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

THIAGO BELL

Laboratório 1

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 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 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.

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 16.04.

1.4 Análise de Complexidade da Heap

A complexidade teórica das operações da heap foram comparadas com os resultados de experimentos. Os quais respeitam a expectativa teórica.

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 o tempo T_i e o número de swaps I_i onde $0 <= i <= 27$. A complexidade dessa operaçao é de $O(2^i \ \dot{i})$. Com isso pode-se estimar o custo $E = 2^i \ \dot{i}$ para cada i. Na figura 1.1 pode-se ver o número de swaps e E em função de 2^i. Para ficar claro que a complexidade é respeitada, a curva E*15 mostra os valores de E multiplicados por uma constante. 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.

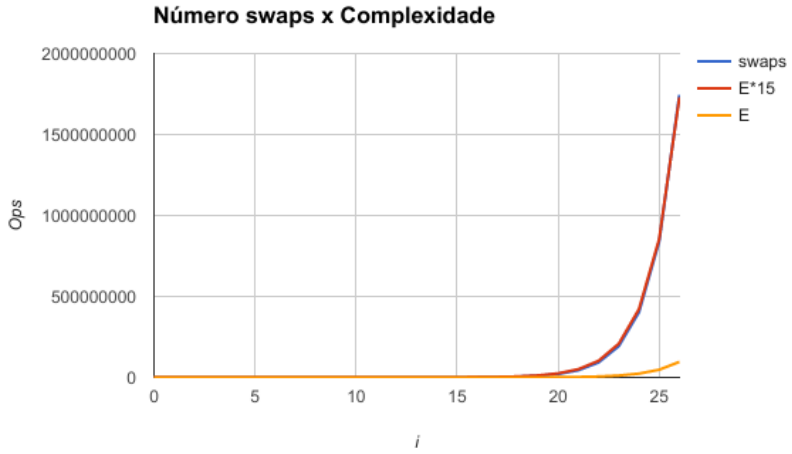

Figura 1.1:

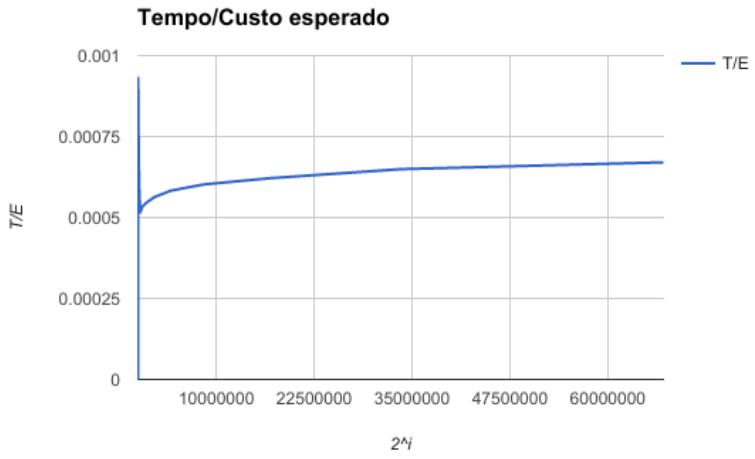

Figura 1.2:

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 $E = 2^i\dot{i}$. 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.

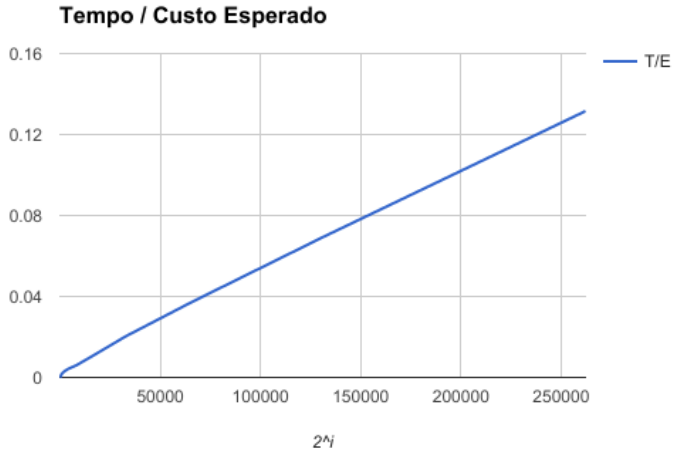

Figura 1.3:

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.

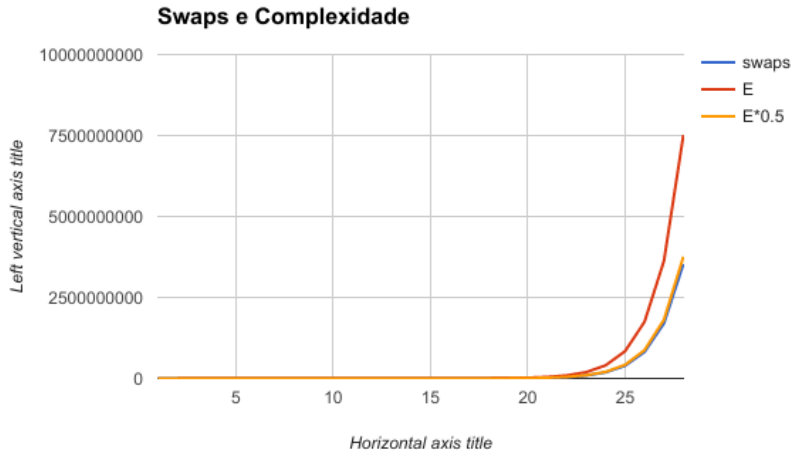

Figura 1.4:

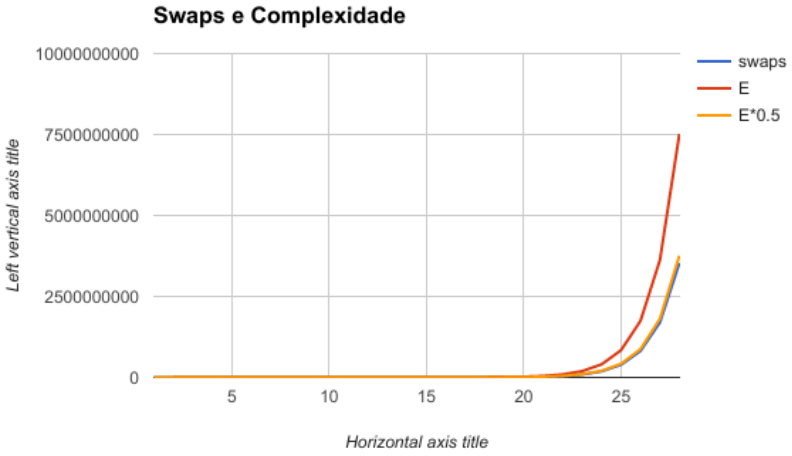

Figura 1.5:

1.5 Análise de Complexidade do Algoritmo de Dijkstra

1.5.1 Variando o número de Arestas

Fixou-se o número de vértices em 2^{15} e variou-se o número de arestas entre 2^{18} e 2^{24}. 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 update menores ou iguais ao de arestas.

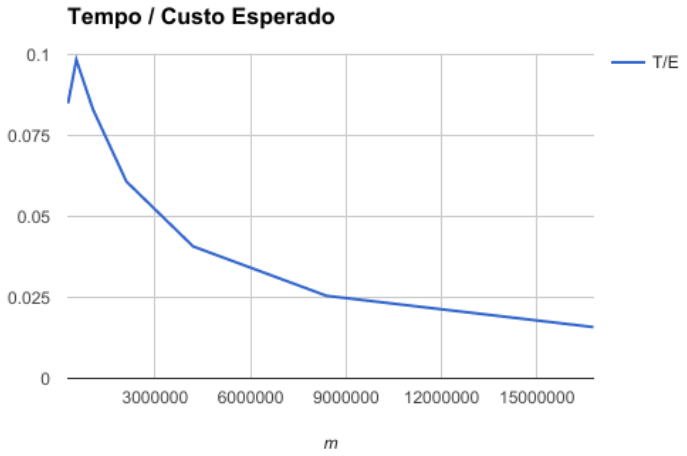

Figura 1.6:

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 tornavo excessivamente esparso e o algoritmo de dijkstra

encerrava sem percorrer parte significante do grafo. Poucas amostras foram obtidas como fica evidente na figura 1.7.

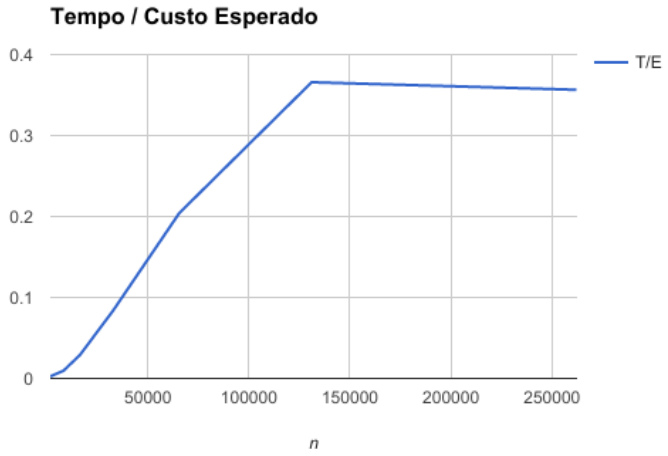

Figura 1.7:

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 memómia máximo de 3.64 GB. 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 aleatóriamente geradas 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. Entretando, para as artificiais a tendência foi de quanto maior o grau da heap, melhor o desempenho ao executar o algoritmo de Dijkstra. A figura 1.8 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 atribuição de grafos o que resulta numa topologia diferente.

1.8 Conclusão

Excluindo-se o caso onde não se obteve resultados experimentais satisfatórios e o onde a medida de tempo divergiu da de swaps, as implementações dos algoritmos respeitaram a complexidade teórica dos mesmos.

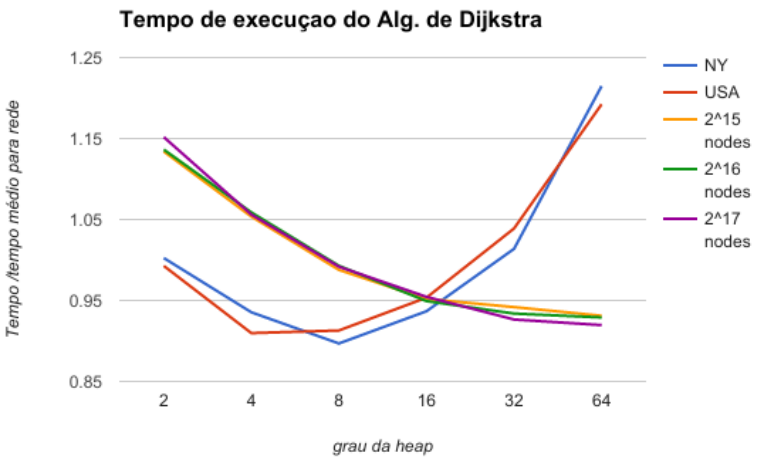
Tempo de execuçao do Alg. de Dijkstra
Tempo /tempo médio para rede
1.25
1.15
1.05
0.95
0.85
2
4
8
16
32
64
grau da heap
NY
USA
2^15 nodes
2^16 nodes
2^17 nodes

Figura 1.8: