UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

THIAGO BELL

Laboratório 1 - Hollow Heaps e Dijkstra

Porto Alegre, 2017

1.1 Tarefa

Implementar uma *Hollow Heap* e usá-la ao implementar o algoritmo de Dijkstra. Verificar desempenho da *Hollow Heap* e do algoritmo de Dijsktra e compará-los com as previsões teóricas. Os testes presentes no plano de teste sugerido foram utilizados neste trabalho.

1.2 Implementação

Os algoritmos foram implementados em C++. A *Hollow Heap* utiliza memória alocada dinamicamente para armazenar os seus nodos. Os grafos são representados por lista de adjacências. Um vetor armazena os vértices, e outro, as arestas.

1.3 Ambiente de Teste

Os experimentos foram realizados usando um processador Intel i7 2600k acompanhado de 8 GiB de RAM. O sistema operacional utilizado foi Ubuntu Linux 16.10.

1.4 Análise de Complexidade da *Hollow Heap*

A complexidade teórica das operações da *Hollow Heap* foram comparadas com os resultados de experimentos. Conclui-se que a implementação respeita as previsões teóricas.

1.4.1 Avaliação das Operações de Inserção

Fixando-se o valor $n = 2^{27} - 1$, adicionou-se n elementos na *Hollow Heap* com chaves no intervalo de $[n, 1]$. Para cada bloco com o intervalo de inserções $[2^{i-1}, 2^i - 1]$ para $i \in [0, 26]$, foi medido o número de nodos criados C_i, o número de melds M_i e tempo de execução T_i. Para cada valor inserido na *Hollow Heap*, um *meld* é executado. Um *meld* é a união de duas *heaps*. Onde um nodo único também é considerado uma heap. Com experimentos, verificou-se que o número de chamadas as rotinas de inserção e *meld* foram chamadas uma vez para cada elemento inserido como era esperado. A inserção de um elemento na heap tem custo $O(1)$ (*meld* do elemento com a *heap*). Logo, inserir n nodos terá custo $O(n)$. Comparando o número de inserções com o número de *melds*, percebe-se que eles são os mesmos indicando que a implementaçao respeita a definição do algoritmo. Comparando-se o tempo de execuçao com o número de operações esperadas pela complexidade, vê-se uma relação constante como pode ser vista na figura 1.1

Considerando as duas comparações, conclui-se que a implementação da inserção na *Hollow Heap* respeita a complexidade do algoritmo.

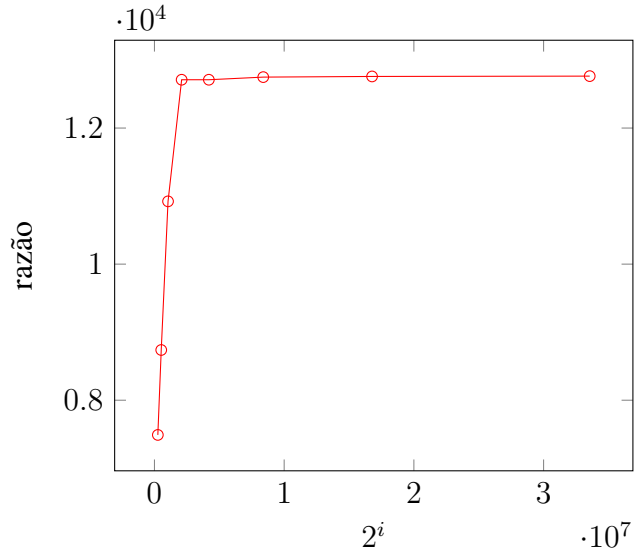

Figura 1.1: Razão entre tempo de execução e o custo esperado.

1.4.2 Avaliação das Operações de Decrescimento de Chaves

Considerando-se uma variável i, $2^i - 1$ chaves são inseridas com valor de $2^i + 1$. Em seguida, 2^i chaves com valor $2^i + 2$ são adicionadas. Medindo-se o tempo, as chaves com valor, $2^i + 2$ são atualizadas para valores decrescentes no intervalo $[2^i, 1]$. As operações de atualização começam por 2^i e a cada operação seguinte a esse valor é decrescido por 1. A complexidade para a operação de *decrease key* é $O(1)$. Logo, para atualizar as 2^i chaves, a complexidade é de $O(2^i)$. Ao comparar essa complexidade com o tempo de execuçao, nota-se a convergência da razão desses valores para um valor como mostra a figura 1.2. Assim, pode-se concluir que o algoritmo escala como previsto.

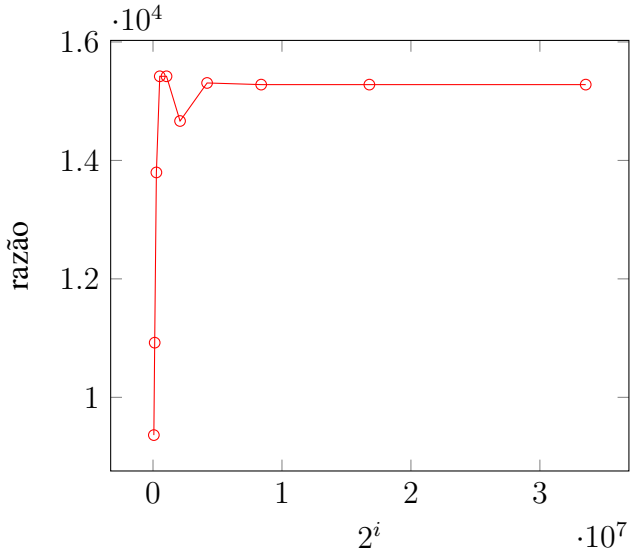

Figura 1.2: Mostra a razão entre o tempo de execução e o custo teórico esperado

1.4.3 Avaliação das Operações de Remoção Mínima

Para um valor i, $n = 2^i - 1$ chaves com valor aleatórios são adicionados. Depois, $m = 2^{i-1}$ chaves são removidas. O tempo de execução e o número de swaps foram

medidos para cada i. A complexidade da operação de *delete min* é de $O(log\ n)$ onde n é o número de nós na heap. Para remover m nodos, a complexidade é $O(log(n) * m)$. Na figura 1.3 é comparado esse custo com o tempo de execução. A convergência a um valor indica que o tempo de execução cresce conforme a complexidade do algoritmo.

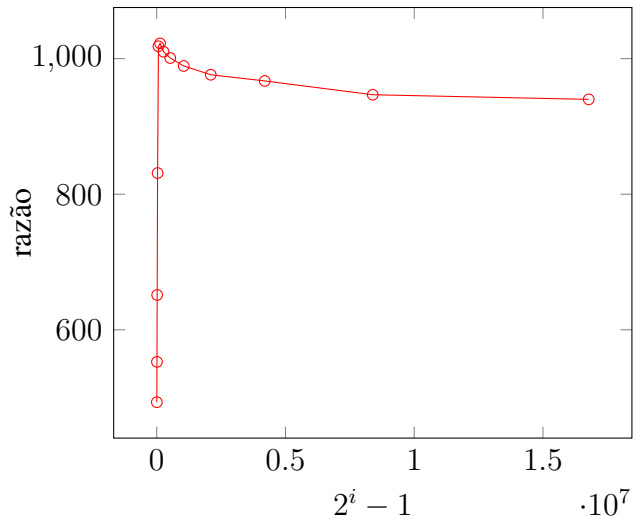

Figura 1.3: Mostra a razão entre o tempo de execução e o custo teórico esperado

1.4.4 Comparando *Binary Heap* e *Hollow Heap*

O tempo de execução para as operações descritas acima nos dois tipos de heaps foram comparados. Essas comparações estão plotadas nas figuras 1.4, 1.5 e 1.6. Com exceção das operações de *delete min*, a *Hollow Heap* teve melhor desempenho.

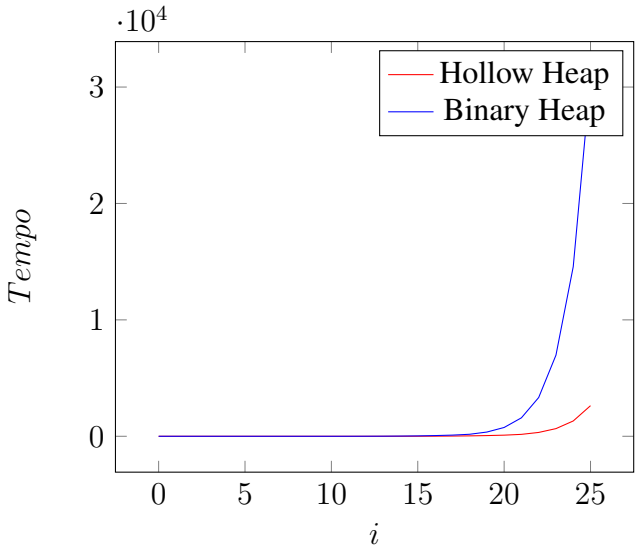

Figura 1.4: Tempo de execução para testes de inserção

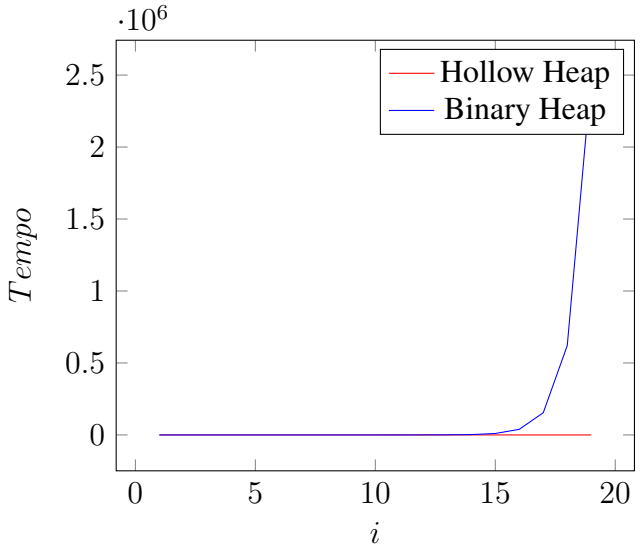

Figura 1.5: Tempo de execução para testes de *decrease key*

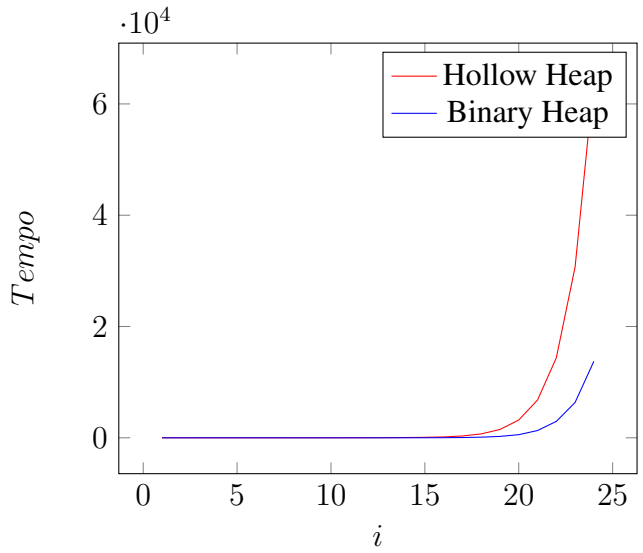

Figura 1.6: Tempo de execução para testes de *delete min*

1.5 Análise de Complexidade do Algoritmo de Dijkstra

1.5.1 Variando o número de Arestas

Fixando-se o número de vértices em 2^{15}, variou-se o número de arestas entre 2^{23} e 2^{27}. Esse intervalo foi pequeno devido a restrições de memória no limite superior e o de precisão na medição de tempo no inferior. A complexidade do algoritmo de Dijkstra é de $O(m + n * log(n))$. Comparando o tempo de execuçao com essa complexidade - através da divisão da complexidade esperada para o tamanho da instância pelo tempo - obteve-se a curva na figura 1.7. Nota-se uma convergência para um valor constante indicando que a implementação segue os limites teóricos. Os números de *inserts* e *delete mins* foram menores ou iguais ao de vértices e o de *decrease min*, menor ou igual ao de arestas. Uma regressão linear na figura 1.8 sobre os dados obtidos mostrou que o tempo de execução cresce exponencialmente de acordo com a soma do número de arestas.

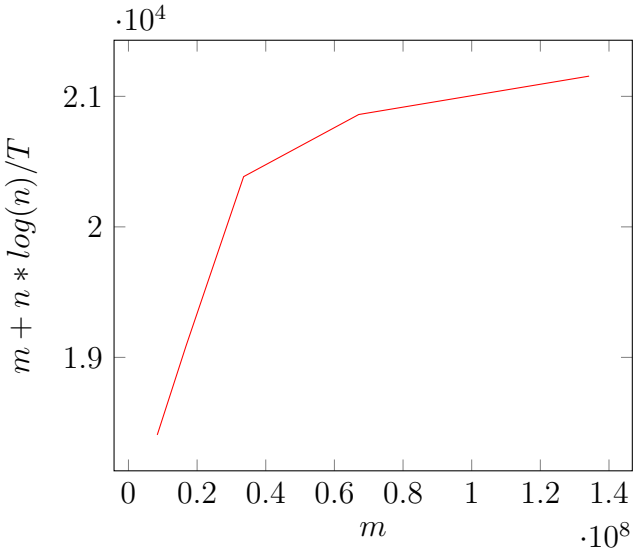

Figura 1.7: Mostra a razão entre o custo teórico e o tempo de execução

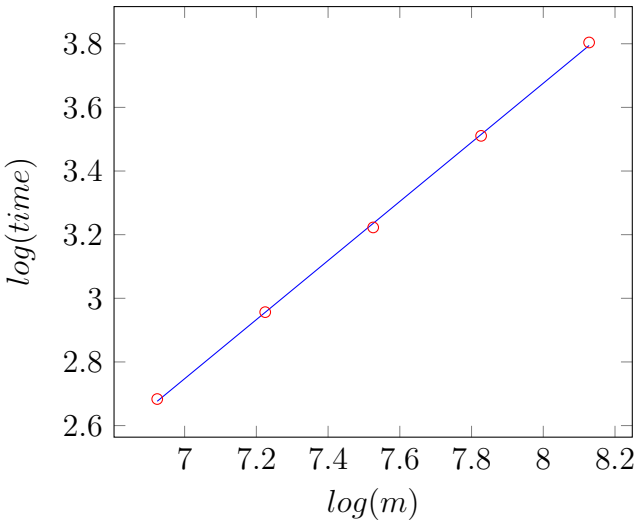

Figura 1.8: Regressão linear. O número de vértices foi desconsiderado pois é constante.

1.5.2 Variando o número de Vértices

Fixou-se o número de arestas em 2^{20} e variou-se o número de vértices entre 2^{11} e 2^{18}. A complexidade é a mesma do caso anterior: $O(m+n*log(n))$. Comparando-se o tempo de processamento como no caso anterior, obteve-se os resultados mostrados na figura 1.9. O gráfico mostra que o valor tende a uma constante indicando que a implementação respeita a complexidade do algoritmo. Uma regressão linear na figura 1.10 sobre os dados obtidos mostrou que o tempo de execução cresce exponencialmente de acordo com a soma do número de vértices.

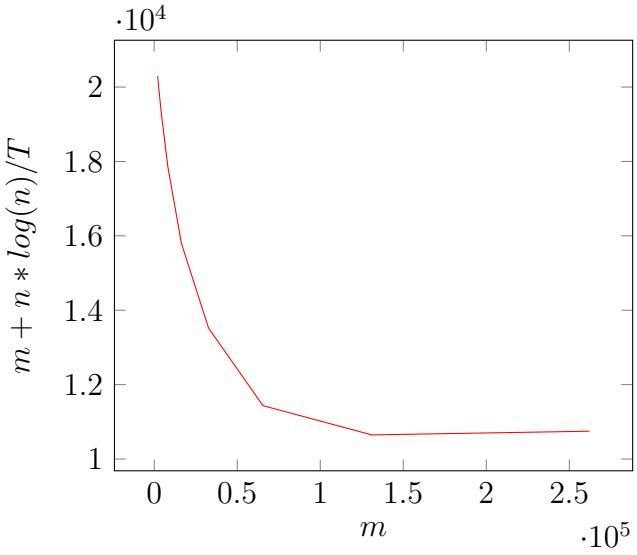

Figura 1.9: Mostra a razão entre o custo teórico e o tempo de execução

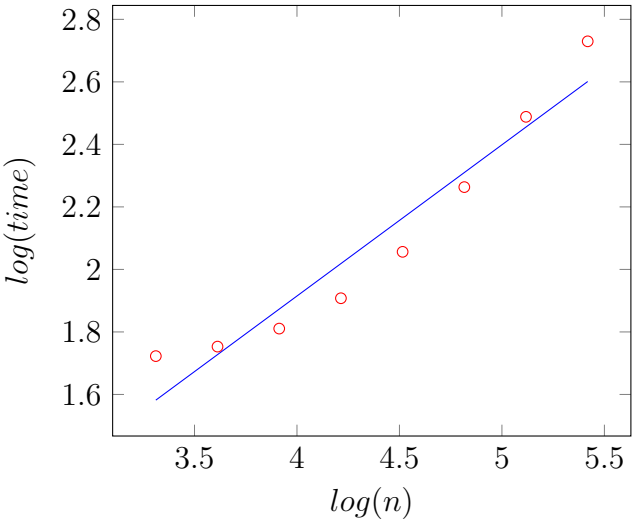

Figura 1.10: Regressão linear. O número de arestas foi desconsiderado pois é constante.

1.5.3 Comparando *Binary Heap* e *Hollow Heap*

Os tempos de execução para o algoritmo de Dijkstra usando as duas heaps foram comparados. Essas comparações estão plotadas nas figuras 1.11 e 1.12. Em ambos os casos a *Hollow Heap* teve melhor desempenho.

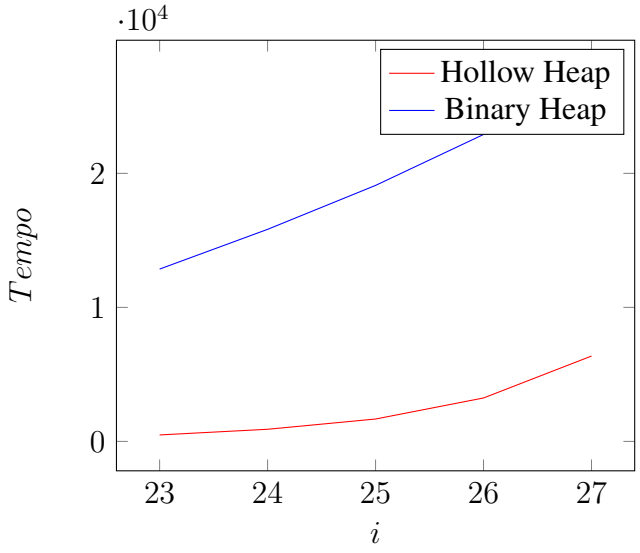

Figura 1.11: Tempo de execução para testes de inserção

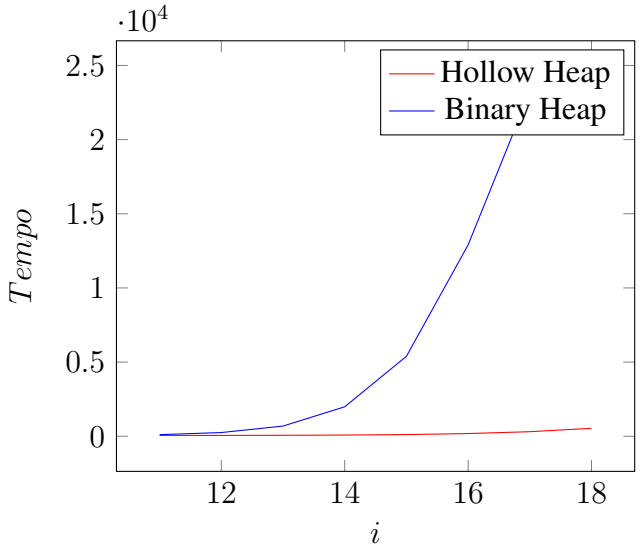

Figura 1.12: Tempo de execução para testes de *decrease key*

1.6 Verificando a implementação de Dijkstra com grandes instâncias

Testou-se a implementação do algoritmo com duas instâncias conforme recomendado no plano de teste. Comparou-se, também, com os resultados obtidos com as *Binary heaps* implementadas anteriormente. A comparação para os dois casos pode ser vista nas tabelas 1.6 e 1.6. O caso usando *Hollow Heaps* teve um tempo de execução menor em ambos os casos a custo de um consumo maior de memória.

rede	tempo de execução (s)	máximo de memória (MiB)
NY	1.62	54.41
USA	141.81	5204

Tabela 1.1: Tempo de execução do algoritmo de Dijkstra usando Hollow Heaps

rede	tempo de execução (s)	máximo de memória (MiB)
NY	1.83	42.67
USA	169.78	3555

Tabela 1.2: Tempo de execução do algoritmo de Dijkstra usando Binary Heaps

1.7 Conclusão

Implementou-se a estrutura de dados *Hollow Heap* assim como o algoritmo de Dijkstra usando essa estrutura. A implementação respeitou as previsões teóricas. Ao calcular a regressão linear para os experimentos envolvendo o algoritmo de Dijkstra, obteve-se no entretanto uma comprovação de uma hipótese exponencial o que contraria a complexidade do algoritmo. Comparou-se a performance da *Hollow Heap* e da *Binary Heap* nos vários experimentos. A primeira teve melhores tempos de execução na maior parte dos casos. Essa melhora tem o custo de maior uso de memória.