Simpler Universally Optimal Dijkstra

Ivor van der Hoog
IT University of Copenhagen, Denmark

Eva Rotenberg
IT University of Copenhagen, Denmark

Daniel Rutschmann
IT University of Copenhagen, Denmark

Abstract

Let G be a weighted (directed) graph with n vertices and m edges. Given a source vertex s, Dijkstra's algorithm computes the shortest path lengths from s to all other vertices in $O(m + n \log n)$ time. This bound is known to be worst-case optimal via a reduction to sorting. Theoretical computer science has developed numerous fine-grained frameworks for analyzing algorithmic performance beyond standard worst-case analysis, such as instance optimality and output sensitivity. Haeupler, Hladík, Rozhoň, Tarjan, and Tětek [FOCS '24] consider the notion of universal optimality, a refined complexity measure that accounts for both the graph topology and the edge weights. For a fixed graph topology, the universal running time of a weighted graph algorithm is defined as its worst-case running time over all possible edge weightings of G. An algorithm is universally optimal if no other algorithm achieves a better asymptotic universal running time on any particular graph topology.

Haeupler, Hladík, Rozhoň, Tarjan, and Tětek show that Dijkstra's algorithm can be made universally optimal by replacing the heap with a custom data structure. Their approach builds on Iacono's [SWAT '00] working-set bound $\phi(x)$. This is a technical definition that, intuitively, for a heap element x, counts the maximum number of simultaneously-present elements y that were pushed onto the heap whilst x was in the heap. They design a new heap data structure that can pop an element x in $O(1 + \log \phi(x))$ time. They show that Dijkstra's algorithm with their heap data structure is universally optimal.

In this work, we revisit their result. We use a simpler heap property that we will call timestamp optimality, where the cost of popping an element x is logarithmic in the number of elements inserted between pushing and popping x. We show that timestamp optimal heaps are not only easier to define but also easier to implement. Using these time stamps, we provide a significantly simpler proof that Dijkstra's algorithm, with the right kind of heap, is universally optimal.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Shortest paths

Keywords and phrases Graph algorithms, instance optimality, Fibonnacci heaps, simplification

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.71

Related Version *Full Version*: `https://arxiv.org/abs/2504.17327`

Funding *Ivor van der Hoog*, *Eva Rotenberg*, and *Daniel Rutschmann* are grateful to the Carlsberg Foundation for supporting this research via Eva Rotenberg's Young Researcher Fellowship CF21-0302 "Graph Algorithms with Geometric Applications". This work was supported by the the VILLUM Foundation grant (VIL37507) "Efficient Recomputations for Changeful Problems" and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987.

33rd Annual European Symposium on Algorithms (ESA 2025).
Editors: Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman; Article No. 71; pp. 71:1–71:9
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Let $G = (V, E)$ be a (di)graph with edge weights w, n vertices, and m edges. Let s be a designated source vertex. In the *single-source shortest path* problem, the goal is to sort all the vertices in V by the length of the shortest path from s to v. Dijkstra's algorithm with a Fibonacci heap solves this problem in $O(m + n \log n)$ time, which is worst-case optimal.

However, worst-case optimality can be overly coarse as a performance measure. A stronger notion, called *universal optimality*, requires an algorithm to run as fast as possible on *every* graph topology. Recently, Haeupler, Hladík, Rozhoň, Tarjan, and Tětek [6] proved that Dijkstra's algorithm, when paired with a suitably designed heap, achieves universal optimality. The paper was well-received, as it won the best paper award at FOCS'24. Their contribution is threefold: First, they observe that Dijkstra's efficiency relies on a heap supporting Push and DecreaseKey in amortized constant time, and Pop in $O(\log n)$ time. They refine this analysis using the *working-set bound* by Iacono [9] (we forward reference Definition 1). Intuitively, for a heap element x, the working set size $\phi(x)$ is the maximum number of elements y that are simultaneously present in the heap while x resides in it (excluding all y' that already were in the heap). Haeupler et al. [6] design a new heap data structure for in the word-RAM model in which the Pop operation for element x takes $O(1 + \log \phi(x))$ time.

Secondly, they show, for any fixed graph topology $G = (V, E)$ and designated source s, a universal lower bound of $\Omega(m + n + \sum_{x_i \in V \setminus \{s\}} \log \phi(x_i))$. This lower bound matches the running time of Dijkstra's algorithm, when it is equipped with their heap. They thereby prove that Dijkstra's algorithm is universally optimal.

Third, they refine their analysis by distinguishing between edge weight comparisons and all other operations. This argument is technical, and we only refer to it on a high level. For completeness, we explain this result in great detail in the full version. Imagine $m_G \leq m$ to be some technical parameter that only depends on G. Note that, for any fixed weighted graph, Dijkstra's algorithm always performs the same sequence of push and pop operations on the heap. Therefore, the weighted graph uniquely determines the working set size of each element. For a fixed graph topology G, for any weighting w of G with corresponding working set sizes $\phi(x_i)$, they prove a universal lower bound of $\Omega(m_G + \sum_{x_i \in V \setminus \{s\}}(1 + \log(\phi(x_i))))$ on the number of weight comparisons used to solve the single shortest path problem. They adapt Dijkstra's algorithm so that the number of edge weight comparisons performed is also asymptotically universally optimal. A recent paper by Haeupler et al. [7] shows a simple instance optimal result for the bidirectional problem: Compute, given G and two vertices (s, t), the shortest st-path. Although the problems are related, the techniques and bounds are different, and thus, we do not consider the st-path problem in the paper at hand.

Contribution. We offer a significantly more concise proof of Dijkstra's universal optimality. Our main insight is to replace the working-set bound with a conceptually simpler measure called *timestamps*.[1] We maintain a heap H and global time counter that increments after H.Push. For each heap element x_i, let a_i and b_i denote the time it was pushed and popped, respectively. We define H to be *timestamp optimal* if popping x_i takes $O(1 + \log(b_i - a_i))$ time. Timestamps are not only conceptually simpler than working-set sizes, timestamp optimality is also considerably simpler to implement.

[1] We note that the original work by Iacono [10] included several notions of working-set bounds. One of these notions, not present in [6, 9], but used in [5], matches our notion of timestamp optimality. By working set, we explicitly refer to the definition used in [6, 9]. We use the more distinct term timestamps to properly distinguish between the concepts in this work and [6]. This also highlights that our notion yields a family of intervals $[a_i, b_i]$, which is more convenient than having just the working set sizes $\phi(x_i)$.

We design a heap data structure that supports Push and DecreaseKey in amortized constant time, and Pop in amortized $O(1 + \log(b_i - a_i))$ time. Using timestamps, we show a universal lower bound of $\Omega(m + \sum_{x_i \in V \setminus \{s\}}(1 + \log(b_i - a_i)))$ for any fixed graph topology $G = (V, E)$ and source s. We consider this to be our main contribution, as it offers a significantly simpler proof that Dijkstra's algorithm can be made universally optimal. Finally, our formulation of timestamps allow us to apply recent techniques for universally optimal sorting [5, 11] to show that Dijkstra's algorithm, with our timestamp optimal heap, matches our universal lower bound. Additionally, we show in the full version that our algorithm is equally general to that of [6], as it can be made universally optimal with respect to edge weight comparisons. While this proof is indeed new, different, and perhaps simpler, it is not self-contained; therefore, we relegate it to the full version for the sake of completeness.

2 Preliminaries

Let $G = (V, E)$ be a (directed) graph and let w assign to each edge a positive weight. For ease of notation, we say that G has $n + 1$ vertices and m edges. In the *single-source shortest path* problem (SSSP), the input consists of G, w, and a designated source vertex s. The goal is to order all n vertices in $V \setminus \{s\}$ by the length of the shortest path from s to each vertex $v \in V$. We follow [6] and assume that the designated source s can reach every vertex in $V \setminus \{s\}$ and that no two vertices are equidistant from s.

For weighted graph algorithms A that solves SSSP, the running time $\text{Runtime}(A, G, s, w)$ depends on the graph topology $G = (V, E)$, the designated source vertex s, and the edge weights w. Let $\mathbb{G}_n$ denote the set of all (directed) graph topologies $G = (V, E)$ with $|V| = n + 1$. For a fixed graph topology G, let $\mathbb{W}(G)$ denote the (infinite) set of all possible edge weightings of G. Then, for a fixed input size n, the worst-case running time of an algorithm A that solves the single-source-shortest path problem is a triple maximum:

$$\text{Worst-case}(A, n) := \max_{G \in \mathbb{G}_n} \max_{s \in V(G)} \max_{w \in \mathbb{W}(G)} \text{Runtime}(A, G, s, w).$$

An algorithm A is worst-case optimal if there exists a constant c such that, for sufficiently large n, there exists no algorithm A' with $\text{Worst-case}(A, n) > c \cdot \text{Worst-case}(A', n)$.

Haeupler, Wajc, and Zuzic [8] introduce a more fine-grained notion of algorithmic optimality for weighted graphs called *universal optimality*. Universal optimality requires an algorithm to perform optimally for every fixed graph topology. Formally, in the single-source-shortest path problem, it removes the outer two maximizations: For a fixed algorithm A, graph topology $G = (V, E)$, and source s, the universal running time is

$$\text{Universal}(A, G, s) := \max_{w \in \mathbb{W}(G)} \text{Runtime}(A, G, s, w).$$

An algorithm A is *universally optimal* if there exists a constant c such that for every fixed topology G and every $s \in V$, no algorithm A' satisfies $\text{Universal}(A, G, s) > c \cdot \text{Universal}(A', G, s)$.

2.1 Dijkstra's algorithm

Dijkstra [2] presented an algorithm for the single-source shortest paths problem in 1959. Since the invention of Fibonnacci heaps [3], Dijkstra's algorithm can be implemented in a way that is worst-case optimal, by utilising these heaps in the implementation of Dijkstra's algorithm. A heap H maintains a set of elements where each $x \in H$ is associated with a unique *key* $\gamma(x)$. A heap H typically supports the following five operations:

Algorithm 1 Dijkstra's algorithm with a black-box heap H.

```
Require: (Di)graph G = (V, E), weighting w, designated source s ∈ V
  distance ← array with n entries equal to ∞
  H ← empty min-heap
  distance[s] = 0
  H.PUSH(s, 0)
  while H is not empty do
    u ← H.POP()
    OUTPUT.append(u)
    for Each edge (u, v) ∈ E do
      if distance[v] is ∞ then
        distance[v] ← distance[u] + w((u, v))
        H.PUSH(v, distance[v])
      else if v is in H and distance[v] > distance[u] + w((u, v)) then
        distance[v] ← distance[u] + w((u, v))
        H.DECREASEKEY(v, distance[v])
      end if
    end for
  end while
```

- PUSH(x, y): Insert x into H where the key $\gamma(x)$ is y.
- DECREASEKEY(x, y): For $x \in H$ and $y \leq \gamma(x)$, update $\gamma(x)$ to y.
- POP(): Remove and return the element $x \in H$ with the smallest key.
- PEAK(): Return an element $x \in H$ with the smallest key.
- MELD(H'): Return a heap H'' on the elements in $H \cup H'$.

Fibonacci heaps support PUSH and DECREASEKEY in $O(1)$ amortized time, POP in amortized $O(\log n)$ time, and PEAK and MELD in worst-case $O(1)$ time [3]. Dijkstra's algorithm only uses the first three heap operations. Using Fibonacci heaps, it runs in $O(m + n \log n)$ time which is worst-case optimal. Dijkstra's algorithm (Algorithm 1) is conceptually simple: it initializes the heap with all vertices v that s can directly reach, where the key $\gamma(v)$ is the edge weight of (s, v). Then, it repeatedly pops the vertex u with the smallest key and it appends u to the output. Each time a vertex u is popped, its neighbors v are examined. For each neighbor v, the tentative distance d_v is computed as the distance from s to u plus the edge weight $w(u, v)$. If v is not in the heap, it is pushed with key $\gamma(v) = d_v$. If v is already in the heap and $\gamma(v) > d_v$ then the key $\gamma(v)$ is decreased to d_v.

2.2 Universally optimal Dijkstra

Iacono [9] introduced a refined measure for the running time of heap operations via the concept of *working-set size*:

▶ **Definition 1** ([9]). *Consider a heap supporting* PUSH *and* POP *and let x_i be an element in the heap. For each time step t between pushing x_i and popping x_i, we define the working set of x_i at time t as the set of elements $W(x_i, t)$ inserted after x_i and still present at time t. We include x_i itself in $W(x_i, t)$. Fix any time t_0 that maximizes the value of $|W(x_i, t)|$; we call the set $W(x_i, t_0)$* working-set *of x_i and $\phi(x_i) := |W(x_i, t_0)|$ the* working-set size.

Haeupler, Hladík, Rozhoň, Tarjan, and Tětek [6] design a heap H that supports Push and DecreaseKey in amortized $O(1)$ time, and supports Pop for an element $x \in H$ in $O(1 + \log \phi(x))$ time. The basis of their data structure are Fibonacci heaps F which support Push and DecreaseKey in amortized constant time and Pop in amortized $O(\log N)$ time where N is the number of elements in the heap.

The core idea of their data structure is to partition elements into buckets B_j of doubly exponentially increasing size. Each bucket B_j contains a Fibonacci heap F_j. Denote by m_j the minimum key of F_j. The set of $O(\log \log n)$ values m_j is stored in a fusion tree [4]. Fusion trees of size at most $O(\log \log n)$ have constant update time in the word-RAM model. By carefully merging and managing buckets, they ensure that each element x_i is stored in a Fibonacci heap F_j whose size is proportional to $\phi(x_i)$. We refer to [6] for full details; we only note here that ensuring that Push and DecreaseKey take amortized constant time whilst Pop takes $O(1 + \log \phi(x_i))$ time requires non-trivial maintenance and analysis.

Using this heap, Dijkstra's algorithm takes $O(m+\sum_{x_i \in V \setminus \{s\}}(1+\log \phi(x_i)))$ time. They [6] also prove a universal lower bound: for any fixed (directed) graph topology $G = (V, E)$ and source $s \in V$, no algorithm A' achieves runtime $o(m + n + \sum_{x_i \in V \setminus \{s\}} \log \phi(x_i))$ for all edge weightings $w \in \mathbb{W}(G)$. Dijkstra's algorithm with this heap is therefore universally optimal.

3 Timestamp optimal heaps

Instead of designing heaps with a running time proportional to the working-set size of a heap element, we use a simple timestamp approach:

▶ **Definition 2.** *For any heap H, we define* time t *as a counter that increments after each H.Push(x, y). For any element $x_i \in H$, we denote by a_i the time when x_i was pushed onto the heap, and by b_i the time when x_i was popped from the heap. Observe that $b_i > a_i$.*

▶ **Definition 3.** *A heap H is* timestamp optimal *if it supports Push and DecreaseKey in amortized constant time, and Pop with output x_i in amortized $O(1 + \log(b_i - a_i))$ time.*

Our Data Structure. Let H denote our abstract heap data structure, supporting operations H.Push(x, y), H.DecreaseKey(x, y), and H.Pop$()$. We implement H as follows:

▶ **Invariant 1** (Figure 1)**.** *We partition the elements of H into an array of buckets B. Each bucket $B[j]$ contains one or two Fibonacci heaps. Every element of H lies in exactly one Fibonacci heap. Along with each Fibonacci heap F we store a half-open interval $I_F \subset \mathbb{R}$. We maintain the following invariants; for all times t:*

(a) *An element $x_i \in H$ is in F if and only if $a_i \in I_F$,*
(b) *The interval I_F for $F \in B[j]$ has size 2^j,*
(c) *All intervals of $B[j-1]$ lie to the right of the intervals in $B[j]$,*
(d) *All intervals collectively partition $[0, t)$.*

By Invariant 1, for all elements $x_i \in B[j]$, $(t - a_i)$ is at least 2^{j-1}, but less than 2^{j+2}. We define M as an array of size $\lceil \log n \rceil$, where $M[j]$ stores the minimum key in $B[j]$. Furthermore, we maintain a bitstring S_M where $S_M[j] = 1$ if for all $k > j$, we have $M[j] \leq M[k]$.

▶ **Lemma 4.** *For any element $x_i \in H$ we can, given a_i and t, compute the bucket containing x_i and the Fibonacci heap that contains x_i in constant time.*

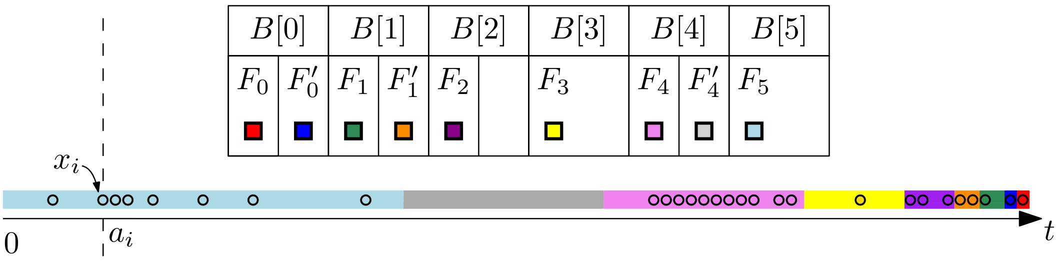

Figure 1 We illustrate our invariant. Each bucket $B[j]$ contains one or two Fibonacci heaps. Each heap F is associated with an interval on the number line in $[0, t)$. The elements x_i stored in a Fibonacci heap F are visualised by placing them at positions a_i along the number line. Importantly, when the buckets in B are indexed from left to right (i.e., from low to high indices), the corresponding intervals of the Fibonacci heaps appear in the reverse order along the number line.

Proof. Let a_i be the insertion time of x_i. We now want to determine the possible values j such that $B[j]$ contains x_i. From invariant 1.b we get: $t - a_i \geq 2^{j-1}$, or equivalently, $j \leq \log(t-a_i)+1$; and $t-a_i < 2^{j+2}$, or equivalently, $j > \log(t-a_i)-2$. In particular, for $\ell = \lfloor \log(t-a_i) \rfloor$, the Fibonacci heap containing x_i lies in one of the buckets $B[\ell-1], B[\ell], B[\ell+1]$. For each of these at most 6 corresponding intervals of the at most 6 heaps contained in those 3 buckets, we test, in constant time, whether a_i is contained in the interval. This procedure returns us the interval containing a_i, and thus, the heap (and bucket) containing x_i. ◀

▶ **Theorem 5.** *The above data structure is a timestamp optimal heap H.*

Proof. We prove that the data structure supports all three heap operations within the specified time bounds while maintaining Invariant 1.

Push(x_i, y). To push x_i onto H, we set $a_i = t$ and increment t. We then add a new Fibonacci heap F to $B[0]$ with $I_F = [a_i, a_i+1)$. Whilst a bucket $B[j]$ contains three Fibonacci heaps, we select its oldest two Fibonacci heaps F_1, F_2. We MELD F_1 and F_2 into a new Fibonacci heap F' in worst-case constant time, and we move F' to $B[j+1]$. We define the corresponding interval $I_{F'}$ as the union of the consecutive intervals I_{F_1} and I_{F_2}. Since Property (a) held for both F_1 and F_2, it now holds for F'. Moreover, $|I_{F'}| = 2^{j+1}$ and so Property (b) is maintained. Since, before the update, all intervals collectively partitioned $[0, t)$ and we only added the interval $[t, t+1)$ we maintain Properties (c) and (d). Every MELD operation decreases the number of Fibonacci heaps and every push operation creates only one heap. Hence, each push operation performs amortized $O(1)$ MELD operations.

Finally, we update M and the string S_M. After we have merged two heaps in a bucket $B[j]$, only $M[j]$ and $M[j+1]$ change. Moreover, $M[j]$ can only increase in value and $M[j+1]$ can only decrease to at least the original value of $M[j]$. Therefore, all bits $S_M[k]$ for $k \notin \{j, j+1\}$ remain unchanged and we update S_M in amortized constant time. Thus, a push operation takes amortized constant time.

DecreaseKey(x_i, y). By Lemma 4, we find the Fibonacci heap F and the bucket $B[j]$ that contain x_i in constant time. We perform F.DECREASEKEY(x_i, y) in amortized constant time. We update $M[j]$ and identify the index k of the leftmost 1-bit in S_M after index j in constant time using bitstring operations [4]. Note that $S_M[j] = 1$ if and only if $M[j] \leq M[k]$.

If $S_M[j] = 1$, then we recursively find the next $j' < j$ for which $S_M[j'] = 1$ using the same bitstring operation. If $M[j] < M[j']$, then we set $S_M[j'] = 0$ and we recurse. If $M[j] \geq M[j']$ then for all $j'' < j'$ with $S_M[j''] = 1$, $M[j] \geq M[j'']$ and our update terminates. The recursive call takes amortized constant time since it can only decrease the number of 1-bits in S_M.

Pop(). We obtain the index j of the leftmost 1-bit of S_M in constant time using bitstring operations. Per definition, the minimum key of H is in $B[j]$. We perform the PEAK operation on both heaps in $B[j]$. For the heap F containing the minimum key, we do F.POP() in $O(1 + \log 2^j) = O(j)$ time to return the element $x_i \in H$ with the minimum key. Finally, we update $M[j]$ and S_M. We update $M[j]$ in constant time and iterate from j to 1. For each iteration ℓ, let $k > \ell$ be the minimum index where $S_M[k] = 1$. We obtain k in constant time using bitstring operations and update $S_M[\ell]$ by comparing $M[\ell]$ to $M[k]$. Thus, this update takes $O(j)$ time. By Invariant 1, $\log(b_i - a_i) \in O(j)$ and thus the H.POP() takes $O(1 + \log(b_i - a_i))$ amortized time. ◀

▶ **Corollary 6.** *Algorithm 1 can be adapted to use* $O(m + \sum_{x_i \in V \setminus \{s\}} (1 + \log(b_i - a_i))$ *time.*

4 Universal Optimality

We show that categorizing the running time of Dijkstra's algorithm by timestamps substantially simplifies its proof for universal optimality. For a given graph $G = (V, E)$ and a designated source vertex s, we define a *linearization* L of (G, s) as any permutation of $V \setminus \{s\}$ such that there exists some choice of edge weights $w' \in \mathbb{W}(G)$ where L is the solution to the single-source-shortest path problem on (G, s, w'). Let $\ell(G, s)$ denote the number of distinct linearizations of (G, s). Observe that $\Omega(\log \ell(G, s))$ is a comparison-based universal lower bound for the single-source shortest path problem. Indeed, any comparison-based algorithm has a corresponding binary decision tree with leaves representing possible outputs. With $\Omega(\ell(G, s))$ distinct outputs, there must exist a root-to-leaf path of length $\Omega(\log \ell(G, s))$.

Our approach. We fix the input $(G = (V, E), s, w)$ and execute Dijkstra's algorithm, which defines a set of n timestamp intervals $\{[a_i, b_i]\}_{i \in [n]}$. Each interval $[a_i, b_i]$ has a minimum size of 1 and corresponds to a unique vertex $x_i \in V \setminus \{s\}$. Dijkstra's algorithm operates in $O(m + \sum_{x_i \in V \setminus \{s\}} (1 + \log(b_i - a_i)))$ time. Since any algorithm must spend $\Omega(n + m)$ time to read the input, it remains to show that $\sum_{x_i \in V \setminus \{s\}} (1 + \log(b_i - a_i)) \in \Omega(n + \log \ell(G, s))$.

▶ **Lemma 7.** *For each vertex $x_i \in V \setminus \{s\}$, choose a real value $r_i \in [a_i, b_i]$. If all r_i are distinct, then the sequence L, which orders all x_i by r_i, is a linearization of (G, s).*

Proof. Consider the initial execution of Dijkstra's algorithm on (G, s, w). For each vertex $x_i \in V \setminus \{s\}$, there is a unique edge (u, x_i) that, upon inspection, pushes the element x_i onto the heap H. The set of these edges forms a spanning tree T rooted at s.

Denote by $\text{rank}(r_i)$ the rank of r_i in the ordered sequence L. We construct a new integer edge weighting w' such that running Dijkstra's algorithm on (G, s, w') outputs L. We set $w'(e) = \infty$ for all edges $e \in E \setminus T$. We then traverse T in a breath-first manner. For each edge $(x_j, x_i) \in T$, we observe that in our original run of Dijkstra's algorithm, the algorithm popped the vertex x_j before it pushed the vertex x_i onto the heap. It follows that for the intervals $\{[a_j, b_j], [a_i, b_i]\}$, the value $b_j \leq a_i$. Since $r_j \in [a_j, b_j]$, $r_i \in [a_i, b_i]$, and $r_j \neq r_i$ we may conclude that $\text{rank}(r_i) > (r_j)$. We assign to each edge $(x_j, x_i) \in T$ the positive edge weight $w'((x_j, x_i)) = \text{rank}(r_i) - \text{rank}(r_j)$. It follows that the distance in (G, E, w') from s to any vertex x_i is equal to $\text{rank}(r_i)$ and so Dijkstra's algorithm on (G, s, w') outputs L. ◀

What remains is to show that the number of ways to select distinct $r_i \in [a_i, b_i]$, correlates with the size of the timestamp intervals. This result was first demonstrated by Cardinal, Fiorini, Joret, Jungers, and Munro [1, the first 2 paragraphs of page 8 of the ArXiV version]. It was later paraphrased in Lemma 4.3 in [5].

▶ **Lemma 8** (Originally in [1], paraphrased by Lemma 4.3 in [5]). *Let $\{[a_i, b_i]\}$ be a set of n integer intervals where each interval has size at least 1. Let Z be the set of linear orders realizable by real numbers $r_i \in [a_i, b_i]$. Then $\sum_{i\in[n]} \log(b_i - a_i) \in O(n + \log|Z|)$.*

Proof. To illustrate the simplicity of this lemma, we quote the proof in [5]: For each i between 1 and n inclusive, choose a real number r_i uniformly at random from the real interval $[0, n]$, independently for each i. With probability 1, the r_i are distinct. Let L be the permutation of $[n]$ obtained by sorting $[n]$ by r_i. Each possible permutation is equally likely. If each $r_i \in [a_i, b_i]$ then L is in Z. The probability of this happening is $\prod_{i=1}^{k}(b_i - a_i)/n$. It follows that $|Z| \geq n! \cdot \prod_{i=1}^{n}(b_i - a_i)/n$. Taking logarithms gives $\log|Z| \geq \sum_{i=1}^{n} \log(b_i - a_i) + \log n! - n \log n$. By Stirling's approximation of the factorial, $\log n! \geq n \log n - n \log e$. The lemma follows. ◀

It follows that we can recreate the main theorem from [6]:

▶ **Theorem 9** (Theorem 1.2 in [6]). *Dijkstra's algorithm, when using a sufficiently efficient heap data structure, has a universally optimal running time.*

Proof. The running time of Dijkstra's algorithm with the input (G, s, w) is $O(m)$ plus the time needed for push and pop operations. With a timestamp-optimal heap, by Corollary 6, the algorithm thus takes $O(m + \sum_{x_i \in V\setminus\{s\}} (1 + \log(b_i - a_i)))$ total time. By Lemma 8, the total running time is then $O(m + n + \log|Z|)$, where $|Z|$ denotes the number of distinct linear orders created by choosing distinct real values $r_i \in [a_i, b_i]$. By Lemma 7: $|Z| \leq \ell(G, s)$, and so Dijkstra's algorithm with any timestamp-optimal heap runs in $O(m + n + \log \ell(G, s))$ time. Conversely, any algorithm that solves SSSP needs $\Omega(m + n)$ time to read the input, and $\Omega(\log \ell(G, s))$ is an information-theoretical universal lower bound. ◀

5 Conclusion

We study Dijkstra's algorithm for the single-source-shortest path problem and give an alternative proof that Dijkstra's algorithm is universally optimal. We consider our construction to be considerably simpler than the one in [6]. We find time stamps easier to define than the working set size of an element, and regard our timestamp optimal heap to be simpler in both its construction and analysis than the heap used in [5]. As an added benefit, we only rely on Fibonacci trees and bitstring manipulation and no involved data structures (e.g., the *fusion trees* used in [6]). We regard Section 4 as our main contribution, as it gives a very concise proof of universal optimality. We understand that algorithmic simplicity is subjective, and we leave it to the reader to judge the simplicity of our approach. We note, however, that proofs are unarguably shorter since the full version of [6] counts over 50 pages. We observe that both our paper and [6] must assume a word RAM and we consider it an interesting open problem to show universal optimality on a pointer machine.

Finally, we note that our data structure is equally general to [6]. In the full version, we show that our data structure, just as the one in [6], is also universally optimal if algorithmic running time is exclusively measured by the number of edge weight comparisons that the algorithm performs. While our proof is indeed new, different, and perhaps simpler than the previous one, it is not self-contained; for this reason, we relegate it to our full version.

References

1 Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M Jungers, and J Ian Munro. Sorting under partial information (without the ellipsoid algorithm). In *ACM Symposium on Theory of Computing (STOC)*, pages 359–368, 2010. doi:10.1145/1806689.1806740.

2 Edsger Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

3 Michael Fredman and Robert Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987. doi:10.1145/28869.28874.

4 Michael Fredman and Dan Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

5 Bernhard Haeupler, Richard Hladík, John Iacono, Václav Rozhoň, Robert E Tarjan, and Jakub Tětek. Fast and simple sorting using partial information. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3953–3973. SIAM, 2025. doi:10.1137/1.9781611978322.134.

6 Bernhard Haeupler, Richard Hladík, Václav Rozhoň, Robert Tarjan, and Jakub Tětek. Universal optimality of dijkstra via beyond-worst-case heaps. In *Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2024. doi:10.1109/FOCS61266.2024.00125.

7 Bernhard Haeupler, Richard Hladík, Václav Rozhoň, Robert E Tarjan, and Jakub Tětek. Bidirectional dijkstra's algorithm is instance-optimal. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 202–215. SIAM, 2025. doi:10.1137/1.9781611978315.16.

8 Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In *ACM Symposium on Theory of Computing (STOC)*, pages 1166–1179, 2021. doi:10.1145/3406325.3451081.

9 John Iacono. Improved upper bounds for pairing heaps. In *Scandinavian Workshop on Algorithm Theory*, pages 32–45. Springer, 2000. doi:10.5555/645900.672600.

10 John Iacono and Michael L. Fredman. *Distribution-sensitive data structures*. PhD thesis, Rutgers University, New Jersey, 2001.

11 Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. Simpler optimal sorting from a directed acyclic graph. In *SIAM Symposium on Simplicity in Algorithms (SOSA)*, 2025. doi:10.1137/1.9781611978315.26.