

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

THIAGO BELL

## **Laboratório 1 - Binary Heaps e Dijkstra**

Porto Alegre, 2017

## 1.1 Tarefa

Implementar uma *heap*  $n$ -ária e usá-la ao implementar o algoritmo de Dijkstra. Verificar desempenho da *heap* e do algoritmo de Dijkstra 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 *heap* utiliza um vetor da *standard library* da linguagem para o armazenamento da árvore de forma eficiente. 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 *Heap*

A complexidade teórica das operações da *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 *heap* com chaves no intervalo de  $[n, 1]$ . Como se usou uma *heap* de grau 2, isso corresponde a 27 níveis completos na árvore. Ao adicionar os elementos, após inserir  $2^i - 1$  chaves, mede-se o tempo  $T_i$  e o número de *swaps*  $I_i$  onde  $0 \leq i \leq 27$ . A complexidade desta operação é de  $O(2^i)$ . Com isso pode-se estimar o custo  $E = 2^i$  para cada  $i$ . Na figura 1.1 pode-se ver o número de *swaps* e  $E$  em função de  $2^i$ . Se essas curvas crescerem da mesma forma, a complexidade é respeitada. A curva  $E*15$  mostra os valores de  $E$  multiplicados por uma constante mostrando, assim, que assintoticamente o custo é o mesmo.

Na figura 1.2, mostra-se a razão entre o tempo obtido experimentalmente e o custo teórico. Nota-se que a curva para valores maiores de  $2^i$  tende a um valor constante o que indicaria que o tempo e o custo estão crescendo na mesma taxa.

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

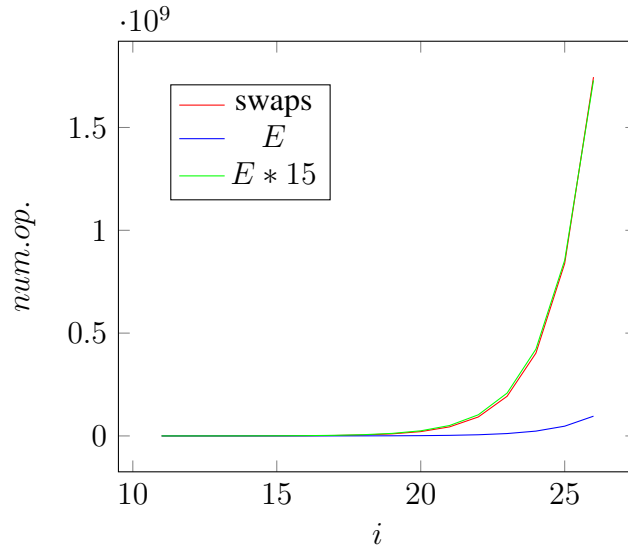


Figura 1.1: Comparação entre o número de swaps e a complexidade através do custo esperado ( $E$ ) e seu múltiplo ( $E * 15$ ).

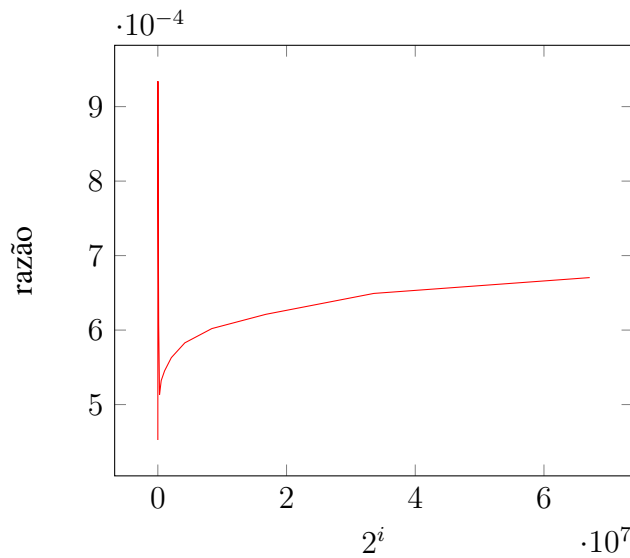


Figura 1.2: Razão entre tempo de execução e o custo esperado.

### 1.4.2 Avaliação das Operações de Atualização

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 e o número de swaps, 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 esse valor é decrescido por 1.

O custo teórico esperado também é de  $E = 2^i i$ . Ao comparar com o número de swaps obtido para cada iteração, percebeu-se que estes valores são iguais. Logo, pelo menos em número de swaps, a complexidade é obedecida. No entanto, ao comparar o tempo de execução, o resultado obtido foi uma reta inclinada o que indicaria que o tempo de execução cresce mais que o esperado. Não se soube determinar o motivo de isso ter ocorrido. É possível que seja devido ao fato de que uma busca pelo nodo com o id que

deseja-se atualizar ocorre. Caso, ao invés de o id, a posição do elemento na árvore ser informada, o tempo medido poderia ter crescido da mesma forma que a previsão teórica.

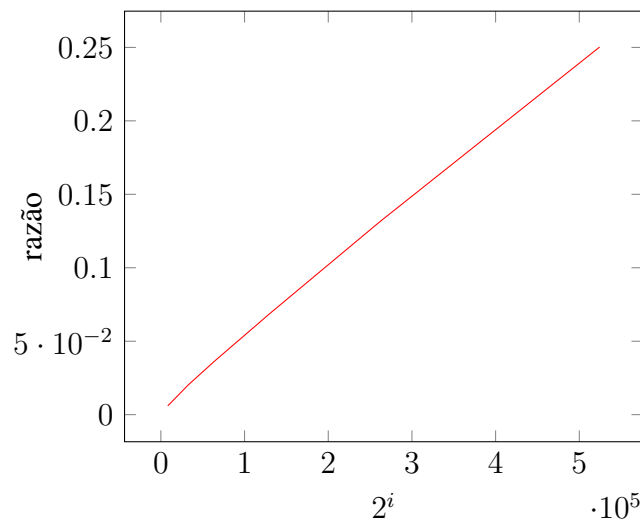


Figura 1.3: Mostra a razão entre o tempo de execução e o custo teórico esperado

### 1.4.3 Avaliação das Operações de Remoção

Para um valor  $i$ ,  $2^i - 1$  chaves com valor aleatórios são adicionados. Depois,  $2^{i-1}$  chaves são removidas. O tempo de execução e o número de swaps foram medidos para cada  $i$ . O custo teórico dessas operações é de  $E = 2^i i$ . Na figura 1.4 é comparado esse custo com o número de swaps. De forma análoga a feita anteriormente, também se multiplicou o valor de  $E$  por uma constante. A comparação do tempo de execução com  $E$  pode ser vista na figura 1.5. A convergência a um valor indica que o tempo de execução cresce conforme a complexidade do algoritmo.

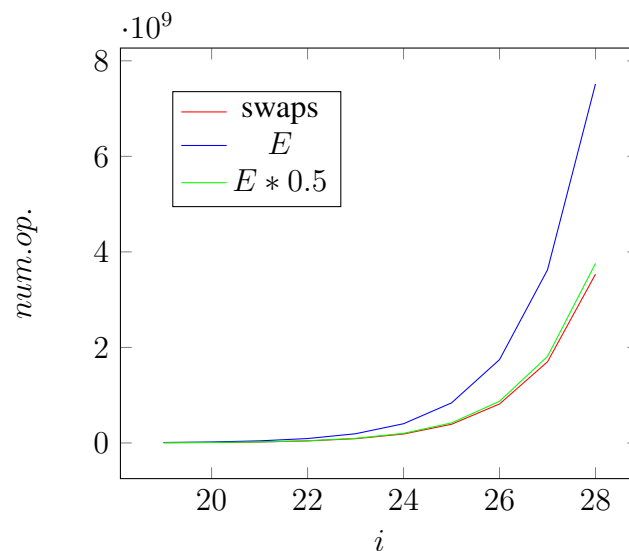


Figura 1.4: Comparação entre o número de swaps e a complexidade através do custo esperado ( $E$ ) e seu múltiplo ( $E * 0.5$ )

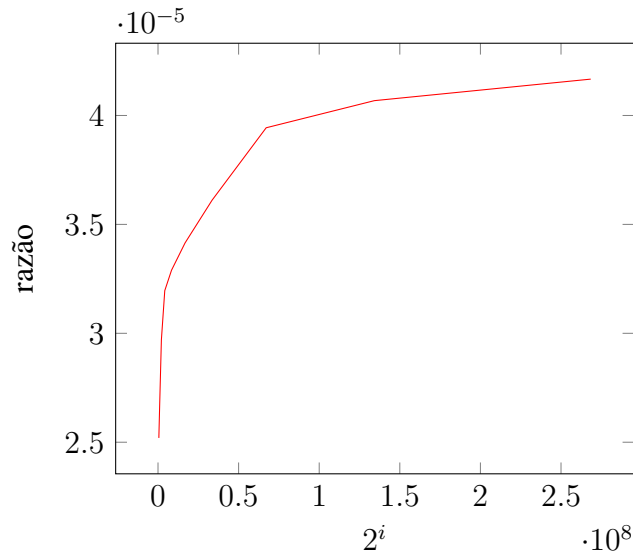


Figura 1.5: Mostra a razão entre o tempo de execução e o custo teórico esperado

## 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  $2^{18}$  e  $2^{28}$ . Ao comparar o tempo de execução com o custo teórico de  $E = (n + m) * \log(n)$ , obteve-se uma convergência a um valor constante como pode ser visto na figura 1.6. Os números de *inserts* e *deletes* foram menores ou iguais ao de vértices e o de *updates* menor ou igual ao de arestas. Uma regressão linear na figura 1.7 sobre os dados obtidos mostrou que o tempo de execução cresce exponencialmente de acordo com a soma do número de arestas.

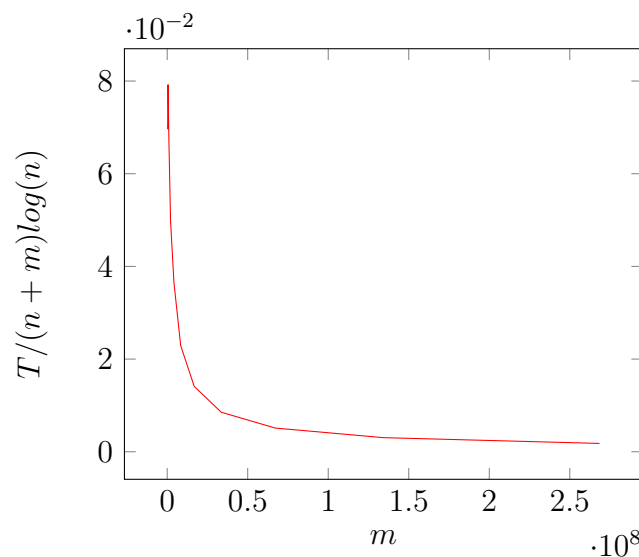


Figura 1.6: Mostra a razão entre o tempo de execução e o custo teórico esperado

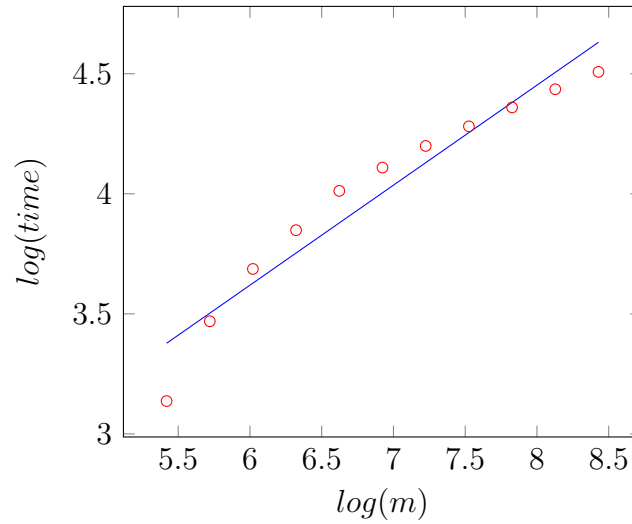


Figura 1.7: Regressão linear. O número de vértices foi desconsiderado pois é constante.

### 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  $2^{18}$ . O custo teórico é o mesmo do caso anterior,  $E = (n + m) * \log(n)$ . Encontrou-se um problema onde o grafo se tornava excessivamente esparsa e o algoritmo de dijkstra encerrava sem percorrer parte significativa do grafo. Poucas amostras foram obtidas como fica evidente na figura 1.8.

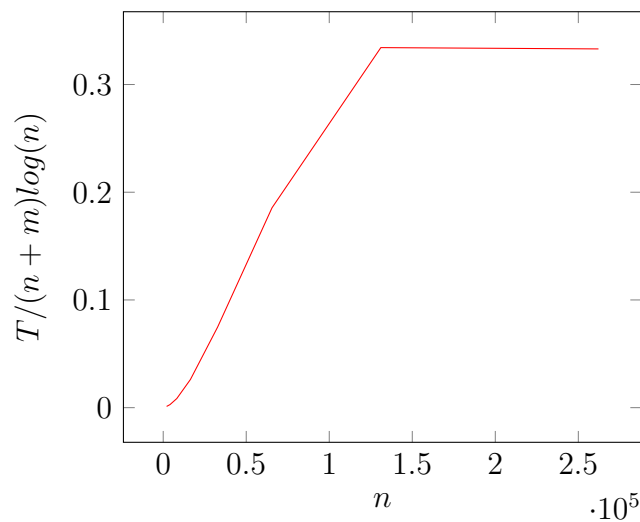


Figura 1.8: Mostra a razão entre o tempo de execução e o custo teórico esperado

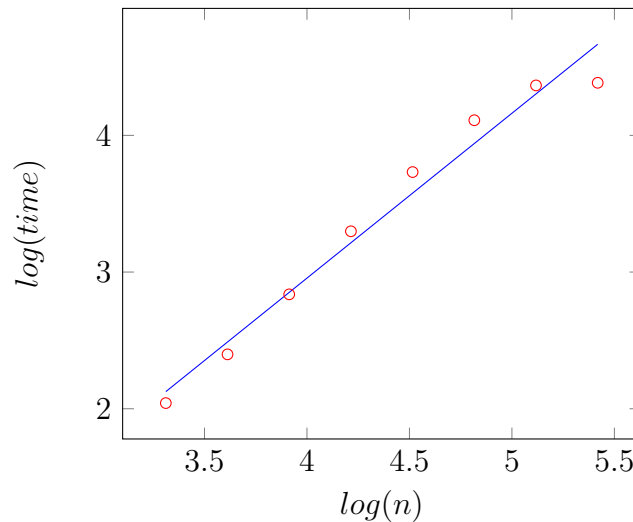


Figura 1.9: Regressão linear. O número de arestas foi desconsiderado pois é constante.

## 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. Para o caso de teste consistindo da rede completa dos Estados Unidos provida na definição do trabalho, obteve-se um tempo médio de execução de 169 segundos e consumo de memória máxima de 3.64 GiB. Para a rede de Nova York, foram 1.83 segundos e 43.7 MB de memória.

## 1.7 Testando Grau Ótimo da Heap

Testou-se a implementação da heap variando-se o grau da mesma e usando as redes de Nova York e dos Estados Unidos assim como outras geradas aleatoriamente com uma versão modificada do algoritmo gerador oferecido. Um comportamento interessante foi obtido. Para as redes reais, os resultados foram melhores para graus 4 e 8. Entretanto, para as artificiais a tendência foi de quanto maior o grau da heap, melhor o desempenho ao executar o algoritmo de Dijkstra. A figura 1.10 mostra os resultados. Os tempos de execução de cada grafo foram normalizados em relação ao tempo médio obtido para aquela rede. Acredita-se que esse comportamento possa ser explicado pela aleatoriedade na inserção de arestas o que resulta numa topologia diferente daquelas encontradas em redes reais.

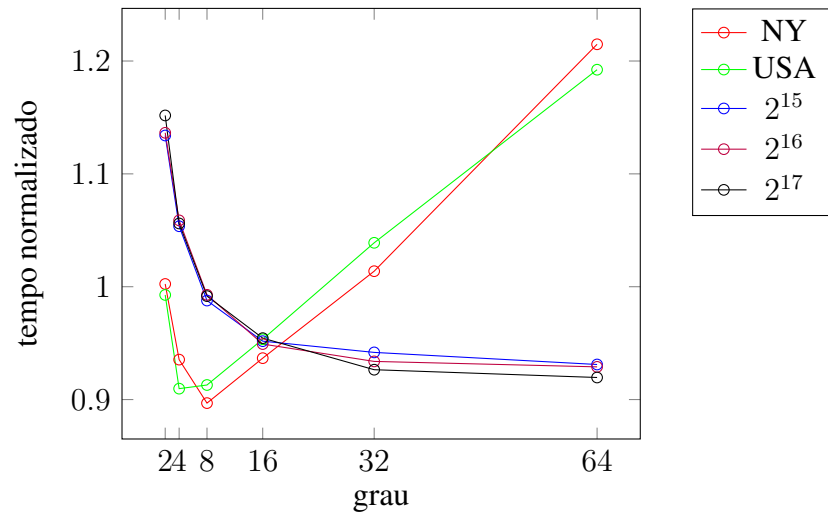


Figura 1.10: Tempo normalizado de execução do algoritmo de Dijkstra

## 1.8 Conclusão

Excluindo-se o caso do tempo de execução das atualizações quando comparado com a expectativa teórica, as implementações dos algoritmos respeitaram a complexidade teórica dos mesmos. Procurou-se, também, o grau da heap com o valor mais adequado ao algoritmo de Dijkstra. Chegou-se a conclusão que a rede com a qual se testa pode influenciar os resultados desse teste.