# Análise assintótica e empírica de filas de prioridade avançadas aplicadas ao Algoritmo de Dijkstra

**Ana Carla Fernandes**[1], **Lucas Castro**[1], **Thailsson Clementino** [1], **Rosiane de Freitas**[1]

[1] Instituto de Computação – Universidade Federal do Amazonas (IComp/UFAM)
Manaus – AM, Brasil
{ana.fernandes, rosiane}@icomp.ufam.edu.br

***Abstract.*** *The shortest path problem is fundamental to numerous computational applications. Despite theoretical advances, Dijkstra's Algorithm with priority queue optimization remains one of the best-performing practical solutions. This work theoretically and experimentally compares heaps and bucket queues, exploring optimizations that arise from monotonicity and graph weight constraints. The results highlight the potential of bucket queues in sparse graphs with integer weights.*

***Resumo.*** *O problema do caminho mínimo é fundamental em diversas aplicações computacionais. Apesar dos recentes avanços teóricos, o Algoritmo de Dijkstra otimizado com filas de prioridade permanece como a solução de melhor desempenho prático. Este trabalho compara teórica e empiricamente heaps e bucket queues, explorando otimizações resultantes da monotonicidade e da restrição dos pesos dos grafos. Os resultados atestam o potencial das bucket queues multinível em grafos esparsos com pesos inteiros.*

## 1. Introdução

O Problema do Caminho Mínimo (do inglês, *Shortest Path Problem* (SPP)) é um problema fundamental em teoria dos grafos. Dado um grafo direcionado ponderado $G = (V, A, w)$, onde $V$ é o conjunto de vértices, $A$ é o conjunto de arestas, o SPP consiste em encontrar um caminho de menor custo entre um vértice de origem $s \in V$ e um vértice de destino $t \in V$ [Madkour et al. 2017]. Algoritmos que solucionam esse problema estão presentes em diversas aplicações, de sistemas autônomos em transporte [Grujic and Grujic 2025] a algoritmos de processamento de dados e aprendizado de máquina [Tenenbaum et al. 2000]. Os primeiros algoritmos para solução do SPP surgiram nos anos 1950. Desde então, foram propostas novas soluções e otimizações que exploram os avanços teóricos e tecnológicos da computação.

Um dos algoritmos pioneiros para o SPP foi proposto por Dijkstra (1959), utilizando uma estratégia gulosa para encontrar o caminho mínimo de única origem em um grafo com pesos não negativos. Por décadas, o Algoritmo de Dijkstra combinado com fila de prioridade, como o *heap* binário [Williams 1964], foi considerado ótimo por parte da comunidade científica para essa variação do problema, com custo $O((n + m) \log n)$ ($n = |V|$ e $m = |A|$). Porém, Duan et al. (2025) proporam um novo algoritmo com custo $O(m \log^{2/3} n)$ para grafos esparsos. Apesar do avanço teórico, Castro et al. (2025) apontam que ainda existe uma lacuna em sua aplicação prática devido ao *overhead* do padrão de acesso à memória e às altas constantes ocultas. Nesse sentido, o Algoritmo de Dijkstra permanece relevante em aplicações reais.

O maior gargalo do Algoritmo de Dijkstra é a necessidade de selecionar o vértice de menor distância à origem a cada iteração, etapa que pode ser otimizada por uma estrutura de dados auxiliar: a fila de prioridade. O *heap* binário [Williams 1964] foi a primeira fila de prioridade aplicada a esse algoritmo. Desde então, outras estruturas foram propostas para essa finalidade [Brodal 2013]. Este trabalho discute e compara filas de prioridade e seus impactos no desempenho teórico e prático do Algoritmo de Dijkstra.

## 2. O Algoritmo de Dijkstra

Costa et al. (2025) descrevem o Algoritmo de Dijkstra como um algoritmo de *label-setting*, cuja solução para o SPP consiste em construir uma árvore $T_s$ enraizada em $s$ que contém os menores caminhos de $s$ para todos os vértices alcançáveis por ele. Ao mesmo tempo, calcula-se $d_s(v)$, a distância entre $s$ e $v$ para cada $v \in V$:

1. Define-se $T = \{s\}$, $d_s(s) = 0$ e $d_s(v) = \infty, \forall v \in V(G) \setminus \{s\}$;
2. Seleciona-se uma aresta $uv$, que conecta um vértice $u \in T$ a um vértice $v \in V(G) \setminus T$ e que minimiza $d_s(u) + w(uv)$. Adiciona-se $uv$ em $T_s$ e define-se $d_s(v) = d_s(u) + w(uv)$;
3. Repete-se o passo 2 até que $T = V(G)$ — ou seja, até que $T$ contenha todos os vértices de $G$ alcançáveis por $s$.

Nota-se que o ponto crítico desse processo é o passo 2, em que ocorre a seleção da aresta de menor peso a cada iteração. Na proposição original de Dijkstra (1959), essa etapa era realizada por meio de uma busca sequencial, com complexidade total $O(n^2+m)$.

## 3. Filas de prioridade aplicadas ao Algoritmo de Dijkstra

Knuth (1998) define fila de prioridade como uma estrutura de dados abstrata que mantém um conjunto cuja ordem de remoção dos elementos depende das chaves associadas ao invés do momento em que eles foram inseridos. Uma fila de prioridade mínima deve suportar as seguintes operações:

- *insert(a, Q)* adiciona o elemento $a$ no conjunto $Q$;
- *extractMin(Q)* remove e retorna o elemento de maior prioridade do conjunto $Q$;
- *decreaseKey(a, k, Q)* atualiza a chave do elemento $a$ para $k$.

A operação *decreaseKey* foi proposta por Fredman e Tarjan (1987) para o *heap* de Fibonacci. As filas de prioridade não necessariamente suportam essa operação. No Algoritmo de Dijkstra com *heap* binário é comum que, ao invés de atualizar as chaves, os vértices sejam reinseridos quando um novo melhor caminho para um vértice ativo é encontrado, o que aumenta o consumo de memória e a quantidade das operações. Apesar desses ônus, o *heap* binário, frequentemente supera o *heap* de Fibonacci devido à sua simplicidade e melhor eficiência no uso de memória *cache* [Larkin et al. 2014].

Brodal (2013) documenta a vasta diversidade de implementações de filas de prioridade, desde o *heap* binário até estruturas mais complexas como o *heap* de Fibonacci. A maioria delas atende propósitos gerais, porém, observar as demandas específicas do Algoritmo de Dijkstra amplia as possibilidades de otimização. As filas de prioridade monotônicas são estruturas otimizadas para o caso em que nunca será inserido um elemento cuja chave seja menor do que a chave do último elemento removido [Thorup 2000]. Essa restrição ocorre no Algoritmo de Dijkstra, pois as chaves inseridas correspondem à distância acumulada de cada vértice para a origem. Costa, Jonas et al. (2025) descrevem algumas dessas estruturas e suas aplicações ao SPP, como as *bucket queues* multinível.

## 4. Análise Assintótica das Filas de Prioridade para o Algoritmo de Djkstra

As principais filas de prioridade são os *heaps*, estruturas baseadas em árvores e que mantêm a prioridade entre os elementos por meio de trocas entre os nós. Larkin et al. (2014) separam os *heaps* em implícitos e explícitos. Os *heaps* implícitos utilizam travessia hierárquica em uma estrutura contígua na memória, enquanto os explícitos utilizam alocação de nós e ponteiros. A principal vantagem dos *heaps* implícitos é a melhor localidade de cache e menor sobrecarga, enquanto os *heaps* explícitos alcançam melhores limites teóricos por meio de operações como o *decreaseKey*.

O *heap* binário é frequentemente representado de forma implícita, consistindo em uma árvore binária, na qual basta percorrer no máximo sua altura para reajuste das propriedades a cada operação, com custo $O(\log(n))$ [Williams 1964]. É comum que suas implementações não suportem o *decreaseKey*. Em contrapartida, o *Heap* de Fibonacci é implementado de forma explícita e é baseado em uma floresta de árvores binomiais relaxadas. Sua organização permite postergar o ajuste estrutural para o momento do *extractMin*, alcançando $O(1)$ amortizado nas operações mais frequentes: *insert* e *decreaseKey* [Fredman and Tarjan 1987].

Dial (1969) apresentou uma estratégia para grafos com pesos inteiros, que aproveita o padrão monotônico observado no Algoritmo de Dijkstra, utilizando *1-level bucket queue*. Utiliza-se uma fila de prioridade baseada em uma sequência de *buckets*, onde os vértices são organizados conforme a sua distância à origem, possibilitando *insert* e *decreaseKey* em $O(1)$. A operação *extractMin* percorre a estrutura em busca do *bucket* não vazio de menor índice. Em um grafo cujo maior peso é $C$, a monotonicidade garante o uso de apenas $C + 1$ *buckets*, possibilitando *extractMin* em $O(C)$. Denardo e Fox (1979) aprimoraram essa ideia propondo as *bucket queues* multinível, que dividem a estrutura em intervalos para saltar *buckets* vazios consecutivos e reduzir a complexidade assintótica. A Tabela 1, apresenta um quadro comparativo das complexidades assintóticas das operações das filas de prioridades analisadas.

| **Filas de Prioridade** | ***insert*** | ***extractMin*** | ***decreaseKey*** | **Dijkstra** |
|---|---|---|---|---|
| *Heap* Binário | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O((m+n)\log(n))$ |
| *1-Level Bucket* | $O(1)$ | $O(C)$ | $O(1)$ | $O(m+nC)$ |
| *2-Level Bucket* | $O(1)$ | $O(\sqrt{C})$ | $O(1)$ | $O(m+n\sqrt{C})$ |
| *k-level Bucket* | $O(1)$ | $O(k+C^{1/k})$ | $O(1)$ | $O(km+n(k+C^{1/k}))$ |
| *Heap* de Fibonacci | $O(1)$* | $O(\log(n))$* | $O(1)*$ | $O(m+n\log(n))$ |

*Amortizado

**Tabela 1. Complexidades assintóticas das filas de prioridade para um grafo com $n$ vértices, $m$ arestas e peso máximo $C$.**

## 5. Experimentos e Análise Empírica

Muitos estudos investigam a lacuna entre os desempenhos teórico e prático na solução do SPP. Larkin et al. (2014) mostram que *heaps* implícitos apresentam menor tempo de execução e uso de memória que os explícitos em grafos esparsos, enquanto Goldberg e Silverstein (1997) comparam as *bucket queues* multinível entre si. Porém, não foram encontrados estudos comparativos mais abrangentes que incluam essas estruturas.

Neste trabalho, foram implementados em C++20 o *heap* binário e as *bucket queues* de 1 à 6 níveis. As filas foram aplicadas ao Algoritmo de Dijkstra para resolver o SPP em grafos esparsos com pesos inteiros da malha rodoviária dos Estados Unidos

[Demetrescu et al. 2006]. Uma implementação do *heap* de Fibonacci também foi incluída [Lisitsyn et al. 2013]. Os tempos de execução foram medidos em uma máquina com 8 GB de memória, processador Intel Core i7-1165G7 2.80GHz e Linux Ubuntu 24.04.3.

O gráfico da Figura 1 mostra que as *bucket queues* apresentaram menor tempo de execução, com destaque à *1-level bucket queue* com suporte à *decreaseKey*. Isso contradiz o esperado, pois nas instâncias utilizadas, os valores de $C$ são mais de 10 vezes maiores do que os de $\log(n)$, indicando um custo maior que que os *heaps*, conforme a Tabela 1. Essa discrepância pode ser atribuída à simplicidade e melhor localidade de cache das *bucket queues*, além do custo $O(1)$ das operações mais frequentes — *insert* e *decreaseKey*, que contrastam com os custos logarítmicos e amortizados dos *heaps*. Observa-se que mais níveis não necessariamente implicam em melhor desempenho. Por outro lado, os valores de $C$ nessas instâncias não ultrapassam $10^6$. Para pesos maiores, o custo da iteração por grandes intervalos pode superar a sobrecarga de gestão de níveis, o que torna as *bucket queues* multinível preferíveis.

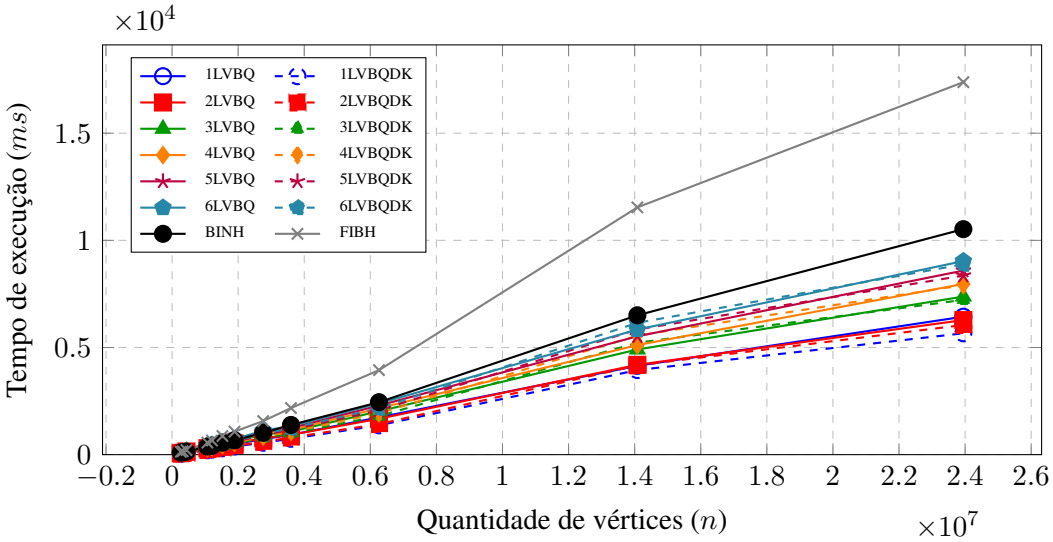


**Figura 1. Tempo médio de execução em função da quantidade de vértices.**

## 6. Considerações Finais

Este trabalho revisitou as *bucket queues* multinível, comparando-as teórica e empiricamente com *heaps* na execução do Algoritmo de Dijkstra para instâncias reais de rodovias. A análise evidencia a discrepância entre os limites teóricos e os resultados empíricos: as *bucket queues*, em especial as com poucos níveis, apresentam menor tempo de execução que os *heaps* nas instâncias avaliadas, apesar do maior custo assintótico. Isso decorre da simplicidade proporcionada pela monotonicidade e restrição de pesos, propriedades teóricas com impacto prático direto e que podem ser exploradas em novas aplicações e estruturas. As *bucket queues* merecem maior atenção em aplicações que utilizem grafos com pesos inteiros, bem como em estudos comparativos mais abrangentes que investiguem eficiência no uso de memória, desempenho em outras classes de grafos e aplicações, além dos impactos da quantidade de níveis e do valor de $C$.

## Referências

Brodal, G. S. (2013). *A Survey on Priority Queues*, pages 150–163. Springer Berlin Heidelberg, Berlin, Heidelberg.

Castro, L., Clementino, T., and de Freitas, R. (2025). Implementation and brief experimental analysis of the Duan et al. (2025) algorithm for single-source shortest paths.

Costa, Jonas, Castro, Lucas, and de Freitas, Rosiane (2025). Exploring monotone priority queues for dijkstra optimization. *RAIRO-Oper. Res.*, 59(5):2419–2436.

Demetrescu, C., Goldberg, A. V., and Johnson, D. S., editors (2006). *9th DIMACS Implementation Challenge: Shortest Paths*. American Mathematical Society.

Denardo, E. V. and Fox, B. L. (1979). Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27(1):161–186.

Dial, R. B. (1969). Algorithm 360: shortest-path forest with topological ordering [h]. *Commun. ACM*, 12(11):632–633.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271.

Duan, R., Mao, J., Mao, X., Shu, X., and Yin, L. (2025). Breaking the sorting barrier for directed single-source shortest paths. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, STOC '25, page 36–44, New York, NY, USA. Association for Computing Machinery.

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615.

Goldberg, A. V. and Silverstein, C. (1997). Implementations of dijkstra's algorithm based on multi-level buckets. In Pardalos, P. M., Hearn, D. W., and Hager, W. W., editors, *Network Optimization*, pages 292–327, Berlin, Heidelberg. Springer Berlin Heidelberg.

Grujic, Z. and Grujic, B. (2025). Optimal routing in urban road networks: A graph-based approach using dijkstra's algorithm. *Applied Sciences*, 15(8).

Knuth, D. E. (1998). *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Massachusetts, 2 edition.

Larkin, D. H., Sen, S., and Tarjan, R. E. (2014). A back-to-basics empirical study of priority queues. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, page 61–72, USA. Society for Industrial and Applied Mathematics.

Lisitsyn, S., Widmer, C., and Garcia, F. J. I. (2013). Tapkee: An efficient dimension reduction library. *Journal of Machine Learning Research*, 14(36):2355–2359.

Madkour, A., Aref, W., Rehman, F., Rahman, A., and Basalamah, S. (2017). A survey of shortest-path algorithms.

Tenenbaum, J. B., de Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323.

Thorup, M. (2000). On ram priority queues. *SIAM Journal on Computing*, 30(1):86–109.

Williams, J. (1964). Algorithm 232: Heapsort. *Communications of the ACM*, 7:347 – 348.