Construindo Heap em tempo linear

Eduardo Miranda Pedrosa Filho

LAVRAS

MAIO/2019

INTRODUÇÃO

O objetivo deste trabalho é a implementação da função de construção de um Heap(Build Heap) em tempo linear. O desafio consiste em fazer alterações na maneira mais conhecida de construir Heap, fazendo com que fique em uma complexidade menor.

O Heap é uma estrutura de dados baseada em árvore que possui algumas características específicas. No caso de um Min Heap, cada elemento é sempre menor que seus dois filhos, fazendo com que o menor elemento fique na raiz da árvore. No caso de um Max Heap, cada elemento é maior que seus dois filhos e o maior elemento pode ser encontrado na raiz.

DESENVOLVIMENTO

A função mais conhecida possui complexidade de O(n*log(n)), sendo n o número de entradas do Heap. Isso acontece porque cada vez que um elemento é inserido no Heap é necessário no máximo log(n) alterações na estrutura. Como isto é feito n vezes(uma vez para cada elemento inserido), então no final temos no pior caso O(n*log(n)).

Para construir uma função de inserção no heap em tempo linear, ou seja, O(n), é necessário fazer algumas alterações na maneira como esses dados entram no programa. Para isso, considera-se que os dados estão dispostos em um vetor que não precisa estar ordenado. Caso os dados inicialmente não estejam em um vetor, basta inseri-los em qualquer ordem, o que tem complexidade de O(n). A partir daí começa a função de heapify que será explicada a seguir.

Explicando em alto nível, o algoritmo começa a comparar da metade para trás do Heap, isso porque sabe-se que metade dos elementos de um Heap são folha, isto é, não possuem filhos. Assim começando do elemento(tamanho/2 -1) faz-se verificações com todos os elementos que possuem filhos na árvore. A cada elemento é verificado qual o menor filho e então comparado com o elemento em questão. Caso esse elemento for maior que o menor filho, então é trocado a posição dos dois e na próxima iteração o elemento em questão será o elemento da iteração anterior, mas na nova localização(porque foi trocado com o filho). Assim o número

máximo de alterações que cada elemento vai sofrer depende do nível, isto é, da altura em que esse nó se encontra no Heap. Também existe um número de elementos para cada nível na árvore, como será explicado melhor em seguida. Ao final, temos um Min Heap construído em O(n).

Pseudo-código:

BuildHeap

Entrada: Vetor desordenado h

Saída: Heap no vetor h

```
1 início
2     para cada elemento "i" do meio para o início faça:
3         se((i possuir filho) e (menorFilho(i) < i)) faça:
4             troca(i,menorFilho);
5             acabou = false;
6             enquanto((i possuir filhos) e (!acabou)) faça:
7                 se(menorFilho(i) < i) faça:
8                     troca(i,menorFilho);
9                     i = menorFilho;
10                fim-se
11                se nao faça:
12                    acabou = true;
13                fim-se
14            fim-enquanto
15        fim-se
16    fim-para
17 fim
```

Analisando a complexidade:

Das linhas 2 a 5 temos complexidade em O(n), isso acontece porque é executada metade do número de entradas e assintoticamente encontramos que a função de limite superior é linear.

As linhas 10, 13, 15 e 17 estão no pseudo-código apenas para melhorar a visualização, mas elas não possuem código de máquina, portanto a análise de complexidade não se aplica.

Para as linhas de 6 a 12 a análise será feita a seguir.

O laço de repetição encontrado na linha 6 funciona da seguinte maneira: para cada elemento é verificado se ele é maior que o menor filho, ou seja, se tem necessidade de trocar sua posição. Como estamos verificando o pior caso, considera-se que essa condição sempre será satisfeita e portanto queremos encontrar a função que expressa o número máximo de vezes em que esse laço será executado.

Como existem elementos em diferentes alturas no Heap, onde o elemento se encontra está diretamente relacionado com a quantidade de vezes em que o laço interno será executado (Michelle Hugue, 1998). Para isso temos a função que expressa quantos elementos tem em cada nível, primeiro nível (n/2), segundo nível (n/4) e assim por diante, como expressado na função a seguir:

$$\frac{n}{2^{h+1}}$$

Sendo n o número elementos no vetor de entrada e h a altura em que se encontra o elemento (as folhas estão na altura 0). Então, podemos expressar a complexidade assintótica do laço da seguinte maneira:

$$O(\sum_{h=0}^{lg(n)} h \cdot \frac{n}{2^{h+1}})$$

Sendo um somatório de todas as alturas, em que cada altura tem-se a quantidade de elementos presentes naquela altura $\frac{n}{2^{h+1}}$ multiplicado pela quantidade de vezes que cada elemento pode ser alterado h, totalizando a quantidade de vezes que todos os elementos vão ser alterados no pior caso, o que é também a quantidade de vezes em que o laço interno das linhas de 6 a 12 é executado.

Manipulando e usando uma aproximação para a soma temos:

$$O(n \cdot \sum_{h=0}^{lg(n)} \frac{h}{2^h}) < O(n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h})$$

Sabe-se que $2^{h+1} = 2 \cdot 2^h$, e como estamos usando a notação assintótica do “Big O”, ignora-se constantes. Sabe-se pela série geométrica que :

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$$

Então, fazendo a derivada e multiplicando os dois lados por x, temos:

$$\sum_{h=0}^{\infty} hx^{h-1} = \frac{1}{(1-x)^2} \qquad \sum_{h=0}^{\infty} hx^{h} = \frac{x}{(1-x)^2}$$

Finalmente, substituindo o x por ½, temos:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2$$

Portanto:

$$O(n \cdot 2) = O(n)$$

Dessa forma, mostramos que a complexidade do algoritmo no pior caso é na ordem de O(n), ou seja, linear.

A implementação do build heap modificado está disponível em: <https://github.com/EduardoPedrosa/BuildHeapInN>

RESULTADOS

Abaixo estão gráficos demonstrando a melhoria do algoritmo em relação ao anterior. Assim foram escolhidas diferentes tipos de entrada de dados. Os resultados obtidos foram da execução do algoritmo implementado em um computador Acer Aspire F 15, Intel Core i5-7200U e CPU 2.5 GHz com 8GB de memória RAM.

1. O primeiro gráfico mostra a execução do Build Heap no algoritmo mais conhecido e também do algoritmo modificado no pior caso (caso em que os números de entrada estão em ordem decrescente).
2. O terceiro gráfico mostra a execução do Build Heap no algoritmo mais conhecido e também no algoritmo modificado no caso em que os números são aleatórios, para simular uma entrada real.

Como pode ser visto nos gráficos a seguir, encontra-se uma melhora significativa em relação ao algoritmo anterior, principalmente em relação ao pior caso. É importante observar que nas duas estruturas foram ignoradas todas as instruções que não fazem parte do BuildHeap para manter uma análise mais relevante.

Gráfico 1:

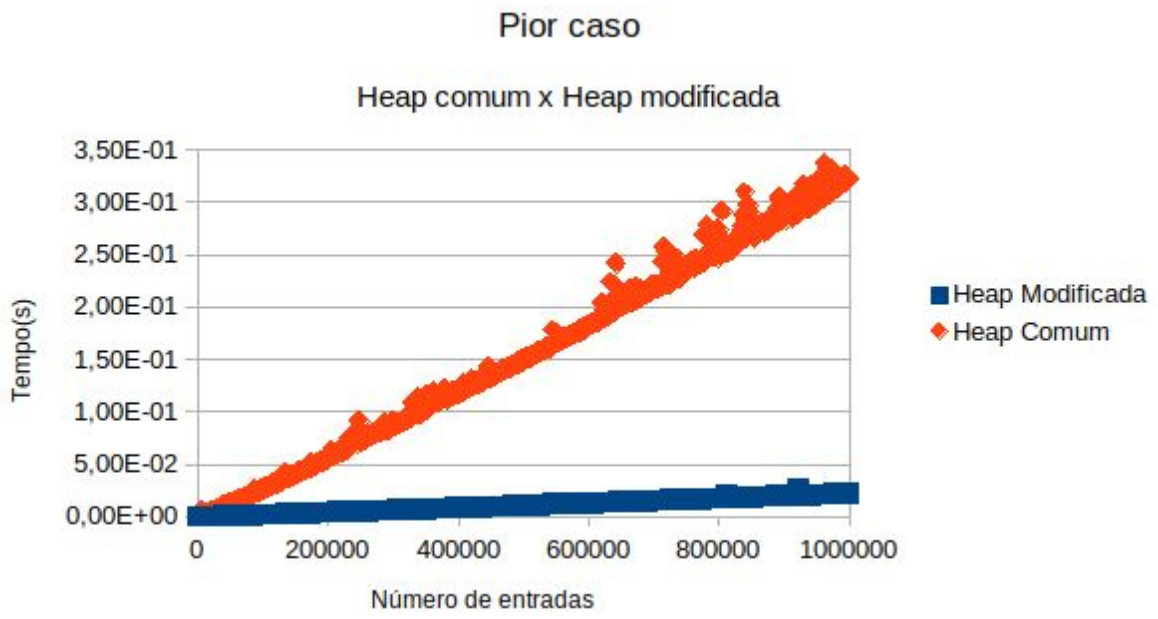
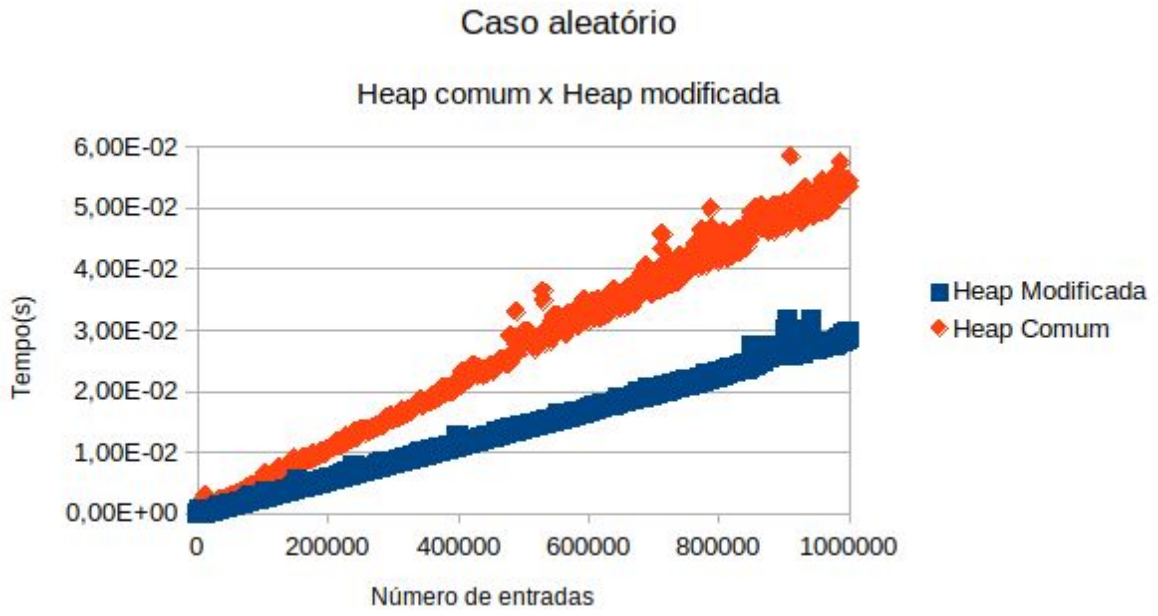


Gráfico 2:



CONCLUSÕES

Neste trabalho, foi implementado uma função de construção de um heap em tempo linear. Conclui-se que esse algoritmo pode ser usado como parte de um outro, como o de Dijkstra ou de Prim, por exemplo. Assim é possível diminuir a complexidade e o custo em relação ao tempo total do algoritmo, isto porque a melhora em relação ao algoritmo anterior é significativa, como demonstrado nos gráficos.

Desenvolver este trabalho foi importante para usar conceitos vistos em aula de forma prática, incentivar a pesquisa e o estudo sobre o assunto e entender a importância de se projetar um bom algoritmo para que ele execute com o menor custo possível.

REFERÊNCIAS

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009.

HUGUE, Michelle. BuildHeap Analysis. **HeapSort Analysis and Partitioning**, 1998. Disponível em: <http://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf>. Acesso em: 05 jun. 2019.

TAYLOR, David Scot. Linear Time BuildHeap. **Youtube**, 05 fev. 2015. Disponível em <https://www.youtube.com/watch?v=MiyLo8adrWw>. Acesso em: 05 jun. 2019.