Documentação | Trabalho 3 | INF 610 - Estruturas de Dados e Algoritmos

> Implementação completa disponível no Github. Essa documentação visa atender os requisitos do trabalho prático nº 3 da disciplina INF610 do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Viçosa.

Força Bruta x Programação Dinâmica

A primeira atividade tem como objetivo comparar a solução de um problema cuja abordagem de força bruta seja menos eficiente do que a abordagem de programação dinâmica.

O Problema

O problema escolhido foi da distância Levenshtein (ou distância de edição). Essa distância é uma maneira de quantificar o quão diferente são duas strings, contando a quantidade mínima de operações necessárias para transformar uma string A em uma string B. Cada edição de caracteres em uma string para resultar em outra é contado como uma unidade nessa métrica, ou seja, inserções, substituições e exclusões de caracteres.

Por exemplo, a distância Levenshtein entre as palavras `carro` e `barcos` é 3, devido as 3 operações de caracteres necessárias:

> 'carro' -> '**b**arro' (subsituição de **c** por **b**)
> 'barro' -> 'bar**c**o' (substiuição de **r** por **c**)
> 'barco' -> 'barco**s**' (inserção de **s** no fim)

Casos

O problema possui 3 casos de cálculo de operações.

Caso 1: Uma das substrings está vazia.

Se uma substring X está vazia, basta inserir os caracteres de Y em X, e o custo dessa operação é a quantidade de caracteres em Y. Exemplo:

> ('', 'SOL') -> ('SOL', 'SOL') (3 inserções, custo = 3)

Caso 2: O último caractere das substrings X e Y são iguais.

Nesse caso, basta calcular a distância para as substrings antes desses dois caracteres ($X[0...i-1]$ e $Y[0...j-1]$). Como nenhuma operação é realizada, o custo será 0.

> ('SOL', 'CAROL') = ('SO' -> 'CARO') = ('S' -> 'CAR')

Caso 3: O último caractere das substrings X e Y são diferentes.

Nesse caso uma das 3 operações serão necessárias: inserção, remoção ou substituição de caracteres.

Caso 3.1 - Inserção

Insira o último caractere de Y em X. O tamanho de Y reduz em 1 e o de X se mantém. Ou seja, conta como $X[0...i]$, $Y[0...j-1]$, já que movemos na substring alvo mas não na substring fonte.

> ('SOL', 'SOA') -> ('SOL**A**', 'SOA') = ('SOL', 'SO') (usando o Caso 2)

Caso 3.2 - Remoção

Remova o último caractere de X. O tamanho de X reduz em 1 e o de Y se mantém. A operação conta portanto em $X[0...i-1]$, $Y[0...j]$, já que movemos na substring fonte mas não na alvo.

> ('MAN**O**', 'MACA') -> ('MAN', 'MAC')

Caso 3.3 - Substituição

O caractere atual de X se torna o caractere atual de Y. O tamanho de ambos X e Y reduzem em 1 já que ao substituir os caracteres movemos em ambas substrings: $X[0...i-1]$, $Y[0...j-1]$.

> ('BRASIL', 'HEXA') -> ('BRASIA', 'HEXA') = ('BRASI', 'HEX') (usando o Caso 2)

Definição formal do problema

Formalizando, temos portanto um problema que pode ser definido recursivamente. Seja X e Y, as substrings de entrada e sejam m e n o tamanho (quantidade de caracteres) dessas substrings, respectivamente.

```
dist(m, n) =
    max(m, n); quando min(m, n) = 0
    dist(m − 1, n − 1); quando X[m − 1] = Y[n − 1]
    1 + min(dist(m − 1, n), dist(m, n − 1), dist(m − 1, n −1)); quando X[m − 1] ≠ Y[n − 1]
```

Força Bruta

A implementação abaixo, de força bruta, usa a prórpia definição formal do problema diretamente para realizar o cálculo da distância de Levenshtein. O código abaixo demonstra a sua implementação, que faz chamadas recursivas de acordo com a definição do problema, até que o critério de parada seja atendido, que no caso é quando uma das substrings esteja vazia ($m = 0|n = 0$).

```rust
// mod brute_force;
fn distance(x: &String, m: usize, y: &String, n: usize, evaluations: &mut i32) -> usize {
    *evaluations += 1;
    if m == 0 { return n; }
    if n == 0 { return m; }

    let cost = if x.chars().nth(m - 1) == y.chars().nth(n - 1) { 0 } else { 1 };

    let a = distance(x, m - 1, y, n, evaluations) + 1;
    let b = distance(x, m, y, n - 1, evaluations) + 1;
    let c = distance(x, m - 1, y, n - 1, evaluations) + cost;

    return min(min(a, b), c);
}
```

A função recursiva recebe como entrada as substrings X e Y, além dos indíces m e n, como a definição do problema. Como esse problema só está interessado em encontrar a distância e não alterar as strings em si, os valores de m e n são meros apontadores de delimitação da substring da chamada atual da função para as strings originais. Dessarte, as strings originais **não** são mutadas. Um apontador mutável é o último parâmetro, que incrementa a cada chamada da função para a análise de comparação realizada nas próximas seções.

Dado duas strings como entrada uma função pública inicializa os indíces das substrings e contador de chamadas, retornando a distância de Levenshtein e a quantidade de chamadas em uma tupla, como mostra o *snippet* a seguir.

```rust
// mod brute_force;
pub fn levenshtein_distance(x: &String, y: &String) -> (usize, i32) {
    let m = x.chars().count();
    let n = y.chars().count();
    let mut evaluations = 0;

    let d = distance(x, m, y, n, &mut evaluations);

    return (d, evaluations);
}
```

Programação Dinâmica

O problema descrito possui a característica de se resolver subproblemas que se sobrepõem. A partir do snippet da função `distance/5` apresentado na seção de força bruta, é possível notar que a cada chamada recursiva outras 3 são realizadas, no caso de duas substrings não-vazias. A primeira chamada decrementa o primeiro índice em 1, a segunda decrementa o segundo índice em 1 e o terceiro decrementam ambos. Na sub-árvore de execuções da primeira chamada, eventualmente a sua execução também invocorá a segunda chamada que decrementa o segundo índice em 1, resultando na mesma chamada de decrementar ambos índices em 1. Esse subproblema portanto estaria sendo computado repetidamente e poderia ser otimizado com a abordagem de programação dinâmica.

Por exemplo, a distância entre as strings `Samba` e `Fama` pode ser visto como o resultado da distância entre as strings `Samba` e `Fam` somado com o custo da operação para adicionar o último caractere no segundo argumento. Dessa forma é possível construir uma matriz de maneira bottom-up que, a partir de resoluções dos subproblemas menores até chegar na entrada completa, resultaria como:

		S	**A**	**M**	**B**	**A**
	0	1	2	3	4	5
F	1	**1**	2	3	4	5
A	2	2	**1**	2	3	4
M	3	3	2	**1**	2	3
A	4	4	3	2	**2**	**2**

Utilizando-se de uma matriz dessa, o custo da avaliação de um dado i-ésimo índice da entrada X e j-ésimo índice da entrada Y depende apenas dos valores na matriz acima, à esquerda, e na diagonal superior esquerda, além do custo de qual dos casos do problema a comparação entre a entrada nesses índices resulta. A resposta das strings completas estará na última linha e na última coluna, portanto.

```rust
// mod dynamic;
pub fn levenshtein_distance(x: &String, y: &String) -> (usize, i32) {
    let m = x.chars().count();
    let n = y.chars().count();

    let mut evaluations: i32 = 0;
    let mut table = vec![vec![0; n + 1]; m + 1];

    // Initialize first row and first column
    for i in 1..(m + 1) { table[i][0] = i; }
    for j in 0..(n + 1) { table[0][j] = j; }

    // Bottom-up evaluation
    for i in 1..(m + 1) {
        for j in 1..(n + 1) {
            evaluations += 1;
            let cost = if x.chars().nth(i - 1) == y.chars().nth(j - 1) { 0 } else { 1 };

            let a = table[i - 1][j] + 1;
            let b = table[i][j - 1] + 1;
            let c = table[i - 1][j - 1] + cost;

            table[i][j] = min(min(a, b), c);
        }
    }

    return (table[m][n], evaluations);
}
```

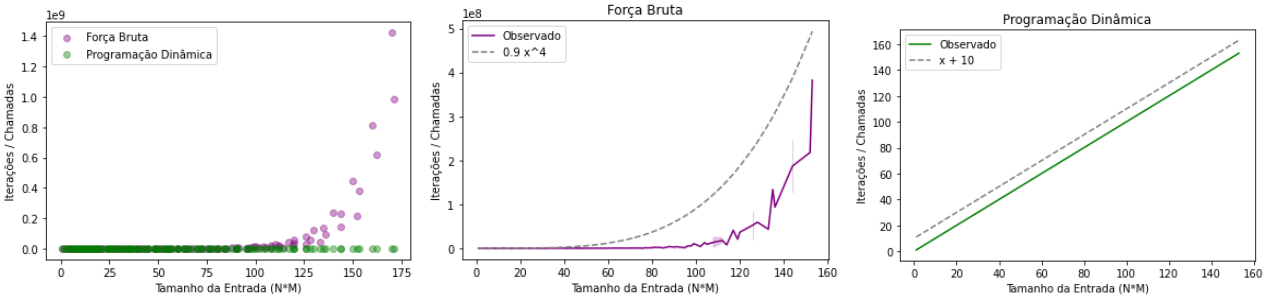
Assim como na implementação de Força Bruta, a função recebe duas strings de entrada, e retorna a distância de Levenshtein e a quantidade de chamadas (iterações) realizadas, a partir de um contador inicializado no início da função e incrementado a cada chamada do sub-problema que preenche a matriz. Entretanto, no início a primeira linha e coluna é inicializada com os valores já conhecidos de entrada entre uma substring vazia e uma substring com os caracteres até a i-ésima linha ou j-ésima coluna, o que é considerado apenas uma inicialização e não a execução do subproblema de fato.

A lógica do corpo do subproblema permanece inalterado, com a diferença de que ao invés de realizar chamadas recursivas para analisar o custo para as substrings é realizado uma simples consulta na matriz nos índices de linha e coluna imediatamente anteriores ao do índice atual. Como os laços começam no índice 1 e o indíce 0 já foi preenchido na etapa de incialização, o laço realiza as exatas mesmas operações a cada chamada.

Comparação das Abordagens

Como o problema recebe duas entradas, o tamanho da entrada depende da quantidade de caracteres de ambas strings. Para facilitar a análise, consideraremos o tamanho da entrada como o produto da quantidade de caracteres dessas strings, que será representada no eixo x dos gráficos a seguir. No eixo y, a quantidade de chamadas do sub-problema, seja por invocações da função recursiva ou de uma nova iteração no laço.

132 execuções diferentes foram realizadas para as duas abordagens, variando o tamanho da entrada de 1 à 171. Strings aleatórias de entradas foram geradas a cada execução, já que o conteúdo da string não é relevante no algoritmo. As Figuras abaixo mostram a comparação das execuções.



A primeira imagem mostra um gráfico de dispersão com cada ponto representando uma execução realizada. É notável como os pontos em verde, das execuções de Programação Dinâmica, se mantiveram muito próximos ao variar o tamanho da entrada, enquanto os roxo, de Força Bruta, cresceram mais rapidamente. As duas figuras seguintes mostram o comportamento individual de cada abordagem, assim como uma equação de referência para comparar o nível de crescimento. Em Força Bruta o crescimento foi exponencial, enquanto com Programação Dinâmica o crescimento foi exatamente linear.

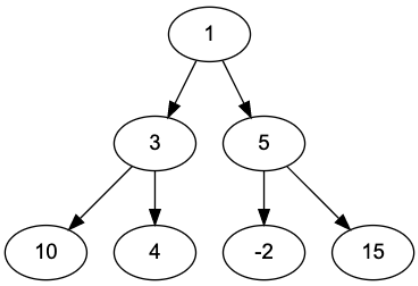
Greedy x Divisão e Conquista

Nessa segunda seção, discutiremos um problema onde a abordagem guloso não é ótima, mas a abordagem de divisão e conquista sim, e comparando as complexidades.

O Problema

O problema escolhido foi o *Binary Tree Maximum Path Sum* (MPS), onde a partir de uma árvore binária, deseja-se encontrar o caminho até um nó folha cuja soma dos valores de seus nós seja o máximo. O caminho é uma sequência de nós que só pode aparecer uma única vez, não podendo ter ciclos, e que obrigatoriamente inicia-se no nó raiz e termine em um nó folha.

A Figura abaixo mostra um exemplo de árvore, e o seu MPS seria de 21, no caminho que saindo da raiz 1 soma 5 e finalmente 15.



Árvore Binária

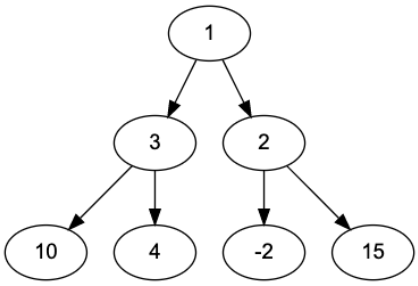
Uma estrutura simples de árvore binária foi implementada como mostra a *struct* abaixo. Para esse trabalho, o tipo de dado utilizado no atributo `value` desse nó foi um inteiro de 32 bytes (`i32`). Uma função auxiliar também foi implementada para gerar as árvores na Linguagem Dot e posteriormente visualizada no GraphViz Online, que gerou as figuras dessa documentação.

```
pub struct BTNode<T> {
    pub left: ChildNode<T>,
    pub right: ChildNode<T>,
    pub value: T,
}
```

Abordagem *Greedy* (Gulosa)

Uma abordagem possível para resolver o problema seria de a partir do nó-raiz sempre escolher o filho que possua o maior valor. Por exemplo, no exemplo da figura da seção do Problema, a partir do nó raiz se compara qual o maior valor entre os filhos 3 e 5. Como 5 é maior, esse seria o escolhido no caminho final e essa mesma lógica seria aplicada novamente, até que se mova até um nó raiz. Entretanto, essa abordagem não garante a solução ótima, uma vez que em um dado nível uma decisão pior pode ser tomada antes de se saber que o caminho ótimo inclui essa pior decisão. Na Figura abaixo, a abordagem resultaria num MPS = 14 (1 + 3 + 10), sendo que o resultado ótimo seria MPS = 18 (1 + 2 + 15).

Essa abordagem é gulosa portanto, devido ao fato que sempre escolhe a melhor decisão localmente, não retornando ou recomputando valores, e não garantindo o resultado ótimo.



O trecho de código a seguir detalha a implementação dessa abordagem, retornando o valor da MPS e a quantidade de iterações realizadas, inciando no nó raiz passado por parâmetro. Existem 3 casos na avaliação de cada nó: se tiver 2 filhos, escolhe-se o que possui o maior valor entre eles como o próximo nó a ser avaliado; se tiver apenas um filho, escolhe-se ele como próximo nó; se não tiver nenhum filho, ou seja, se atingir um nó raiz, o laço se quebra e o algoritmo retorna. A cada nó visitado um contador de iterações é incrementado em 1 assim como a soma (MPS).

```
// mod greedy;
pub fn mps(node: &Box<BTNode<i32>>) -> (i32, i32) {
    let mut evaluations = 0;
    let mut sum = 0;

    let mut current_node = node;
```

```
    loop {
        evaluations += 1;
        sum += current_node.value;

        // Case 1: there is a value in both children
        if let (Some(left), Some(right)) = (&current_node.left, &current_node.right) {
            current_node = if left.value >= right.value { left } else { right };
            continue;
        }

        // Case 2: there is only a left child
        if let (true, Some(left)) = (current_node.right.is_none(), &current_node.left) {
            current_node = left;
            continue;
        }

        // Case 2: there is only a right child
        if let (true, Some(right)) = (current_node.left.is_none(), &current_node.right) {
            current_node = right;
            continue;
        }

        // Case 3: It's a leaf
        break;
    }

    return (sum, evaluations);
}
```

Abordagem de Divisão e Conquista

O problema do MPS para uma árvore pode ser dividido em subproblemas menores, cujo resultado desses subproblemas pode ser utilizado para obter a solução do problema maior. Para cada sub-árvore avaliada, quebra-se em dois subproblemas: a avaliação da sub-árvore à esquerda e a avaliação da sub-árvore à direita. Na etapa de conquista, basta escolher o máximo entre as duas soluções onde o problema foi dividido.

O *snippet* abaixo mostra a implementação do problema em uma função recursiva. Por restrições de apontadores nulos de memória da linguagem de programação implementado, em alguns casos a chamada da função gera apenas mais uma chamada, pois as chamadas com a entrada de um apontador nulo são suprimidas.

```
// mod divide_and_conquer;
fn mps_step(node: &Box<BTNode<i32>>, evaluations: &mut i32) -> i32 {
    *evaluations += 1;

    // Case 3: It's a leaf
    if node.left.is_none() && node.right.is_none() {
        return node.value;
    }

    // Case 1: there is a value in both children
    if let (Some(left), Some(right)) = (&node.left, &node.right) {
        let left_mps = mps_step(left, evaluations);
        let right_mps = mps_step(right, evaluations);

        return max(left_mps, right_mps) + node.value;
    }

    // Case 2: there is only a left child
    if let (true, Some(left)) = (node.right.is_none(), &node.left) {
        return mps_step(left, evaluations);
    }

    // Case 2: there is only a right child
    if let (true, Some(right)) = (node.left.is_none(), &node.right) {
        return mps_step(right, evaluations);
    }
}
```

Dado o nó raiz de uma árvore binária como entrada uma, função pública inicializa o contador de chamadas, retornando o valor de MPS quantidade de chamadas em uma tupla, como mostra o snippet a seguir.

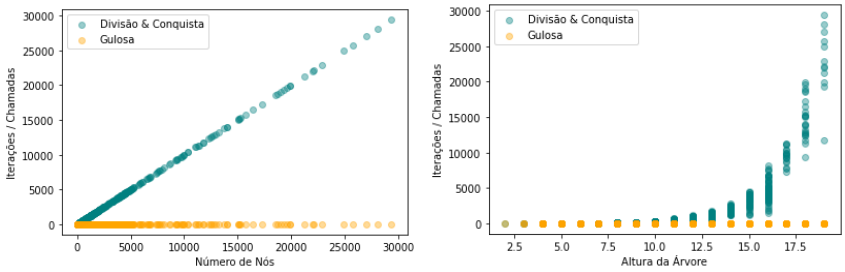
```rust
// mod divide_and_conquer;
pub fn mps(node: &Box<BTNode<i32>>) -> (i32, i32) {
    let mut evaluations = 0;
    let sum = max_subsum(node, &mut evaluations);

    return (sum, evaluations);
}
```

Comparação das Abordagens

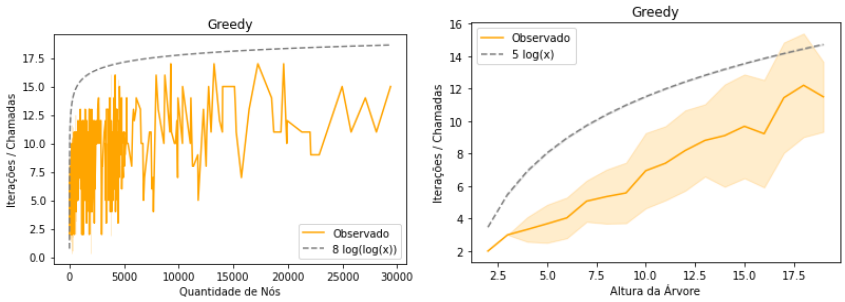
Para a realização da comparação das abordagens, duas interpretações distintas do tamanho da entrada é apresentado: a primeira é a de quantidade de nós na árvore, o que é uma decisão mais clara na abordagem de divisão e conquista que computa todos os nós existentes na árvore; a segunda é a da altura da árvore, o que é uma entrada mais clara para a abordagem *greedy*.

Foi implementado, de forma auxiliar, um gerador de árvores aleatórias, tendo como entrada a probabilidade de um dado nó ser um nó folha. Dessarte, diversas execuções foram realizadas gerando árvores aleatórias como entrada, aumentando incrementalmente a probabilidade de geração de nós folhas em uma taxa, e ajustando essa taxa a cada execução para gerar árvores de tamanhos diferentes.

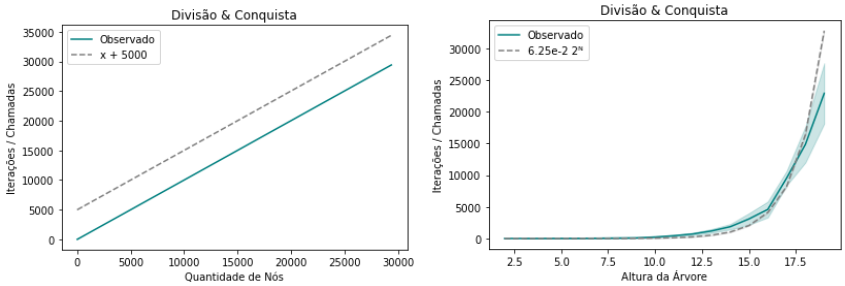
No total, 674 execuções distintas foram executadas, com a quantidade de nós variando de 3 até 29.399 e a altura variando de 2 até 19. A Figura abaixo mostra gráficos de dispersão para as duas interpretações de entradas, com o número de nós ou com a altura da árvore. A Divisão e Conquista com o número de nós se mostra claramente linear, enquanto a Gulosa demonstra não ter um crescimento significativo. Com a altura da árvore (que é sabido ser $\log_2(n)$ em árvores binárias onde n é a quantidade de nós), mostra o crescimento exponencial, mas mesmo assim difícil de ver o crescimento da abordagem gulosa.



Entretanto, olhando para os gráficos da abordagem gulosa individualmente, é possível notar o comportamento logarítmico com a altura da árvore, e consequentemente parecido com a função de nível $\log(\log(x))$ para a quantidade de nós. Em cinza, equações de referência foram traçadas para comparar o comportamento do observado com a função de referência.



De maneira similar, o comportamento de divisão e conquista é claramente linear com a quantidade de nós, e exponencial com a da altura da árvore.



Concluindo, a abordagem greedy possui uma ordem de crescimento muito mais baixa que a de divisão e conquista, e consegue calcular o resultado do problema para árvores que demorariam horas ou dias na outra abordagem, além da necessidade de mais recursos computacionais em memória para a pilha de chamadas. Apesar da abordagem não ser ótima e poder dar respostas ruins, para árvores grandes suficientes é uma abordagem válida tendo em vista a necessidade de uma resposta aproximada em tempo ágil.