Pontifícia Universidade Católica do Rio Grande do Sul
Escola Politécnica
Ciência da Computação

Desenho, Análise e Implementação de Algoritmos Greedy e de Divisão e Conquista

Gabriel W. Piazenski, Gustavo W. M. da Silva e Lorenzo D. More

Trabalho I da disciplina "Projeto e Otimização de Algoritmos"

Professor: Rafael Scopel

Porto Alegre
Abril de 2024

O Problema 1 (Algoritmo Greedy)

O objetivo do primeiro problema é analisar e implementar um algoritmo do tipo *greedy* para encontrar uma maneira eficiente de detectar padrões em sequências de eventos que ocorrem ao longo do tempo. Essa situação surge da mineração de dados temporais, e um exemplo típico ocorre no mercado de ações: os eventos podem ser a compra e a venda de ações, enquanto o tempo é o período no qual o mercado está aberto e há a possibilidade de negociar essas ações. Vale destacar que um evento pode ocorrer diversas vezes na sequência de transações do dia.

Conforme o dia passa, as ações são negociadas e inseridas em uma ordem de movimentações. Quando o mercado fechar, existirá uma sequência **S** que contém todas as movimentações das ações no mercado naquele dia. O objetivo da detecção de padrões, nesse caso, é saber se algumas movimentações específicas, representadas por **S'** ocorreram nesse dia ou não. Assim, **S'** é considerada subsequência de **S** se for possível deletar determinados eventos de **S** para tornar os restantes, em ordem, iguais aos contidos em **S'**.

O Algoritmo

A seguir, apresenta-se um algoritmo capaz de percorrer dois vetores (de tamanho m e n) e verificar se um está contido no outro. Assume-se que as entradas são válidas. Ou seja, não faz sentido procurar uma subsequência em um dia sem transações ou procurar uma subsequência vazia em uma sequência de transações (vazia ou não).

Algorithm 1 boolean hasTrend(String[] S, String[] S_line)

```
m ← S_line.length
n ← S.length
j ← 0
i ← 0
while i < n and j < m do
    if S[i] equals S_line[j] then
        j ← j + 1          ▷ increments j every time it finds an element of S_line in S
    end if
    i ← i + 1
end while
return Whether j equals m          ▷ if j equals m, S_line is a subsequence of S
```

Análise do Algoritmo

Em primeiro lugar, busca-se provar que a decisão *greedy*, a cada passo, sempre resulta em uma solução global ótima. Para tanto, assume-se que tal solução ótima S^* possua um número k de verificações de transações para um problema de tamanho m. Considera-se, para a solução S^*, somente o número de verificações que deram certas (quando um evento de **S'** está presente em **S**). Essa solução deve conter uma solução ótima para um problema de $m-1$. Então existe uma solução de $k-1$ verificações na solução do problema $m-1$ que foi utilizado na solução para o problema de tamanho m. Assim, se fosse possível utilizar uma solução com $k-1$ verificações, poderia-se utilizar uma solução que contém menos que k verificações para um problema de tamanho m. Isso contradiz a premissa de que a solução ótima utiliza k verificações. Dessa forma, considera-se que a subestrutura

do problema é ótima nesse caso.

Analisando a decisão tomada dentro do *if* no algoritmo 1, é possível perceber que a escolha *greedy* é respeitada, pois se opta por comparar os elementos das duas sequências ao mesmo tempo, mas só há avanço na sequência **S'** quando for encontrado um elemento correspondente na sequência **S**. A decisão, portanto, garante que cada elemento de **S'** seja verificado, desde que presente em **S**. A forma escolhida para lembrar dessa decisão é incrementar uma variável j cada vez que encontrar um elemento correspondente; no final, compara-se j com o tamanho do vetor de **S'**.

Um caso que não passe por todos os elementos de **S'** resultará em uma resposta *false* porque a sequência **S** foi toda percorrida e ainda faltam elementos em **S'** para verificar se estão presentes. Nesse caso, $j \neq m$. O fato de incrementar a variável j é um indicativo para seguir para o próximo elemento de **S'**. Quando $j = m$, isso significa que todas as movimentações de **S'** foram encontradas, em ordem, em **S**, e a resposta será *true*.

Implementação e Tempo de Execução

Considera-se que o algoritmo proposto depende do tamanho da entrada n, que corresponde ao tamanho da sequência **S**. Ou seja, a execução depende do número de eventos ocorridos. De qualquer forma, a solução sempre percorrerá, no pior caso, todos os elementos da sequência **S**. Assim, é possível concluir que o algoritmo executa em $O(n)$.

O código-fonte comentado dos dois problemas pode ser encontrado nos arquivos *.java* enviados junto desse relatório. A análise do problema 2 inicia na próxima página.

O Problema 2 (Algoritmo de Divisão e Conquista)

A multiplicação de matrizes desempenha um papel crucial em diversas áreas, desde a ciência da computação até a matemática, que envolve o cálculo de cada elemento e sua subsequente soma, pode se tornar ineficiente à medida que o tamanho das matrizes aumenta. Isso se deve à sua complexidade assintótica, que é de $O(n^3)$, onde n é a ordem das matrizes.

O problema da multiplicação de matrizes consiste em calcular o produto de duas matrizes. Dadas duas matrizes A e B, onde A tem dimensões $m \times n$ e B tem dimensões $n \times p$, o produto $C = A \times B$ será uma matriz $m \times p$.

O método tradicional para multiplicação de matrizes envolve a multiplicação de cada elemento individual e a soma dos resultados, o que resulta em uma complexidade assintótica de $O(n^3)$, onde n é a ordem das matrizes.

Em campos como aprendizado de máquina, processamento de imagem e simulações computacionais, lidar com conjuntos de dados volumosos requer eficiência na multiplicação de matrizes. Portanto, há uma demanda por algoritmos mais eficientes que possam reduzir o tempo de processamento e o consumo de recursos para multiplicação de matrizes, levando ao desenvolvimento de técnicas como o algoritmo de Strassen, que busca otimizar esse processo e reduzir a complexidade assintótica.

O Algoritmo

O algoritmo de Strassen pode ser resumido da seguinte forma:

Algorithm 2 int[][] multiply(int[][] A, int[][] B)

```
for i to n/2 do
    for j to n/2 do
        for k to n/2 do
            Calculate product of blocks
        end for
        Sum results in k loop and redirect to resulting matrix
    end for
end for
    return Resulting matrix
```

Análise do Algoritmo

À medida que a necessidade de lidar com grandes conjuntos de dados crescia, várias propostas de algoritmos surgiram com o intuito de melhorar a complexidade assintótica da multiplicação de matrizes, que tradicionalmente era de $O(n^3)$. No entanto, foi com o algoritmo de Strassen que uma solução significativamente mais eficiente e escalável emergiu.

A abordagem de divisão e conquista é fundamental para o entendimento e a eficácia do algoritmo de Strassen. Ele introduziu uma abordagem inovadora que divide as matrizes em submatrizes menores e realiza operações matriciais utilizando um conjunto reduzido de multiplicações e adições. Essa técnica permitiu reduzir a complexidade assintótica para aproximadamente $O(n^{2.81})$, representando um avanço significativo em relação aos métodos anteriores.

Com essa melhoria, o algoritmo de Strassen se tornou uma solução viável para lidar com grandes conjuntos de dados, oferecendo uma alternativa mais eficiente e escalável para aplicações que exigem alta performance multiplicação de matrizes.

Para implementar o algoritmo, as matrizes são divididas em quatro submatrizes de igual tamanho, e sete novas matrizes temporárias M_n são calculas a partir das equações apresentadas abaixo:

$\mathrm{M}_1 \leftarrow (A_{1,1} + A_{2,1}) \cdot (B_{1,1} + B_{1,2})$

$\mathrm{M}_2 \leftarrow (A_{1,2} + A_{2,2}) \cdot (B_{2,1} + B_{2,2})$

$\mathrm{M}_3 \leftarrow (A_{1,1} - A_{2,2}) \cdot (B_{1,1} + A_{2,2})$

$\mathrm{M}_4 \leftarrow (A_{1,1} \cdot (B_{1,2} - B_{2,2})$

$\mathrm{M}_5 \leftarrow (A_{2,1} + A_{2,2}) \cdot B_{1,1}$

$\mathrm{M}_6 \leftarrow (A_{1,1} + A_{1,2}) \cdot B_{2,2}$

$\mathrm{M}_7 \leftarrow (A_{2,2} \cdot (B_{2,1} - B_{1,1})$

$\mathrm{P} \leftarrow M_2 + M_3 - M_6 - M_7$

$\mathrm{Q} \leftarrow M_4 + M_6$

$\mathrm{R} \leftarrow M_5 + M_7$

$\mathrm{S} \leftarrow M_1 - M_3 - M_4 - M5$

$\mathrm{C} \leftarrow P \bowtie Q \bowtie R \bowtie S$

Essas matrizes temporárias são então somadas para obter o valor das submatrizes da matriz resultante C.

Essas divisões são realizadas recursivamente até que as submatrizes sejam constituídas de apenas um elemento.

A implementação do Algoritmo de Strassen contém três laços, similar ao utilizado no algoritmo tradicional. No entanto, no laço mais interno, são efetuadas sete multiplicações em vez de oito como no algoritmo tradicional. É nesse fator que o algoritmo de Strassen se diferencia e obtém uma complexidade menor, o que implica em uma melhor performance.

Implementação e Tempo de Execução

Como foi dito, uma multiplicação de matrizes normalmente apresenta um algoritmo com $O(n^3)$.

Já no caso do algoritmo de Strassen, o número de adições e multiplicações necessárias para realizar a multiplicação de matrizes apresenta uma notação assintótica diferente, e por se tratar de um algoritmo do tipo de Divisão e Conquista podemos confirmar essa diferença por intermédio do Teorema O Método Mestre:

$T(n) = aT(n \div b) + f(n)$

a = quantidade de recursão

b = quantidade de vezes que o problema é particionado

f(n) = curso de combinação e divisão, neste caso $O(n^2)$

$T(n) = 7T(n \div 2) + O(n^2)$

$T(n) = O([7 + o(1)]^n) = O(n^{\log_2 7 + o(1)}) \approx O(n^{2.807})$

Também se nota que é um caso em que o peso aumenta geometricamente da raiz para as folhas. As folhas mantêm uma fração constante do peso total.

No entanto, a redução no número de operações aritméticas vem ao preço de uma estabilidade numérica um pouco reduzida.