

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

117536 - Projeto e Análise de Algoritmos
Turma: B

Análise Assintótica e Corretude do Algoritmo
MergeSort Utilizando PVS

Gabriel Levi - 16/0006490
Gabriel Nunes - 16/0006597

3 de dezembro de 2019

1 Introdução

A verificação formal de algoritmo, no que tange ao seu comportamento assintótico e a corretude do algoritmo é interesse central da Ciência da Computação. Por vezes, a prova via argumentação, isto é, lápis e papel pode ser suficiente para que o escritor convença o leitor de que o algoritmo está correto. Contudo, esse modelo de prova se sustenta muita vezes em passos de pura intuição, assumições que ambas as partes enxergam como um axioma, e saltos lógicos que por mais naturais que pareçam escondem um conjunto não-trivial de conceitos. A ocorrência desses aspectos em uma formalização pode ocultar falhas que, de fato, provem a incorretude do algoritmo e desmonstre um comportamento assintótico pior do que o esperado.

Como forma de minimizar o exposto anteriormente, introduz-se os sistemas de verificação de provas. Os verificadores garantem que os passos realizados dentro de uma prova respeitem o conjunto de regras de sua lógica intrínseca. Então, dadas premissas corretas e um ponto factível onde se deseja chegar, qualquer passo intermediário tem que, necessariamente, estar correto. Obviamente, construções ruins de objetivos e premissas podem levar a provas, ainda sim, incorretas ou impossíveis. Podemos concluir então que os sistemas de verificação pressupõe que uma prova, ou pelo menos a ideia da mesma, já exista e o usuário interessado o utilize para demonstrar que de fato aquela construção vale.

Um dos verificadores, PVS - Prototype Verification System - é a linguagem de especificação e provador automatizado de teoremas que aqui será utilizado. PVS trabalha com implementações em distribuições de diferentes versões de LISP. Em um arquivo, o usuário define premissas - um algoritmo - e teoremas. O arquivo é dado como entrada para o provador que requisita as regras a serem aplicadas até que o teorema desejado seja provado. As regras reconhecidas pelo PVS tratam-se simplesmente de regras da lógica de primeira-ordem. O PVS permite também que uma prova possa ser revisitada em uma representação gráfica que revela cada passo bem como suas dependências.

O objetivo deste trabalho é analisar assintoticamente o custo de tempo e a corretude do algoritmo de ordenação *Merge Sort* via PVS. O Merge Sort foi criado em meados de 1945 por John Von Neumann. A escolha deste algoritmo se deu por ser de implementação muito simples para múltiplas estruturas de dados tais como vetores e listas ligadas e por, ainda sim, ser um algoritmo com múltiplas aplicações dado o seu custo de execução. Este relatório apresentará

a ideia de prova formalizada via provador e fica a critério do leitor visitar o repositório para verificar a mesma. A formalização completa pode ser encontrada em github.com/paa-2019-2/levi-nunes.

Esse documento se organizará, daqui em diante, por um capítulo 2 de revisão teórica. Em seguida, no capítulo 3, a apresentação do algoritmo Merge Sort. O capítulo 5 apresentará a ideia abordada pelos autores para a análise de assintótica do algoritmo bem como argumentação sobre o pior e o melhor caso do mesmo. O capítulo 4 apresentará, como o capítulo anterior, a ideia abordada e sua argumentação. Por fim, o capítulo de conclusão, apresentará um resumo dos resultados aqui encontrados acompanhado de um meta-texto discorrendo sobre as principais dificuldades do trabalho com o verificador formal de provas.

2 Revisão teórica

Este capítulo apresentará conceitos importantes para o entedimento do que se segue nos capítulos posteriores. Caso sintá-se a vontade com o conceito, cujo o nome será enunciado no título de cada subseção, não há nenhum mal em pular. Cada um dos conceitos será apresentado de maneira simples mais preocupado com ser inteligível para o leitor do que com um profundo formalismo. Em compensação, uma boa bibliografia de apoio será indicada para aqueles interessados em se aprofundar ou que não acharam que o texto de uma ou mais subseções foi suficiente.

// Se não existir nada além de Notação Assintótica, converter pra uma seção de revisão sobre Notação assintótica.

2.1 Notação assintótica

2.2

3 O algoritmo Merge Sort

O algoritmo fundamenta-se na técnica Dividir-para-Conquistar. A técnica consiste em, dada uma instância do problema quebra-la em partes até que as mesmas sejam fáceis ou triviais de se resolver, por fim, cada pequena solução é combinada a fim de obter a solução para o problema maior. Apesar de parecer similar, esta técnica é bem diferente de programação dinâmica pois

cada subproblema é disjuncto dos demais. Desta forma, o algoritmo recebe uma lista como entrada uma lista de elementos comparáveis e divide tal lista até que cada sublista possua tamanho unitário - trivialmente ordenável - e então combina as listas para obter a lista de entrada ordenada. O processo pode ser visualizado na figura 1 onde a parte superior diz respeito a dividir e a parte inferior diz respeito a conquista.

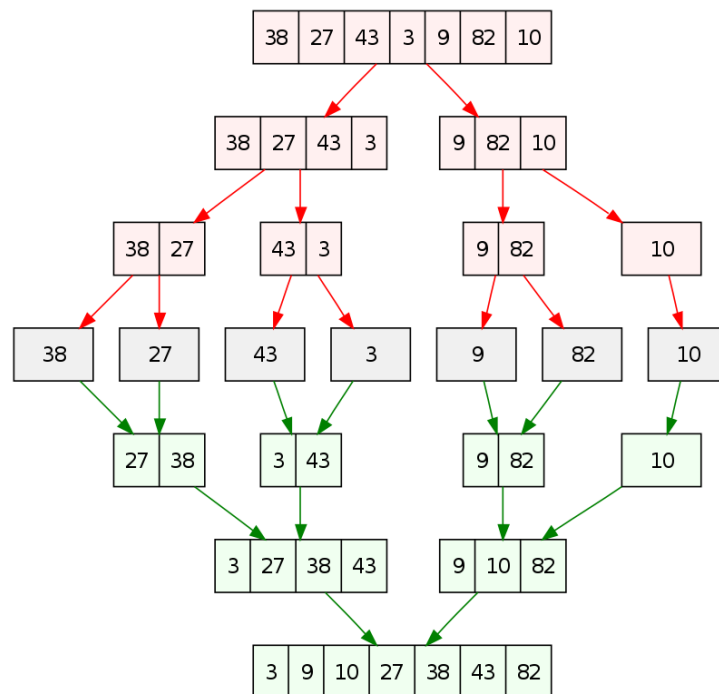


Figura 1: Ordenação da instância [38, 27, 43, 3, 9, 82, 10]

Por mais poderosa que seja a ideia, a implementação do Merge Sort é bem simples e pode ser deduzida a partir do pseudo-código abaixo, para fins de facilitar a leitura, denotamos a cabeça da lista τ como $car(\tau)$ e a sua calda como $cdr(\tau)$, o operador $+$ significa inserção de um elemento no início

de uma lista ou a concanetação de duas listas.

Algorithm 1: MERGE

Data: ρ, η : *Sorted lists*
Result: Sorted merge of ρ and η
if $\rho = Nil$ or $\eta = Nil$ **then**
| **return** $\rho + \eta$
else
| **if** $car(\rho) \leq car(\eta)$ **then**
| | **return** $car(\rho) + MERGE(cdr(\rho), \eta)$
| **else**
| | **return** $car(\eta) + MERGE(\rho, cdr(\eta))$

Algorithm 2: MERGESORT

Data: τ : *List of comparable elements*
Result: *Sorted permutation of τ*
if $length(\tau) \leq 1$ **then**
| **return** τ
else
| $prefix \leftarrow MERGESORT(first_half(\tau));$
| $suffix \leftarrow MERGESORT(second_half(\tau));$
| **return** $MERGE(prefix, suffix);$

4 Corretude do algoritmo Merge Sort

A corretude de um algoritmo passa por demonstrar que o mesmo possui certas características independente da instância e que essas características se verifiquem antes, durante e depois da execução. Para um algoritmo de ordenação, é esperado que o mesmo responda para qualquer entrada uma permutação ordenada da mesma, isto é, a saída não só deve estar ordenada como também o número de ocorrências de cada elemento deve ser o mesmo da lista de entrada.

A análise de MERGESORT nos leva a uma série de resultados intermediários que fortalecem a argumentação da corretude do algoritmo. Tais resultados estão expostos abaixo nesta seção e a formalização em PVS dos mesmos pode ser encontrado no repositório apresentado previamente na introdução deste texto.

Lema 4.1. Para quaisquer ρ e η , listas, e n valor, o número de ocorrências de n em $MERGE(\rho, \eta)$ é igual ao número de ocorrências de n em ρ mais o número de ocorrências de n em η .

Demonstração. □

Lema 4.2. Para quaisquer entradas ρ e η , listas, o tamanho da lista resultante de $MERGE(\rho, \eta)$ é a soma dos tamanhos de ρ e η .

Demonstração. □

Lema 4.3. Para quaisquer entradas ρ e η , $MERGE(\rho, \eta)$ é uma permutação de concatenação de η e ρ .

Demonstração. □

Lema 4.4. Para quaisquer entradas ρ e η , se ambas as listas estão ordenadas então a resultante de $MERGE(\rho, \eta)$ também será uma lista ordenada.

Demonstração. Basta demonstrar que para qualquer iteração de $MERGE$, o menor elementos dentre ambas as listas é escolhido para inserção no final da lista parcialmente resultante. Uma vez que ambas as listas são ordenadas, necessariamente o menor elemento será a cabeça de ρ ou de η . Utilizando da definição de $MERGE$ e induzindo sobre o tamanho de ambas as listas 3 situações emergem:

1. Alguma das listas ρ e η tem tamanho 0.
2. A cabeça de ρ é menor ou igual que a cabeça de η .
3. A cabeça de ρ é maior que a cabeça de η .

Na primeira situação, no máximo uma das listas possui elementos, então o elemento tomado para a composição da lista resultante será a cabeça (o menor elemento) daquela que ainda não é vazia, por conveniência, o algoritmo já retorna a lista inteira, mas se de fato tivesse que escolher apenas um, seria o menor disponível. O segundo e o terceiro caso são similares, a cabeça de ρ é igual ao menor elemento de ρ e o mesmo vale para a cabeça de η , logo, o menor entre os ambos será o menor dentre todos e este será escolhido para a inserção ao final da lista resultante, a chamada recursiva posterior cai novamente em algum dos 3 casos.

□

Lema 4.5. *Para qualquer entrada τ , lista, o tamanho de τ é igual ao tamanho de $MERGESORT(\tau)$.*

Demonstração. □

Lema 4.6. *Para qualquer lista τ , a resultante de $MERGESORT(\tau)$ é uma permutação de τ .*

Demonstração. □

Lema 4.7. *Para qualquer lista τ , a resultante de $MERGESORT(\tau)$ é uma lista ordenada.*

Demonstração. □

Teorema 4.8. *Para qualquer lista τ , a resultante de $MERGESORT(\tau)$ é uma permutação ordenada de τ .*

Demonstração. □

5 Análise de complexidade do Merge Sort

A complexidade de um algoritmo de ordenação se baseia em contar o número de comparações realizadas durante sua execução. Neste sentido, existe uma classificação de algoritmos de ordenação chamada algoritmos de ordenação por comparação, que, segundo Cormen [1], são algoritmos baseados em comparações entre dois elementos e efetuam pelo menos $\Omega(n \lg n)$ comparações no pior caso.

O Merge Sort é um algoritmo de ordenação por comparação, e portanto, sua análise de complexidade se baseia em contar quantas comparações são efetuadas para ordenar uma lista de tamanho n . Para tanto, foram necessárias modificações no algoritmo, de forma a incluir um contador de comparações. O pseudo-código abaixo representa a ideia da inclusão de um contador no Merge Sort:

Algorithm 3: COUNT-MERGE

Data: ρ, η : Sorted lists, σ : natural

Result: Sorted merge of ρ and η , number of comparisons

if $\rho = Nil$ or $\eta = Nil$ **then**

return $\rho + \eta, \sigma$

else

if $car(\rho) \leq car(\eta)$ **then**

return $car(\rho) + MERGE(cdr(\rho), \eta, \sigma + 1)$

else

return $car(\eta) + MERGE(\rho, cdr(\eta), \sigma + 1)$

Algorithm 4: COUNT-MERGESORT

Data: τ : List of comparable elements

Result: Sorted permutation of τ , number of comparisons

if $length(\tau) \leq 1$ **then**

return $\tau, 0$

else

$prefix, \sigma \leftarrow MERGESORT(first_half(\tau));$

$suffix, \iota \leftarrow MERGESORT(second_half(\tau));$

return $MERGE(prefix, suffix, \sigma + \iota);$

A adição de um contador no Merge Sort permite a análise de comparações, porém, não é possível assegurar que o contador não inseriu um erro no algoritmo. Portanto, é necessário provar os seguintes lemas, que asseguram a equivalência entre o Merge Sort com e sem contador:

Lema 5.1. $\forall \rho, \eta$ listas de naturais, a resultante de $MERGE(\rho, \eta)$ é igual à resultante de $COUNT - MERGE(\rho, \eta, \sigma)$, a não ser por σ .

Demonstração. A ideia da prova é mostrar que a lista retornada por $MERGE$ é igual à lista retornada por $COUNT - MERGE$. Utilizando indução forte sobre a soma tamanho das listas ρ e η , temos que para quaisquer listas cujo a soma dos tamanhos das mesmas é menor que a soma do tamanho das listas, a propriedade vale. Portanto basta expandir a definição de $MERGE$ e $COUNT - MERGE$ e analisar os seguintes casos, no momento da chamada recursiva:

1. cabeça de ρ é maior ou igual que a cabeça de η
2. a cabeça de ρ é menor que a cabeça de η

Em ambos, temos uma construção de uma lista no qual podemos ver que a propriedade da indução pode ser aplicada, pois temos que tamanho da cauda de ρ + tamanho de η é menor que tamanho de ρ + tamanho de η , e precisamos mostrar que inserir a cabeça de ρ na lista construída resulta na mesma lista para as duas funções. De fato, ao aplicar a hipótese de indução, temos que inserir o mesmo elemento em duas listas iguais resulta em duas listas iguais, o que é trivialmente verdade. \square

Lema 5.2. $\forall \rho$, lista de naturais, a resultante de $MERGESORT(\rho)$ é igual à resultante de $COUNT - MERGESORT(\rho)$, a não ser pelo contador retornado.

Demonstração. De forma semelhante ao lema anterior, a ideia da prova é mostrar que o retorno de $MERGESORT$ e $COUNT - MERGESORT$ são equivalentes. Com indução forte, temos que para quaisquer listas cujo a soma dos tamanhos das mesmas é menor que a soma do tamanho das listas, a propriedade vale. Com isso, aplicamos a hipótese de indução para **prefix** e **suffix** e expandimos as definições de $MERGESORT$ e $COUNT - MERGESORT$. Basta aplicar o lema anterior para $MERGE$ e $COUNT - MERGE$ na expansão de $MERGESORT$ e $COUNT - MERGESORT$, e o seguinte é provado. \square

O lema a seguir é um lema auxiliar, necessário para a prova do lema 5.6.

Lema 5.3. $\forall \rho$, lista de naturais, o tamanho da resultante de $COUNT - MERGESORT$ é igual ao tamanho de ρ .

Demonstração. Para tal, basta utilizar o lema 5.2 e o lema 4.5. \square

Os lemas seguintes são acerca da complexidade do Merge Sort, utilizando os contadores de comparações.

Lema 5.4. $\forall \rho, \eta$, listas de naturais, o valor do contador de comparações retornado pela chamada $COUNT - MERGE(\rho, \eta, 0)$ é menor ou igual que $n + m$, onde n é o tamanho de ρ e m é o tamanho de η

Demonstração. \square

Lema 5.5. $\forall \rho, \eta$, listas de naturais, e σ , um natural, o valor do contador de comparações retornado pela chamada $COUNT-MERGE(\rho, \eta, \sigma)$ é menor ou igual que $\mathbf{n} + \mathbf{m} + \mathbf{c}$, onde \mathbf{n} é o tamanho de ρ , \mathbf{m} é o tamanho de η e \mathbf{c} é o número de comparações realizadas anteriormente, ou seja, σ .

Demonstração. □

Lema 5.6. $\forall \rho$, lista de naturais, tal que tamanho de ρ é maior que zero, o valor do contador de comparações retornado pela chamada $COUNT-MERGESORT(\rho)$ é menor ou igual a $\mathbf{n} + \mathbf{n} \log \mathbf{n}$, onde \mathbf{n} é o tamanho de ρ .

Demonstração. □

6 Conclusão

Referências

- [1] CORMEN, T., LEISERSON, C., AND STEIN, R. *Algoritmos: teoria e prática*. ELSEVIER EDITORA, 2012.