Avaliação do desempenho de *Multilevel Bucket Queues* na determinação de caminhos mínimos em arquiteturas modernas

Ana Carla Fernandes[1] **, Ricardo Miranda**[1] **, Rosiane de Freitas**[1]

[1] Instituto de Computação (IComp) – Universidade Federal do Amazonas (UFAM)
Av. Gen. Rodrigo Octávio, 6200, Coroado I, 69080-900 – Manaus – AM – Brasil

{ana.fernandes, ricardo.filho, rosiane}@icomp.ufam.edu.br

Abstract. *Multilevel bucket queues exploit integer weight constraints to outperform general-purpose priority queues in Dijkstra's Algorithm. However, the hardware-level reasons for this advantage are still poorly understood. This work compares 1- to 6-level bucket queues with binary and Fibonacci heaps in twelve instances of DIMACS road networks. The 1-level variant with decrease-key operation achieves up to 2× speedup compared to the binary heap and 3.2× compared to the Fibonacci heap, which is explained by a smaller number of branch instructions and better cache utilization. For high maximum weights ($C \geq 10^6$), variants with fewer levels exhibit degradation due to memory manipulation, with the 5-level variant showing the best scalability.*

Resumo. *Filas compartimentadas (buckets) multinível exploram restrições de pesos inteiros para superar filas de prioridades de propósito geral no Algoritmo de Dijkstra. Entretanto, as razões em nível de hardware para essa vantagem ainda são pouco compreendidas. Neste trabalho são comparadas as filas de buckets de 1 a 6 níveis com heaps binário e de Fibonacci, em doze instâncias de redes viárias do Benchmark DIMACS. A variante de 1 nível com operação decrease-key alcança até 2× de aceleração em relação ao heap binário e 3,2× em relação ao heap de Fibonacci, o que se explica por um menor número de instruções de desvio (branch instructions) e por uma melhor utilização do cache. Para pesos máximos elevados ($C \geq 10^6$), variantes com menos níveis apresentam degradação devido à manipulação da memória, sendo a variante de 5 níveis a que se observou oferecer melhor escalabilidade.*

1. Introdução

O problema do caminho mínimo (do inglês *Shortest Path Problem* – SPP) está presente em diversos contextos, como redes de telecomunicações, sistemas de transporte e ciência de dados. Diante dessas demandas e do volume abundante de dados na atualidade, as soluções exigem precisão e baixo custo computacional. Nesse sentido, o Algoritmo de Dijkstra (1959) se destaca ao utilizar uma estratégia gulosa para encontrar o caminho mínimo de uma única origem para todos os vértices de um grafo com pesos não negativos. Seu maior gargalo é a necessidade de selecionar o próximo menor passo a cada iteração, etapa otimizada com uma estrutura de dados auxiliar: a fila de prioridade. A eficiência dessa estrutura determina o desempenho do algoritmo.

Por mais de meio século o Algoritmo do Dijkstra foi considerado por muitos com uma solução ótima para o problema do caminho mínimo em $O(m + n\log(n))$ para um grafo com n vértices e m arestas. Porém, Duan et al. (2025) surpreenderam a comunidade científica como uma solução em $O(m\log^{2/3} n)$, apesar de que Castro et al. (2025) apontarem que, devido às altas constantes ocultas, o novo algoritmo não supera o desempenho do Algoritmo de Dijkstra, apesar da menor complexidade assintótica.

Situação similar ocorre com estruturas de dados, como filas de prioridade. A implementação pioneira foi o *heap* binário, proposto por Williams (1964). Apesar da diversidade de novas estruturas propostas desde então [Brodal 2013], o *heap* binário permanece como a escolha predominante em aplicações reais, como na Biblioteca STL da Linguagem C++ [cppreference.com 2024]. No entanto, explorar características específicas do problema e as restrições nos dados de entrada pode viabilizar soluções especializadas com desempenho superior, especialmente quando também se consideram aspectos práticos da execução, como constantes ocultas e hierarquia de memória.

Tendo isso em vista, este trabalho tem como objetivo investigar o desempenho prático das *multilevel bucket queues* de Denardo e Fox (2007) aplicadas ao Algoritmo de Dijkstra. Foram realizados experimentos empíricos com grafos esparsos com pesos inteiros não negativos para compará-las a outras filas de prioridade. Este artigo está estruturado como segue: na Seção 2 são apresentados os conceitos teóricos necessários, como o Algoritmo de Dijkstra, filas de prioridade e as *bucket queues*; na Seção 3 discute-se o problema e os trabalhos relacionados; na Seção 4 é descrito o projeto de experimentos; na Seção 5 são apresentados e analisados os resultados; por fim, na Seção 6 são apresentadas as considerações finais e direções para trabalhos futuros.

2. Fundamentação Teórica

Nesta seção são apresentados os conceitos nos quais este trabalho se baseia.

2.1. O Custo Dominante do Algoritmo de Dijkstra

O problema do caminho mínimo pode ser definido da seguinte forma: seja um grafo $G(V, A, w)$, em que V é o conjunto de vértices, A o conjunto de arestas ponderadas e w os pesos das arestas, um vértice de origem s e um vértice de destino t. Encontre o menor caminho entre s e t. A função d_s indica a distância para o vértice de origem [Madkour et al. 2017].

O Algoritmo de Dijkstra resolve esse problema para grafos com pesos não negativos utilizando uma estratégia gulosa. $S(v) \in \{inalcançável, não\ completo, completo\}$ indica o *status* dos vértices e $\pi(v)$, o vértice pai de v. Inicialmente, todos os vértices possuem $d_s(v) = \infty$ e $S(v) = inalcançável$. O algoritmo começa definindo $d_s(s) = 0$ e $S(s) = não\ completo$. A cada iteração, seleciona-se um vértice $não\ completo$ com menor $d(v)$ e examina-se todas as arestas $A(vu)$. Se $d(v) + w(vu) < d(u)$, essa se torna a nova distância $d(u)$, $S(u)$ é definido $não\ completo$ e $\pi(u)$ é definido como v. Após processar todos os seus vértices vizinhos, $S(v)$ se torna $completo$. O algoritmo termina quando não há mais vértices $não\ completos$ a serem processados [Goldberg and Silverstein 1997].

O custo computacional do algoritmo é dominado pela busca pelo vértice $não\ completo$ de menor distância estimada a cada iteração. No algoritmo original

[Dijkstra 1959], essa etapa é realizada por meio de uma busca sequencial em um vetor, resultando na complexidade total $O(n^2 + m)$. A otimização com filas de prioridade permite manter os vértices de forma a facilitar o acesso e remoção daquele de menor chave.

2.2. Estratégias de Implementação de Filas de Prioridade

Filas de prioridade são estruturas de dados abstratas que mantém um conjunto cuja ordem de remoção dos elementos depende das chaves associadas ao invés do momento em que eles foram inseridos [Knuth 1998]. A prioridade pode ser baseada no maior ou menor valor. Neste trabalho consideram-se apenas as filas de prioridade mínima, que devem 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.

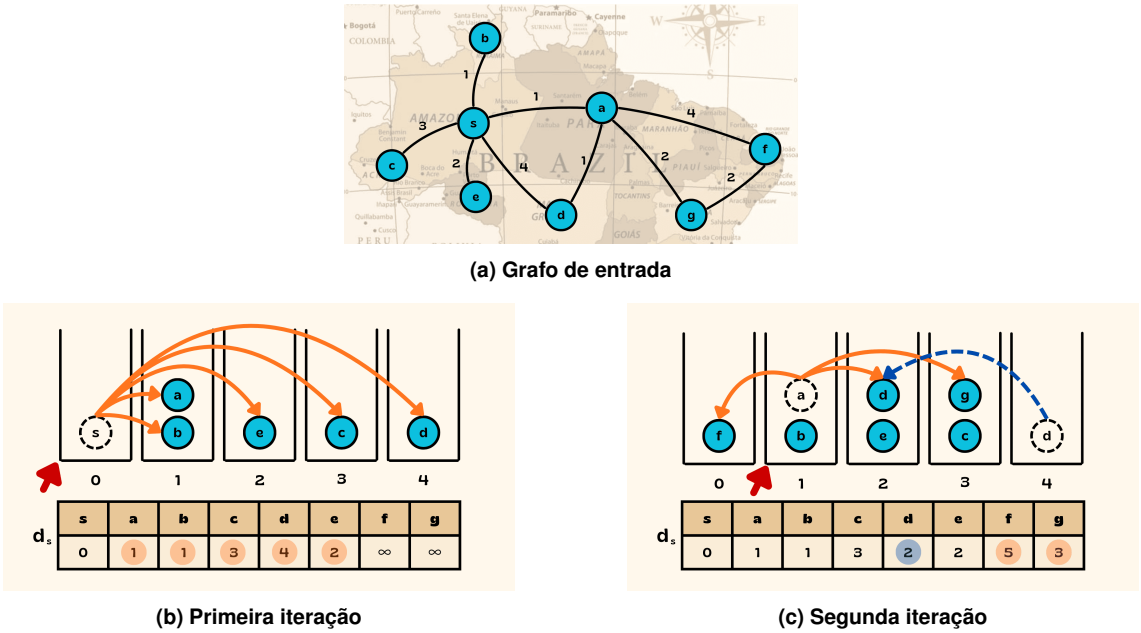
As principais implementações de filas de prioridade são os *heaps*, estruturas baseadas em árvores e que mantêm suas propriedades por meio de trocas entre os nós. Os *heaps* podem ser classificados como implícitos e explícitos [Larkin et al. 2014]. Os *heaps* implícitos se baseiam em estruturas contíguas na memória e utilizam aritmética de índices para gerenciar a hierarquia entre os nós. Os *heaps* explícitos são baseados em alocação de nós e ponteiros, mantendo a hierarquia por meio de referências diretas.

Os *heaps* explícitos permitem maior flexibilidade para realizar operações mais complexas, o que lhes proporciona alcançar melhores limites teóricos. Por exemplo, o *heap* de Fibonacci introduziu a operação *decreaseKey* [Fredman and Tarjan 1987]. Sem ela, ocorrem inserções repetidas (*lazy*) quando um novo melhor caminho para um vértice que já está presente na fila é encontrado. Essa estratégia aumenta o consumo de memória da estrutura, além de necessitar de mais operações para gerenciar elementos extras na fila. Comparações entre as duas estratégias são abordadas nas Seções 3.1 e 5.

2.3. Monotonicidade e *Multilevel Bucket Queues*

As filas de prioridade monotônicas são estruturas em que a chave mínima não decresce ao longo das operações [Thorup 2000]. Em outras palavras, após a remoção de um elemento com determinada chave, nenhum elemento posteriormente inserido possuirá chave menor que essa. Essas estruturas apresentam otimizações para casos que obedecem a essa restrição, como o Algoritmo de Dijkstra, em que a chave de um novo vértice v inserido é a distância acumulada da origem até ele: $d_s(u) + w(uv)$.

Costa, Jonas et al. (2025) descrevem algumas filas de prioridade monotônicas e suas aplicações ao algoritmo de Dijkstra. Dentre elas, eles explicarm as *multilevel bucket queues*. Essas filas são baseadas em *buckets*, estruturas dinâmicas que armazenam os elementos e cujos índices são associados à prioridade na fila. A aplicação das *bucket queues* ao problema do caminho mínimo foi proposta por Dial (1969). O algoritmo de Dial é baseado no de Dijkstra e utiliza a *1-level bucket queue*, estrutura composta por um *array* de *buckets* B. A Figura 1, exemplifica as iterações do Dial. Na primeira iteração (Figura 1b), o vértice origem s é inserido no *bucket* $B[0]$. Em seguida, seus vértices vizinhos são distribuídos de forma que $B[i]$ contenha apenas os vértices cuja a distância à origem é i. Finalizada a distribuição, s é removido e procura-se o próximo *bucket* não vazio (Figura 1c). O processo se repete até que todos os todos os *buckets* estejam vazios.



(a) Grafo de entrada

(b) Primeira iteração

(c) Segunda iteração

Figura 1. Exemplos de iterações do algoritmo de Dial. d_s registra as menores distâncias encontradas à cada iteração. Essas figuras foram baseadas no trabalho de Costa, Jonas et al. (2025)

O funcionamento do Dial impõe uma restrição aos grafos de entrada: os pesos devem ser inteiros, pois são utilizados para definir o índice na distribuição de elementos. Além disso, surge uma questão: como definir a quantidade de *buckets*? Uma implementação ingênua utilizaria a maior distância possível, $(n-1) * C$ para um grafo com n vértices e com peso máximo C. Esse valor pode ser grande e iterar por essa quantidade de *buckets* seria ineficiente.

Apenas $C+1$ *buckets* são necessários. Suponha que o vértice v $B[i]$ o *bucket* e o vértice $v \in B[i]$ terá seu vizinho u distribuído. A inserção deve ser feita conforme a distância total à origem, ou seja, no índice $k = i + w(vu)$. Se esse valor ultrapassar $C+1$, pode-se utilizar uma operação modular $B[k \mod (C+1)]$ para distribuir os vértices nos *buckets* anteriores à $B[i]$. Esse processo ocorre na Figura 1c. A propriedade circular do conjunto de *buckets* é possível devido à monotonicidade — novos vértices sempre serão inseridos em índices maiores que i. Devido à necessidade de examinar os $C+1$ *buckets*, a complexidade assintótica do Dial é $O(m+nC)$ (Tabela 1).

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.

Denardo e Fox (1979) aprimoraram essa ideia propondo as *multilevel bucket queues*, que organizam os elementos em níveis hierárquicos de *buckets* para evitar a var-

redura de longos intervalos vazios. Em vez de um único conjunto linear de *buckets*, a estrutura divide os *buckets* em segmentos maiores no nível superior e refina progressivamente apenas o segmento ativo nos níveis inferiores, distribuindo os vértices por meio da subrotina expansão. Dessa forma, apenas intervalos relevantes são examinados a cada momento. Essa abordagem aplicada ao algoritmo de Dijkstra alcança custo $O(km + n(k + C^{1/k}))$ para k níveis. A Tabela 1 apresenta as complexidades assintóticas das filas de prioridades mencionadas.

3. O Desempenho Prático de Filas de Prioridade em Arquiteturas Modernas

Existe uma lacuna na conversão de resultados teóricos em desempenho prático. Brodal (2013) relata a evolução das filas de prioridade, comentando os avanços teóricos e seus impactos no problema de caminhos mínimos e outras aplicações. No entanto, o autor enfatiza que, devido à complexidade dadas máquinas modernas, melhores modelos teóricos não necessariamente realizam a menor quantidade de instruções possível. Isso ocorre pois os comportamentos dos algoritmos frente à hierarquia de memória e à previsibilidade dos fluxos de controle também são onerosos para o desempenho prático.

Além disso, houve um deslocamento no gargalo do desempenho de sistemas computacionais modernos. Tradicionalmente, a eficiência de um algoritmo era avaliada principalmente pela quantidade de operações computacionais. Contudo, avanços no *hardware*, na capacidade de armazenamento e nas arquiteturas de processamento fizeram com que a velocidade das CPUs crescesse muito mais rapidamente do que a velocidade de acesso à memória [Chen et al. 2007, Roche 2007]. Assim, não é mais suficiente reduzir a quantidade de operações, é necessário também considerar a hierarquia de memória.

As *multilevel bucket queues* apresentam tempos de execução promissores em relação aos tradicionais *heaps* [Cherkassky et al. 1996, Goldberg and Silverstein 1997], no entanto, não foram encontrados trabalhos que explorem em profundidade as razões práticas para esse comportamento. Investigar esse potencial pode contribuir para melhor compreensão dos fatores que determinam o desempenho real de estruturas de dados em arquiteturas modernas, bem como orientar o desenvolvimento de soluções mais eficientes na prática, mesmo quando não apresentam vantagens assintóticas no plano teórico.

3.1. Trabalhos Relacionados

Esta subseção apresenta trabalhos relevantes sobre o tema. A Tabela 2 sintetiza as características dos trabalhos, evidenciando as filas de prioridade e as métricas comparadas. Todos os estudos incluem *k-ary heaps*, que se refere ao *heap* binário e suas variações cujos nós contém k filhos. As métricas de comparação de desempenho mais comuns são tempo de execução e *cache miss*.

Cherkassky et al. (1996) comparam diferentes algoritmos de caminhos mínimos, dentre eles o Algoritmo de Dijkstra com filas de prioridade. Nesse trabalho, a *2-level bucket queue* se destacou. Goldberg & Silverstein (1997) investigaram a influência da quantidade de níveis das *k-level bucket queues* no tempo de execução, além de contarem as quantidades de redistribuições de vértices e exame de *buckets* vazios. As versões com 3 e 4 níveis mostraram-se mais robustas que as implementações com 1 e 2 níveis, mantendo desempenho consistente em uma ampla variedade de entradas.

Os demais trabalhos incluem métricas associadas à arquitetura, como *cache miss*, quantidade de instruções realizadas pelo processador e erros de previsão de desvios (*branch mispredictions*, associadas a *loops* e estruturas condicionais). No entanto, apesar de utilizarem o Algoritmo de Dijkstra como *benchmarking*, esses trabalhos não incluem as *multilevel bucket queues*, seja por incluírem pesos não inteiros ou por considerarem aplicações não monotônicas das filas de prioridade.

Chen et al. (2007) investigam os impactos do suporte ou não à operação *decreaseKey* em contexto *in-core* e *out-of-core*. Os autores mostram que estruturas que utilizam a estratégia de reinserção ao invés de *decreaseKey* apresentaram melhores resultados em ambos contextos devido a maior eficiência de *cache*. Similarmente, Larkin et al (2014) expõem a forte correlação entre a menor taxa de *cache L1 miss* e menor tempo de execução. Mrena et al. (2019) exploram implementações avançadas de filas de prioridade em comparação ao *heap* binário, mostrando que fatores como constantes ocultas e as características dos grafos podem limitar a aplicabilidade prática de estruturas teoricamente mais eficientes e favorecer implementações mais simples.

<table>
<tr><th colspan="3">Filas de Prioridade</th><th>Referências</th><th colspan="4">Métricas</th></tr>
<tr><td rowspan="10">K-ary Heap</td><td rowspan="5">K-level Bucket</td><td>Heap de Fibonacci</td><td rowspan="2">Cherkassky et al. (1996)</td><td rowspan="2"></td><td rowspan="2"></td><td rowspan="4"></td><td rowspan="10">Tempo de Execução</td></tr>
<tr><td>Radix Heap</td></tr>
<tr><td></td><td>Goldberg & Silverstein (1997)</td><td>Movimentação de vértices e Expansões</td><td rowspan="3">Buckets Vazios</td></tr>
<tr><td>Hot Queue</td><td>Cherkassky et al. (1999)</td><td>insert
decreaseKey</td></tr>
<tr><td>Heap de Fibonacci</td><td>Este Trabalho</td><td>Quantidade de instruções
Branch mispredictions</td><td rowspan="8">Cache Miss</td></tr>
<tr><td rowspan="4">Pairing Heap</td><td>Sequence Heap</td><td rowspan="2">Chen et al. (2007)</td><td rowspan="2">Ciclos por operação
insert
decreaseKey</td><td rowspan="6"></td></tr>
<tr><td>Buffer Heap</td></tr>
<tr><td>Heap de Fibonacci</td><td rowspan="2">Larkin et al (2014)</td><td rowspan="2">Linhas de código
Leituras de cache
Branch mispredictions
Instruções de CPU</td></tr>
<tr><td>Heap Binomial</td></tr>
<tr><td rowspan="2"></td><td>Heap de Fibonacci</td><td rowspan="2">Mrena et al.(2019)</td><td rowspan="2">Tamanho da fila</td></tr>
<tr><td>Brodal Queue</td></tr>
</table>

Tabela 2. Comparação de trabalhos sobre filas de prioridade aplicadas ao Dijkstra.

Por fim, este trabalho tem o objetivo de preencher a lacuna na avaliação do desempenho prático das *multilevel bucket queues* em grafos esparsos com pesos inteiros. Diferente dos trabalhos anteriores, foram incluídas métricas relacionadas à hierarquia de memória e processamento, além do tempo de execução e avaliação das operações internas das filas. Também são analisados os impactos do suporte à operação *decreaseKey* e da quantidade de níveis de *buckets*. Essas estruturas foram comparadas aos *heaps* binário e de Fibonacci a fim de investigar e explicar as razões práticas para as diferenças de desempenho observadas.

4. Projeto de Experimentos

Esta seção descreve o projeto experimental adotado. São detalhadas as implementações das filas de prioridade, as características das instâncias utilizadas, as métricas coletadas,

bem como as ferramentas empregadas na medição.

4.1. Implementações

Foram implementadas em Linguagem C++20 as *1-level*, *2-level* e *k-level bucket queues* (para $k = 3, \ldots, 6$), o *heap* binário e o *heap* de Fibonacci [Lisitsyn et al. 2013]. Cada estrutura foi encapsulada em uma interface comum que expõe as operações *insert*, *extractMin* e *decreaseKey*, permitindo substituí-las sem alterar o código do Dijkstra.

As *bucket queues* foram dimensionadas como potências de 2, substituindo a operação de módulo por AND bit a bit no cálculo de índices. Na *1-level bucket queue*, são utilizados $2^{\lceil \log_2 C \rceil}$ *buckets*; na *2-level bucket queue*, cada nível possui $2^{\lceil \log_2 \sqrt{C} \rceil}$ *buckets*; na *k-level bucket queue*, $2^{\lceil \log_2 C^{1/k} \rceil}$ *buckets* por nível. Todas as variantes foram implementadas com e sem suporte à operação *decreaseKey*. Nas versões sem suporte, as filas foram aplicadas ao Dijkstra com reinserção *lazy* em std::vector. Nas versões com suporte, os *buckets* utilizam uma lista duplamente encadeada em *pool* estático para manter as operações em $O(1)$ e evitar alocações dinâmicas durante a execução.

O *heap* binário foi implementado com `std::vector` para armazenamento contíguo. O *heap* de Fibonacci utiliza a variante com *decreaseKey*. A correção de todas as implementações foi verificada por meio da comparação dos vetores de distância resultantes com os produzidos pelo gabarito com `std::priority_queue` da STL do C++, abortando a execução em caso de divergência. Os códigos estão disponíveis publicamente em https://github.com/anacarlaaf/IC-2025-26-pqs-dijsktra-spp.

4.2. *Benchmark* DIMACS e Instâncias sintéticas

Os experimentos de tempo e cache utilizaram grafos de redes rodoviárias dos Estados Unidos disponibilizados pelo *9th DIMACS Implementation Challenge* [Demetrescu et al. 2006], grafos dirigidos e esparsos com pesos inteiros positivos. A Tabela 3 apresenta as características das instâncias utilizadas.

Grafo	n	m	C
NY	264.346	733.846	36.946
BAY	321.270	800.172	49.887
COL	435.666	1.057.066	67.999
FLA	1.070.376	2.712.798	99.807
NW	1.207.945	2.840.208	110.803
NE	1.524.453	3.897.636	108.737
CAL	1.890.815	4.657.742	130.202
LKS	2.758.119	6.885.658	200.374
E	3.598.623	8.778.114	194.862
W	6.262.104	15.248.146	264.628
CTR	14.081.816	34.292.496	376.445
USA	23.947.347	58.333.344	368.855

Tabela 3. Instâncias da malha rodoviária dos EUA do 9th DIMACS Challenge, com número de vértices (n), arestas (m) e maior peso (C).

Para os experimentos de escalabilidade em relação à C, foram utilizados grafos sintéticos produzidos por um gerador inspirado no padrão *DIMACS* [Castro et al. 2025]. O n foi fixado e variou-se o tamanho de C.

4.3. Métricas e Ferramentas de Medição

A máquina utilizada nos experimentos possui 8 GB de memória RAM, processador Intel Core i7-1165G7 2,80 GHz e sistema operacional Linux Ubuntu 24.04.3. A Tabela 4 apresenta as métricas coletadas, organizadas por categoria.

Cada experimento de tempo e cache executou o algoritmo de Dijkstra **10 vezes** por par (grafo, fila), selecionando um vértice de origem aleatório. A adoção de múltiplas repetições segue a prática comum em avaliações empíricas de algoritmos de caminho mínimo, reduzindo a variabilidade experimental e produzindo métricas médias mais estáveis [Demetrescu et al. 2009]. Trabalhos clássicos de engenharia de implementações do algoritmo de Dijkstra, como Cherkassky et al. (1996), também empregam execuções repetidas para comparação de desempenho.

Como todas as estruturas foram avaliadas sob as mesmas condições, eventuais *overheads* introduzidos pelas ferramentas de medição afetam igualmente todos os experimentos, não comprometendo a validade das comparações relativas.

Categoria	**Métrica**	**Ferramenta**
Tempo	Tempo de CPU (ms)	`std::clock_t`
Hierarquia de memória	*Cache L1 miss* *Cache LLC miss* (último cache)	`perf_event_open`
Processador	Ciclos de clock Instruções de CPU *Branch misspredictions*	
Operações da fila	*extractMins* extras (inserções *lazy*) *Buckets* vazios percorridos	Contadores manuais

Tabela 4. Métricas coletadas nos experimentos, organizadas por categoria e ferramenta de medição.

5. Análise dos Resultados

Nesta seção são apresentados os resultados dos experimentos.

5.1. Desempenho em Instâncias Rodoviárias

A Figura 2 mostra que as *bucket queues* apresentaram menores tempos de execução que os *heaps* binário e de Fibonacci em todas as instâncias. Destaca-se que a *1-level bucket queue* com suporte à *decreaseKey* alcançou menor tempo de execução em todas as instâncias. Isso contrasta com o esperado diante dos limites teóricos observados na Tabela 1, Na Tabela 3, é possível observar que C chega a ser mais de dez mil vezes maior que $log(n)$. No entanto, Goldberg & Silverstein (1997) observam que certas instâncias podem favorecer as *bucket queues* de 1 ou 2 níveis. Além disso, as versões com suporte à *decreaseKey* alcançaram tempos de execução similares e até menores do que suas contrapartes sem suporte, contrariando a tendência observada em *heaps* [Chen et al. 2007].

A Tabela 5 elucida esses comportamentos, mostrando as métricas medidas durante a execução de cada fila de prioridade no grafo da malha rodoviária completa dos Estados Unidos. Os valores são as médias relativas aos mínimos obtidos por qualquer fila em cada métrica. As métricas inclusas são, em ordem: tempo de cpu, tempo real, leitura e falhas de *cache L1* e *L2*, total de *branches*, *branch mispredictions*, total de instruções e total de *buckets* vazios examinados.

A fila com menor tempo de execução (1LVBQDK) também é a que apresenta menores *cache L1 miss*, total de desvios e instruções de CPU. O 6LVBQ apresentou menor *cache miss*, apesar de apresentar uma das maiores quantidades de *branches* totais. Isso sugere que, para essas instâncias, eficiência de *cache* não foi determinante para o tempo de execução, diferente dos resultados de Larkin et al (2014).

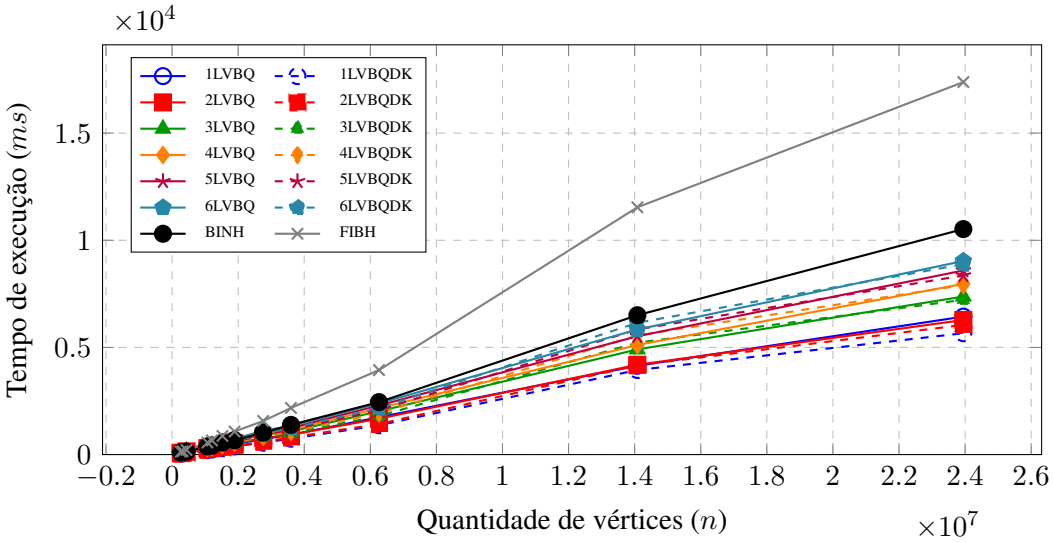


Figura 2. Tempo médio de execução em função da quantidade de vértices.

Fila	tempo	l1_a	l1_m	llc_a	llc_m	b	b_m	inst	bkt_vaz
1LVBQ	1,14	1,14	1,38	1,57	1,53	1,39	1,00	1,20	1,00
1LVBQDK	1,00	1,00	1,60	1,52	1,30	1,00	1,24	1,00	1,08
2LVBQ	1,11	1,52	1,23	1,10	1,05	1,87	1,01	1,58	1,88
2LVBQDK	1,07	1,24	1,75	1,42	1,23	1,11	1,25	1,21	1,88
3LVBQ	1,30	2,04	1,05	1,02	1,01	2,37	1,11	2,12	1,86
3LVBQDK	1,28	1,76	1,79	1,42	1,20	1,50	1,33	1,71	1,75
4LVBQ	1,41	2,30	1,01	1,03	1,02	2,62	1,12	2,38	1,77
4LVBQDK	1,40	2,01	1,87	1,45	1,25	1,63	1,36	1,95	1,63
5LVBQ	1,52	2,51	1,02	1,05	1,05	2,84	1,25	2,61	1,64
5LVBQDK	1,48	2,23	1,90	1,39	1,20	1,74	1,52	2,15	1,59
6LVBQ	1,60	2,72	1,00	1,00	1,00	3,05	1,29	2,83	1,73
6LVBQDK	1,57	2,40	2,02	1,48	1,26	1,84	1,51	2,32	1,36
BINH	1,86	2,95	1,17	1,05	1,05	3,06	3,18	3,00	-
FIBH	3,07	4,43	3,22	2,44	2,09	2,75	5,39	3,28	-

Tabela 5. Métricas médias relativas ao mínimo por fila de prioridade na instância USA. Valores em azul indicam o mínimo e em vermelho, o máximo.

Os resultados do *heap* binário podem decorrer do quantidade extra (cerca de 7% a mais) de operações *extractMin* devido à inserção *lazy*, do custo $O(\log(n))$ da operação *insert* em contraste ao custo $O(1)$ das *bucket queues*, além de apresentar a maior quantidade de *branches*. Isso provavelmente decorre das sucessivas comparações e trocas de posição necessárias para manter a propriedade de *heap* a cada inserção ou remoção, enquanto as *bucket queues* realizam cálculos de índice diretos. Como esperado, o *heap* de Fibonacci apresentou pior eficiência de *cache*, maior quantidade de *branch mispredictions* e instruções.

5.2. Escalabilidade em Relação ao Tamanho de C

As *1* e *2-level bucket queues* não apresentaram boa escalabilidade em relação ao valor de C. A Figura 3 mostra que a partir de 10^6 ambas as filas perdem sua vantagem tanto em relação às contrapartes de maior nível quanto aos *heaps* binário e de Fibonacci. Nesse cenário, a *5-level bucket queue* com suporte à *decreaseKey* (5LVBQDK) apresentou melhor escalabilidade e menor tempo de execução.

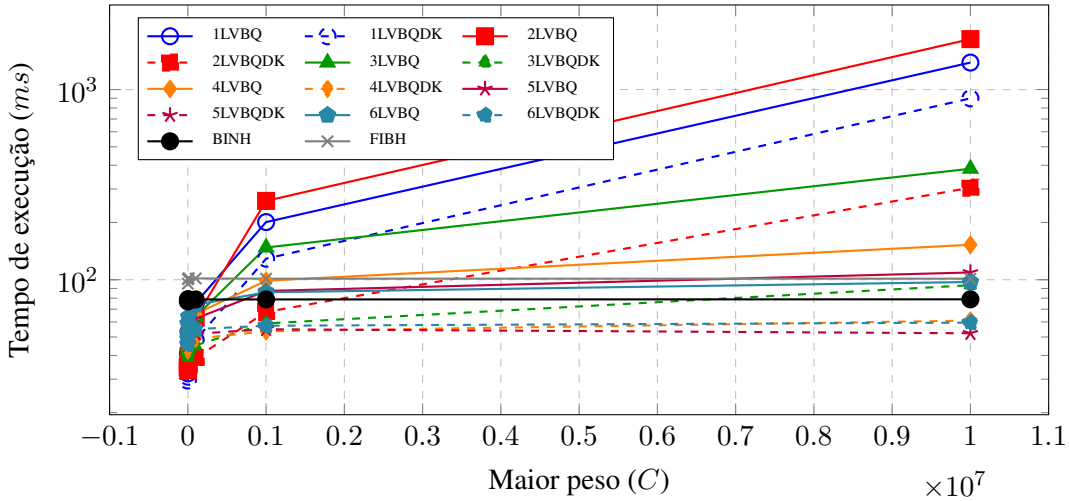


Figura 3. Tempo médio de execução em função do maior peso. Os grafos gerados têm n fixo e o C variam de 10^5 a 10^8.

Fila	tempo	l1_a	l1_m	llc_a	llc_m	b	b_m	inst	bkt_vaz
1LVBQ	26,51	44,43	87,33	3,51	10,07	40,45	1,42	40,77	378,54
1LVBQDK	17,19	32,68	29,63	3,62	10,51	19,86	1,00	24,90	372,49
2LVBQ	35,22	48,82	34,14	1,63	1,69	86,72	1,80	53,36	143,57
2LVBQDK	5,83	11,45	10,90	1,59	1,34	7,19	1,21	7,94	126,91
3LVBQ	7,33	9,68	1,44	1,07	1,07	16,28	1,88	10,46	24,86
3LVBQDK	1,79	2,50	1,49	1,44	1,28	1,75	1,55	1,98	17,27
4LVBQ	2,92	3,56	1,12	1,02	1,03	5,86	1,90	3,85	7,03
4LVBQDK	1,17	1,29	1,40	1,45	1,26	1,18	1,64	1,20	4,17
5LVBQ	2,09	2,40	1,04	1,01	1,02	3,87	2,11	2,63	3,27
5LVBQDK	1,00	1,00	1,30	1,31	1,13	1,00	1,68	1,00	1,57
6LVBQ	1,86	2,08	1,00	1,00	1,03	3,32	2,17	2,31	1,88
6LVBQDK	1,14	1,12	1,49	1,46	1,27	1,14	1,89	1,15	1,00
BINH	1,51	1,41	1,54	1,42	1,00	2,04	4,80	1,56	0,00
FIBH	1,94	1,44	3,00	2,86	2,52	1,30	6,70	1,17	0,00

Tabela 6. Métricas médias relativas ao mínimo por fila de prioridade em grafo sintético com C=10^8. Valores em azul indicam o mínimo; em vermelho, o máximo.

A Tabela 6 mostra que, para grafos com grandes valores de C, as *bucket queues* com menos níveis apresentam menos eficiência de cache. Isso está relacionado à dimensão das estruturas auxiliares. Para o grafo em questão, seriam necessários cerca de 10^8 *buckets* para 1 nível, refletindo em mais verificações de *buckets* vazios, desperdiçando operações e memória. O fato de que as *bucket queues* de 6 níveis não superarem as de 5 níveis mostra que, contrário à intuição proporcionada pelos limites teóricos, mais níveis

não necessariamente proporcionam melhor desempenho. A 5LVBQDK apresentou melhor eficiência de *cache*, além de menos desvios e instruções totais, equilibrando as vantagens da menor quantidade de *buckets* e os custos da gestão de múltiplos níveis.

5.3. Generalização dos Resultados

Os resultados confirmam e justificam o melhor desempenho das *bucket queues* nas instâncias analisadas, o que indica que são recomendadas para determinar caminhos mínimos em problemas cujos grafos são esparsos, com pesos inteiros positivos, graus equilibrados e valores de C moderados em relação a n, como redes rodoviárias. Em grafos com topologia de grade longa (*long grids*) ou com distribuição de pesos não uniforme *bucket queues* podem perder a vantagem em relação aos *heaps* [Goldberg and Silverstein 1997]. Trabalhos futuros podem investigar o desempenho nessas topologias, bem como em grafos com pesos reais e instâncias densas.

6. Considerações Finais

Este trabalho investiga, sob a perspectiva de comportamento de hardware, por que *multilevel bucket queues* superam *heaps* no Algoritmo de Dijkstra para grafos esparsos com pesos inteiros. Os resultados mostram que a principal vantagem está na redução de desvios condicionais, e não apenas na eficiência de *cache*. Além disso, diferentemente dos *heaps*, o suporte a *decrease-key* não penalizou as *bucket queues*. Para $C \geq 10^6$, variantes com poucos níveis sofrem degradação por pressão de memória, enquanto a versão de 5 níveis apresentou melhor escalabilidade. Os achados indicam que eficiência de *cache* e previsibilidade do fluxo de controle devem ser consideradas no projeto de estruturas de dados de alto desempenho.

Como próximos passos, destacam-se a comparação das *bucket queues* com *k-ary heaps* alinhados à hierarquia de memória, *sequence heaps* e *hot queues* sob as mesmas métricas de *hardware*, o que permitiria situar as estruturas avaliadas em um panorama mais amplo de filas de prioridade especializadas. Por fim, a extensão para grafos com outras topologias e densidades, bem como a generalização para pesos reais via mapeamento para domínio inteiro [Otte 2016] e para cenários *out-of-core*, testaria a robustez dos resultados além das condições investigadas neste estudo.

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.

Chen, M., Chowdhury, R. A., Ramachandran, V., Roche, D. L., and Tong, L. (2007). Priority queues and dijkstra's algorithm. Technical report, Computer Science Department, University of Texas at Austin Austin, TX, USA.

Cherkassky, B. V., Goldberg, A. V., and Radzik, T. (1996). Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174.

Cherkassky, B. V., Goldberg, A. V., and Silverstein, C. (1997). Buckets, heaps, lists, and monotone priority queues. Association for Computing Machinery, New York, NY (United States).

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

cppreference.com (2024). std::priority_queue. Acesso em: 30 mar. 2026.

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

Demetrescu, C., Goldberg, A. V., and Johnson, D. S. (2009). The shortest path problem: Ninth dimacs implementation challenge. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 74. AMS.

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.

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.

Mrena, M., Sedlacek, P., and Kvassay, M. (2019). Practical applicability of advanced implementations of priority queues in finding shortest paths. In *2019 International Conference on Information and Digital Technologies (IDT)*, pages 335–344.

Otte, M. W. (2016). On solving floating point SSSP using an integer priority queue. *CoRR*, abs/1606.00726.

Roche, D. L. (2007). Experimental study of high performance priority queues. *University of Texas Computer Sciences Undergraduate Thesis*.

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.