O problema do DNA Alienígena em questão: uma proposta de solução com lista encadeada

Mateus Freitas Charloto*
Escola Politécnica da PUCRS

19 de setembro de 2023

Resumo

Este artigo tem como objetivo prover uma solução algorítmica para o problema do DNA Alienígena apresentado na disciplina de Algoritmos e Estruturas de Dados II. A solução projetada prevê o uso de uma estrutura de dados baseada em lista de encadeamento simples. Os resultados obtidos mostraram que a implementação é capaz de resolver plenamente os casos de teste fornecidos em tempo finito. Além disso, os resultados indicaram um desempenho superior da estrutura de dados implementada em relação às estruturas fornecidas pela API do Java. Ainda que haja otimizações a serem realizadas, e há a menção às tais, é possível afirmar que a solução não foi à toa, podendo servir como rudimento para futuras implementações mais complexas, como as baseadas em encadeamento duplo.

Introdução

Na disciplina de Algoritmos e Estruturas de Dados II, foi apresentado o problema do DNA Alienígena, descrito a seguir. Após a queda de uma nave espacial e fuga dos alienígenas que a pilotavam, cientistas fizeram a análise do material genético encontrado no objeto, a partir da qual descobriram a composição do DNA destes seres: 3 bases chamadas “D”, “N” e “A”. Além disso, descobriram que, quando duas bases diferentes estão dispostas uma do lado da outra, estas se fundem e geram a terceira base, a qual é inserida ao final da cadeia. Essa fusão ocorre sempre na dupla mais à esquerda do DNA. Um exemplo desse processo é dado a seguir:

* mateus.charloto@edu.pucrs.br

$$DNADNA \rightarrow ADNAA \rightarrow NAAN \rightarrow AND \rightarrow DD$$

Assim, a sequência DNADNA converte-se em DD, o que marca o fim do processo, já que bases iguais não apresentam o comportamento observado.

Com um número pequeno de bases, é possível gerar a sequência resultante sem grande dificuldade; ocorre que, porém, as cadeias de DNA tendem a ser largamente extensas, o que torna inviável o trabalho manual. A fim de contornar esta limitação, busca-se, neste artigo, prover uma solução algorítmica que seja capaz de, dada uma cadeia qualquer, retornar em tempo finito as bases resultantes ao fim do processo de fusão.

Design e implementação do algoritmo

O ponto de partida para a solução do problema foi a escolha de qual estrutura de dados seria mais adequada para armazenar o DNA. Em virtude de seu comportamento dinâmico, ponderou-se que, idealmente, seria necessário fazer uso de uma estrutura cujas operações de acesso, inserção e remoção fossem eficientes de forma conjunta. Isso tornava inviável o uso de forma "pura" das estruturas mais comuns: se implementada uma lista com arranjo, por exemplo, ter-se-ia acesso eficiente, porém as operações de inserção e remoção impactariam negativamente a performance do algoritmo; se implementada uma lista encadeada, ter-se-ia o problema oposto dos arranjos, isto é, operação de acesso ineficiente. Diante disso, surgiu a necessidade de implementação de uma estrutura personalizada para a solução do problema.

Antes de partir para a implementação, porém, fez-se uma análise sobre a forma do DNA, ao que se descobriu que as sequências tendem a apresentar frequentemente repetições contíguas, especialmente em seu início. Levando em conta que somente bases diferentes realizam fusão, tornou-se evidente não ser necessário ler a sequência inteira de bases repetidas de modo contíguo na lista, pois somente a última é significativa, isto é, ela quem de fato se combina com a base seguinte, que é distinta. Dessa maneira, chegou-se à conclusão de que as bases da sequência de repetição se mostram irrelevantes até que a última base desta sequência sofra fusão (e assim continuamente). Com isso em vista, inferiu-se que essas sequências poderiam entrar em uma espécie de modo de espera, sendo processadas somente quando de fato necessário.

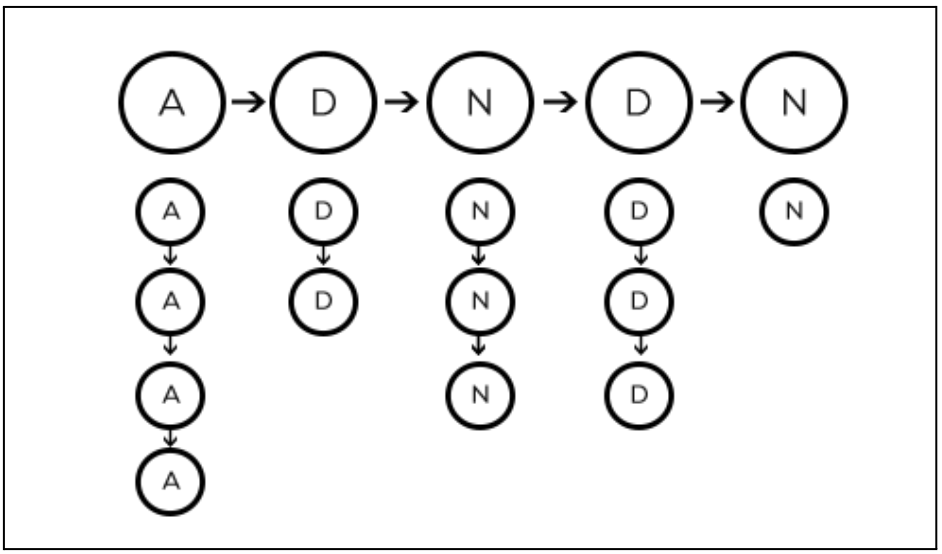
Para implementar essa solução, pensou-se na possibilidade de haver duas listas: uma principal, que armazenaria a primeira ocorrência da sequência repetida, e uma interna associada a esta primeira ocorrência, que armazenaria o restante dos elementos repetidos. Assim, a lista principal teria a propriedade de que seus elementos imediatamente contíguos deveriam ser distintos (no momento em que os caracteres são adicionados à lista), ao passo que a interna somente armazenaria repetições. Tendo sido projetado um acesso mais otimizado ao evitar ler desnecessariamente sequências repetidas, o próximo passo, então, foi garantir que as operações de inserção e remoção fossem eficientes em ambas as listas. A estrutura base que pareceu atender melhor a esta condição foi a lista encadeada, em que, ao ser personalizada, seus nodos poderiam, além de armazenar a base, ter como atributo uma referência para uma lista encadeada contendo o restante das sequências repetidas. Além disso, a lista poderia ter um atributo "tail" para guardar a referência para o último nodo, de modo que para adicionar uma base no final bastaria fazer a referência apontar para ela, garantindo, assim, uma inserção O(1). Dessa forma, fez-se a implementação desta classe, cuja complexidade das operações e ilustração encontram-se no quadro e imagem que se seguem, respectivamente:

Quadro 1 - Complexidade das operações da estrutura implementada

	Lista principal	**Lista interna**
Acesso	O(N)	O(1) (sempre, apenas o primeiro elemento é acessado)
Inserção (ao final)	O(1)	O(1) (sempre, a inserção ocorre no início)
Remoção	O(N)	O(1) (sempre, a remoção se dá no primeiro elemento)

Fonte: Autor

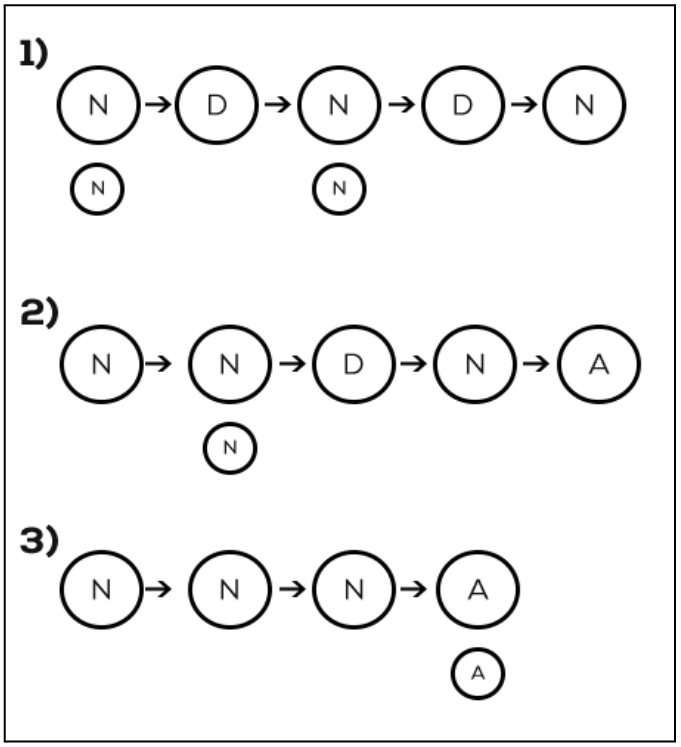
Imagem 1 - Ilustração da estrutura implementada



Fonte: Autor.

O próximo passo, então, foi projetar um algoritmo que, dada uma sequência de DNA, retornasse a cadeia resultante após ocorridas as fusões entre as bases distintas. Partindo da posição inicial da sequência *i=0*, a próxima etapa seria verificar se a base contígua, *i+1*, é distinta ou igual a ela. No caso de ser igual, deve-se partir para a próxima base *i+1* e realizar novamente a comparação com a base contígua *i+1+1*, e assim sucessivamente. No entanto, caso as bases sejam distintas, deve-se verificar qual base é gerada pela combinação destas e inseri-la ao final da cadeia. Após isso, poder-se-ia simplesmente retornar para o início da sequência para realizar as comparações, porém é preciso considerar um ponto importante. Embora no momento em que as bases são inseridas na lista (parte de pré-processamento) se tenha a garantia de que elementos contíguos são diferentes, à medida que ocorrem fusões, perde-se essa garantia. Isso fica evidente no caso a seguir, em que *1)* é a sequência recém-processada:

Imagem 2 - Caso especial



Fonte: Autor.

Quando, no caso *3)*, a última base "N" for comparada com a base "A", gera-se uma nova base "D"; intuitivamente se voltaria para o início da lista. Entretanto, existem casos para os quais isso não é necessário: ao haver uma sequência de repetições contíguas, e sabendo que bases iguais não se combinam entre si, quando houver uma fusão, é possível iniciar a próxima comparação a partir da base anterior à última da sequência de repetições. A exceção a essa regra é o caso de não haver uma sequência de repetições contíguas, caso em que, obrigatoriamente, deve-se realizar as comparações a partir da primeira base. Dessa forma, ganha-se em performance algorítmica, já que comparações que não resultam em bases novas são reduzidas de forma significativa. É importante salientar, todavia, que isso não significa um acesso O(1) às bases. Em uma lista encadeada, normalmente para se chegar em um dado elemento, é necessário percorrer desde o primeiro elemento até que se chegue nele de fato. Apesar disso, este não é exatamente o caso aqui, uma vez que, pela propriedade já explicada da lista principal, ela não apresenta o tamanho igual ao número total caracteres, e a tendência é que, quanto maior o número de repetições contíguas, menor essa lista seja. Nos casos de testes que serão comentados nas próximas seções, por exemplo, ela chegou a ter tamanho igual a aproximadamente 33% do número total de caracteres.

A seguir, está a implementação em pseudocódigo do algoritmo acima descrito, sendo *lista* uma instância da classe personalizada criada:

```
1.  i ←0
2.  enquanto o tamanho da lista>1, faça:
3.    caracter1 ← caracter da posição i da lista
4.    caracter2 ← caracter da posição i+1 da lista
5.    se caracter1 != caracter2, então:
6.      adicione ao final da lista a base gerada a partir da fusão do caracter1 com o caracter2
7.      remova da lista o caracter1
8.      remova da lista o caracter2
9.      se i>0, então:
10.       i ← i - 2
11.     do contrário:
12.       i ← i - 1
13. i ← i + 1 e volta para a linha 2 (associado ao loop)
```

É importante mencionar que o *i* é decrementado em duas unidades porque, ao verificar a condição e executar o bloco, ele é incrementado automaticamente em uma unidade pelo laço. Sendo assim, como se deseja acessar o elemento imediatamente à esquerda da repetição, é preciso fazer este decremento; raciocínio semelhante vale para i<0: faz-se o decremento para acessar o elemento da posição 0, e não 1.

Resultados obtidos

O algoritmo modelado na seção anterior foi implementado na linguagem de programação Java[1]. Foram executados oito casos de testes[2], tendo o primeiro 10 caracteres e o último 30 milhões.

[1] O código está disponível na íntegra no link a seguir: *https://github.com/mateusfch*.
[2] A máquina em que foram realizados os testes dispõe de 16GB de RAM, e o sistema operacional é Ubuntu versão 23.04.

Para se aproximar de um número mais fidedigno, cada teste foi executado 5 vezes e, a partir disso, fez-se a média simples dos valores, a partir da qual obteve-se como consta no quadro:

Quadro 2 - Resultado da execução com os casos de teste

Número de caracteres	Tempo médio de execução (em segundos)	Tamanho da lista principal em relação ao número total de caracteres dos casos (%)
10	0,00358187	70,00
10^2	0,00487374	64,00
10^3	0,00861098	50,70
10^4	0,01743592	47,57
10^5	0,06092066	39,91
10^6	0,42538588	33,34
10^7	6,26346034	33,34
3.10^7	36,24624045	33,34

Fonte: Autor.

O resultado obtido em cada teste foi atestado como verdadeiro com base no gabarito disponibilizado pelo professor da disciplina de Algoritmos e Estruturas de Dados II.

A fim de comparação, os casos de teste também foram executados a partir de uma lista de encadeamento simples e outra com arranjo, ambas “puras”, isto é, da forma como são fornecidas pela API do Java. O mesmo algoritmo citado na seção anterior foi empregado. Abaixo estão os resultados em comparação com a implementação personalizada:

Quadro 3 - Resultado da execução com os casos de teste em comparação com as estruturas da API do Java

Número de caracteres	Tempo médio de execução da classe personalizada (em segundos)	Tempo médio de execução da classe Lista Encadeada da API do Java (em segundos)	Tempo médio de execução da classe Lista com Arranjo da API do Java (em segundos)
10	0,00358187	0,00357690	0,003304306
10^2	0,00487374	0,00471396	0,004141542

Número de caracteres	Tempo médio de execução da classe personalizada (em segundos)	Tempo médio de execução da classe Lista Encadeada da API do Java (em segundos)	Tempo médio de execução da classe Lista com Arranjo da API do Java (em segundos)
10^3	0,00861098	0,01062387	0,007853352
10^4	0,01743592	0,10227634	0,023899228
10^5	0,06092066	12,17603038	0,824068905
10^6	0,42538588	*	82,036363407
10^7	6,26346034	*	*
3.10^7	36,24624045	*	*

Fonte: Autor.

Como é possível perceber, até 10^4 caracteres, as classes apresentaram desempenho semelhante. No entanto, a partir desse ponto, as classes da API do Java começam a ter um tempo de execução longo de tal forma, que se torna impraticável seu uso. O * indica que o algoritmo não foi capaz de ser executado em tempo inferior a 15 minutos. Desse modo, fica claro que ambas as estruturas "puras" tem um funcionamento restrito a casos em que o tamanho das sequências é inferior ou igual (no caso da lista com arranjo) a 10^6 caracteres. Ainda assim, nesse caso, apresenta um desempenho muito inferior à estrutura personalizada, que performa em 0,5% do tempo total da estrutura com arranjo.

Conclusões

Com base no exposto acima, é possível afirmar que o objetivo de prover uma solução algorítmica em tempo finito para o problema do DNA alienígena foi plenamente alcançado. De fato, todos os casos de teste puderam ser executados e geraram os resultados esperados. É evidente, todavia, que a solução ainda pode ser otimizada a fim de reduzir o tempo de execução.

Algo que percebi somente após a escrita deste artigo (e, portanto, não tive tempo hábil para implementar) foi em relação ao nodo da lista principal: no lugar de criar um atributo do tipo referência para uma lista encadeada, imagino que seja possível simplesmente criar um atributo do tipo inteiro para armazenar o número de repetições contíguas e, partir daí, realizar as operações.

Acredito, porém, que a performance de ambos não se distanciaria de modo tão significativo, já que a lista interna associada à principal não é percorrida inteiramente (apenas o primeiro elemento é acessado), ou seja, essa operação é O(1).

Outra questão a ser considerada é em relação ao fato da lista principal (que de fato afeta mais a performance do algoritmo) ter implementação baseada em encadeamento simples. Ainda que o acesso aos seus elementos não seja tão demorado em função de elementos iguais contíguos serem armazenados em uma lista interna durante a etapa de pré-processamento, o uso de uma lista duplamente encadeada poderia gerar resultados em tempo menor. Isso porque, armazenada a referência de um elemento *n*, seria possível acessar *n-1* de forma otimizada, já que haveria uma referência/atributo "previous" em cada nodo. Assim, eliminar-se-ia a necessidade de percorrer todos os elementos anteriores a *n* para acessá-lo, porém teria de se lidar com uma estrutura mais complexa, e neste trabalho buscou-se soluções mais simples/genéricas.

De qualquer forma, a solução apresentada não foi à toa, pois somente foi possível chegar às conclusões acima com base nela e em sua implementação. Assim, é possível considerá-la como rudimento para outras soluções mais otimizadas e mais complexas, envolvendo, por exemplo, estruturas duplamente encadeadas. No futuro, pretende-se testar o algoritmo com as modificações assinaladas e publicar os resultados neste repositório: *https://github.com/mateusfch*.