 });
(2) T'' ← localiza(T', v_n, { n });
(3) para i de 1 até n-1 faça
(4) T'' ← identifica(T'', v_i);
(5) fim-para;

Inicialmente, são separadas do multiconjunto T todas as seqüências que iniciam pelo vértice v_0 e estas seqüências são colocadas em T' (linha 1). O passo seguinte é separar de T' todas as seqüências que são finalizadas pela subseqüência do vértice v_n e estas seqüências são separadas em T''(linha 2). Em seguida, para cada vértice v_i, são armazenadas em T'' somente as seqüências que possuem a subseqüência que representa o vértice v_i (linha 4).

4.4 Recuperação da Solução

Após construído o conjunto inicial de palavras e este ser submetido às operações de peneira, resta saber se existe uma solução viável que satisfaça o problema. Dependendo do problema, para obter sua solução, basta saber se o conjunto de soluções viáveis é vazio ou não, caso seja vazio, a resposta para o problema é NÃO, caso contrário, é SIM (problemas de decisão). Se o problema for de localização, qualquer elemento do conjunto de soluções viáveis é uma solução para o problema, basta apenas recuperar uma delas.

Para verificar se existe uma solução viável ou recuperar uma delas, utilizam-se as operações de teste que são definidas na seção 4.4.1.

4.4.1 Operações de Teste

As operações de teste são caracterizadas no método de Filtragem Seqüencial por verificar se o multiconjunto está vazio ou não ou por selecionar um elemento do multiconjunto. As operações de teste são divididas em:

- **Detecção**: esta operação determina se existe alguma palavra no multiconjunto T; caso exista, é retornado “verdadeiro”, senão é retornado “falso”.
  Notação: detecta(T)
- **Seleção**: esta operação seleciona um elemento aleatório de T, se o conjunto T é vazio então é retornado “vazio”.
  Notação: seleciona(T)

A implementação biológica destas operações consiste em verificar se no *eppendorf* restante, ou seja, aquele que já passou pela fase de peneira, ainda existe alguma seqüência de DNA.

Para ambas as operações o processo inicial é o mesmo. É aplicada a técnica de reação em cadeia da *polimerase* (PCR) para aumentar o número das seqüências da solução, se é que elas existem, para melhor identificá-las. Após ser aplicada a operação de PCR, na solução, ela é colocada em uma cuba retangular, chamada cuba de eletroforese (FIGURA 4.7), a qual é colocado o gel de agarose, sendo então é submetida a um campo eletro-magnético. Assim, as seqüências irão ser separadas uma das outras, na solução, e através de um corante colocado no gel de agarose pode-se visualizar as seqüências quando elas forem submetidas a uma luz ultravioleta.

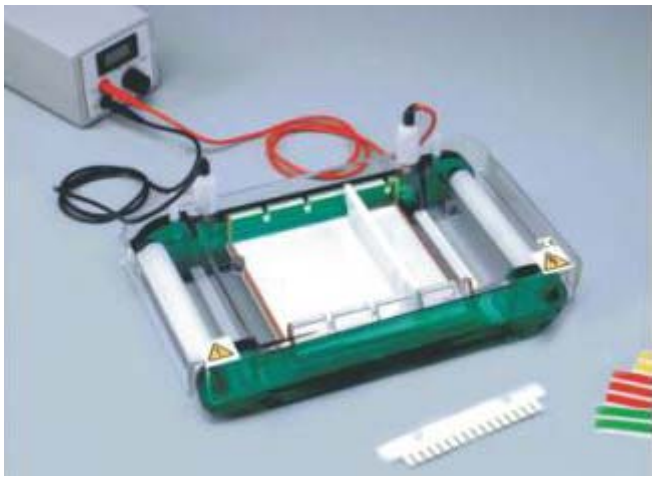


Figura 4.7: Cuba de Eletroforese

A partir deste ponto, se a operação for de detecção e caso exista alguma seqüência resultante, a solução para o problema é “verdadeiro”, caso contrário, é “falso”. Já, se a operação for de seleção, basta identificar se existe alguma seqüência restante e então pegá-la. Como as seqüências estarão separadas e são visíveis a luz ultravioleta, é simples separar uma delas. Depois de separada a seqüência, sua informação pode ser decodificada através de um seqüenciador de DNA.

5 ILUSTRAÇÃO DA DEFINIÇÃO DO MÉTODO DE FILTRAGEM SEQÜENCIAL

Neste capítulo serão apresentados alguns problemas solucionados através da computação com DNA. Os problemas serão representados, usando a notação proposta por Garey e Johnson [GAR79], pois ela apresenta o problema de forma clara e de fácil entendimento. Os algoritmos foram coletados na literatura existente.

5.1 Problema 3-vértice-colorível

Este problema foi provado ser NP-Completo, por Karp [KAR72] e é definido a seguir:

Problema 3-vértice-colorível

Entrada: Um grafo G: (V, E).

Pergunta: G é 3-vértice-colorível?

Um grafo G é 3-vértice-colorível se for possível atribuir cores (de um conjunto de três cores) a seus vértices, de tal forma que, dois vértices adjacentes não possuam a mesma cor.

O conjunto das soluções possíveis para este problema consiste em todas as colorações possíveis para o grafo (tal conjunto possui cardinalidade 3^n, onde n é o número de vértices do grafo) e o conjunto de soluções viáveis é aquele que satisfaz a condição do problema, ou seja, que não atribui a mesma cor a vértices adjacentes. Pode-se observar que, partindo do conjunto de soluções possíveis e através de operações de peneira, chega-se ao conjunto de soluções viáveis.

A solução do problema 3-vértice-colorível foi proposta por Adleman [ADL95] e posteriormente, por Amos [AMO97]; cada um deles utilizou suas operações para propor a solução do problema. A seguir, são descritos os algoritmos propostos por cada autor, em detalhes, para obtenção das soluções viáveis.

5.1.1 Proposta de Adleman

Adleman desenvolveu um algoritmo para o problema 3-vértice-colorível, onde o conjunto inicial T é um multiconjunto que consiste de palavras: $c_1 \bullet c_2 \bullet ... \bullet c_n$, onde $c_i \in$

{r_i, g_i, b_i} e n=|V|. Os nodos do grafo G são enumerados 1, 2, ..., n e $c_i = r_i$ significa que o $i^{ésimo}$ nodo é colorido com a cor "*red*", $c_i = g_i$, significa que o $i^{ésimo}$ nodo é colorido com a cor "*green*" e $c_i = b_i$, significa que o $i^{ésimo}$ nodo é colorido com a cor "*blue*". Assume-se que todas as possíveis colorações são representadas no conjunto inicial T, em número suficiente.

O algoritmo é o seguinte:

Algoritmo de Adleman

{ *Entrada*: Conjunto inicial T.

Saída: solução (solução = V se o conjunto T é não vazio, caso contrário, solução = F se o conjunto T é vazio).}

Algoritmo	**Método de Filtragem Seqüencial**
(1) solução ← V;	Inicialização
(2) para i de 1 até n faça	Iteração [2 –11]
(3) $T_r \leftarrow +(T, r_i)$; $T_{bg} \leftarrow -(T, r_i)$; (4) $T_b \leftarrow +(T_{bg}, b_i)$; $T_g \leftarrow -(T_{bg}, b_i)$;	Operações de separação por identificação
(5) para cada j tal que <i,j> ∈ E faça	Iteração [5 - 9]
(6) $T_r \leftarrow -(T_r, r_j)$; (7) $T_g \leftarrow -(T_g, g_j)$; (8) $T_b \leftarrow -(T_b, b_j)$;	Operações de separação por identificação
(9) fim-para;	
(10) T ← *merge*(T_r, T_g, T_b);	Operação de unificação por união
(11) fim-para;	
(12) solução ← *detect*(T);	Operação de teste por detecção

Na linha 1 atribui-se o valor verdadeiro para a solução. Então, a cada iteração [2 - 11], para o vértice $v_i \in V$, realizam-se os seguintes passos: divide-se T em 3 conjuntos T_r, T_g e T_b, onde T_r contém somente as palavras, contendo na posição i o símbolo r_i, T_g contém somente as palavras, contendo os símbolos g_i na posição i e T_b contém somente as palavras, contendo o símbolo b_i na posição i (linhas 3 e 4). Quando a execução chega à linha 5, pela primeira vez, T_r contém todas as colorações possíveis com o primeiro vértice "red"; T_b contém todas as colorações possíveis com o primeiro vértice "blue" e T_g o mesmo para "green". Então, para cada aresta <i, j> ∈ E, removem-se destes conjuntos (T_r, T_g e T_b) qualquer palavra contendo $c_i = c_j$ (isto é, essas palavras codificam colorações, onde os vértices adjacentes i e j são da mesma cor – solução não viável) (linhas 5 a 9). Então, estes conjuntos (T_r, T_g e T_b) são unidos, formando um novo conjunto T (linha 10) e o algoritmo segue para o próximo vértice e para a próxima iteração. Após todos os vértices terem sido considerados, executa-se uma verificação (linha 12); se T não for vazio, então qualquer palavra em T codifica uma coloração de três cores possível para o grafo G, isto é, o grafo é 3-vértice-colorível.

5.1.2 Proposta de Amos

Amos também definiu a solução, segundo seu conjunto de operações, para o problema 3-vértice-colorível. Amos apresentou a descrição de um método de filtragem paralelo. Este método foi o primeiro a prover um padrão formal para a descrição dos algoritmos de DNA, para qualquer problema da classe NP.

O conjunto inicial T consiste de todas as palavras da forma $p_1c_1p_2c_2....p_nc_n$, onde n = |V| é o número de vértices no grafo. Para todo i, p_i codifica unicamente o nodo i e cada c_i é qualquer uma das "cores" 1, 2 ou 3, tal que para cada i, a cor c_i é atribuída ao vértice i, onde cada palavra representa uma possível atribuição de cores aos vértices do grafo.

Algoritmo de Amos

{ *Entrada*: Conjunto inicial T.

Saída: solução (retorna uma atribuição de cores válida para o problema ou vazio).}

Algoritmo	**Método de Filtragem Seqüencial**
(1) solução ← V;	Inicialização
(2) para j de 1 até n faça	Iteração [2 – 8]
(3) *copy*(T,{T_1,T_2,T_3});	Operação de unificação por amplificação
(4) para i de 1 até 3 e todo k tal que (j, k) ∈ E faça	Iteração [4 - 6]
(5) *remove* (Ti,{$p_j \neq i$, $p_k i$});	Operações de separação por identificação
(6) fim-para;	
(7) T ← *union*({T_1,T_2,T_3}, T);	Operação de unificação por união
(8) fim-para;	
(9) solução ← *select*(T);	Operação de teste por detecção

O algoritmo inicia atribuindo um valor verdadeiro para a solução (linha 1). A seguir, inicia-se uma iteração [2 – 8], onde j varia de 1 até n, sendo j a posição do respectivo vértice nas palavras e n o número de vértices. Em seguida, são criadas três cópias (T_1, T_2 e T_3) do conjunto inicial T. Para cada palavra representada em T_i, com i variando de 1 até 3, faz-se (iteração [4 – 6]) a remoção de T_i (linha 5):

- Todas as palavras que, para o vértice correspondente a j, não codificam a cor correspondente a i ($p_j \neq i$).
- E de todas as palavras que codificam vértices adjacentes, ao correspondente a j, com a mesma cor ($p_k i$).

Concluída a iteração [4 – 6], os conjuntos T_1, T_2 e T_3 são unidos formando o novo conjunto T e a computação retorna à iteração [1 – 7]. Após a $j^{ésima}$ iteração do laço, a computação assegura que, nas palavras restantes para qualquer vértice correspondente a j, não existirão vértices adjacentes da mesma cor. Assim, quando o algoritmo finaliza, T

somente codifica colorações válidas, se elas existem. Certamente, toda a coloração válida estará representada em T.

A principal diferença da solução proposta por Amos em relação à de Adleman, é que na solução de Amos, nos passos de separação, as palavras que são removidas, são descartadas e não participam de computações futuras. Já na solução de Adleman, as palavras são conservadas e podem ser utilizadas posteriormente na computação.

5.2 Problema do caminho Hamiltoniano

Adleman também apresentou uma solução para o problema do caminho Hamiltoniano, que é um problema NP-Completo (provado por [KAR72]). Este foi o primeiro problema computacional a ser resolvido através da computação com DNA.

Problema do caminho Hamiltoniano de v a w
Entrada: Um grafo G: (V, E) direcionado e v, w ∈ V.
Pergunta: Existe um caminho Hamiltoniano de v para w para este grafo?

Este problema consiste em determinar se existe um caminho que comece em um vértice inicial e termine em um vértice final, passando por todos os vértices remanescentes, exatamente uma vez.

O conjunto das soluções possíveis para este problema consiste de todos os caminhos possíveis entre os vértices do grafo, sendo que o conjunto de soluções viáveis é aquele que satisfaz a condição do problema, ou seja, começa no vértice inicial e termina no vértice final, passando uma única vez pelos demais vértices do grafo.

A obtenção do conjunto inicial T, para este problema, foi demonstrado na seção 4.1. A seguir, descreve-se o algoritmo de Adleman.

Algoritmo de Adleman

{ *Entrada*: Conjunto inicial T.

Saída: solução (solução = V existe o caminho Hamiltoniano, caso contrário, solução = F).}

Algoritmo	**Método de Filtragem Seqüencial**
(1) solução ← V;	Inicialização
(2) T ← B (T, v_0); (3) T ← E(T, v_n);	Operações de separação por localização
(4) para i de 1 até n-1 faça	Iteração [4 - 6]
(5) T ← +(T, v_i);	Operação de separação por identificação
(6) fim-para;	
(7) solução ← *detect*(T);	Operação de teste por detecção

Na linha 1, atribui-se o valor verdadeiro para solução. Então, retiram-se do conjunto T todas as palavras que não começam com o vértice inicial (linha 2). Em seguida, retiram-se do conjunto T todas as palavras que não terminam com o vértice final (linha 3). Então, para cada vértice v_i, armazena-se em T somente as seqüências que possuem a palavra que representa o vértice v_i (linha 5). Após todos os vértices serem checados, executa-se uma verificação (linha 7), se T não for vazio, então qualquer palavra em T codifica o caminho Hamiltoniano para o grafo G.

5.3 Problema *Satisfability*

O SAT, uma abreviação de *Satisfability*, foi o primeiro problema mostrado NP-Completo (através do teorema de Cook [COO71]).

Problema Satisfability
Entrada: Um conjunto V de variáveis lógicas e uma coleção de cláusulas C, sobre V.
Pergunta: Existe uma atribuição verdade para cada variável em V, tal que torne todas as cláusulas de C verdadeiras?

O problema SAT consiste de uma sentença lógica na forma normal conjuntiva, isto é, composta de cláusulas ligadas pelo operador lógico "*and*", e cada cláusula é composta de literais (variáveis ou sua negação) ligados pelo operador lógico "*or*".

Para resolver uma instância I do SAT, gera-se todas as possíveis atribuições verdade para as variáveis de V (são 2^n atribuições) e testa-se se alguma torna todas as cláusulas de C verdadeiras.

O conjunto de soluções possíveis, para este problema, são todas as possíveis seqüências que podem ser formadas, atribuindo valores verdadeiros e falsos para as variáveis de V. Já o conjunto de soluções viáveis, é formado somente pelas seqüências que satisfazem a condição do problema, ou seja, tornam todas as cláusulas de C verdadeiras.

O conjunto inicial T contém as 2^n palavras, que codificam uma seqüência simples n-bit, representando uma atribuição verdade para as n variáveis, onde "1" representa o valor verdadeiro e "0" o valor falso. Logo o conjunto de soluções possíveis T para o problema, contém todas as seqüências n-bit.

A solução do SAT foi proposta por Lipton [LIP94], Beigel [BEI98a] e Liu [LIU96], cada um deles utilizou seu conjunto de operações para propor a solução do problema. A seguir, apresenta-se as soluções propostas pelos autores.

5.3.1 Proposta de Lipton

Lipton definiu o seguinte algoritmo para solucionar o SAT:

(1) Criar o conjunto inicial T

(2) solução = "Verdadeiro"

(3) para cada cláusula c ∈ C faça

(4) para cada literal v_i de c faça

(5) se $v_i = v_j$ (para algum i) então extraia de T as palavras codificadas com $v_i = 1$

(6) senão extraia de T as palavras codificadas com $v_i = 0$

(7) fim-para

(8) criar novo conjunto T com a união das palavras extraídas

(9) fim-para

(10) se $T \neq \varnothing$ então solução = "verdadeiro"

Assim, o pseudocódigo do algoritmo pode ser expresso formalmente como:

Algoritmo de Lipton

{ *Entrada*: Conjunto inicial T.

Saída: solução (solução = V se I é satisfazível, caso contrário, solução = F, o conjunto T é vazio).}

Algoritmo	**Método de Filtragem Seqüencial**
(1) solução ← V;	Inicialização
(2) para a de 1 até |C| faça	Iteração [2 – 11]
(3) para b de 1 até | C_a| faça	Iteração [3 – 9]
(4) se $v_b^a = v_j$ então	Condição
(5) T_b ← *extract*(T, $v_b^a = 1$);	Operação de separação por localização
(6) senão	
(7) T_b ← *extract*(T, $v_b^a = 0$);	Operação de separação por localização
(8) fim-se;	
(9) fim-para;	
(10) T ← *merge* (T_1, T_2, …, T_b);	Operação de unificação por união
(11) fim-para;	
(12) solução ← *detect*(T);	Operação de teste por detecção

Na linha 1 atribui-se o valor verdadeiro para a solução. Então para cada cláusula $C_a = \{v_1^a, v_2^a, ..., v r_a^a\}$ (linha 3), onde $r_a = |C_a|$, executam-se os passos seguintes:

Para cada literal v_b^a (linha 4), onde v_b^a é o índice em V da variável do $b^{ésimo}$ literal da $a^{ésima}$ cláusula:

- Se v_b^a computa um literal positivo (ou seja, não negado), então extrai-se de T todas as palavras codificadas por 1 na posição correspondente a v_b^a, colocando-se estas palavras em T_b, isto é, T_b satisfaz a cláusula C_a (linha 5).
- Se v_b^a computa um literal negado, extrai-se de T todas as palavras codificadas por 0 na posição correspondente a v_b^a, colocando-se estas palavras em T_b (linha 6).

Após |C| interações (nº de cláusulas), constroem-se as atribuições que fazem C_a = "verdadeiro". Cria-se então, um novo conjunto T resultante da união dos conjuntos T_1, T_2,...., Tr_a (linha 9); neste ponto, T consiste de todas as atribuições verdade para V que satisfazem a cláusula C_a. Como a operação é acumulativa, ao terminar as variáveis de |C| iteração (linha 10), T terá somente as atribuições verdade que satisfazem todas as cláusulas e então, bastará saber se T é vazio ou não (linha 11). Se restar alguma cadeia de símbolos em T, então I é satisfazível.

5.3.2 Proposta de Beigel

Beigel apresentou uma solução para uma instância do problema SAT, cuja sentença lógica a ser avaliada é I : $(x \vee y) \wedge (\bar{x} \vee \bar{y})$

Neste caso o conjunto inicial contém 4 palavras (2^n, onde n é número de variáveis), que codificam uma seqüência n-bit, representando uma atribuição verdade para as variáveis. O conjunto inicial é formado, pelo menos, pelas seguintes seqüências: 00, 01, 10, 11, que correspondem a todos os valores possíveis para as duas variáveis (00 - x é falso; 01 - x é verdadeiro; 10 - y é falso; 11 - y é verdadeiro).

Algoritmo de Beigel e Fu

{ *Entrada*: Conjunto inicial T.

Saída: solução (solução = V se I é satisfazível, caso contrário, solução = F, o conjunto T é vazio).}

Algoritmo	**Método de Filtragem Seqüencial**
(1) *separate*(T, 1, 1, T_1, T_2); (2) *separate*(T_2, 1, 2, T_2, T_3);	Operações de separação por localização
(3) *merge*(T_1, T_2, T_2);	Operação de unificação por união
(4) *separate*(T_2, 0, 1, T_1, T_2); (5) *separate*(T_2, 0, 2, T_2, T_4);	Operações de separação por localização
(6) *merge*(T_1, T_2, T);	Operação de unificação por união
(7) solução ← *test*(T)	Operação de teste por detecção

Na linha 1, são colocadas em T_1 todas as palavras que tornam o primeiro literal da primeira cláusula verdadeiro e, em T_2 todas as palavras restantes. Em seguida, coloca-se

em T_2 todas as palavras de T_2 que tornam o segundo literal da primeira cláusula verdadeiro (linha 2). Note que a partir da execução das linhas 1 e 2, todas as palavras que tornavam a primeira cláusula falsa foram excluídas. Na linha 3, os multiconjuntos T_1 e T_2 são unidos em T_2. Nas linhas 4 e 5 são excluídas de T_2 todas as palavras que tornam a segunda cláusula falsa. Na linha 6 são unidos os multiconjuntos T_1 e T_2 em T e finalmente, é feita a verificação, em T, se a sentença lógica foi satisfeita.

5.3.3 Proposta de Liu

Liu define um método denominado "*surface-based*". Segundo Liu, suas operações são mais eficazes que as demais, pois reduzem significativamente, a perda de moléculas de DNA durante a execução dos passos de seus algoritmos, uma vez que, suas operações são de simples implementação biológica.

Liu demonstrou a eficiência de suas operações, apresentando dois algoritmos para solucionar o problema SAT, os algoritmos propostos por ele são as seguintes:

1ª algoritmo proposto por Liu

{ *Entrada*: Conjunto inicial T.

Saída: solução (solução = V se existe uma solução para o problema, caso contrário, solução = F, o conjunto T é vazio).}

Algoritmo	**Método de Filtragem Seqüencial**
(1) solução ← V;	Inicialização
(2) para cada cláusula ($x_{i_1} \vee ... \vee x_{i_k} \vee \overline{x}_{j_1} \vee ... \vee \overline{x}_{j_m}$) faça	Iteração [2 – 5]
(3) T ← *mark* ((i_1, 0), …, (i_k, 0), (j_1, 1), . . . , (j_m, 1));	Operação de separação por localização
(4) T ← *destroy-marked*(T);	Operação de separação por identificação
(5) fim-para;	
(6) solução ← *test-if-empty*(T);	Operação de teste por detecção

Neste algoritmo Liu, propõe marcar todas as cláusulas (palavras) do conjunto inicial que possuem valor negativo, ou seja, que tornam a sentença lógica falsa. Na linha 3, são marcadas todas as palavras que possuem: valor 0 (falso) para os literais positivos e valor 1 (verdadeiro) para os literais negativos. Em seguida, são destruídas todas as palavras marcadas. Estas operações são repetidas para todas as cláusulas da sentença (iteração [2 - 5]). Na linha 6, é realizada a verificação se existe alguma atribuição que satisfaça a sentença lógica.

2ª algoritmo proposto por Liu

{ *Entrada*: Conjunto inicial T.

Saída: solução (solução = V se existe uma solução para o problema, caso contrário, solução = F, o conjunto T é vazio).}

Algoritmo	Método de Filtragem Seqüencial
(1) solução ← V;	Inicialização
(2) para cada cláusula C faça	Iteração [2 – 9]
(3) para cada variável não-negativa x_i de C faça	Iteração [3 – 5]
(4) T ← *mark* (i, 1);	Operação de separação por localização
(5) fim-para;	
(6) para cada variável negativa x_i de C faça	Iteração [6 – 8]
(7) T ← *mark* (i, 0);	Operação de separação por localização
(8) fim-para;	
(9) fim-para;	
(10) T ← *destroy-unmarked*(T);	Operação de separação por identificação
(11) solução ← *test-if-empty*(T);	Operação de teste por detecção

Ao contrário do de seu primeiro algoritmo, neste Liu propõe que sejam marcadas todas as palavras que tornam a sentença lógica verdadeira. Na iteração [3 - 5], são marcadas todas as palavras que possuem literais positivos com valor 1 (verdadeiro). Já na iteração [6 - 8], são marcadas todas as palavras que possuem literais negativos com valor 0 (falso). Esta marcação é realizada para cada cláusula da sentença (iteração [2 - 9]). Em seguida são destruídas todas as palavras não marcadas do multiconjunto T (linha 10). Por fim, é realizada a verificação da existência ou não da palavra que satisfaz a sentença lógica.

5.4 Problema da Permutação

Amos definiu a solução para o problema das permutações, que consiste em criar o conjunto P_n de todas permutações dos inteiros {1, 2,...., n}.

Problema das Permutações

Entrada: Conjunto de n inteiros.

Procura: O conjunto P_n de todas as permutações dos n inteiros.

O conjunto de soluções possíveis para este problema é formado pela seqüência dos n inteiros, podendo inclusive conter elementos repetidos. Já o conjunto de soluções viáveis é formado pelas seqüências que satisfazem o problema, ou seja, apenas aquelas que contém todas as permutações dos n inteiros, sem que haja elementos repetidos na permutação.

O conjunto inicial T consiste de todas as palavras com o formato $p_1i_1p_2i_2....p_ni_n$, onde, para todo j, p_j codifica unicamente a "posição j" e cada i_j pertence a {1, 2,.., n}. Cada palavra de T consiste da seqüência de n inteiros com, possivelmente, ocorrências do mesmo inteiro.

Para obter-se o conjunto inicial de todos elementos pode-se proceder da seguinte maneira: geram-se subseqüências únicas, codificando cada p_ki_k, onde $1 \leq k \leq n$ e $1 \leq i_k \leq i$. Assim, forma-se um número polinomial n_k de diferentes subseqüências. A tarefa agora é combinar as subseqüências para formar o conjunto inicial desejado. Isto é feito como segue: para cada par, (p_ki_k, $p_{k+1}i_{k+1}$), é construída uma seqüência que é a concatenação do complemento da segunda parte da subseqüência p_ki_k, com o complemento da primeira parte da subseqüência $p_{k+1}i_{k+1}$. Também são formadas seqüências que são o complemento da primeira parte da subseqüência p_1i_1 e da segunda parte da subseqüência p_ni_n. Tem-se assim, um total de 2ni+1 seqüências na solução. O efeito de adicionarem-se estas novas seqüências será formar, no *eppendorf*, uma fita que será o elemento do conjunto inicial desejado. As novas seqüências, através de hibridização, agem como "suportes" para unir as primeiras seqüências nas seqüências desejadas. Estes "suportes" podem então, ser removidos da solução (assumindo que eles são biotilinizados[7]), obtendo assim, o conjunto inicial esperado.

Algoritmo de Amos

{ *Entrada*: Conjunto inicial T.

Saída: conjunto P_n, contendo todas as permutações dos n inteiros.}

Algoritmo	**Método de Filtragem Seqüencial**
(1) para j de 1 até n-1 faça	Iteração [1 – 7]
(2) *copy*(T, {T_1, T_2, ..., T_n});	Operação de unificação por amplificação
(3) para i de 1 até n e para todo k>j faça	Iteração [3 – 5]
(4) *remove*(T_i, {$p_j \neq i$, p_ki});	Operação de separação por localização
(5) fim-para;	
(6) *union*({T_1, T_2, ..., T_n}, T);	Operação de unificação por união
(7) fim-para;	
(8) $P_n \leftarrow T$;	

O algoritmo inicia com uma iteração [1 - 7] de j variando de 1 até n-1, onde j representa a posição de cada inteiro na palavra. Em seguida, criam-se tantas cópias de T, quanto o número de inteiros (n) a serem permutados (linha 2). Para cada palavra representada em T_i, com i variando de 1 até n, faz-se (iteração [3 – 5]) a remoção de T_i (linha 4):

[7] Seqüências biotilinizados são seqüências marcadas com biotina permitindo sua detecção através da avidina.

- De todas as palavras que possuem na posição j, o inteiro diferente ao correspondente de i ($p_j \neq i$).
- E de todas as palavras que possuem o inteiro correspondente a i, nas posições maiores que j ($p_k i$).

Concluída a iteração [3 – 5], os conjuntos T_n são unidos formando o novo conjunto T e a computação retorna à iteração [1 – 7]. Após a $j^{ésima}$ iteração do laço, a computação assegura que, nas palavras restantes, o inteiro i_j não está duplicado nas posições k>j da palavra. O inteiro i_j pode ser qualquer número do conjunto {1, 2, ..., n}. Ao final da computação, cada palavra restante contém exatamente uma ocorrência de cada inteiro do conjunto {1, 2, ..., n} e assim, representa uma das possíveis permutações. Dada a entrada específica, é fácil ver que P_n será o conjunto de todas as permutações do número, dos n primeiros naturais.

5.5 Problema do circuito Hamiltoniano direcionado

O problema do circuito Hamiltoniano direcionado foi provado ser um problema NP-Completo, por Karp em [KAR72].

Problema do circuito Hamiltoniano direcionado

Entrada: Um grafo direcionado G (V, E).

Pergunta: Existe um circuito Hamiltoniano para este grafo?

Este problema consiste em determinar se existe um circuito que comece e termine em um determinado vértice, passando por todos os vértices remanescentes exatamente uma vez.

A solução para este problema foi apresentada por Fitzgibbons em [FIT2000]. Nele Fitzgibbons não descreve seu algoritmo através de nenhum conjunto de operações, e sim, apenas descreveu uma seqüência de passos para solucionar o problema.

Fitzgibbons, definiu sua codificação para a entrada do problema, baseado na codificação usada por Adleman, onde cada vértice ($x_i y_i$) é representado por uma seqüência formado por 20 bases, sendo que x_i representa as 10 primeiras bases e y_i representa as 10 bases restantes. Os passos propostos por Fitzgibbons, para a solução do problema, são os seguintes:

Passos propostos	Método de Filtragem Seqüencial
(1) Codificar cada vértice (x_iy_i) por uma seqüência formado por 20 bases; (2) Codificar cada aresta (v_i, v_j) $\in$ E como segue: a. Para cada aresta que parte de v_s (v_s, v_i) representa-se à aresta pelas bases complementares de $x_sy_sx_j$; b. Para cada aresta que termina em v_s (v_i, v_s) representa-se à aresta pelas bases complementares de $y_ix_sy_s$; c. Demais aresta (v_i, v_j) serão representadas pelas bases complementares y_ix_j; (3) Localizar todos os caminhos que contém exatamente n+1 vértices;	Obtenção do conjunto inicial
(4) Localizar todos caminhos que começam é terminam com v_s; (5) Localizar todos os caminhos que contém cada vértice uma única vez;	Peneira através da operação de separação por identificação
(6) Verificar se existe algum caminho restante.	Recuperação da solução através da operação de teste por detecção

Pode-se notar, através deste paralelo, que os passos propostos por Fitzgibbons poderiam ser descritos através do método de Filtragem Seqüencial.

5.6 Problema do Caixeiro Viajante

O problema do caixeiro viajante é um problema NP-Difícil, provado por [KAR72].

Problema do Caixeiro viajante

Entrada: Um conjunto C de m cidades, uma distância $d(c_i, c_j) \in Z^+$, para cada par de cidades $c_i, c_j \in C$, e um inteiro positivo B.

Pergunta: Existe uma rota em C que seja de comprimento menor ou igual a B, isto é, uma permutação $<c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)}>$ de C, tal que $\left(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B$?

Neste problema, além de se desejar saber se existe um caminho que parta de uma cidade inicial e termine em uma cidade final, passando por todas as demais cidades

(vértices), deseja-se também saber se existe uma rota que seja menor ou igual a um determinado valor pré-estabelecido.

A solução para uma instância do problema do Caixeiro Viajante foi proposta por Narayanan, em [NAR97] e [NAR98]. Esta instância é uma versão do problema do Caixeiro Viajante, como um problema de decisão.

O conjunto de entrada deste problema pode ser visto como um grafo, onde as cidades são os vértices e as distâncias entre as cidades são as arestas rotuladas (caminhos binários), assim fica fácil enxergar o problema como uma variação do problema do Caminho Hamiltoniano.

Narayanan propôs os seguintes passos para solucionar este problema:

Passos propostos	**Método de Filtragem Seqüencial**
(1) Codificar uma única seqüência para cada cidade; (2) Gerar seqüências de DNA para cada caminho binário entre duas cidades X e Y como segue: a. Para o caminho X → Y, una n ocorrências da seqüência de DNA de X com uma ocorrência da seqüência de DNA de Y, onde n é o valor da aresta (X, Y); b. Para o caminho Y → X, una n ocorrências da seqüência de DNA de Y com uma ocorrência da seqüência de DNA de X; c. Forme uma fita complementar onde as junções assegurem o caminho binário; (3) Para formar rotas maiores, junte dois caminhos P_1 e P_2 se, e somente se, o código da cidade final para P_1 coincidir com o nodo da cidade inicial para P_2. Descarte a metade do código do nodo da cidade final em P_1 e a metade do código da cidade inicial em P_2 (que são coincidentes); (4) Para prevenir *loops*, não forme rotas longas, se a cidade final em P_2 coincidir com a cidade inicial em P_1; (5) Repita os dois passos anteriores, até que mais nenhuma rota possa ser formada;	Obtenção do conjunto inicial Obtenção do conjunto inicial
(6) Localizar todas rotas que iniciem com a seqüência que codifica a cidade inicial; (7) Localizar todas as rotas que terminem com a seqüência da cidade final;	Peneira através da operação de separação por localização
(8) Localizar todas as seqüências que passam por todas as demais cidades;	Peneira através da operação de separação por identificação
(9) Verificar se existe alguma seqüência restante e localizar a menor seqüência, esta representa a menor rota entre a cidade de início e a cidade de destino.	Recuperação da solução através da operação de teste e aplicação da operação de separação por verificação

5.7 Problema do *Shortest Common Superstring*

O *shortest common superstring* foi provado ser um problema NP-Completo por Maier em [MAI77].

Problema do shortest common superstring

Entrada: Alfabeto finito Σ, conjunto finito R = $\{x_1, x_2,..., x_n\}$,n $\geq$ 1, de palavras de Σ* e um inteiro positivo K.

Pergunta: Existe uma palavra w $\in$ Σ* com comprimento |w| $\leq$ K, tal que cada palavra x $\in$ R é uma subseqüência de w, isto é, w = $w_0x_1w_1x_1x_2w_2...x_kw_k$, onde cada w_i $\in$ Σ* e x = $x_1x_2...x_k$?

Este problema consiste em determinar se existe uma "superpalavra" de comprimento menor ou igual a K, a qual contenha todas as palavras do conjunto de entrada como subpalavras.

A solução para o problema s*hortest common superstring* foi apresentada por Gloor, em [GLO98].

A seguir, serão apresentados os passos propostos por Gloor, sendo feito um paralelo com o método de Filtragem Seqüencial.

Passos propostos	**Método de Filtragem Seqüencial**
(1) Codificar todas as palavras $\{x_1, x_2,..., x_n\}$ do conjunto R em seqüências de DNA; (2) Gerar todas as possíveis seqüências de DNA, w, de comprimento: max$\{\|x_i\|, 1 \leq i \leq n\}$ e K;	Obtenção do conjunto inicial
(3) Sendo x_1 uma palavra pertencente a R. Para cada palavra gerada no passo anterior, selecione somente aquelas seqüências que contenham x_1 como uma subseqüência; (4) Do novo conjunto de palavras obtidas, selecione somente as palavras que contenham $x_2 \in$ R, como subseqüência; (5) Repita este passo para cada seqüências x_i em R, com $1 \leq i \leq n$;	Peneira através da operação de separação por localização
(6) Verificar se existir alguma seqüência w restante.	Recuperação da solução através da operação de teste por detecção

O método de Filtragem Seqüencial resolve problemas de decisão e problemas de localização, de forma satisfatória. Se um problema é de decisão, para obter sua solução, basta saber se o conjunto de soluções viáveis é vazio ou não, caso seja vazio, a resposta

para o problema é NÃO, caso contrário, é SIM. Se o problema for de localização, qualquer elemento do conjunto de soluções viáveis é uma solução para o problema, basta apenas recuperar uma delas. Já os problemas de otimização requerem uma diferenciação entre soluções viáveis, pois deseja-se a melhor (a ótima) entre todas soluções viáveis. Esta otimalidade é definida por uma função objetivo que deve ser maximizada ou minimizada; a mesma, geralmente, toma valores no conjunto dos números reais. A representação dos reais em fitas de DNA ainda não é um problema satisfatoriamente resolvido, o que dificulta a solução de problemas de otimização na computação com DNA.

Neste capítulo mostrou-se a usabilidade do método de Filtragem Seqüencial e, além disso, foi evidenciado como esta técnica é bastante usada.

6 CONCLUSÃO

Este trabalho apresenta os fundamentos da computação com DNA: como estão se desenvolvendo a programação e a codificação dos dados, tendo sempre presente a preocupação com a viabilidade de implementação das operações.

Basicamente são utilizadas três maneiras de se desenvolver um programa com DNA. O método de construção, que usa o raciocínio usual da programação convencional e, portanto, facilita o desenvolvimento de algoritmos, porque o projetista não precisa mudar sua forma de raciocínio. Por outro lado, a complexidade de implementação de suas operações, dificulta a utilização do método.

O método proposto por este trabalho, que culminou na proposta do método de Filtragem Seqüencial, tenta entender as leis biológicas e tirar proveito delas para resolver problemas. Está baseado nas reações em paralelo, que acontecem numa solução (tubo de teste), quando é executado um experimento. Baseado no fato deste paralelismo modelar uma computação não determinística, presente nos modelos teóricos da computação, como a máquina de Turing, e incapaz de ser copiada em sistemas reais pela exigência ilimitada de processadores, gera-se uma grande expectativa em volta da capacidade de computação nos sistemas de DNA. A demanda por sistemas que simulem, eficientemente, modelos não determinísticos é uma realidade presente. A programação com DNA não resolverá o problema “P = NP?”, mas pode ser uma alternativa eficiente (polinomial) para resolver problemas NP-Completos.

Este método, Filtragem Seqüencial, é bastante geral e no capítulo 5 mostrou-se que ele está sendo muito usado. O capítulo 4 é uma tentativa de tornar o método mais claro e facilitar seu uso.

O terceiro método, o de *splicing*, não deve ser rejeitado, pois ele é viável e pode ser muito útil na solução de problemas e, também, utiliza operações usuais da biologia molecular, portanto tem viabilidade clara e explora as potencialidades de representação e de manipulação biológica. O que parece uma desvantagem em relação ao método de Filtragem Seqüencial, neste momento, é a deficiência do método em tratar problemas combinatoriais, que são exatamente os que não tem um tratamento aceitável na computação convencional.

A programação teve seu início em máquinas abstratas, como a máquina de Turing, resolvendo problemas simples de palavras. A criação do computador proporcionou a solução de problemas mais complexos e essa evolução se deu gradualmente, em paralelo com o desenvolvimento de estruturas de dados mais complexas.

Nos primórdios da computação, os problemas que se apresentavam eram tão simples que estruturas de dados simples eram suficientes. A evolução na facilidade de programação aconteceu naturalmente, com a criação de novas estruturas. Em paralelo surgiram novos desafios de solução de problemas. O estudo de qualquer linguagem de

programação inicia com o uso de variáveis, vetores, matrizes, entre outras estruturas simples de dados e, à medida que os problemas vão se tornando mais complexos, outras estruturas são exigidas como, listas, grafos e objetos.

A programação com DNA, assim como a programação usual, requer o uso de estruturas de dados adequadas, mas a sua evolução não pode seguir os mesmos passos da programação digital, porque a exigência dos problemas a serem resolvidos (problemas intratáveis) está muito acima do disponível em termos de estruturas de dados. A computação com DNA está na fase em que é preciso desenvolver essas estruturas.

Considerando a classificação de problemas de Garey e Johnson [GAR79], pode-se dizer que a programação com DNA resolve de forma satisfatória problemas de decisão e de localização. Já as deficiências em representar os números reais e em tratar de problemas de otimização ainda são alguns dos desafios para esta forma de programação.

Este trabalho teve como objetivo traçar as linhas básicas de uma metodologia de programação, baseada nas operações de manipulação das fitas de DNA, sempre se preocupando com a viabilidade biológica.

Conclui-se que a técnica construtiva, usual em programação tradicional, não é a mais utilizada e a mais adequada para computação com DNA. Desenvolver uma metodologia que incorpora as técnicas mais utilizadas e guia o programador menos experiente.

No desenvolvimento deste trabalho, o grande aprendizado foi a forte dependência da codificação dos dados com a técnica de programação. Na tarefa de tornar a programação com DNA mais acessível ao programador iniciante, contribui-se com a primeira parte que é o “jeito de programar” com DNA, mas fica a sensação que esta é a fase mais fácil. A próxima etapa, que é o desenvolvimento de codificações padrão, está por ser feita.

REFERÊNCIAS

[ADL94] ADLEMAN, L. Molecular Computational of Solutions to combinatorial Problems. **Science**, [S.l.], n. 266, p. 1021-1024, 1994.

[ADL95] ADLEMAN, L. **On constructing a Molecular Computer**. [S.l.]: Department of Computer Science, University of Southern California, 1995.

[ALB94] ALBERTS, B. et al. **Molecular Biology of the Cell**. New York: Garland Publishing, 1994.

[AMO96] AMOS, M.; GIBBONS, A.; HODGSON, D. Error-resistant Implementation of DNA Computations. In: ANNUAL MEETING ON DNA BASED COMPUTERS, 2., 1996. **Proceedings…** [S.l.]: Princeton University, 1996. p. 87-101.

[AMO97] AMOS, M. **DNA computation**. 1997. PhD thesis. University of Warwick, UK.

[AMO98] AMOS, M.; GIBBONS, A.; DUNNE, P. **The complexity and viability of DNA computations**. England: University of Liverpool, 1998.

[BEI97] BEIGEL, R.; FU, B. On Molecular Approximation Algorithms for NP Optimization Problems. In: DIMACS MEETING ON DNA BASED COMPUTERS, 3., 1997. **Proceedings…** [S.l.: s.n.], 1997.

[BEI98a] BEIGEL, R.; FU, B. Solving Intractable Problems with DNA Computing. In: ANNUAL IEEE CONFERENCE ON COMPUTATIONAL COMPLEXITY, 13., 1998. **Proceedings…** [S.l.: s.n.], 1998. p. 154-169.

[BEI98b] BEIGEL, R.; FU, B. Molecular Computing, Bounded Nondeterminism, and Efficient Recursion. In: INTERNATIONAL COLLOQUIUM AUTOMATA, LANGUAGES AND PROGRAMMING, 24., 1998. **Proceedings…** [S.l.: s.n], 1998. p. 816-826.

[BLU96] BLUSTEIN, J. **Encoding Methods for DNA Computers**. Disponível em: <www.csd.uwo.ca/~jamie/.Refs/Courses/CS881/encoding.ps>. Acesso em: 25 mar. 2002.

[BON95] BONEH, D.; DUNWORTH, C.; LIPTON, R. **Breaking DES Using a Molecular Computer**. USA: Department of Computer Science, Princeton University, USA, 1995. (Technical Report CS TR489-95).

[BOV93] BOVET, D.; CRESCENZI, P. **Introduction to the Theory of Complexity**. New York: Prentice Hall, 1993.

[CER2002] CERVO, L. V. **Modelagem e Simulação de Algoritmo Paralelo baseados em Operações com DNA**. 2002. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[COM74] CONSTABLE R. L.; HUNT H. B.; SAHNI S. **On the Computational Complexity of Scheme Equivalence**. Ithaca, NY: Department of Computer Science, Cornell University, 1974. (Report n. 74-201).

[COO71] COOK, S. A. The Complexity of Theorem-Proving Procedures. In: ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, 3., 1971. **Proceedings…** Ohio: ACM, 1971. p. 151-158.

[DAS96] DASSEN, J. H. M. **Molecular Computation and Splicing Systems**. 1996. Msc Thesis. Leiden University, USA.

[DAS98] DASSEN, J. DNA computing: Promises, problems and perspective. **IEEE Potentials**, [S.l.], v.16, n.5, p. 27-28, 1998.

[DEA96] DEATON, R. et al. **Genetic search of reliable encodings for DNA based computation**. Disponível em: <www.csce.uark.edu/~rdeaton/dna/papers/gp-96.pdf>. Acesso em: jun. 2002.

[EIS89] EISENFELD, J.; LEVINE, D. L. Biomedical systems modeling and simulation. In: IMACS WORLD CONGRESS ON SCIENTIFIC COMPUTATION, 12., 1989. **Proceedings…** [S.l.:s.n.], 1989. p.18-22.

[FIT2000] FITZGIBBONS, B. **An Exploration of the DNA Computing Model**. Disponível em: <gatech.edu/~bradf/cs7001/dna_comp.ps>. Acesso em: nov. 2002.

[FU98a] FU, B.; BEIGEL, R. Length Bounded Molecular Computing. In: DIMACS WORKSHOP ON DNA BASED COMPUTERS, 4., 1998. **Proceedings…** Philadelphia:[s.n.], 1998

[FU98b] FU, B.; BEIGEL, R. An $\tilde{O}(2^n)$Volume Molecular Algorithm for Hamiltonian Path. **Biosystems**, [S.l.], v. 52., n. 1, p. 217-226, 1998.

[GAR79] GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability**: A Guide to the Theory of NP-Completeness. San Francisco: W. H. Freeman, 1979.

[GIF94] GIFFORD, D. K. On the path to Computation with DNA. **Science**, [S.l.], n. 266, p. 993-994, 1994.

[GRA98] GRAMB, T. et al. **Non-Standard Computation**: Molecular Computation – Cellular Automata – Evolutionary Algorithms – Quantum Computers. [S.l.]: Wiley-VCH, 1998.

[HAR95] HARTMANIS, J. On the weight of computations. In: EUROPEAN ASSOCIATION FOR THEORETICAL COMPUTER SCIENCE, 55., 1995. **Proceedings…** [S.l.: s.n.], 1995. p. 136-138.

[KAR72] KARP, R. M. Reducibility Among Combinatorial Problems. In: COMPLEXITY OF COMPUTER COMPUTATIONS, 1972. **Proceedings…** New York: Plenum Press, 1972.

[LIP94] LIPTON, R. J. **Speeding Up Computations via Molecular Biology**. 1994. Princeton University, New Jersey.

[LIP95] LIPTON, R. J. DNA solution of hard computational problems. **Science**, [S.l.], n. 288, p. 542-545, 1995.

[LIU96] LIU, Q. et al. **A Surface-Basead Approach to DNA Computation**. Madison: University of Wisconsin, 1996. (WI 57306).

[MAI77] MAIER, D.; STORER, J. A. **A note on the complexity of the superstring problem**. NJ: Computer Science Laboratory, Princeton University, 1977. (Report n. 233).

[MEN97] MENEZES, P. F. B. **Linguagens formais e autômatos**. Porto Alegre: Instituto de Informática da UFRGS, 1997.

[NAR97] NARAYANAN, A. **Representing arc labels in DNA algorithms**. Exeter: University of Exeter, 1997. (Report TR360).

[NAR98] NARAYANAN, A.; ZORBALAS, S. DNA algorithms for computating shortest paths. In: ANNUAL CONFERENCE GENETIC PROGRAMMING, 3., 1998. **Proceedings…** Exeter: University of Exeter, 1998.

[PAU98] PAUN, G.; ROZEMBERG, G.; SALOMAA, A. **DNA computing**: new computing paradigms. Berlin: Springer, 1998.

[REI95] REIF, J. Parallel Molecular Computation. In: ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 7., 1995. **Proceedings…** Santa Barbara: ACM, 1995.

[REI98] REIF, J. H. Parallel Biomolecular Computation: Models and Simulations. **Algorithnica**, New York, v. 25, n. 2, p. 142-175, 1998.

[ROO96] ROOB, D.; WAGNER, K. On the power of DNA-Computing. **Information and Computation**, San Diego, v. 131, p. 95-109, 1996.

[SAL73] SALOMAA, A. **Formal Languages**. New York: Academic Press, 1973.

[TOS2000] TOSCANI, L. V.; VELOSO, P. A. S. **Complexidade de Algoritmos**. Porto Alegre, Instituto de Informática da UFRGS: Sagra Luzzatto, 2000.

[WAS2000] WASIEWICZ, P. et al. Adding Numbers with DNA. In: IEEE INTERNATIONAL CONFERENCE ON SYSTEMS, 2000. **Proceedings...** Nashville: IEEE, 2000.

ANEXO A CODIFICANDO DNA

A codificação de estruturas de dados, em seqüências de DNA, ainda é uma limitação para a programação com DNA. Desenvolver uma forma padrão para representar diferentes estruturas, trará grandes benefícios à computação com DNA.

Neste capítulo serão apresentadas algumas formas de codificar seqüências de DNA para representar informações; diferentes autores propuseram formas de representá-las e a seguir, serão descritas estas formas de codificação para grafos e números inteiros não negativos.

Para definir um sequenciamento estável de nucleotídeos para formar as seqüências que codificação dados de um problema deve-se observar algumas restrições como tamanho da seqüência e os padrões de combinações entre nucleotídeos. As seqüências devem ser formadas por, no mínimo 20 nucleotídeos. Os nucleotídeos G e C devem, na medida do possível, serem predominantes na seqüência, pois estes apresentam uma ligação e 3 pontes de hidrogênio, enquanto que os nucleotídeos A e T possuem apenas a ligação de 2 pontes de hidrogênio.

A.1 Codificando Grafos

Entre os problemas solucionados através da computação com DNA, muitos são problemas que envolvem estruturas de grafos. Adleman [ADL 94] e Narayanan [NAR 98] propuseram formas de codificar estas estruturas. A seguir serão apresentadas as formas de codificação utilizadas por cada autor.

Para fins didáticos, irá se definir estas seqüências como tendo duas partes, caso a seqüência seja formada por 20 nucleotídeos, os 10 primeiros nucleotídeos da seqüência serão denominados de “metade inicial” e os 10 nucleotídeos restantes serão denominados “metade final”. (FIGURA A.1)

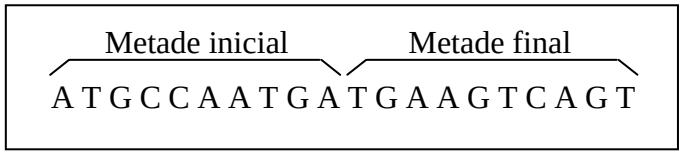


Figura A.1: Seqüência representativa (vértice ou aresta)

A.1.1 Codificação de Adleman

Adleman codificou um grafo direcionado e não rotulado como seqüências sintetizadas (vértices e arestas), representando cada elemento por uma seqüência de nucleotídeos. A codificação dos vértices e arestas dá-se da seguinte forma:

- **Codificação dos vértices**: cada vértice (v) do grafo é sintetizado como uma seqüência específica de nucleotídeos aleatórios e de direção 5' – 3'.
- **Codificação das aretas**: cada aresta ($a_{i \to k}$), que parte de um v_i e chega em um v_k, é representada por uma seqüência de nucleotídeos, onde a metade inicial da aresta é o complemento (complementaridade de Watson-Crick) da metade final do vértice v_i e a metade final da aresta é o complemento da metade inicial do vértice v_k; esta seqüência terá a direção 3' – 5'. A FIGURA A.2 demonstra a construção de uma aresta.

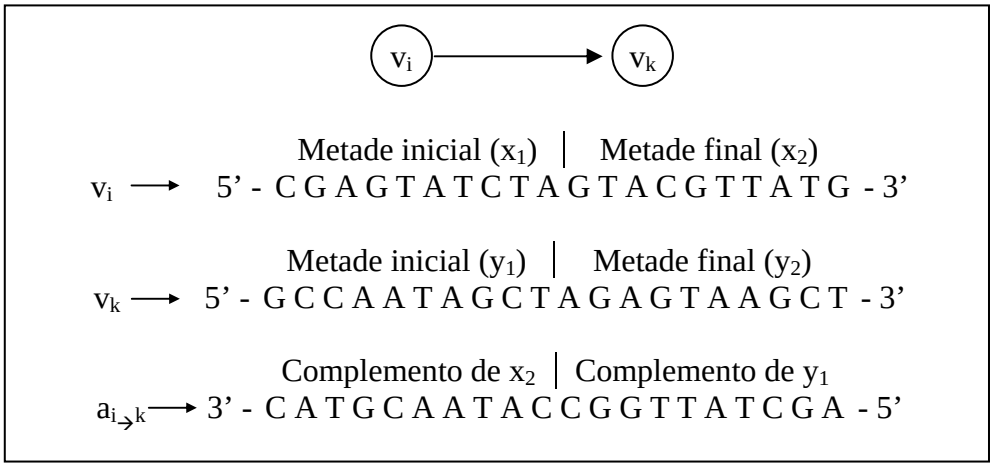


Figura A.2: Codificação das arestas

Uma exemplificação mais detalhada da codificação completa de um grafo, foi apresentada no capítulo 4. Essa codificação representa bem o grafo não valorado. Na seção seguinte é apresentada uma codificação para grafos com arcos valorados.

A.1.2 Codificações de Narayanan

Narayanan, utilizando a proposta de codificação de Adleman, apresentou duas formas de codificar grafos rotulados, com valores naturais.

A.1.2.1 Primeira Proposta

A codificação dos vértices e arcos de um grafo rotulado (FIGURA A.3) dá-se da seguinte forma:

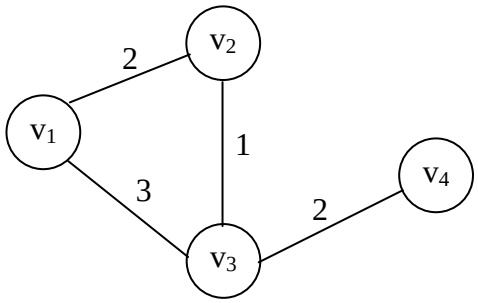


Figura A.3: Grafo rotulado

- **Codificação dos vértices**: cada vértice (*v*) do grafo é representado como uma seqüência específica de nucleotídeos aleatórios. (FIGURA A.4)

Vértices	Seqüências
v_1	AAAA
v_2	GGGG
v_3	CCCC
v_4	TTTT

Para melhor compreensão do exemplo, neste caso, são utilizados códigos claramente distinguíveis para representar os vértices do grafo, mas esta forma de codificação não é aconselhada para computação com DNA.

Figura A.4: Codificação dos vértices

- **Codificação dos arcos**: cada arco (e_{i-k}), com valor *n*, é representado pelas seqüências de nucleotídeos, da seguinte forma (Tabela A.1):
  - Para $i \rightarrow k$, onde *i* é o vértice de partida e *k* o vértice de chegada, sintetize *n* ocorrências da seqüência de v_i com uma ocorrência da seqüência de v_k, com direção 3' – 5'.
  - Para cada seqüência, construída anteriormente, sintetize a seqüência complementar que garanta a união de cada arco. Esta seqüência é formada pelo complemento da metade final do vértice de partida com o complemento da metade inicial do vértice de chegada, tendo a direção 5' – 3'.

Tabela A.1: Codificação dos arcos.

v_i	v_k	*N*	*Seqüência*
v_1	v_3	3	T T G G A A A A | A A A A | A A A A | C C C C V_1 v_1 v_1 v_3
v_3	v_1	3	G G T T C C C C | C C C C | C C C C | A A A A v_3 v_3 v_3 v_1
v_1	v_2	2	T T C C A A A A | A A A A | G G G G v_1 v_1 v_2
v_2	v_1	2	C C T T G G G G | G G G G | A A A A v_2 v_2 v_1
v_2	v_3	1	C C G G G G G G | C C C C v_2 v_3
v_3	v_2	1	G G C C C C C C | G G G G v_3 v_2
v_3	v_4	2	G G A A C C C C | C C C C | T T T T v_3 V_3 v_4
v_4	v_3	2	A A G G T T T T | T T T T | C C C C v_4 V_4 v_3

Para codificar os caminhos do grafo, as seqüências dos arcos do caminho são concatenadas da seguinte forma: dois arcos e_1 e e_2 formam um caminho se, e somente se, o vértice final do arco e_1 for igual ao vértice inicial do arco e_2. Sendo assim, devem ser removidos os nucleotídeos da metade final do vértice de chegada de e_1 e os nucleotídeos da metade inicial do vértice partida de e_2 (FIGURA A.5).

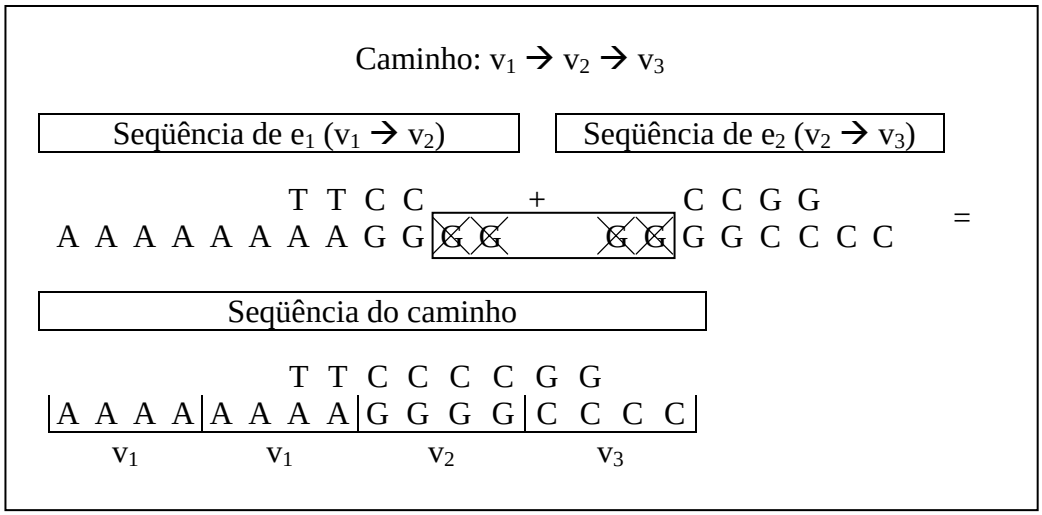


Figura A.5: Codificação dos caminhos

Esta forma de codificação apresenta alguns problemas, entre eles estão: a enorme massa de DNA requerida para representar as informações (valores altos) do grafo. Hartmanis [HAR 95] afirmou que a massa de DNA necessária para representar um grafo rotulado, seria suficientemente grande para tornar esta forma de codificação inviável. Outro problema é a degradação das seqüências. Como nesta forma de codificação faz-se necessária a remoção de nucleotídeos de locais específicos da seqüência, há grandes chances de, ao final da computação, o resultado não ser correto [NAR 98].

A.1.2.2 Segunda Proposta

Para tentar evitar os problemas descritos na forma anterior, Narayanan propôs uma segunda forma de codificação. Nesta forma os valores dos arcos entre os vértices devem ser ordenados, em ordem crescente e não repetida, em D (seqüência dos valores dos arcos). Para exemplificar esta forma de codificação, será usado o grafo apresentado anteriormente na FIGURA A.3, tendo o vértice v_1 como sendo o vértice inicial .

A codificação dos vértices, rótulos e arcos dá-se da seguinte forma:

- **Codificação dos vértices**: cada vértice (*v*) do grafo é representado por uma seqüência específica de nucleotídeos aleatórios, de comprimento fixo. (tabela A.2)

Tabela A.2: Codificação dos vértices.

Vértices	Seqüências
v_1	TATT
v_2	GCGG
v_3	GTCG
v_4	AGAA

- **Codificação dos rótulos**: cada rótulo (valor) pertencente a D é codificado como uma seqüência específica de nucleotídeos aleatórios, de comprimento variável. O comprimento (C) de cada seqüência de DNA é proporcional à localização do valor do arco, correspondente em D. O comprimento de cada seqüência aumenta em um fator constante K, isto é, K: $\forall$ C $\in$ 1,...., |D| $\Rightarrow$ C = d * K. (Por exemplo, se K = 2 e D = <2, 5, 9>, então 2 é representado por uma fita de comprimento de 2 bases, 5 por uma fita de comprimento de 4 bases, e 9 por uma fita de comprimento de 6 bases). Para o grafo da FIGURA A.3, com D = <1, 2, 3> e K = 3, é dada a representação pela tabela A.3.

Tabela A.3: Codificando os valores dos arcos do grafo da FIGURA A.3.

Distâncias	Seqüências
D{1}	AAT
D{2}	TCGATT
D{3}	ACGACGTTAATT

- **Codificação dos arcos**: cada arco (e_{i-k}) é representado por uma seqüência de nucleotídeos de tamanho variado, seguindo as seguintes orientações:
  - Se o vértice v_i (vértice de partida) do arco for o vértice inicial do caminho, a seqüência representativa deste arco é formada pela seqüência do vértice *i*, seguida pela seqüência do rótulo do arco e_{i-k} e seguida da metade final da seqüência do vértice *k* (FIGURA A.6a).
  - Para os demais arcos (e_{i-k}), as seqüências representativas são formadas pela metade inicial da seqüência de *i*, seguida pela seqüência do rótulo do arco e_{i-k} e seguida da metade final da seqüência de *k* (FIGURA A.6b).

(a)

Arco (e_{A-B})

v_A	Valor do arco v_{A-B}	Metade final (v_B)	Seqüência e_{A-B}
TATT +	TCGATT +	GG =	TATTTCGATTGG

Arco (e_{A-C})

v_A	Valor do arco v_{A-C}	Metade final (v_C)	Seqüência e_{A-C}
TATT +	ACGACGTTAATT +	CG =	TATTACGACGTTAATTCG

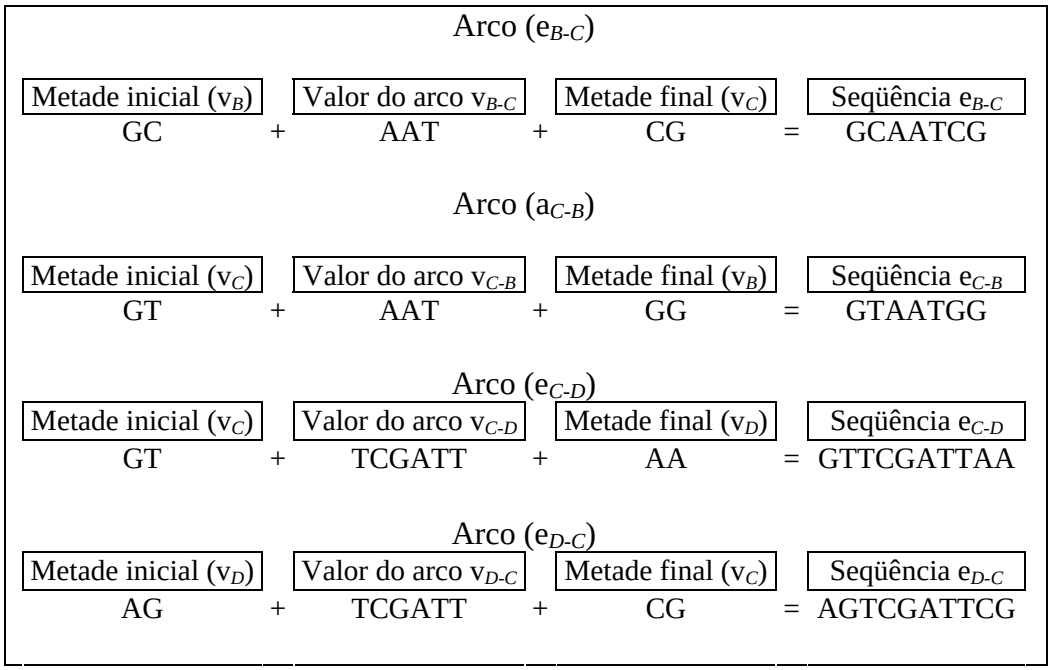


Figura A.6: Codificação dos arcos

Esta forma de codificação é eficiente somente para grafos que possuem os arcos com valores que crescem de forma constante e seqüencial, por exemplo, 1, 2, 3 ou 5, 10, 15, 20. Para os demais grafos valorados, representar os valores utilizando um fator K, não garante que o resultado esteja correto. O ideal seria que as seqüências, que representam os caminhos, aumentassem proporcionalmente ao tamanho das distâncias, ou seja, para um D=<1, 50, 70> ao invés de ter-se seqüências como D{50} com 2*K nucleotídeos, ter-se-ia seqüências com 50*K nucleotídeos, o que garante a integridade da proporção entre as distâncias. Porém, esta solução não é viável, pois a massa de DNA necessária para representar tais seqüências seria muito grande [HAR 95].

A.1.3 Paralelo entre as codificações de Adleman e Narayanan

Para grafos não valorados, a forma de codificação de Adleman apresenta uma boa forma de representação, entretanto as tentativas de representar grafos valorados estão longe de serem satisfatórias, até mesmo porque existe a necessidade de uma boa representação dos números (valores).

A codificação de caminhos em um grafo definida por Adleman, é uma ligação típica de montagem de seqüências (usado por crianças montando estruturas com *Lego*/*Playmobil* - FIGURA A.7). Note, que a técnica usada é a de “grudar” subseqüências complementares (arestas) para unir duas peças (vértices), sendo esta uma técnica natural e bem promissora, no entanto agregar mais informações, como associar valores às arestas, não é um caso resolvido. Espera-se que uma representação de números naturais ajude a agregar tais informações.

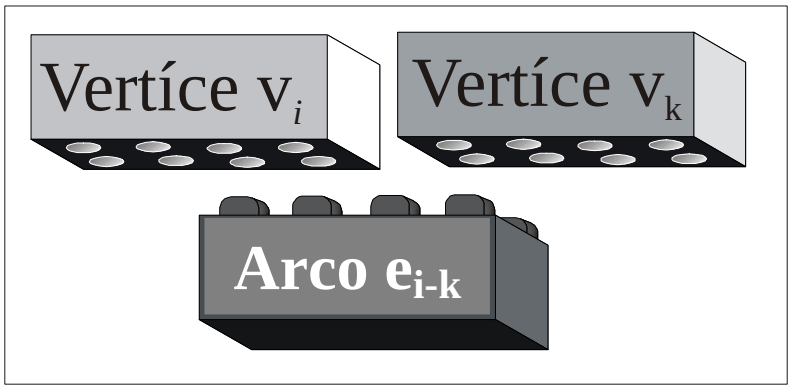


Figura A.7: Estruturas de *Lego*

A.2 Codificação de Números

Guarnieri, Fliss e Bancroft [GAU 96] propuseram uma forma de codificar números inteiros e não negativos de 2 dígitos. Já Wasiewicz e Mulawka propuseram, em [WAS2000], uma forma de codificar quaisquer números inteiros e não negativos, em seqüências de DNA. Em ambos casos, para codificar os números, é usada a representação do sistema binário.

A.2.1 Sistema Binário

O sistema binário é baseado em dois algarismos (*bits*): 0 (desligado) e 1 (ligado). Cada número do sistema binário é representado pela concatenação de seus algarismos, por exemplo 1101. A cada algarismo do número, é atribuída uma posição que inicia em zero e aumenta da direita para a esquerda (FIGURA A.8).

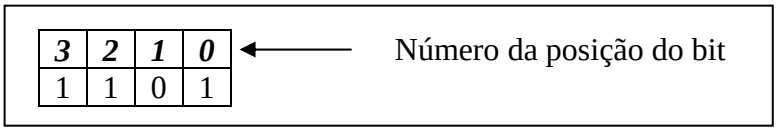
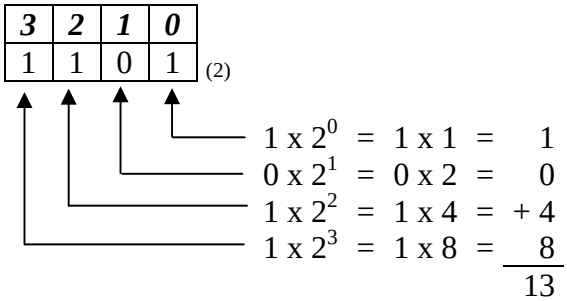


Figura A.8: Representação binária

O valor numérico de um número binário no sistema decimal, dá-se da seguinte forma:



A.2.2 Codificação de Guarnieri, Fliss e Bancroft

Na codificação de Guarnieri, Fliss e Bancroft dois números binários inteiros e não negativos de 2 dígitos são codificados. A codificação dos números binários é construída a partir de uma codificação inicial padrão, onde cada dígito é representado pela concatenação de algumas das subseqüências ($0_{(0)}$, $0_{(1)}$, $1_{(0)}$, $1_{(1)}$, $1_{(2)}$, $X_{(0)}$, $X_{(1,0)}$, $X_{(1)}$, $X_{(2,1)}$, $Y_{(0)}$, $Y_{(1,0)}$, $Y_{(1)}$, $Y_{(2,1)}$, $C_{(1)}$ e seus complementos, por exemplo $\overline{X}_{(0)}$ e a subseqüência complementar de $X_{(0)}$), formando uma fita simples única e não complementar (FIGURA A.9).

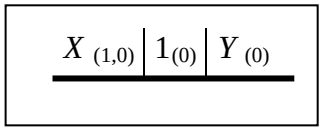


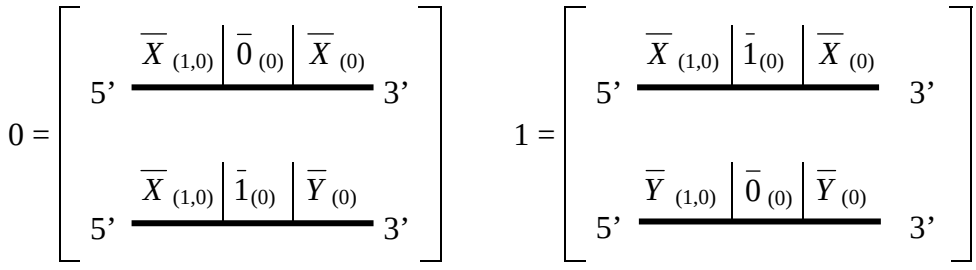
Figura A.9: Seqüência exemplo

Na FIGURA A.8 as subseqüências X e Y representam o operador de posição e a subseqüência 1 (podendo também ser a 0) indica o valor do dígito. Os números entre parênteses, quando únicos, representam a posição do dígito e quando em par, a transição da posição, por exemplo, (1,0) representa a transição da posição 0 para a posição 1, neste caso, o operador é denominado operador de transição da posição. Como o propósito desta codificação é realizar a soma binária entre os números, foi necessário definir uma subseqüência $C_{(1)}$ que representa o bit carregador.

Cada dígito tem uma codificação específica dependendo da posição que este se encontra. As codificações serão as seguintes:

Para os dígitos na posição 0, as seqüências têm o seguinte formato:

- **Operando 1**: Cada dígito é codificado por duas fitas distintas tendo a direção 5'- 3'. Cada fita é formada pelo operador de transição da posição ($\overline{X}_{(1,0)}$ ou $\overline{Y}_{(1,0)}$), seguido do elemento representativo do valor do dígito na posição 0 ($\overline{0}_{(0)}$ ou $\overline{1}_{(0)}$) e o operador de posição ($\overline{X}_{(0)}$ ou $\overline{Y}_{(0)}$). A codificação dos números 0 e 1 é demonstrada a seguir:



- **Operando 2**: Cada dígito é codificado por uma fita específica de direção 3’ – 5’, podendo ser $X_{(0)}$ ou $Y_{(0)}$ se o dígito for 0 ou 1, respectivamente. Esta fita servirá como um *primer*, que desencadeará a reação para efetuar a soma. A codificação dos números 0 e 1 é demonstrada a seguir:

$$0 = \left[\; 3' \; \underline{\qquad X_{(0)} \qquad} \; 5' \; \right] \qquad 1 = \left[\; 3' \; \underline{\qquad Y_{(0)} \qquad} \; 5' \; \right]$$

Para os dígitos na posição 1, as seqüências têm o seguinte formato:

- **Operando 1**: Cada dígito é representado por três fitas distintas, tendo a direção 5’- 3’. Cada fita é formada pelo operador de transição da posição ($\overline{X}_{(2,1)}$ ou $\overline{Y}_{(2,1)}$), seguido do elemento representativo do valor do dígito na posição 0 ($\overline{0}_{(1)}$ ou $\overline{1}_{(1)}$) e o operador de posição ($\overline{X}_{(1)}$ ou $\overline{Y}_{(1)}$) ou o carregador ($\overline{C}_{(1)}$). A codificação dos números 0 e 1 é demonstrada a seguir:

$$0 = \left[\begin{array}{l} 5' \; \underline{\qquad \overline{X}_{(2,1)} \mid \overline{0}_{(1)} \mid \overline{X}_{(1)} \qquad} \; 3' \\ 5' \; \underline{\qquad \overline{X}_{(2,1)} \mid \overline{1}_{(1)} \mid \overline{Y}_{(1)} \qquad} \; 3' \\ 5' \; \underline{\qquad \overline{Y}_{(2,1)} \mid \overline{0}_{(1)} \mid \overline{C}_{(1)} \qquad} \; 3' \end{array} \right] \qquad 1 = \left[\begin{array}{l} 5' \; \underline{\qquad \overline{X}_{(2,1)} \mid \overline{1}_{(1)} \mid \overline{X}_{(1)} \qquad} \; 3' \\ 5' \; \underline{\qquad \overline{Y}_{(2,1)} \mid \overline{0}_{(1)} \mid \overline{Y}_{(1)} \qquad} \; 3' \\ 5' \; \underline{\qquad \overline{Y}_{(2,1)} \mid \overline{1}_{(1)} \mid \overline{C}_{(1)} \qquad} \; 3' \end{array} \right]$$

- **Operando 2**: Cada dígito é codificado por duas fitas distintas de direção 5’ – 3’ e cada uma contém o operador de posição ($\overline{X}_{(1)}$ ou $\overline{Y}_{(1)}$) ou o carregador ($\overline{C}_{(1)}$), seguido do operador de transição da posição ($\overline{X}_{(1,0)}$ ou $\overline{Y}_{(1,0)}$). A codificação dos números 0 e 1 é demonstrada a seguir:

$$0 = \left[\begin{array}{l} 5' \; \underline{\qquad \overline{X}_{(1)} \mid \overline{X}_{(1,0)} \qquad} \; 3' \\ 5' \; \underline{\qquad \overline{Y}_{(1)} \mid \overline{Y}_{(1,0)} \qquad} \; 3' \end{array} \right] \qquad 1 = \left[\begin{array}{l} 5' \; \underline{\qquad \overline{Y}_{(1)} \mid \overline{X}_{(1,0)} \qquad} \; 3' \\ 5' \; \underline{\qquad \overline{C}_{(1)} \mid \overline{Y}_{(1,0)} \qquad} \; 3' \end{array} \right]$$

Além de codificar as seqüências de cada dígito, também é criada uma seqüência para representar o bit carregador-final, que será ligado à seqüência resultado, caso a soma

dos dígitos da posição 1 resultem um bit carregador. Esta seqüência será codificada da seguinte forma:

$$\text{Carregador-final} = \left[5' \; \overline{\bar{1}_{(2)} \mid \bar{Y}_{(2,1)}} \; 3' \right]$$

A.2.3 Codificação de Wasiewicz e Mulawka

Na codificação de Wasiewicz e Mulawka, cada posição de bit é codificada por uma seqüência específica, que terá o tamanho suficiente para somente se ligar a sua seqüência complementar. Dependendo do problema que se quer solucionar, utilizando esta codificação, é necessário codificar as seqüências complementares de cada posição de bit. Somente os valores 1 são codificados, tendo um código para cada posição do valor 1; por exemplo na tabela A.4.

Tabela A.4: Representação binária das seqüências de DNA.

Posição de Bit	Seqüência representativa	Seqüência complementar
0	ATGCTAG	TACGATG
1	TTGACTT	AACTGAA
2	GCGCCCG	CGCGGGC
3	AGTATTA	TCATAAT
4	CCAGTAG	GGTCATG
5	CACACAT	GTGTGTA

Após codificar cada posição de bit como uma seqüência específica de nucleotídeos, é formada a representação do número binário através da concatenação das seqüências de posições que contenham o bit 1. Por exemplo, o número 13 possui a representação binária 1101 e sua codificação será a seguinte:

Posição de bit	3	2	1	0
Número binário	1	1	0	1
Seqüência representativa	AGTATTA	GCGCCCG		ATGCTAG

A seqüência que representa o número 13 é AGTATTGCGCCCGATGCTAG.

A.2.4 Paralelo entre as duas codificações

A codificação de Guarnieri, Fliss e Bancroft é projetada com a finalidade de implementar a soma de números binários (dois números de dois dígitos), sendo evidenciada a viabilidade biologia. Já, a codificação de Wasiewicz e Mulawka pretende ser uma codificação geral, onde o algoritmo da soma independe do número de dígitos dos operandos, entretanto a execução do experimento biológico não foi citada.

A.3 Observações Finais

As codificações apresentadas em A.1 e A.2 são somente exemplos para ilustrar o desenvolvimento e as dificuldades de representação de dados usando DNA.

A falta de padrão e as restrições biológicas continuam sendo uma barreira no desenvolvimento de algoritmos moleculares.

Convém notar que, apesar das codificações apresentadas em A.2 representarem números naturais, estas não acrescentam, de maneira direta, um avanço na representação de grafos valorados, uma vez que não é possível utilizar nenhuma dessas codificações para agregar valores aos arcos de um grafo.