UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL KENJI YATSUDA IKUTA

Trabalho 1

Porto Alegre
2025

1.1 Tarefa

Implementar o Algoritmo de Dijkstra que permite o cálculo do valor do caminho mínimo entre um vértice e todos os outros de um grafo direcionado ponderado dado, com complexidade de tempo $O((n+m)log(n))$, onde n é o número de vértices e m é o número de arestas. Queremos validar que a complexidade de tempo é de fato $O((n+m)log(n))$.

1.2 Solução

1.2.1 Implementação

Implementei o algoritmo de Dijkstra em `C++`, utilizando uma fila de prioridades baseada em um heap k-ário. Um dos principais desafios foi determinar o valor ideal de k, o que foi feito por meio de testes experimentais, variando k de 2 a 64. Os testes foram realizados em um conjunto de 48 grafos com diferentes quantidades de vértices e arestas, depois é feito esse mesmo conjunto de testes em um segundo dataset gerado da mesma maneira.

A estrutura dos grafos foi representada por listas de adjacências. Para otimizar o desempenho do heap k-ário, foi utilizado um vetor auxiliar (hashing por índice) de tamanho n — onde n é o número de vértices — permitindo buscas em tempo constante $O(1)$.

Além disso, foi utilizado um `Makefile` para agilizar o processo de compilação, bem como scripts em `bash` para facilitar a execução dos testes e a geração dos grafos no formato DIMACS. Por fim, a linguagem `Python` foi utilizada para processar os dados obtidos e gerar os gráficos que auxiliaram na análise dos resultados.

1.2.2 Criações do Datasets Iniciais

Estes datasets de grafos foi gerado a partir de um código adaptado de um gerador no formato DIMACS, no qual os parâmetros de entrada são o número de vértices e a probabilidade p de criação de uma aresta entre cada par possível de vértices. Foram utilizados grafos com número de vértices dado por 2^i, com $i \in [8, 15]$, para cada um dos seguintes valores de probabilidade: $p \in \{0{,}1;\ 0{,}2;\ 0{,}3;\ 0{,}4;\ 0{,}5;\ 0{,}6\}$.

Dois conjuntos de dados foram gerados utilizando a mesma lógica, com o objetivo de permitir comparações e fornecer insights adicionais. Embora o código utilizado para a geração de ambos os conjuntos seja o mesmo, a aleatoriedade introduzida pelas funções `lrand48()` e `drand48()` (da biblioteca `<cstdlib>`) resulta em grafos estruturalmente distintos entre si, mesmo sob os mesmos parâmetros de entrada.

1.3 Ambiente de Testes

1.3.1 Hardware Utilizado

Os resultados foram obtidos em um sistema com a seguinte configuração de hardware:

- **Processador:** AMD Ryzen 5 5600X
  - Frequência base: *3.7GHz*
  - Frequência máxima (Turbo): *4.6GHz*
  - Cache total: *35MB*

    - Núcleos: *6*
    - Threads: *12*
    - Plataforma: **AM4**
- **Placa-mãe:** Asus TUF Gaming X570-Plus
- **Memória RAM:** 32GB DDR4
    - Módulos: XPG Spectrix D50 RGB
    - Frequências: *3000 MHz* e *3200 MHz*
- **Armazenamento:** SSD WD Black SN750 500GB NVMe
    - Velocidade de leitura: *3430MB/s*
    - Velocidade de gravação: *2600MB/s*
- **Placa de vídeo:** Zotac Gaming NVIDIA GeForce RTX 2060
    - Memória: *6GB GDDR6*

1.3.2 Software Utilizado

Os experimentos foram conduzidos em um ambiente baseado no sistema operacional **Ubuntu 24.04.2 LTS** (codinome *noble*), com os seguintes componentes de software:

- **Python 3.12.3** — utilizando ambiente virtual (`venv`) para isolar as dependências do projeto. As principais bibliotecas utilizadas incluem:
    - `pandas`
    - `numpy`
    - `statsmodels`
    - `matplotlib`
    - `seaborn`
    - `scikit-learn` (`sklearn.preprocessing.StandardScaler`)
    - `os`, `io.StringIO`
- **GCC 13.3.0** — compilador utilizado para códigos escritos em C/C++, obtido via pacote oficial do Ubuntu 24.04.

O uso de `venv` permitiu a instalação e gerenciamento das bibliotecas necessárias de forma isolada, garantindo reprodutibilidade e controle sobre o ambiente de execução.

1.4 Resultados

1.4.1 Testes Iniciais

Para cada grafo dos datasets, foram realizados 30 testes com vértices de origem (*source*) e destino (*destination*) escolhidos aleatoriamente. A média das métricas relevantes foi calculada com base nesses testes, incluindo: `avg_insert`, `avg_extract`, `avg_decrease`, `avg_insert_r`, `avg_extract_r`, `avg_decrease_r` e `avg_time` (em microssegundos).

Nos casos em que não havia caminho entre os vértices sorteados, a distância retornada era infinita (representada por `std::numeric_limits<int>::max()`). Nessas situações, a instância era descartada e não considerada no cálculo das médias.

1.4.2 Procurando o melhor K

Com base nos testes realizados, concluiu-se que o valor ótimo para k é 17 para o dataset 1, como podemos ver na (Figura 1.1). A conclusão de que o valor ótimo para k é 17 pode parecer contra-intuitiva, uma vez que 17 não é uma potência de 2, como poderíamos supor inicialmente. Entretanto, esse resultado se justifica, pois para cada combinação de p e quantidade de vértices há um k ótimo específico. Ao calcular a média de todos os casos, o valor resultante pode não corresponder ao k ideal para um determinado par de p e número de vértices, mas representar um desempenho global superior, mesmo que em alguns cenários o desempenho não seja ótimo, mas em casos não extremos ele será satisfatório.

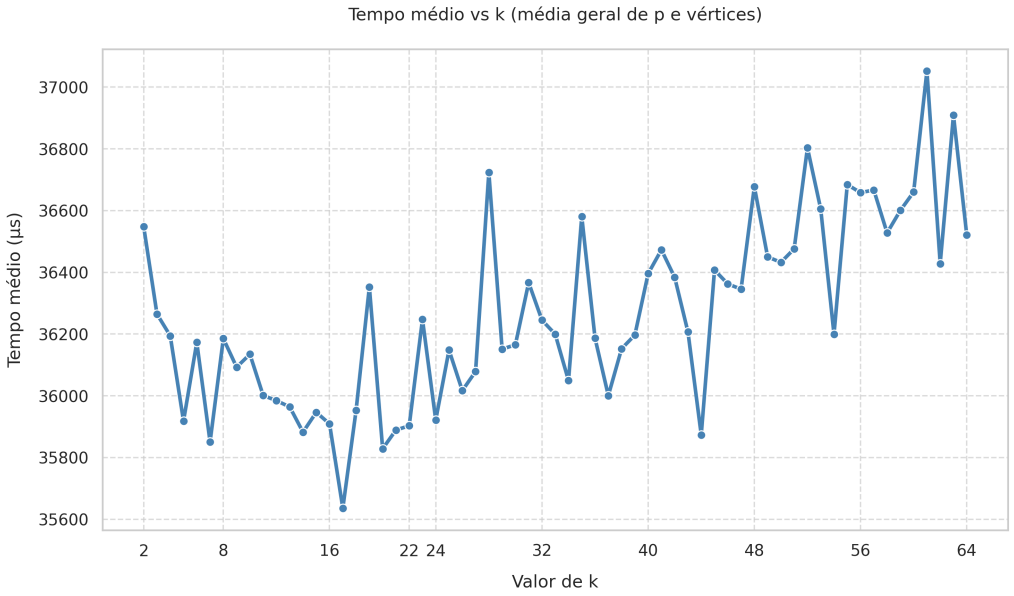

Figura 1.1 – Tempo médio vs k (média geral de p e vértices), `dataset1`

É importante, contudo, fazer algumas ressalvas. Os testes foram conduzidos com grafos de diferentes esparsidades, mas a variedade de instâncias e tamanhos foi limitada, sendo a maioria composta por grafos relativamente pequenos. Mesmo em grafos grandes, o algoritmo de Dijkstra apresenta uma execução extremamente rápida. Dessa forma, o principal fator limitante dos experimentos acabou sendo o tempo necessário para a geração dos grafos e, principalmente, para a leitura dos arquivos a partir do disco, o que tornava o processo consideravelmente mais lento. Por esse motivo, o escopo dos testes precisou ser reduzido para garantir a viabilidade da execução.

Além disso, alguns dos grafos gerados — especialmente os mais densos, com $p = 0{,}6$ — representam instâncias pouco prováveis em aplicações práticas. Esse aspecto também deve ser considerado na interpretação dos resultados obtidos.

1.4.3 Valores Ótimos de k para Diferentes Combinações de Vértices e Densidade p

Nesta seção, analisamos brevemente os resultados para algumas combinações de $p = 0{,}1$ e diferentes quantidades de vértices, com o objetivo de identificar os valores de k que proporcionam o melhor desempenho.

Na Figura 1.2, observamos que o melhor valor de k é 4, apresentando o menor tempo médio de execução.

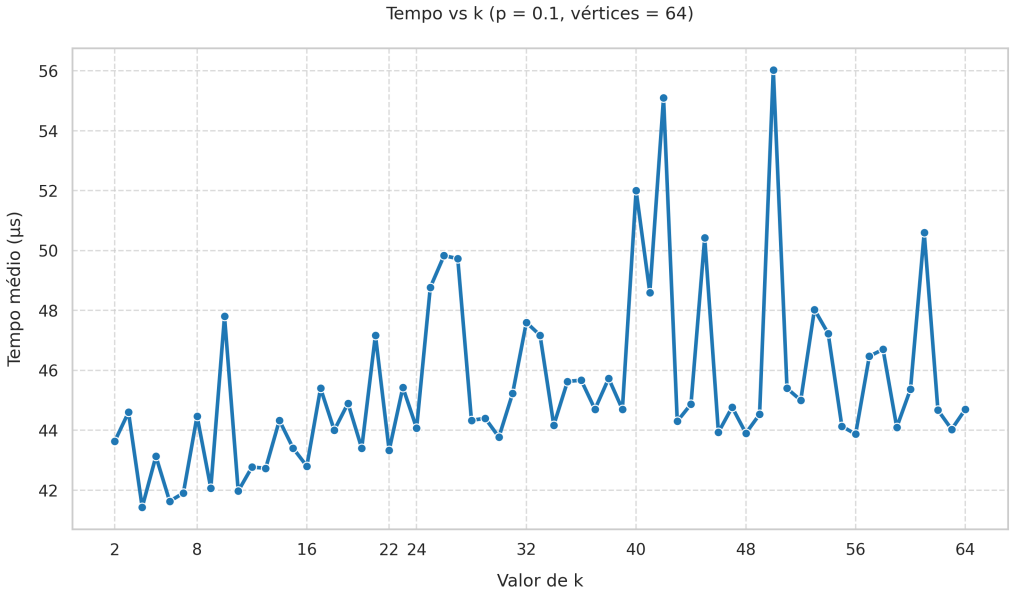

Figura 1.2 – Tempo médio vs k (p = 0,1 e $n = 64$), `dataset1`

Já na Figura 1.3, os menores tempos médios concentram-se entre os valores de k iguais a 6 e 9, indicando uma faixa de desempenho ideal.

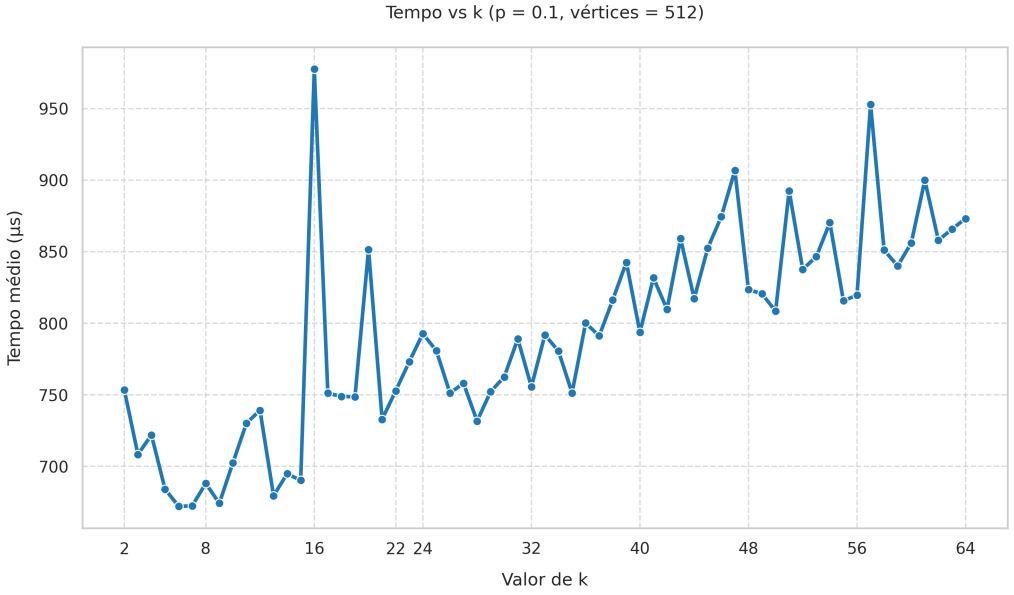

Figura 1.3 – Tempo médio vs k (p = 0,1 e $n = 512$), `dataset1`

Na Figura 1.4, o valor ótimo de k é 7, sendo o que apresentou melhor desempenho médio.

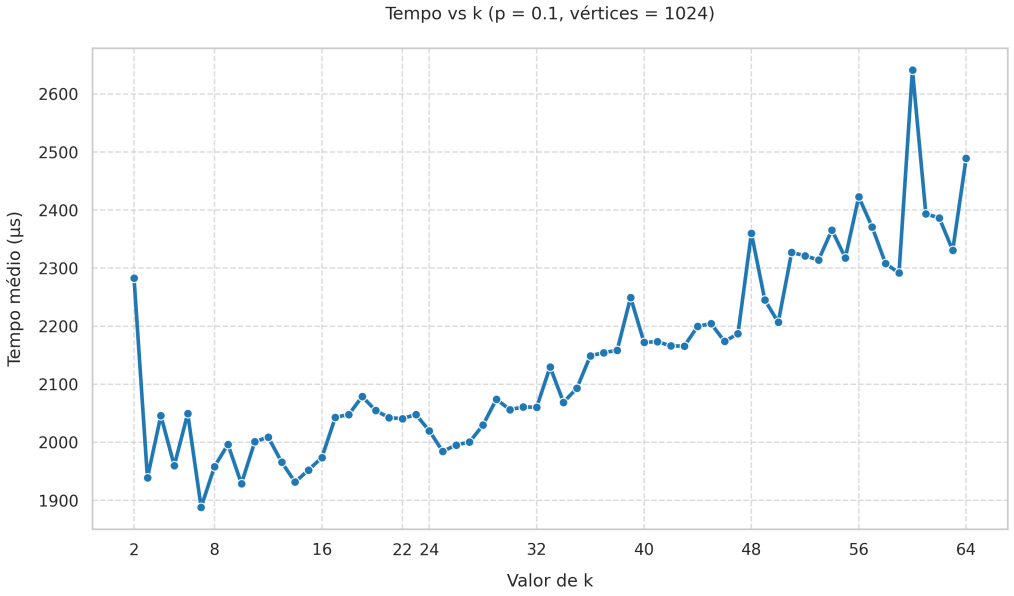

Figura 1.4 – Tempo médio vs k (p = 0,1 e $n = 1024$), `dataset1`

Por fim, na Figura 1.5, o valor de k que obteve o menor tempo médio foi 16, indicando um melhor equilíbrio entre largura da árvore e profundidade.

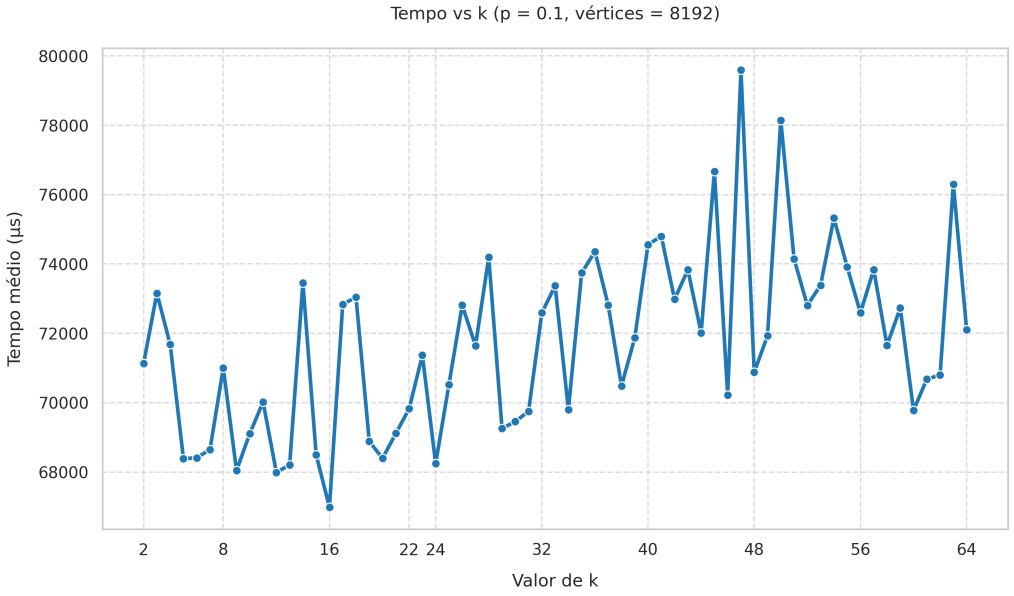

Figura 1.5 – Tempo médio vs k (p = 0,1 e $n = 8192$), `dataset1`

Com base nos gráficos apresentados acima, podemos concluir que a escolha de $k = 17$ não é absurda. A partir deste ponto, adotaremos esse valor de k para as próximas análises, como a avaliação da complexidade por meio da contagem de operações.

No entanto, é importante destacar que os resultados podem apresentar certo grau de ruído devido a diversos fatores, como limitações do hardware, variações na carga do sistema e outros aspectos mencionados anteriormente. Esse comportamento pode ser observado com mais clareza na Figura 1.6, que indica que, em média, o valor $k = 5$ seria o mais eficiente ao fazer os mesmos testes e gráficos com o `dataset2`.

Apesar disso, optamos por manter o valor $k = 17$ nos experimentos subsequentes. Essa decisão evidencia o quão volátil pode ser a escolha do melhor valor de k, especialmente considerando a limitação do tamanho dos datasets utilizado.

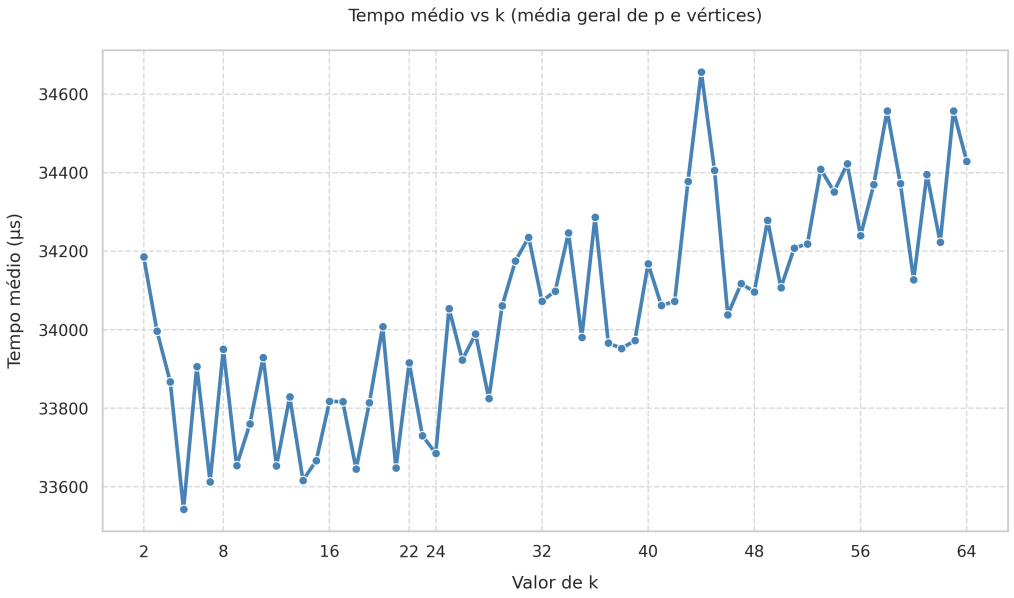

Figura 1.6 – Tempo médio vs k (média geral de p e vértices), `dataset2`

1.4.4 Complexidade por contar operações

O estudo realiza uma análise comparativa das operações computacionais em diferentes níveis de implementação, no nível do heap e no nível do algoritmo, identificando e quantificando os principais pontos de impacto no desempenho.

1.4.4.1 Nível do heap

No nível do heap, foram analisadas as operações elementares *sift-up* e *sift-down*, que são responsáveis por manter as propriedades da estrutura de heap k-ário.

A operação *sift-up* ocorre quando um elemento é inserido ou tem sua prioridade reduzida (como no caso de um `update`), sendo necessário movê-lo para cima na hierarquia do heap. Já a operação *sift-down* acontece quando o elemento no topo do heap é removido (como no `deletemin`), sendo necessário reorganizar a estrutura movendo elementos para baixo.

Cada operação principal do heap possui um número máximo de operações de *sift* no pior caso:

- `insert`: até $\log_k n'$ operações de *sift-up*;
- `deletemin`: até $\log_k n'$ operações de *sift-down*;
- `update` (com redução de prioridade): até $\log_k n'$ operações de *sift-up*.

Aqui, n' representa o número de elementos presentes no heap no momento da operação.

Para cada operação, foi computado o número real de *sift* executado e comparado com o limite superior teórico. Assim, foi definida a razão:

$$r = \frac{\#\text{sift}}{\log_k n'}$$

Essa razão foi calculada individualmente para cada operação e está restrita ao intervalo $0 \leq r \leq 1$. A média dos valores de r, denotada por $\bar{r}$, foi utilizada como uma medida empírica da complexidade das operações no heap.

Todas as estatísticas mencionadas foram obtidas por meio de experimentos conduzidos com uma série de grafos aleatórios, variando-se as quantidades de vértices n e

arestas m. Para cada configuração de grafo, os testes foram repetidos 30 vezes, utilizando diferentes pares de vértices de origem (*source*) e destino (*destination*) escolhidos aleatoriamente. Esse procedimento visou garantir uma maior robustez e confiabilidade na análise dos resultados obtidos.

p	n	m	avg_insert_r	avg_delete_r	avg_update_r
0.1	64	400	0.168153	0.681510	0.111398
0.1	128	1600	0.126828	0.810807	0.055844
0.1	256	6460	0.096795	0.873307	0.044764
0.1	512	26241	0.089693	0.815603	0.038663
0.1	1024	104833	0.076851	0.824165	0.030974
0.1	2048	420016	0.065414	0.843547	0.029517
0.1	4096	1675722	0.057726	0.815993	0.032279
0.1	8192	6709288	0.053656	0.657172	0.043427
0.2	64	810	0.178969	0.730469	0.110387
0.2	128	3268	0.135955	0.815104	0.058370
0.2	256	13202	0.108647	0.876628	0.037448
0.2	512	52547	0.090222	0.807498	0.034372
0.2	1024	209602	0.075896	0.804568	0.032405
0.2	2048	838614	0.068179	0.788946	0.034996
0.2	4096	3354962	0.060435	0.693884	0.045002
0.2	8192	13416779	0.054362	0.489975	0.066594
0.3	64	1164	0.171968	0.713802	0.071777
0.3	128	4859	0.126255	0.823568	0.041602
0.3	256	19350	0.105702	0.871289	0.037823
0.3	512	78134	0.088211	0.792220	0.032306
0.3	1024	314848	0.078749	0.777968	0.035365
0.3	2048	1258105	0.067770	0.743338	0.040773
0.3	4096	5032428	0.060723	0.599392	0.059086
0.3	8192	20131244	0.054329	0.400573	0.055710

Tabela 1.1 – Tabela com p, n, m, avg_insert_r, avg_delete_r, avg_update_r

p	n	m	avg_insert_r	avg_delete_r	avg_update_r
0.4	64	1561	0.175623	0.721124	0.103253
0.4	128	6562	0.131756	0.825984	0.049213
0.4	256	26147	0.107325	0.873611	0.035229
0.4	512	105896	0.088134	0.786932	0.036485
0.4	1024	426382	0.075493	0.758126	0.039872
0.4	2048	1693025	0.067289	0.715892	0.044975
0.4	4096	6789216	0.059876	0.562342	0.062148
0.4	8192	27102852	0.052137	0.380914	0.078965
0.5	64	1964	0.169562	0.708962	0.101487
0.5	128	7856	0.130452	0.827698	0.047125
0.5	256	31389	0.104593	0.870542	0.036517
0.5	512	127593	0.087234	0.781219	0.035794
0.5	1024	513789	0.076451	0.749615	0.040278
0.5	2048	2048124	0.067182	0.703128	0.046391
0.5	4096	8194371	0.058732	0.545128	0.068215
0.5	8192	32783945	0.051789	0.370524	0.081349
0.6	64	2489	0.161482	0.701245	0.097248
0.6	128	9834	0.128579	0.824639	0.045872
0.6	256	39547	0.102563	0.867412	0.038215
0.6	512	160482	0.085417	0.772915	0.034126
0.6	1024	645987	0.074583	0.739284	0.039724
0.6	2048	2584721	0.065278	0.692417	0.047512
0.6	4096	10347912	0.056842	0.525489	0.073158
0.6	8192	41382975	0.049278	0.361278	0.087649

Tabela 1.2 – Continuação da tabela com p, n, m, avg_insert_r, avg_delete_r, avg_update_r

1.4.4.2 Nível do algoritmo

No nível do algoritmos temos as operações elementares *insert*, *deletemin*, e *update*, com uma complexidade pessimista de n, n, e m, respectivamente. Então, similar ao caso anterior, podemos observar o número dessas operações dividido pelo pior caso:

$$i = \frac{\#\textbf{insert}}{n}; \quad d = \frac{\#\textbf{deletemin}}{n}; \quad u = \frac{\#\textbf{update}}{m}.$$

p	k	Vértices	Arestas	I	D	U
0.1	17	64	400	1.0	1.0	0.08425
0.1	17	128	1600	1.0	1.0	0.09711
0.1	17	256	6460	1.0	1.0	0.07390
0.1	17	512	26241	1.0	1.0	0.04950
0.1	17	1024	104833	1.0	1.0	0.03125
0.1	17	2048	420016	1.0	1.0	0.01905
0.1	17	4096	1675722	1.0	1.0	0.01124
0.1	17	8192	6709288	1.0	1.0	0.00647
0.2	17	64	810	1.0	1.0	0.09963
0.2	17	128	3268	1.0	1.0	0.07077
0.2	17	256	13202	1.0	1.0	0.04902
0.2	17	512	52547	1.0	1.0	0.03143
0.2	17	1024	209602	1.0	1.0	0.01905
0.2	17	2048	838614	1.0	1.0	0.01125
0.2	17	4096	3354962	1.0	1.0	0.00650
0.2	17	8192	13416779	1.0	1.0	0.00374
0.3	17	64	1164	1.0	1.0	0.08674
0.3	17	128	4859	1.0	1.0	0.05908
0.3	17	256	19350	1.0	1.0	0.03776
0.3	17	512	78134	1.0	1.0	0.02359
0.3	17	1024	314848	1.0	1.0	0.01407
0.3	17	2048	1258105	1.0	1.0	0.00816
0.3	17	4096	5032428	1.0	1.0	0.00472
0.3	17	8192	20131244	1.0	1.0	0.00259

Tabela 1.3 – Valores de I, D e U para diferentes configurações com p de 0.1 a 0.3

p	k	Vértices	Arestas	I	D	U
0.4	17	64	1603	1.0	1.0	0.06948
0.4	17	128	6540	1.0	1.0	0.04846
0.4	17	256	26006	1.0	1.0	0.03181
0.4	17	512	104282	1.0	1.0	0.01918
0.4	17	1024	419239	1.0	1.0	0.01116
0.4	17	2048	1677952	1.0	1.0	0.00649
0.4	17	4096	6709382	1.0	1.0	0.00375
0.4	17	8192	26835878	1.0	1.0	0.00197
0.5	17	64	2006	1.0	1.0	0.06255
0.5	17	128	8169	1.0	1.0	0.04277
0.5	17	256	32633	1.0	1.0	0.02678
0.5	17	512	130731	1.0	1.0	0.01633
0.5	17	1024	523422	1.0	1.0	0.00943
0.5	17	2048	2094765	1.0	1.0	0.00543
0.5	17	4096	8385703	1.0	1.0	0.00307
0.5	17	8192	33553185	1.0	1.0	0.00158
0.6	17	64	2415	1.0	1.0	0.05833
0.6	17	128	9665	1.0	1.0	0.03804
0.6	17	256	39192	1.0	1.0	0.02341
0.6	17	512	156894	1.0	1.0	0.01405
0.6	17	1024	628618	1.0	1.0	0.00824
0.6	17	2048	2514782	1.0	1.0	0.00470
0.6	17	4096	10060505	1.0	1.0	0.00259
0.6	17	8192	40263525	1.0	1.0	0.00132

Tabela 1.4 – Valores de I, D e U para diferentes configurações com p de 0.4 a 0.6

Novamente temos $0 \leq i, d, u \leq 1$ numa implementação correta. No caso das primeiras duas operações e um grafo conexo, a complexidade resulta em $i = d = 1$. O número de **updates** em geral é bem menor.

1.4.5 Complexidade por medir o tempo

Nesta seção, apresentamos os resultados obtidos a partir da média do número de operações *insert*, *deletemin* e *update*, além do tempo médio de execução do algoritmo em dois conjuntos de grafos. Cada instância do experimento foi repetida 30 vezes para cada grafo, garantindo maior precisão nas métricas coletadas. Em cada repetição, foram escolhidos diferentes vértices de origem (*source*) e destino (*destination*).

O primeiro conjunto de grafos mantém o número de vértices fixo, enquanto a quantidade de arestas varia. Já no segundo conjunto, o número de arestas é mantido constante, enquanto o número de vértices é alterado. Os gráficos que analisam as operações seguem as definições de i, d e u apresentadas na Seção 1.4.1.2. Por outro lado, a análise do tempo de execução foi realizada com base no tempo normalizado, definido como:

$$\frac{T}{(n+m)\log n}$$

onde T representa o tempo de execução do algoritmo em microsegundos.

É importante destacar que os tempos medidos não incluem o tempo gasto na leitura e construção do grafo, considerando apenas a execução do algoritmo de Dijkstra. Além disso, a variação no tempo de execução pode ser influenciada por diversos fatores, como o hardware utilizado, o sistema operacional, o compilador empregado e até mesmo

processos concorrentes no ambiente de execução.

Para a realização dos experimentos, foram gerados dois conjuntos de grafos, um com número de vértices fixo e outro com número de arestas fixo. A geração foi feita utilizando um programa específico (`gen`), que recebe como entrada o número de vértices n e a probabilidade de existência de uma aresta p, gerando um grafo no formato DIMACS.

1.4.5.1 Conjunto com n fixo e m variável

No primeiro conjunto de grafos, o número de vértices foi mantido fixo em $n = 2^{15}$, enquanto o número de arestas m variou de acordo com a seguinte relação:

$$m = 2^{i/2}, \quad \text{com } i \in [30, 44]$$

Para cada valor de i, a quantidade de arestas foi truncada para o inteiro mais próximo, garantindo valores discretos para m. A probabilidade p de existência de uma aresta foi calculada como:

$$p = \frac{m}{n(n-1)}$$

Os grafos gerados foram armazenados no diretório `graphs/part2/fixed_vertice/` com a nomenclatura `graph_n<N>_m<M>.gr`, onde N e M representam, respectivamente, o número de vértices e arestas do grafo.

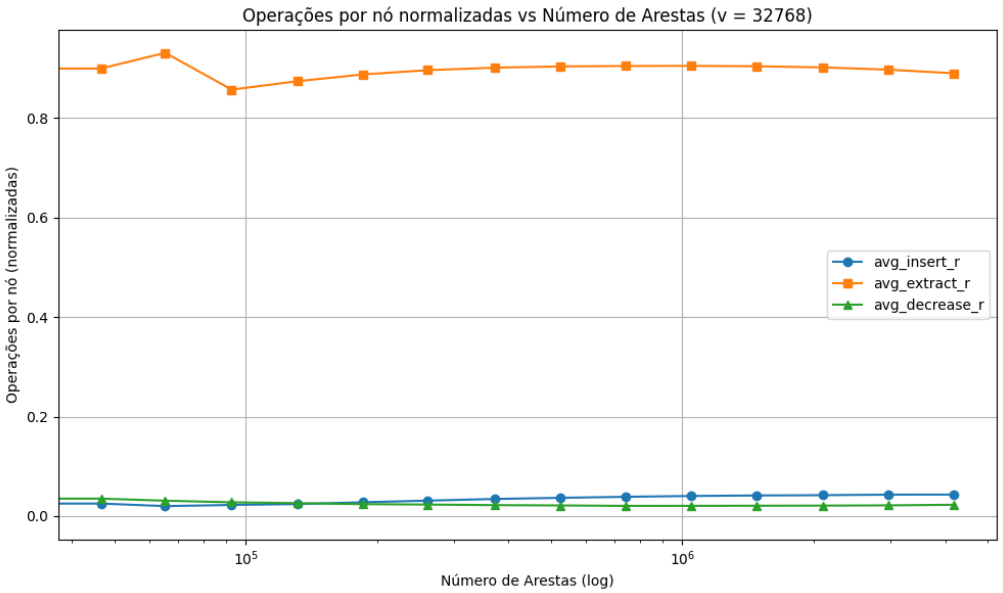

Figura 1.7 – Operações por nó normalizadas vs número de arestas ($V = 32760$).

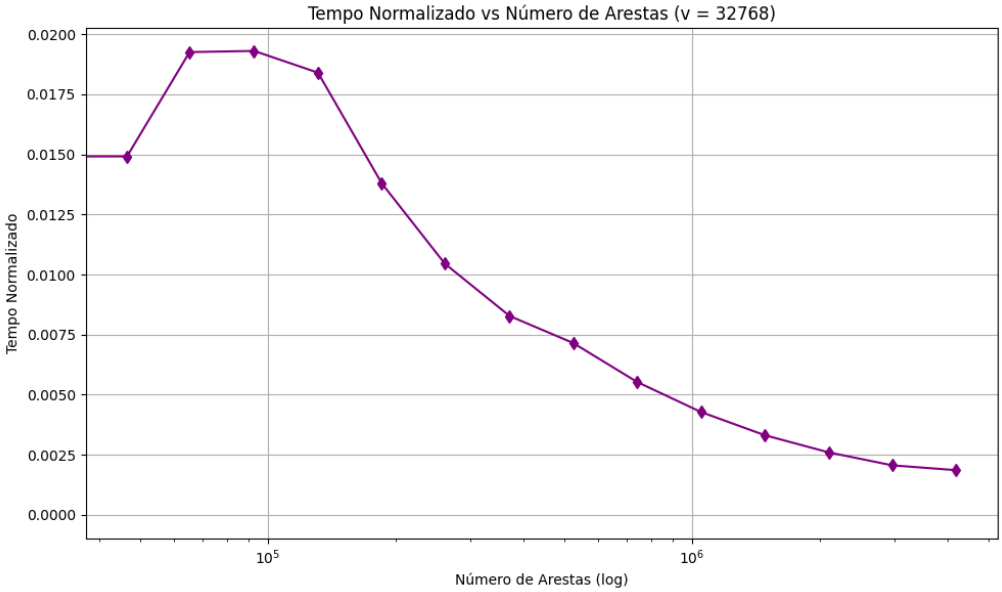

Figura 1.8 – Tempo normalizado vs número de arestas ($V = 32760$).

1.4.5.2 Conjunto com m fixo e n variável

No segundo conjunto de grafos, o número de arestas foi mantido fixo em $m = 2^{20}$, enquanto o número de vértices n variou conforme a seguinte relação:

$$n = 2^{i/2}, \quad \text{com } i \in [21, 35]$$

Assim como no primeiro conjunto, os valores de n foram truncados para os inteiros mais próximos. A probabilidade de existência de uma aresta foi calculada como:

$$p = \frac{m}{n(n-1)}$$

Os grafos desse conjunto foram armazenados no diretório `graphs/part2/fixed_edge/` utilizando a mesma convenção de nomenclatura.

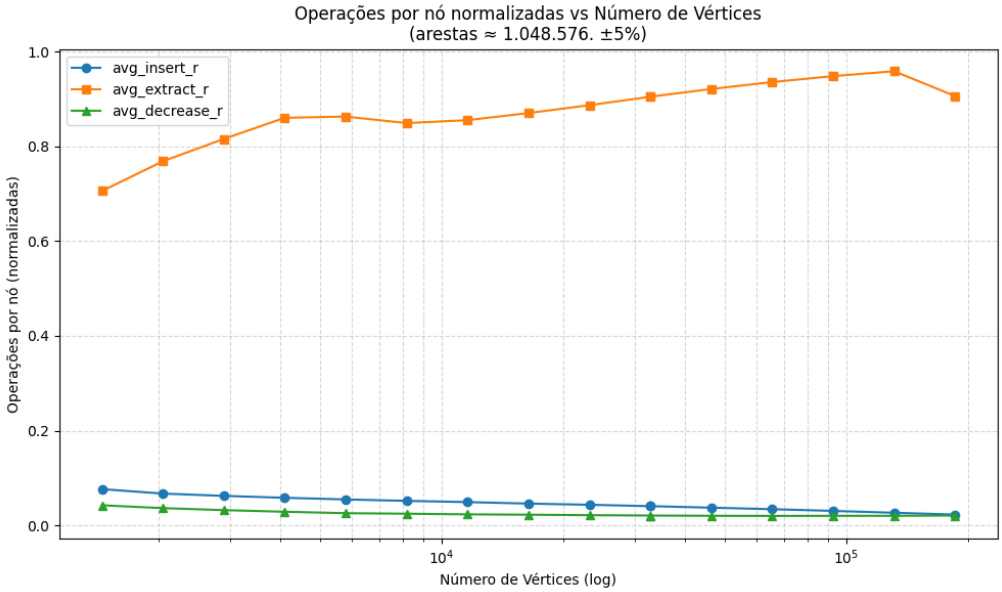

Figura 1.9 – Operações por nó normalizadas vs número de vértices. O número de arestas é aproximadamente $1.048.576$ com uma variação de $\pm 5\%$. São exibidas as razões médias de `insert`, `extract` e `decrease`.

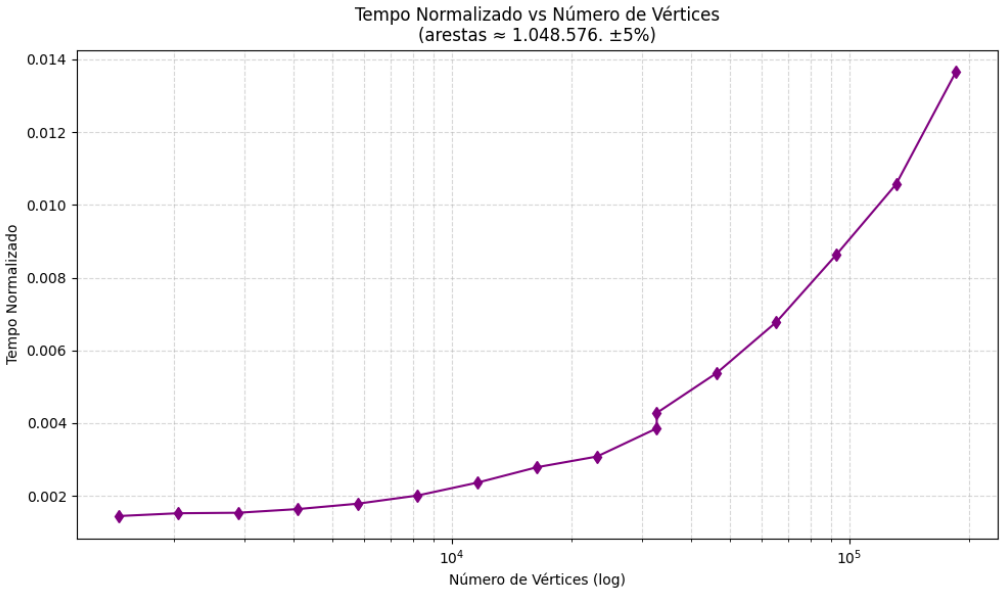

Figura 1.10 – Tempo normalizado vs número de vértices. O número de arestas é aproximadamente $1.048.576$ com variação de $\pm 5\%$.

1.4.5.3 Regressão Linear

Com o objetivo de entender o comportamento assintótico do tempo de execução do algoritmo em função do número de vértices (n) e do número de arestas (m), foi aplicada uma regressão linear log-log aos dados completos dos arquivos A e B. A hipótese é que o tempo médio de execução $T(n, m)$ pode ser modelado por uma função do tipo:

$$T(n, m) \approx a \cdot n^b \cdot m^c$$

Aplicando logaritmo natural em ambos os lados da equação, temos:

$$\log(T) = \log(a) + b \cdot \log(n) + c \cdot \log(m)$$

Ajustando o modelo aos dados disponíveis, obteve-se os seguintes coeficientes:

- $\log(a) = -0{,}1746 \Rightarrow a \approx 0{,}8398$
- Coeficiente de n: $b = 0{,}6022$
- Coeficiente de m: $c = 0{,}3435$

Com isso, a equação final estimada para o tempo de execução é dada por:

$$T(n, m) \approx 0{,}8398 \cdot n^{0{,}6022} \cdot m^{0{,}3435}$$

Essa equação indica que o tempo de execução cresce mais rapidamente com o número de vértices do que com o número de arestas, embora ambos exerçam influência significativa no desempenho do algoritmo. Essa observação é coerente com os dados apresentados nas Figuras 1.8 e 1.10, nas quais se percebe que o tempo normalizado aumenta de forma mais acentuada com a elevação do número de vértices do que com o aumento no número de arestas — ao menos no contexto do **Dataset** utilizado, que é a junção do dataset de vértices fixos e do de arestas fixas.

1.4.6 Avaliando o escalonamento do algoritmo Dijkstra

Para avaliar o desempenho do algoritmo de Dijkstra, realizamos 30 execuções para cada instância de grafo, sempre utilizando o mesmo vértice de origem. Os grafos utilizados foram obtidos a partir da página do *DIMACS Challenge* (<https://www.diag.uniroma1.it/~challenge9/download.shtml>), que disponibiliza grafos reais com grande volume de dados.

A medição do tempo de execução foi feita em microssegundos, e o uso médio de memória foi calculado com base no *Resident Set Size* (RSS), por meio da leitura do arquivo `/proc/self/statm`. A função `memory_used(true)` retorna o número de páginas residentes multiplicado pelo tamanho da página (obtido com `sysconf(_SC_PAGESIZE)`), convertendo o valor para kilobytes. A média foi calculada a partir das 30 execuções de cada instância.

Grafo	**n**	**m**	**T (μs)**	**Memória (KB)**
NY	264346	733846	207044	2776
USA	23947347	58333344	20825053	106130

Tabela 1.5 – Resultados da execução do algoritmo de Dijkstra em dois grafos.

1.4.6.1 Análise dos Resultados

A Tabela 1.5 apresenta os resultados da execução do algoritmo de Dijkstra em dois grafos reais. Observa-se que o tempo de execução cresce quase linearmente com o número de vértices e arestas, o que está de acordo com a complexidade esperada do algoritmo. Embora o grafo *USA* seja aproximadamente 90 vezes maior em vértices que o grafo *NY*, o tempo de execução aumentou cerca de 100 vezes, o que sugere boa escalabilidade da implementação.

O uso de memória também cresce proporcionalmente, mas em menor escala: o grafo *USA* consome cerca de 38 vezes mais memória que o grafo *NY*, indicando que o custo computacional está mais associado ao número de operações da heap do que ao espaço ocupado por estruturas auxiliares. Esses resultados demonstram que a implementação é eficiente e adequada para grafos de grande porte.

1.4.7 Conclusão

Com base nos experimentos realizados, foi possível validar a eficiência do algoritmo de Dijkstra implementado com um heap k-ário, confirmando sua complexidade teórica de $\mathcal{O}((n + m) \log n)$. A escolha do valor $k = 17$ mostrou-se ideal para um desempenho global satisfatório no caso do `dataset1`, embora em casos específicos outros valores de k possam ser mais adequados e em outros `datasets` se encontrem outros valores de k mais interessantes. A análise das operações no heap e no algoritmo revelou que o número de atualizações (*updates*) é significativamente menor que o pior caso teórico, contribuindo para a eficiência prática da implementação.

Os resultados também demonstraram que o tempo de execução é influenciado tanto pelo número de vértices quanto pelo número de arestas, com um crescimento mais acentuado em relação ao número de vértices, conforme indicado pela regressão linear. Além disso, o algoritmo apresentou um desempenho consistente mesmo em grafos de grande escala, como os exemplos `"NY"` e `"USA"`, mantendo um uso de memória razoável.

Em resumo, a implementação proposta atende aos requisitos de complexidade e eficiência, sendo adequada para aplicações práticas em grafos de diferentes tamanhos e densidades.