RAIRO-Oper. Res. 59 (2025) 2419–2436
https://doi.org/10.1051/ro/2025082

RAIRO Operations Research
www.rairo-ro.org

EXPLORING MONOTONE PRIORITY QUEUES FOR DIJKSTRA OPTIMIZATION

Jonas Costa*, Lucas Castro and Rosiane de Freitas

Abstract. The Shortest Path Problem (SPP) is one of the most significant problems in combinatorial optimization. Beyond the vast number of direct applications, the SPP frequently serves as a subroutine in solving other optimization problems. This paper presents a comprehensive review of monotone priority queues, a class of data structures that play a crucial role in solving the SPP efficiently. Monotone priority queues are characterized by the property that their minimum key does not decrease over time, making them particularly effective for label-setting algorithms like Dijkstra's. Some key data structures within this category are explored, emphasizing those derived directly from Dial's algorithm, including variations of multi-level bucket structures and radix heaps. Theoretical complexities and practical considerations of these structures are discussed, with insights into their development and refinement provided through a historical timeline.

Mathematics Subject Classification. 68-02, 68P05, 68Q25.

Received November 25, 2024. Accepted June 17, 2025.

1. Introduction

A *priority queue* is a data structure that stores a dynamic set of elements and generally supports three operations:

- `insert`$(e, priority, Q)$ add the element e in the queue Q;
- `delete` or `remove`(e, Q) removes the element e from Q;
- and `extract-min`(Q) removes and returns an element of Q with minimum priority[1].

The name "priority queue" derives from the idea that the moment an element leaves the queue depends on its priority rather than the time it was inserted, in contrast with the principle *first in, first out* (FIFO) that rules regular queues.

Among several applications, priority queues have a special role in label-setting algorithms for the *Shortest Path Problem* (SPP). This problem has two main versions: to find the shortest path from a source s to all other vertices and find a shortest path between a pair of nodes s and t. The first is sometimes called the Single Source Shortest Path problem, or Shortest Path Tree Problem and, in spite of generalizing the second, it is not

Keywords. Algorithms, data structures, graph algorithms, priority queues, shortest path problem.

PPGI – Instituto de Computação, Universidade Federal do Amazonas, Manaus-AM, Brazil.
*Corresponding author: jonas.costa@icomp.ufam.edu.br

[1]It could be alternatively the maximum. In this paper, only minimum priority queues were considered because of their role in algorithms for the SPP.

© The authors. Published by EDP Sciences, ROADEF, SMAI 2025

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

harder. Therefore, from now on, only the single source version is considered. Typically, for a weighted directed graph $G = (V, A, w)$ and a vertex $s \in V(G)$, a label-setting method for the SPP builds a rooted directed tree T_s containing a shortest path from s to all vertices reachable from s[2]. At the same time, it computes $d_s(v)$ which is the distance from s to v for each vertex $v \in V(G)$ as follows:

(1) Set $T = \{s\}$ and $d_s(s) = 0, d_s(v) = \infty, \forall v \in V(G) \setminus \{s\}$.
(2) Select an arc uv from a vertex $u \in T$ to a vertex $v \in V(G) \setminus T$ that minimizes $d(u) + w(uv)$. Add uv in T and set $d(v) = d(u) + w(uv)$.
(3) Repeat the previous step until either $T = V(G)$ or $t \in T(G)$ depending on the version of the SPP.

The most iconic of these algorithms is due to Dijkstra [9]. While in 1959 the concept of priority queues was not yet established, today it is clear that the complexity of Dijkstra's algorithm is attached to the complexity of the priority queue used in the selection step, *i.e.*, step 2 above. Therefore, optimizing these data structures leads to more efficient algorithms for the SPP, which, as pointed out in [12], is one of the most representative problems of operational research. This is because solving the SPP is often required as a subroutine for many combinatorial optimization problems.

To be efficiently used by a label-setting algorithm, a priority queue must support an extra operation of *decrease-key* which reduces the priority of an element already inside the structure. In general, this operation has the same effect as removing the element and reinserting it with a lower priority, but it was introduced by Fredman and Tarjan in [13] with the goal of providing a more efficient alternative. It is worth noting that Fredman and Tarjan used the term *key* to refer to what has so far been called *priority*. The same convention is used in this paper, where the term *key* is used in a weaker sense, interchangeably with *priority*, to remain consistent with the terminology of the *decrease-key* operation[3].

The way it is presented today, Dijkstra's algorithm maintains the candidates to be added in T in a priority queue, with priority given by the shortest distance from the source vertex s observed so far. However, the addition of a new vertex to T may change these distances and therefore the priority of some vertices. This is why the possibility of changing priorities is important in this context.

The *binary heap* is probably the most popular among the priority queues and was introduced by Williams [34] inside Algorithm 232: Heapsort. Although it was designed for a sorting algorithm, the use of binary heaps in Dijkstra's algorithm dates back to the early 1970s. In [21], Johnson describes how to use d-heaps, a generalization of binary heaps, to find the next node to be added to T and analyzes the performance of this approach. He also points to an older reference that tried a similar approach. However, binary heaps work for general purposes, whereas there are priority queues optimized for shortest paths algorithms.

As defined by Thorup in [26], a priority queue is *monotone* if its minimum priority is non-decreasing over time. This implies, for example, that one cannot insert an element with a key smaller than the key of the last minimum extracted. *A priori*, this is a restriction on the sequence of operations that will be performed rather than a restriction on a data structure. In this context, a monotone priority queue is a data structure designed or optimized to work on such a restricted sequence of operations. They are suited for label-setting algorithms because they naturally "produce" monotone sequences of operations. Sometimes, for further optimization, it is also assumed that the keys are integers and that an upper bound for the maximum key to be inserted is known. Again, this is plausible in the context of shortest path algorithms. In this work, a monotone priority queue is considered in a broader sense of a structure optimized for Dijkstra-like algorithms.

The existence of such priority queues predates their designation as monotone. Dial's algorithm [8] for the shortest path problem implicitly uses a monotone priority queue. Further analyses identified that the time complexity of the operations on this priority queue was bounded by the maximum arc weight rather than by the number of elements inside the queue, as in the case of binary heaps. Over time, Dial's data structure evolved

[2]This particular kind of directed tree is often called arborescence.

[3]In other dynamic sets contexts, a key is regarded an immutable and unique value associated with an element. However, such restrictions are not always required in the context of priority queues.

into more sophisticated priority queues with better bounds. Much of this development was due to DIMACS 5th-Implementation Challenge[4] in 1995 which focused Priority queues, Dictionaries, and Multi-Dimensional Point Sets. Some results in both theoretical and practical fronts date from quite after that edition of the Challenge.

This paper provides an overview of monotone priority queues, emphasizing their application in shortest path algorithms. It aims to elucidate the key characteristics and time complexities of the primary data structures within this category, specifically those that are directly derived from Dial's algorithm and are expected to work well in practice. Theoretical results within the broader context of monotone priority queues are also provided. Additionally, the paper seeks to present a historical perspective on the evolution of these structures, offering insights into their development and refinement over time.

The remainder of this paper is organized as follows: Section 2 provides some groundwork by introducing the shortest path problem and related definitions. Section 3 delves into Dial's algorithm, discussing its origins and implementation. Section 4 explores the evolution into multi-level bucket structures, including two-level, k-level buckets, and hot queues. Section 5 focuses on the radix-heap data structure, reviewing its variants and their respective time complexities. In Section 6, the theoretical limits of Dijkstra's algorithm with different priority queues are examined, under the constraint of integer weights. Finally, Section 7 provides a historical timeline of the key developments in monotone priority queues.

2. Preliminaries

The shortest path problem considered here is defined over a directed graph $G = (V, A, w)$ and a vertex $s \in V(G)$, where $w : A \to \mathbb{Z}_+$ is a non-negative integer *weight* function of the arcs. The letters n and m denote the number of vertices and edges of the graph respectively. The problem asks then for the shortest paths from s to all the other vertices of G. The vertex s is often called *source* or *root* and $C = \max\{w(a) : a \in A(G)\}$ is the maximum weight of an arc of G. One may assume that G does not have loops, *i.e.*, arcs on the form vv. Considering a vertex $u \in V(G)$, the set $N^+(u) = \{v : uv \in A(G)\}$ is the out-neighborhood of u. The distance from the source to u is denoted by $d_s(u)$. The graph G is said to be *dense* if the number of its edges is close to the maximum, that is, $m = \theta(n^2)$. Conversely, G is *sparse* if $m = O(n)$.

This paper deals with bucket-based priority queues. In this context, a *bucket* is a data structure that stores a dynamic set of elements, supporting insertion and deletion. Usually, a double-linked list is enough to implement a bucket, but one may also implement it as a dynamic array. The latter may use more memory but is a more cache-friendly approach. The data structures will be discussed here from a higher level of abstraction so this kind of technicality will be left out.

Unless specified otherwise, an *element* that will be stored in a priority queue is a pair (v, k) where v is a vertex of the graph and k is a non-negative integer priority or label. Concerning the operations of a priority queue, generally, only k is relevant, so an abuse of language is used to compare elements in terms of their keys. For example, for two elements $a = (u, k_1)$ and $b = (v, k_2)$, to say that a is greater than b means that $k_1 > k_2$ and so on.

The most famous algorithm for the SPP is Dijkstra's labeling method [9]. In Algorithm 1, a modern implementation of this method that explicitly uses a priority queue is described. Initially, all vertices are inserted into a priority queue Q with infinite priority, except for s which is inserted with priority 0. Extracting a vertex v from Q is equivalent to adding it to the shortest path tree T, implying that its distance will no longer change. The first extracted vertex is always s because it is the only one with non-infinite priority. The algorithm also keeps two arrays $d_s(v)$ and π so that at the end of the execution $d_s(v)$ holds the distance from s to v and $\pi(v)$ has the predecessor of v in the shortest path from s to v[5]. The `decrease-key`$((v, k), Q)$ function in line 13 is equivalent to removing the element (v, k'), where $k' > k$, from Q, and inserting the element (v, k).

---

[4] `http://dimacs.rutgers.edu/programs/challenge/`.

[5] Here the notation $d_s(v)$ and $\pi(v)$ is used instead of $d_s[v]$ and $\pi[v]$ because these are graph parameters, not merely algorithm variables.

Algorithm 1: Dijkstra's algorithm.

```
Dijkstra(G = (V, A, w), s)
 1 forall v ∈ V(G) \ s do
 2 |   d_s(v) = ∞, π(v) = null;
 3 |   insert((v, ∞), Q);
 4 end
 5 d_s(s) = 0, π(s) = null;
 6 insert((s, 0), Q);
 7 while Q ≠ ∅ do
 8 |   u ← extract-min(Q);
 9 |   forall v ∈ N+(u) do
10 |   |   if d_s(v) > d_s(u) + w(uv) then
11 |   |   |   d_s(v) ← d_s(u) + w(uv) ;
12 |   |   |   π(v) = u ;
13 |   |   |   decrease-key((v, d_s(u) + w(uv), Q));
14 |   |   end
15 |   end
16 end
```

Dijkstra's algorithm performs a *balanced* sequence of operations on the priority queue, *i.e.*, a sequence where the queue starts and ends empty [4]. In particular, it performs initially n operations of insert (lines 3 and 6) and, consequentially, n extract-min operations in line 8. Combined, the loops on lines 7 and 9, will iterate $O(m)$ times (each outgoing arc of each vertex) and therefore the number of decrease-key is at most $O(m)$. Thus, the complexity of this algorithm is $O(nI + nX + mD)$, where I, X, and D are the costs (in the worst case) per insert, extract-min, and decrease-key, respectively.

In the original implementation of Dijkstra's algorithm, the vertices are kept in an array with direct addressing (vertex j at position j). Thus, the insertion and decrease-key are performed in constant time while the extract-min costs $\Theta(n)$ which is the time of scanning the whole array looking for the element with the smallest key. This gives a total complexity of $O(m+n^2) = O(n^2)$. Using a binary heap, the complexity drops to $O((n+m)\log(n))$ because all three operations can be done in $\log(n)$ on this priority queue. However, an efficient decrease-key implementation for binary heaps requires an efficient way to locate and remove a given element, which is not naturally supported by this data structure. This issue is usually solved with the help of an auxiliary structure (*e.g.*, a hash map or a direct-addressing array) that allows constant-time retrieval of an element's position in the heap, such as its index in the underlying array.

3. The Dial's algorithm and the one-level bucket structure

The algorithm presented in this section was also credited to Loubar by Hitchner [20], who pointed to a reference from 1964. More recent references like [12, 19, 23] refer to Dial's algorithm as an alternative implementation of Dijkstra's algorithm. However, in [8], Dial originally built his algorithm as an implementation of Moore's [22] algorithm. Moreover, from the five algorithms studied by Hitchner in [20], the one he attributed to Loubar is the only one that is not tagged as a Dijkstra variation. A decade later, in [7], this algorithm is identified as "Dijkstra address calculation sort" in a work of Dial himself along with Glover, Karney, and Klingman. Back then, different ways of processing vertices were seen as different label-setting algorithms. Over time most of those differences were abstracted by the priority queues, becoming what is called Dijkstra's algorithm today. Thus, Dial's algorithm can indeed be seen as a version of Dijkstra's algorithm that uses an array of buckets as a priority queue. In [19], Goldberg also mentions that the same data structure had been independently proposed in [10, 32].

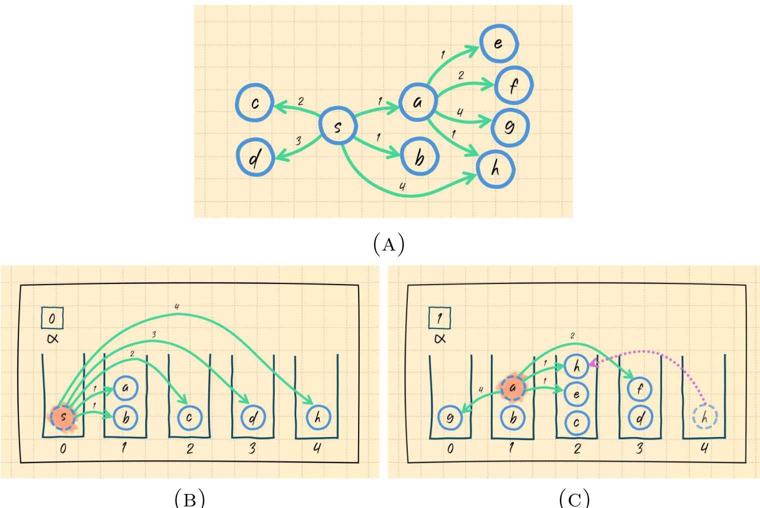
Dial's algorithm provides a good intuition on how bucket-based algorithms work in general, how they take advantage of the monotone structure of the problem as well as the fact that the arc weights are integers from which the maximum is known.

The key idea is to keep an array of buckets $\mathcal{B}$ such that the bucket $\mathcal{B}[i]$ will hold only vertices with distance i from s, that is, vertices v with $d_s(v) = i$. Since a path can have at most $n-1$ arcs, the maximum cost of a path and therefore the largest possible distance from s is $(n-1)C$, where C is the maximum arc weight. Thus, an array of nC buckets is sufficient to represent all possible distances (actually $(n-1)C$ is enough, but nC is simpler). This array will work as a monotone priority queue. The algorithm keeps a pointer α to the lowest-indexed non-empty bucket from which the next vertex to be scanned will be selected. Observe that α points to the bucket containing the vertices of minimum distance in $\mathcal{B}$. The bucket $\mathcal{B}[\alpha]$ is called the *active bucket* and when it gets empty, α is incremented to the next non-empty bucket. Once a vertex is scanned, it is checked whether the distance of its out-neighbors can be lowered. In the affirmative case, those already in $\mathcal{B}$ must be relocated to the buckets relative to their new best distance and the others inserted in the appropriated buckets. The pseudo-code of Dial's algorithm is presented in Algorithm 2.

Algorithm 2: Dial's algorithm.

```
Dial(G = (V, A, w), s)
 1 forall v ∈ V(G) \ s do  d_s(v) = ∞, π(v) = null ;
 2 d_s(s) = 0; π(s) = null;
 3 α ← 0 ;
 4 insert(s, B[α]);
 5 while α ≤ Cn do
 6 │  u ← pop(B[α]);                    /* Return and remove an element of the bucket. */
 7 │  forall v ∈ N+(u) do
 8 │  │  if d_s(v) > d_s(u) + w(uv) then
 9 │  │  │  d_s(v) ← d_s(u) + w(uv) ;
10 │  │  │  π(v) = u ;
11 │  │  │  relocate(v, B);   /* If v is already in B, remove it from its previous bucket. Insert v in
   │  │  │  B[d_s(v)]. */
12 │  │  end
13 │  end
14 │  if B[α] = ∅ then
15 │  │  α ← next(α, B) ;     /* Return the index of next non-empty bucket or ∞ if all buckets larger
   │  │  than α are empty. */
16 │  end
17 end
```

As mentioned earlier, Dial's technique for shortest paths provides an implicit monotone priority queue, which is precisely the array of buckets $\mathcal{B}$. To insert an element v with key k it suffices to call the insert function of the bucket $\mathcal{B}[k]$. Each bucket $\mathcal{B}[i]$ can be implemented, for example, as a double-linked list to allow insertion in constant time, the deletion can also be done in constant time if a reference to the node where each element is stored inside the bucket is kept. The function *relocate* used in Algorithm 2 is equivalent to the decrease-key operation. Since it is an insertion possibly preceded by a removal, it can be done in constant time. The operation of extract-min is the combination of the extraction of an element from the active bucket index (as in Line 6) and the possible update of the active bucket (Line 15). In the case of update (Line 14), by the monotone nature of the problem, one can consider only buckets of index higher than α, this means that a linear search for a non-empty bucket "on the right" of α suffices to implement the *next* (Line 15) function of the Algorithm 2. This priority queue will be called a *one-level bucket structure* for further reference.



(A) (B) (C)

FIGURE 1. Example on how Dial's algorithm works with a one-level bucket. The input graph with $C = 4$ is shown in (a). The numbers around the arcs denote their weights. Outside the main loop, the source vertex s was inserted in $\mathcal{B}[0]$. In the first iteration of the loop, s is removed and its out-neighbors are added into the proper buckets (b). Another vertex is selected in the following step and its neighbors are also added in $\mathcal{B}$. The vertex h is relocated from $\mathcal{B}[4]$ to $\mathcal{B}[2]$ (c).

The Algorithm 2 was stated this way for didactic purposes, but Dial's original technique uses fewer buckets. Indeed, Algorithm 2 can be easily modified to work only with $C + 1$ buckets in $\mathcal{B}$ instead of nC. Given that scanned vertices are selected only from the active bucket (Line 6), the range of labels in the structure is $[\alpha, \alpha+C]$, therefore at most $C + 1$ consecutive buckets can be used at any time of the algorithm's execution. Thus, to determine the position of a vertex inside $\mathcal{B}$, one can use its temporary label modulo $C + 1$ to "wrap around" when the end of $\mathcal{B}$ is reached. Namely, a vertex of label k must be inserted into bucket $\mathcal{B}[k \mod C + 1]$. The active bucket α update must also be modified to return to the beginning when it reaches C and $\mathcal{B}$ is not empty. In this case, α will receive the index of the first non-empty bucket.

When using nC buckets, as the vertex index in $\mathcal{B}$ matches its label, it is possible to modify Algorithm 2 so that the temporary labels are not explicitly stored. For example, the test in Line 8 could be replaced by $i_v > \alpha + w(uv)$, where i_v is the index of v in $\mathcal{B}$. As mentioned earlier, storing i_v is necessary to perform the operations delete and decrease-key in constant time. This approach only makes sense if one is not interested in the cost or length of the shortest path but only in the path itself. However, it highlights that the position of an element inside $\mathcal{B}$ is uniquely determined by its key and *vice-versa*. The same does not happen in other data structures, such as a binary heap, because the other keys present in the data structure may also influence the insertion of a new element. By reducing the number of buckets to $C+1$, it is necessary to keep extra information to infer the label of a vertex from its index in $\mathcal{B}$. One way to do it is to maintain a counter r of how many rounds the active bucket has made around $\mathcal{B}$. So, r is set to 0 at the beginning of the algorithm and is incremented each time α is updated to a smaller value. This way, the label of a vertex is given by $rC + i_v$. In the following sections, one may refer to the status of the data structure on *round* r.

Assuming buckets insertions and deletions in constant time, insert and decrease-key also takes constant time. The extract-min may cost a search in $O(C)$ buckets for updating the active bucket. Thus, the complexity of Dial's algorithm as stated in Algorithm 2 is $O(m + nC)$. A representation of the behavior of a one-level bucket inside Dial's algorithm is shown in Figure 1.

4. The multi-level bucket structure

In 1973, Gilsinn and Witzgall [15] did a performance comparison on labeling algorithms for shortest paths. They considered three techniques to improve the basic label correcting method and four improvements to the basic label setting method. Among these seven, Dial's algorithm showed the best performance, often by a large margin, as pointed out by the authors. In order to extend that study, Dial *et al.* [7] evaluated other procedures for the same problem. However, in contrast with the previous work, they found that different algorithms performed best in varying density scenarios. They considered five label-correcting and four label-setting methods. This time, the label-setting algorithms are Dial's algorithm and three modifications of it. The first is a better idea on when to add the nodes in the data structure. They observed that, when scanning the node u, it is not necessary to add all vertices in $N^+(u)$ in the priority queue but only one. Namely, the vertex v such that $d_s(v) > d_s(u)$ and $w(uv)$ is minimum. Based on this fact, the algorithm was modified to postpone the inclusion of vertices into $\mathcal{B}$.

The other two improvements added changes to the one-level bucket structure and were built upon the first modification. To decrease the number of empty-buckets scans, they partitioned $\mathcal{B}$ into segments of equal size and added a counter for the number of non-empty buckets in each segment. Then, to update the active bucket these counters could be used to skip entire segments of empty buckets. However, Dial *et al.* observed that this strategy did not improve the performance of the algorithm. They also noted that, in their experiments, the non-empty buckets were approximately uniformly distributed along $\mathcal{B}$, probably because w was generated by a uniform probability distribution. This means that for instances where w is not random, this still might be a profitable strategy. The second change consists of grouping each one of these segments into a "larger" bucket. This adjustment decreases the number of buckets but causes vertices with different distances to fall inside the same bucket. To deal with this, when a bucket becomes active, its vertices are sorted by their distances.

Years later in 1996, Prins [23] conducted a performance comparison among several SPP algorithms, focusing on sparse graphs. In his experiments, Dial's algorithm outperformed Dijkstra's algorithm implemented with binary heaps, Fibonacci heaps, and Radix heaps (see Sect. 5) in almost all cases. Interestingly, the pseudocode in [23] suggests that Prins did not incorporate any of the improvements proposed in [7]. For the scenario considered in that study, the simpler version of Dial's algorithm proved to be sufficiently efficient.

4.1. Two-levels of buckets

Denardo and Fox [6] proposed stronger structural modifications on Dial's one-level bucket structure by creating a level hierarchy of buckets. Their data structure is suited to work with floating-point keys, for more general shortest paths scenarios. Goldberg and Silverstein revisited this data structure in [19] and performed empirical experiments comparing the data structure's performance for different numbers of levels, but their implementation was restricted to integer keys.

From a conceptual perspective, a two-level bucket structure works as follows: the top level is an array of $\sqrt{C+1}$ buckets, and each top-level bucket comports a range of $\sqrt{C+1}$ distances. The size of the range of labels that fits into the bucket is called the *width* of that bucket[6]. Moreover, inside a top-level bucket, there are $\sqrt{C+1}$ bottom-level buckets, each of width one, *i.e.*, holding a single distance label. This structure is illustrated in Figure 2. To keep track of the active bucket, α_t is the top-level index while α_b is the bottom-level index, that is, if $\mathcal{B}$ is the whole structure, then $\mathcal{B}[\alpha_t][\alpha_b]$ is the active bucket. To find a position of a new element with key k inside $\mathcal{B}$, one must compute its top-level index $i = \lfloor k/\sqrt{C+1} \rfloor \mod \sqrt{C+1}$ and its bottom-level index $j = k \mod \sqrt{C+1}$.

In this way, the two-level bucket structure has no important advantage over the one-level version because it has the same number of buckets. One can decrease this number by keeping only one array of bottom-level buckets. This is reasonable because grouping the vertices with the same distance inside the same bucket is useful only when they are about to be extracted from the structure (by the extract-min operation). Furthermore, the

[6]The original description by Denardo and Fox [6] is suited to handle a more general distribution of buckets and widths.

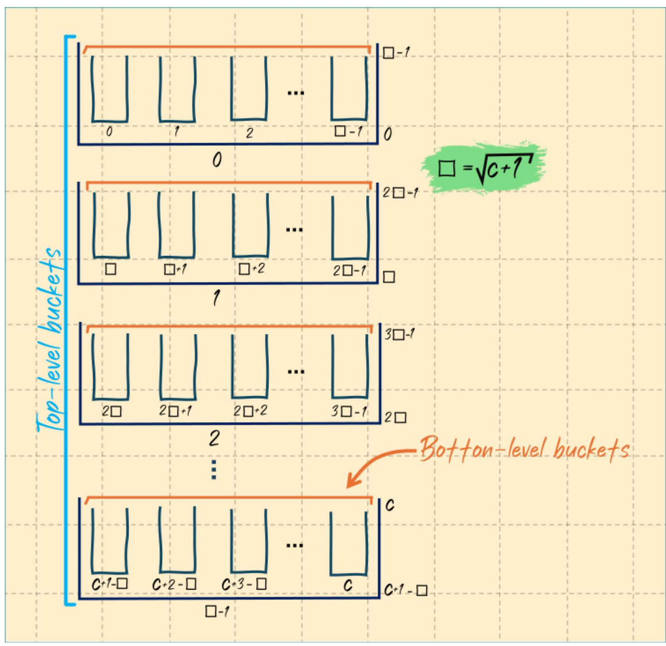


FIGURE 2. Conceptual view of a two-level bucket structure. The numbers on the right side of each top-level bucket stand for its range.

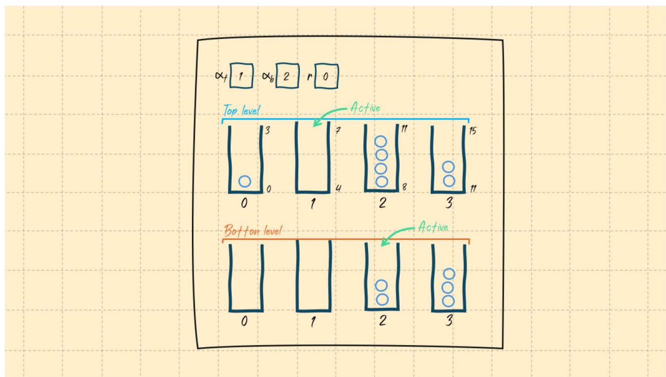


FIGURE 3. The two levels of a two-level bucket implementation for $C = 15$. The numbers on the right side of each top-level bucket stand for its range.

vertices with minimum distance in $\mathcal{B}$ are always in the buckets of $\mathcal{B}[\alpha_t]$. Thus, $\mathcal{B}$ can be modified to work with only two arrays $\mathcal{B}_t$ and $\mathcal{B}_b$ of $\sqrt{C+1}$ buckets, *i.e.*, top and bottom level, respectively. The bottom level $\mathcal{B}_b$ is associated with the active top-level $\mathcal{B}_t[\alpha_t]$. To find the appropriate place of a new vertex of key k, its indexes i and j are computed as before. If $i = \alpha_t$, it is inserted directly in $\mathcal{B}_b[j]$, otherwise, it is inserted in $\mathcal{B}_t[i]$. When the bottom level gets empty, α_t must be updated to the index i of the next non-empty top-level bucket, then the elements of $\mathcal{B}_t[i]$ will be distributed among the buckets of the bottom level, and α_t is set to i. This procedure is called *expansion*.

With this scheme, the number of buckets is reduced to $2\sqrt{C+1}$. The operations insert, remove, and decrease-key can also be performed in constant time. Moreover, the worst-case complexity for the extract-min operation is now $O(\sqrt{C+1})$. This is because after an extract-min it may be necessary to update the indexes of the active buckets, which in the worst case will cause a linear search in nearly all buckets. Recall that each top-level bucket covers a range of distances $\sqrt{C+1}$. Therefore, each empty scan on the top level of a two-level bucket structure skips $\sqrt{C+1}$ empty scans from its one-level counterpart (Fig. 3).

For $i \in \{0, 1, \ldots, \sqrt{C+1}-1\}$, the range of $\mathcal{B}_t[i]$ on round r is $[i\sqrt{C+1}+r\sqrt{C+1}, (i+1)\sqrt{C+1}-1+r\sqrt{C+1}]$ while the range of $\mathcal{B}_b[j]$ is the label $\alpha_t\sqrt{C+1}+r\sqrt{C+1}+j$. If $i < \alpha_t$, that is, $\mathcal{B}_t[i]$ is on the left of the active bucket, instead of r one must consider $r+1$.

The total time of the Dijkstra algorithm using this data structure is $O(m + n(1 + \sqrt{C}))$.

4.2. *k*-levels of buckets

Denardo and Fox [6] also extended the two-level implementation to a more general nested structure with $k \geq 3$ levels. It can also be seen as a k-dimensional matrix of buckets, as in the case of two levels presented earlier, but, as before, the implementation only keeps two dimensions.

The structure $\mathcal{B}$ have k levels $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_{k-1}$, where each level $\mathcal{B}_i$, for $i \in \{0, 1, \ldots, k-1\}$, is an array of $d = \lceil (C+1)^{1/k} \rceil$ buckets. The top-level is $\mathcal{B}_{k-1}$ while the bottom-level is $\mathcal{B}_0$. On each level, all buckets have the same width which narrows from top to bottom. Namely, the width on level i is d^i. In particular, the top and bottom levels have widths d^{k-1} and 1, respectively. Associated with each level i, there is also the index α_i indicating the current active bucket of that level. The buckets on level j correspond to the bucket α_{j+1} expanded, for every $j \in \{0, 1, \ldots, k-2\}$.

The lower bound of the level i in the round r, given by $L_r(i)$, is the smallest label that can be stored in that level on this particular round. Formally, $L_r(k-1) = rd^{k-1}$ and $L_r(i-1) = L_r(i) + \alpha_i d^i$, for $0 \leq i \leq k-2$. In turn, the range of bucket $\mathcal{B}_i[j]$ is $[L_r(i) + jd^i, L_r(i) + (j+1)d^i - 1]$, for $0 \leq j \leq d-1$. Similarly, one can define the upper bound of the level as $U_r(k-1) = L_{r+1}(k-1) - 1$ and $U_r(i-1) = L_r(i) + (\alpha_i + 1)d^i - 1$. The following proposition enlightens the meaning of the above ranges and bounds.

Proposition 4.1. *In any round $r \geq 0$, the range of the active bucket in level i corresponds to the bounds of level $i-1$, for every level $i \in \{1, 2, \ldots, k-1\}$.*

Proof. By definition, $[L_r(i-1), U_r(i-1)] = [L_r(i) + \alpha_i d^i, L_r(i) + (\alpha_i + 1)d^i - 1]$ which is exactly the range of α_i. ☐

With more levels, inserting a new element is less straightforward because one must first determine the correct level. In the initial state, $\mathcal{B}$ is empty and has $r = 0$. Suppose an element x with key 0 (a reasonable assumption for a label setting algorithm) was the first to be inserted, and that no extract-min or decrease-key was performed so far. In this scenario, observe that $\alpha_i = 0$ for every level i. Now, consider the insertion of x' with key y. Let i be the minimum level such that $y \in [L_0(i), U_0(i)]$. The claim is that x' must be inserted in $\mathcal{B}_i[\lfloor y/d^i \rfloor]$, recalling that d^i gives the width of level i. Clearly, x' cannot be inserted in a lower level since y does not match the bounds and, if i is not the top level, by Proposition 4.1 y is in the range of $\mathcal{B}[\alpha_{i+1}]$, which is expanded at level i. Either way, i is the correct level. Let j be the index such that y must inserted into $\mathcal{B}_i[j]$. By definition, the range of $\mathcal{B}_i[j]$ is $[L_r(i) + jd^i, L_r(i) + (j+1)d^i - 1]$ which is $[jd^i, (j+1)d^i - 1]$ since $L_r(i) = 0$. Thus, if $y \in [jd^i, (j+1)d^i - 1]$, then $j = \lfloor y/d^i \rfloor$.

After some operations of extract-min, active buckets may be incremented, pushing the bounds of the levels. But, as the widths and number of buckets on each level remain the same, bounds are always updated by multiples, so it is easy to find the appropriate index. In general, x' must be inserted in $\mathcal{B}_i[j]$, where $i = \min\{z \,|\, y \in [L_r(z), U_r(z)],\ z \in \{0, 1, \ldots, k-1\}\}$ and $j = \lfloor \frac{y - L_r(i)}{d^i} \rfloor$. With a linear search, finding i costs $O(k)$ in the worst case (more efficient methods for finding i will be discussed later). Decreasing the key of an element costs a constant time if it remains at the level and it costs $O(k)$ if a linear search for its proper level is necessary. Note that the decrease-key can only cause elements to descend in the levels of $\mathcal{B}$.

All the extract-min operations should occur at the bottom level. The procedure removes and returns an element of $\mathcal{B}_0[\alpha_0]$. If $\mathcal{B}_0[\alpha_0]$ is empty, α_0 will receive the index of the next non-empty bucket in $\mathcal{B}_0$ or an expansion must be performed. The expansion on a k-level bucket structure is the generalization of the preceding one. One must first determine the index i of the lowest non-empty level. To expand level i, α_i is updated with the index of the first non-empty bucket at level i (since all levels below i just became empty, certainly $j > \alpha_i$). Next, distribute the elements from $\mathcal{B}_i[\alpha_i]$ into the appropriate buckets of level $i-1$ while updating α_{i-1} to the lowest index used in level $i-1$. Continue this process until reaching the level 0, distributing elements and updating active buckets at each level. After the expansion, level 1 is non-empty, and the extract-min can occur normally. Considering the abstract notion of $\mathcal{B}$ as a k-dimensional matrix, the expansion procedure is analogous to finding the bucket with the smallest elements. Finding the level i for starting the expansion may cost $O(k)$ with extra $O(d) = O(C^{1/k})$ to find the first non-empty bucket inside $\mathcal{B}_i$ resulting in the total time of $O(k+C^{1/k})$ for the extract-min operation.

The total time of Dijkstra's algorithm with this k-level buckets structure is $O\left(km + n\left(k + C^{1/k}\right)\right)$ and it uses approximately $kC^{1/k}$ buckets. As shown in [6], this bound can be improved to $O(m\log(k) + n\left(\frac{k+C^{1/k}}{\mathcal{W}}\right)$, where $\mathcal{W}$ is the size of a machine word. This enhanced bound results from more sophisticated techniques to find non-empty levels and buckets. By storing the levels' breakpoints in an array, it is possible to use a binary search to find the correct level for an insertion; reducing the insertion time to $O(\log(k))$. To speed up the expansion one can use a binary k-vector with '1' entries representing non-empty levels. With appropriate bit operations, one can find the first non-zero entry in a machine word in constant time and therefore the first non-empty level in $O(k/\mathcal{W})$ time ($O(1)$ if k fits in a machine word). Applying the same strategy the first non-empty bucket is found in $O(C^{1/k}/\mathcal{W})$.

4.3. Hot queues

In [4], Cherkassky *et al.* introduced what they called *heap-on-top priority queues*. This data structure is a combination of a multilevel bucket priority queue $\mathcal{B}$ and a heap $\mathcal{H}$. The best bounds of time for Dijkstra's algorithm with hot queues are obtained when $\mathcal{H}$ is a monotone s-heap, *i.e.*, a priority queue in which the complexity of the operations grows with the number of elements inside the structure. In particular, these bounds are $O(m + n\log(C)^{\frac{1}{2}})$ when $\mathcal{H}$ is a Fibonacci heap [13], and $O(m + n\log(C)^{\frac{1}{3}+\epsilon})$ using the Thorup's heap of [26] instead. The last assumes a stronger RAM model[7]. Raman argues in [25] that this bound can be reduced to $O(m + n\log(C)^{\frac{1}{4}+\epsilon})$ by using an stronger result of [26].

Cherkassky *et al.* were not only interested in a data structure with lower theoretical bounds but also in one that could work well in practice. Using a stronger RAM model, they give another description of the k-level bucket structure. In their description, there is no need to keep the bounds of the levels because they manage to compute the level of a new element with bit operations between its key and the key of the last minimum extracted. This reduces the insertion and decrease-key times to $O(1)$ and therefore the total time to $O(m+nC^{\frac{1}{k}})$.

The main structural change added to the hot queues, concerning its predecessor, is designed to minimize empty scans after the expansion of a bucket with few elements. For example, consider the expansion of the bucket $\mathcal{B}_i[\alpha_i]$ of a k-level bucket structure $\mathcal{B}$ during one execution of Dijkstra's algorithm. Before the expansion, all levels below i (if $i > 0$) are empty. If the number of elements in $\mathcal{B}_i[\alpha_i]$ is small, some levels below i will be empty or nearly empty. This scenario may lead to successive expansions which is bad because, in terms of time complexity, an expansion is the most expensive procedure in a multi-level bucket. The heap $\mathcal{H}$ is employed to prevent this kind of issue.

Let $\mathcal{B}_\ell[\alpha_\ell]$ denote the bucket containing the element with the smallest key in $\mathcal{B}$, and let $c(B)$ denote the number of elements inside the bucket B. Whenever $c(\mathcal{B}_\ell[\alpha_\ell]) \leq t$ and $\ell > 0$ (the buckets in level 0 do not require

[7]The standard RAM model provide a basis for assessing the efficiency of an algorithm. It defines basic operations like arithmetic (addition, subtraction, multiplication), data manipulation (load, store, copy), and control flow (branching) that execute in constant time ($O(1)$). This model mirrors real-world computing capabilities and is widely used in algorithm design. Stronger RAM models extend this by including operations such as bitwise logic and specialized bit manipulations, making it more theoretical and extending the range of operations beyond practical computer architectures [5,31].

expansions), instead of expanding $\mathcal{B}_\ell[\alpha_\ell]$ its elements are duplicated in $\mathcal{H}$, where t is a predefined threshold. Any operation that should happen in the range of $\mathcal{B}_\ell[\alpha_\ell]$ will be performed both in the bucket and in $\mathcal{H}$, "freezing" the buckets below $\mathcal{B}_\ell[\alpha_\ell]$. On the other hand, inserts and decrease-keys in buckets above $\mathcal{B}_\ell[\alpha_\ell]$ can occur normally. When eventually $\mathcal{B}_\ell[\alpha_\ell]$ gets large enough, *i.e.*, $c(\mathcal{B}_\ell[\alpha_\ell]) > t$, it is expanded, $\mathcal{H}$ is emptied, and the structure returns to operate as a regular k-level bucket structure. The number of elements in $\mathcal{H}$ is always limited this is why using a s-heap is preferable.

The bounds mentioned at the beginning of this section are obtained by properly choosing $\mathcal{H}$ and t as Cherkassky *et al.* proved the following:

Theorem 4.2 ([4]). *Let $I^{\mathcal{H}}(N), D^{\mathcal{H}}(N)$ and $X^{\mathcal{H}}(N)$ the time bounds for the operations of insert, decrease-key, and extract-min in $\mathcal{H}$. Then, for a balanced sequence of operations, the amortized bounds for the hot queue are: $O\left(I^{\mathcal{H}}(t)\right)$ for insert, $O\left(D^{\mathcal{H}}(t) + I^{\mathcal{H}}(t)\right)$ for decrease-key, and $O\left(k + X^{\mathcal{H}}(t) + \frac{kC^{1/k}}{t}\right)$ for the extract-min.*

4.4. Other versions

In [2], Andersson and Thorup implemented a monotone priority queue as part of the DIMACS implementation challenge. From a high-level perspective, their structure can be viewed as a 4-level bucket structure. They optimize it to work with a fixed number of levels given that the keys are 64 bits long. A major difference is that they keep a sorted list with a subset of the elements in the data structure. The idea here is also to avoid expansions of buckets with few elements (as $\mathcal{H}$ in the hot queue). The minimum element will always be in this list, and since it has a fixed maximum size, keeping it sorted takes constant time. This priority queue performed very well in the reported empirical evaluation being substantially faster than a standard heap in most of the problem instances. Andersson and Thorup left the code of their implementation in the appendix of [2].

Goldberg proposed in [17] a label-setting method for the SPP that he called later, in [16,18], the smart queue algorithm. Unlike Dijkstra's algorithm, the smart queue will not necessarily scan the vertex with minimum temporary distance as Raman showed how to detect vertices for which the temporary distance cannot decrease. The algorithm keeps those vertices in a list F scanning them first and in any order. The vertices not satisfying such condition are kept in a multi-level bucket structure $\mathcal{B}$ as usual. If F is empty, the minimum element of $\mathcal{B}$ is extracted and scanned as usual. In [17], Goldberg showed that this algorithm has a linear-time average case. With empirical experiments he verified in [16,18] that the smart queue ran, in the worst case, 2.5 slower than a BFS (breath-first search). He also performed experiments with Dijkstra's algorithm running with a multi-level bucket. The latter also showed a performance close to the BFS even with a large number of levels. An interesting improvement that can be implemented in any of the priority queues presented in this section is what is called in [16] the wide bucket heuristic. Let $\ell = \min\{w(a) : a \in A(G)\}$ be the minimum arc weight of the graph. Then, for $0 < p \leq \ell$, the multi-level bucket structure works correctly if the width of the buckets is multiplied by p on every level.

5. The radix-heap

In 1990, Ahuja *et al.* [1] proposed a monotone priority queue they called *radix heap* and their data structure improved one proposed by D.B. Johnson in 1977. In reference to this work, Denardo and Fox affirm that a preprint of [6] stimulated Johnson to build a variant of their multilevel bucket system. The complexity of Dijkstra's algorithm with Johnson's data structure is $O(m \log\log(C) + n \log(C) \log\log(C))$ as indicated in [1,6]. Ahuja *et al.* present three versions of radix heaps: one-level radix heaps, two-level radix heaps, and two-level radix heaps with Fibonacci heaps. The running times of Dijkstra's algorithm with these data structures are respectively: $O(m + n\log(C)), O\left(m + n\log(C)/\log\log(C)\right)$ and $O\left(m + n\sqrt{\log(C)}\right)$.

To build radix heaps Ahuja *et al.* took particular advantage of the following properties of Dijkstra's algorithm: $d_s(u) \in \{0, 1, 2, \ldots, nC\}$; and $d_s(v) \in \{\mu, \mu+1, \ldots, \mu+C\}$, where μ is the label of the last scanned vertex and v is a vertex in Q (see Algorithm 1) with non-infinity label. They remark that the last implies that successive extract-min operations return vertices with non-decreasing keys.

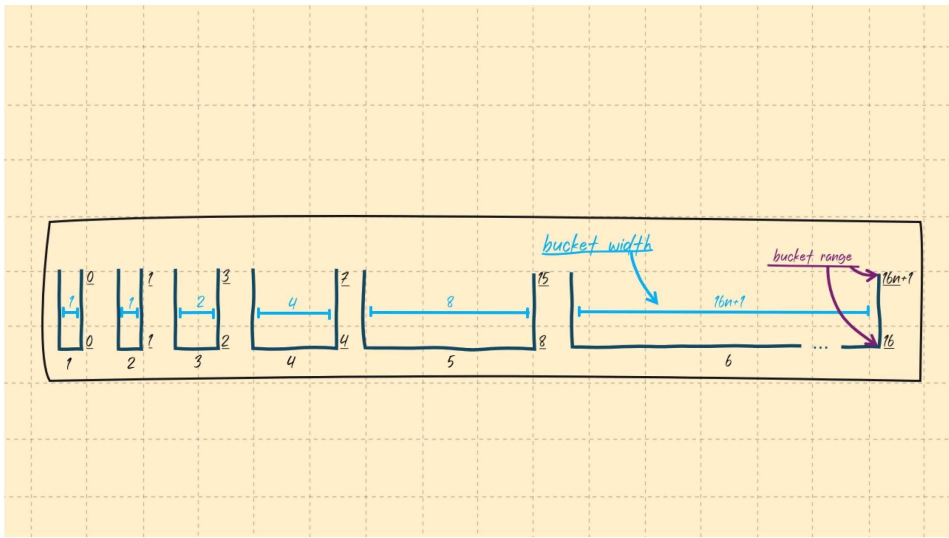


FIGURE 4. Representation of a one-level radix heap at the initial state assuming $C = 15$.

5.1. One-level radix heaps

A one-level radix heap is an array $\mathcal{B}$ with $k = \lceil \lg(C+1) \rceil + 2$ buckets which will be indexed from 1 to k. Let $|\mathcal{B}[i]|$ denote the width of bucket $\mathcal{B}[i]$. Then, $|\mathcal{B}[1]| = 1$, $|\mathcal{B}[i]| = 2^{i-2}$ for $i \in \{2, 3, \ldots, k-1\}$, and $|\mathcal{B}[k]| = nC + 1$. This way, it follows that $\sum_{j=1}^{i-1} |\mathcal{B}[j]| \geq \min\{|\mathcal{B}[i]|, C+1\}$, for every $i \in \{2, 3, \ldots, k\}$. That means that the buckets before $\mathcal{B}[i]$ have enough space to hold all different labels in $\mathcal{B}[i]$ (or $C+1$ labels when $i = k$).

The bucket ranges are chosen to partition the interval $[\mu, \mu+1, \ldots, n(C+1)]$, recalling that μ denotes the label of the last scanned vertex and therefore is initially 0. For each $i \in \{1, 2, \ldots, k\}$, the range of $\mathcal{B}[i]$ is $[U(i-1)+1, U(i)]$ with the convention that $U(0) = \mu - 1$. As in the multi-level bucket structure, these ranges determine which labels can be inserted on each bucket and they vary during the algorithm's execution while the widths are fixed. However, here, the size of the range of a bucket and its width does not necessarily match[8]. It is assumed, that the vertex s (with label 0) will be inserted in the first bucket, then initially the bounds are set as $U(i) = 2^{i-1} - 1$ for $i \in \{1, 2, \ldots, k-1\}$ and $U(k) = nC + 1$. In Figure 4 there is a representation of a radix heap for $C = 15$.

In a one-level bucket structure (see Sect. 3), all buckets have width 1, and the "head" of the structure (the active bucket) moves towards the lower key elements. In contrast, in a one-level radix heap, the lower key elements will be moved toward the "head" of the structure (the first bucket) and the buckets get narrower as they get closer to the head. So, the idea is to keep the elements with minimum label in $\mathcal{B}[1]$ and the algorithm must properly adjust the bucket ranges to maintain this property.

The insertion of an element x with key y proceeds as follows: starting from $i = k$ iterate, in decreasing order, until finding the first i such that $U(i) < y$, then insert x in $\mathcal{B}[i+1]$. Observe that $\mathcal{B}[i+1]$ is the lowest bucket in which x can be inserted. Even if initially there is only one possible bucket for the insertion, eventually the structure may have buckets with overlapping ranges, and therefore one must ensure that the new element is placed in the lowest appropriate bucket. The time spent with an insertion is $O(k) = O(\log(C))$.

To perform the decrease-key of an element x' to a new key y', one should find the index j of the bucket in which x' is stored; remove x' from $\mathcal{B}[j]$; change its key to y', and proceed as regular insertion but starting from

[8] In this case, the term width is an abuse of notation.

$i = j$. As before, the first stage can be done in constant time if each element stores its index in $\mathcal{B}$. On the other hand, finding the new bucket costs $O(k)$ in the worst case. However, the index of an element can only decrease, and at most k times. Indeed this number is the complement to the cost of the first insertion. In other words, an element initially inserted in $\mathcal{B}[\ell]$ can change of bucket at most $k - \ell$ times. Thus, in an amortized analysis, as done in [1], one may charge the cost of every insertion to exactly k and consider that the decrease-key costs $O(1)$. This way, the total time spent with these operations will be $O(m + n\log(C))$. Observe that, the same analysis also applies to the context of a multilevel bucket structure.

The extract-min operation returns and removes an element of $\mathcal{B}[1]$. If $\mathcal{B}[1]$ is empty, the algorithm seeks the first non-empty bucket $\mathcal{B}[j]$. Then, the elements of $\mathcal{B}[j]$ are transferred to a temporary bucket $\mathcal{T}$ and, during this transfer, an element x with the minimum label is isolated. The algorithm proceeds updating the structure by setting $U(0) = d_s(x) - 1, U(1) = d_s(x)$ and $U(i) = \min\{U(i-1) + |\mathcal{B}[i]|, U(j)\}$ for $2 \leq i \leq j-1$. Except for x, the elements in $\mathcal{T}$ are reinserted in the structure with the search for the appropriate bucket starting from $j-1$, as in the decrease-key operation, and, finally, x is returned. It takes $O(k) = O(\log(C))$ to find $\mathcal{B}[j]$, and roughly speaking it is also necessary $\log(C)$ steps to transfer each element in $\mathcal{B}[j]$. But in an amortized sense, this last $\log(C)$ factor can be discounted from the insertion. This way, the time of a single extract-min is $O(\log(C))$. Therefore, Dijkstra's algorithm runs in $O(m + n\log(C))$ with this priority queue.

This update performed by the decrease-key operation is similar to the expansion procedure on the multi-level bucket structure and it is possible because, as mentioned earlier, the buckets before $\mathcal{B}[j]$ have enough range to accommodate all the elements in $\mathcal{B}[j]$. However, two situations may occur after the update: buckets with invalid or overlapping ranges. The first happens when $U(i-1) + 1 > U(i)$, for some $i \in \{2, 3, \ldots, k-2\}$. It is not a problem because $\mathcal{B}[i]$ will be considered empty and not chosen during the insertion since any key in its range is also in the lower bucket $\mathcal{B}[i-1]$ range. The second is when there is some i such that $U(i) = U(j)$, where j is the index found during the extract min. In this case, the bucket $\mathcal{B}[j]$, which is now empty, is the one that will be ignored by insertions and remain empty until some update occurs starting from a bucket with an index higher than j.

5.2. Two-level radix heaps

Ahuja *et al.* [1] remarked that reducing the number of reinsertions of elements into buckets was crucial for decreasing the running time of Dijkstra's algorithm using radix heaps. They also observed that this could be done by increasing bucket widths but that would break the relation of bucket ranges. They overcame this issue by adapting the two-level bucket structure of Denardo and Fox [6] (see Sect. 4.1). The idea is to split each bucket into *inner-buckets* of the same size, where an inner-bucket is analogous to a bottom-level bucket (see Sect. 4.1).

The number of buckets of a two-level radix heap is $k = \lceil \log_\Delta(C+1) \rceil + 1$ where Δ is the number of inner-buckets within each bucket. From a high-level perspective, the behavior of this priority queue is the same as the one-level version but with additional steps of accessing the correct inner-bucket inside the current bucket. The bucket widths need to be redefined in terms of Δ: for $i \in \{1, 2, \ldots, k-1\}$, $|\mathcal{B}[i]| = \Delta^i$ and $|\mathcal{B}[k]|$ remains $nC + 1$.

Assuming that initially $\mu = 0$, it follows that the first bucket ranges from 0 to $\Delta - 1$ because $|\mathcal{B}[1]| = \Delta$. Therefore, $U(1) = \Delta - 1$ and $U(2) = \Delta - 1 + |\mathcal{B}[2]| = \Delta - 1 + \Delta^2$. Thus, generally speaking, the initial upper bounds are $U(i) = \sum_{j=1}^{i} \Delta^j - 1$, for $i \in \{1, 2, \ldots, k-1\}$ and $U(k) = nC + 1$.

The width of each bucket is equally shared among its inner-buckets. The inner-bucket j inside the bucket $\mathcal{B}[i]$ is denoted by $\mathcal{B}[i][j]$ and has width $|\mathcal{B}[i][j]| = |\mathcal{B}[i]|/\Delta = \Delta^{i-1}$[9]. If a key y is in the range of the bucket i, then $y = U(i-1) + d$ with $1 \leq d \leq \Delta^i = |\mathcal{B}[i]|$. Furthermore, $\lceil d/\Delta^{i-1} \rceil$ gives the index j of the proper inner-bucket for y in $\mathcal{B}[i]$, in other words, $j = \lceil \frac{y - U(i-1)}{\Delta^{i-1}} \rceil$.

[9]The two-dimensional array notation is preferable here because the buckets of the first dimension are only virtual, the actual buckets here are the inner-buckets.

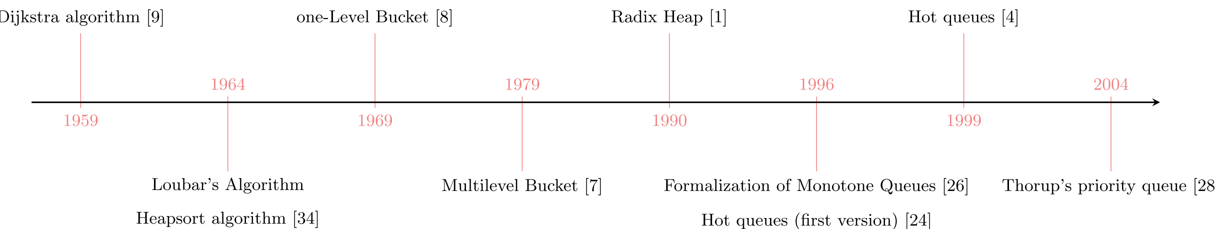


FIGURE 5. Timeline showing key points of the evolution of monotone priority queues.

To insert a new element the index i of the proper bucket is found as in a one-level radix heap. Then the element is inserted in $\mathcal{B}[i][j]$, where j is computed in constant time with the above-mentioned formula. The decrease-key operation is also analogous to the simpler version. The total time spent with these two operations is $O(m + nk) = O(m + \log_\Delta(C))$.

To perform an extract-min, there is an extra step of finding the first non-empty inner-bucket within the first non-empty bucket. Moreover, only the content of this inner-bucket is relocated during the update. The other details of the update can be easily modified from the one-level radix heap. Using a linear search, it takes $O(k)$ time to find the first non-empty bucket and $O(\Delta)$ for the proper inner-bucket. A vertex is extracted only once and can be lowered k times during updates. Thus, the total time for extract-min operations will be $O(nk+n\Delta)$, and the total time for Dijkstra's algorithm is $O(m + n(k + \Delta))$. As pointed out by Ahuja *et al.* [1], one can get the bound of $O(m + n\log(C)/\log\log(C))$ by merely choosing $\Delta = O(\log(C)/\log\log(C))$ (recalling that k is defined as $\lceil\log_\Delta(C + 1)\rceil + 1$).

5.3. Radix heaps with Fibonacci heaps

The third approach for radix heaps presented by Ahuja *et al.* aims to speed up the time for finding non-empty inner-buckets. The main idea is to keep the indices of non-empty inner-buckets in a Fibonacci heap. This priority queue supports insert and decrease-key in constant amortized time and extract-min in $O(\log(n))$ amortized. They extended the Fibonacci heaps from [13] so that when working with keys in the set $\{1, 2, \ldots, N\}$, the amortized time for the extract-min operation is $O(\log(\min\{n, N\}))$, while the insert and decrease-key operations remain constant in amortized time. The number of inner-buckets is $N = k\Delta$. Thus, by choosing $\Delta = 2^{\lceil\sqrt{\lg(C)}\rceil}$ one gets $k = O(\log_\Delta(C)) = O(\sqrt{\log(C)})$ and therefore $\log(N) = O(\sqrt{\log(C)})$. It takes constant time to decide when to operate the Fibonacci heap because they are only necessary when an inner-bucket becomes empty or when there is an insertion into an empty inner-bucket.

As seen in Section 4.3, the time complexity of $O(m + n\sqrt{\log(C)})$ for Dijkstra's algorithm was also achieved with hot queues operating with Fibonacci heaps. It should be noted that radix heaps preceded hot queues, and hot queues achieve better amortized bounds by using Thorup's heaps. However, it is important to mention that Fibonacci and Thorup's heaps are complex data structures, and combining these priority queues is useful to provide good expected times, but, in practice, this may not be efficient.

6. THEORETICAL LIMITS

It is well-established that $O(m + n\log(n))$ is the best time possible for Dijkstra's algorithm when using a comparison-based priority queue and arbitrary arc weights [1, 28]. In this section, some results regarding the equivalent limit to the restricted case of non-negative integer weights are discussed. In particular, the bound above is attained using the Fibonacci heaps developed by Fredman and Tarjan [13] which performs insertions and decrease-key in amortized constant time and extract-min in $O(\log(n))$ amortized. As discussed in Section 5.3, Ahuja *et al.* improved this data structure to obtain a better bound for the extract-min when the keys are non-negative integers limited by a constant N. At the time, they also discussed the problem of finding a bound

TABLE 1. Table of with the time of complexities for different priority queues.

Priority queue	Insert	Extract-min	Decrease-key	Time complexity of Dijkstra
Binary heap	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O((m+n)\log(n))$
One-level bucket	$O(1)$	$O(C)$	$O(1)$	$O(m+nC)$
Two-level bucket	$O(1)$	$O(\sqrt{C})$	$O(1)$	$O(m+n\sqrt{C})$
Multi-level bucket	$O(1)$	$O\left(k+C^{\frac{1}{k}}\right)$	$O(k)$	$O\left(km+n\left(k+C^{\frac{1}{k}}\right)\right)$
One-level radix heaps	$O(\log(C))$	$O(\log(C))$	$O(1)$	$O(m+n\log(C))$
Two-level radix heaps	$O(\log(C)/\log\log(C))$	$O(\log(C)/\log\log(C))$	$O(1)$	$O(m+n\log(C)/\log\log(C))$
Hot queues with Fibonacci heap	$O\left(\log^{\frac{1}{2}}(C)\right)$	$O\left(\log^{\frac{1}{2}}(C)\right)$	$O(1)$	$O\left(m+n\log(C)^{\frac{1}{2}}\right)$
Thorup's heap	$O(1)$	$O(\log\log(n))$	$O(1)$	$O(n\log\log(n))$

on the form $O(m+nf(C))$ for the restricted case of integer arc weights, where f is minimum. Their result implied that $f(C) \leq \sqrt{C}$, but based on the existence of the van Emde Boas priority queue (vEB)[10] [11], which performs all operations in time $O(\log\log(C))$, they raised the question of whether $f(C)$ could be reduced to $O(\log\log(C))$.

This question was answered by Thorup [28] with the following result:

Theorem 6.1 ([28]). *One can implement a priority queue that, for n elements with keys in the range $[0, N-1]$, supports insert and decrease-key in constant time, and extract-min in $O(\log\log(\min\{n, N\}))$ time.*

His proof is constructive, so he presented how to build such a data structure and it is important to note that the times of Theorem 6.1 are not amortized. Observe that one can use this priority queue in a hot queue, and by carefully choosing the parameters in Theorem 4.2 obtain the amortized time of $O(m+n\log\log(C))$ for Dijkstra's algorithm. However, Thorup shows as a corollary of Theorem 6.1 that this time can be achieved by using his data structure and some extra buckets. The extra buckets will work as the highest level of a multi-level bucket structure to ensure that, at each time, the main priority queue maintains only elements in the range $[iC, (i+1)C-1]$. He does not say it explicitly in [28] but two buckets are enough for this purpose, because if one keeps, for each $i \in \{0, 1, \dots, n-2\}$, a bucket B_i for holding elements with distances in the range $[iC, (i+1)C-1]$, only two buckets are non-empty in any given time.

The above-mentioned work of Thorup is one in a series of studies of RAM priority queues. These are not necessarily monotone but, as defined in [3], store non-negative integers and have running time depending on the maximum value stored N, assuming that this value is known *a priori*. The vEB is considered the first data structure of this category. Using a vEB one gets a solution for the SPP in time $O(m\log\log(C))$. The vEB operates with $O(N)$ space in the version of [30]. Willard [33] achieved the same bounds of a vEB with a data structure he called Y-fast tries and reduced the space used to $O(n)$. But since he used dynamic perfect hashing, the bounds are amortized randomized $O(\log\log(N))$.

Going further on the idea of a limited universe of keys, namely, assuming that N fits in a RAM word, one gets that $N \leq 2^{\mathcal{W}}$ and therefore the number of distinct keys is also limited by $\mathcal{W}$. Then, by bucketing elements with the same key one may assume that all elements in the queue have distinct keys, and therefore $n \leq N$. In this scenario Fredman and Willard [14] introduced the fusion trees with insert and extract-min in $O(\log(n)/\log\log(n))$ opening the door for priority queues with sublogarithm operations independent of the word size, as pointed out in [3]. Fredman and Tarjan also observed that with fusion trees one obtains a sorting algorithm (inserting all elements and then extracting them) that surpasses the limitations of the information-theoretic lower bound, *i.e.*, sorting n numbers requires at least $n\log(n)$ comparisons. In [26], Thorup showed a

[10] The data structure was designed by the first author (see the acknowledgments of [11]) so it is mainly known by its name.

RAM priority queue that achieves the time $O(\log\log(n))$ per operation, which in turn, gives a sorting algorithm of time $O(n\log\log(n))$ and also the time $O(m\log\log(n))$ for Dijkstra's algorithm. Raman [24] reduced these bounds to $O(n\sqrt{\log(n)\log\log(n)})$ and $O(m + n\sqrt{\log(n)\log\log(n)})$ with another RAM priority queue. From these three data structures, only the first need to assume constant time multiplication.

In [26], Thorup also proved the following result:

Theorem 6.2 ([26])**.** *For a RAM with arbitrary word size, if one can sort n keys in time $nS(n)$, for a non-decreasing function S, then there is a monotone priority queue with capacity for n keys, supporting insert in constant time and decrease-key in $O(S(n))$.*

In other words, there is an equivalence between sorting in a RAM and monotone priority queues. Thorup extended this equivalence for general RAM priority queues. These results imply that the development of better bounds for RAM priority queues is attached to the improvement of RAM sorting algorithms and *vice versa*. In particular, the bounds of Theorem 6.1 are currently the best for the general cases of directed graphs and non-negative integer weights. However, there are linear bounds for some restricted cases, see [25] for a more detailed list. There are also linear bounds for the shortest path problem in undirected graphs, as shown by Thorup in [27].

7. Final remarks

This paper has provided a review of monotone priority queues, emphasizing those derived from Dial's algorithm and tracing their evolution through subsequent improvements. By bucketing together vertices with the same distance, Dial's technique for the shortest path problem implicitly used a monotone priority queue even before the concept of priority queue was well-established.

Dial's method requires knowledge of the maximum possible value to be stored, but this method has the advantage that the complexity of the operations – insert, decrease-key, and extract-min – depends on this maximum value rather than on the number of elements in the queue. This property is particularly interesting for the SPP, because the maximum value is always known and is often smaller than the number of elements.

Dial's implicit data structure was further improved, giving rise to various versions of monotone priority queues. What follows is a timeline (Fig. 5) highlighting the key points of this evolution, along with a table (Tab. 1) summarizing the time complexities discussed throughout this paper.

Funding

This work was conducted as part of the Research, Development, and Technological Innovation Project (PD&I) “SWPERFI – Artificial Intelligence Techniques for Analysis and Optimization of Software Performance”, been partially supported by Motorola Mobility Comércio de Produtos Eletrônicos Ltda and Flextronics da Amazônia Ltda, under the terms of Federal Law No. 8.387/1991, through agreement No. 004/2021, signed with ICOMP/UFAM. Also, this work was partially supported by the following Brazilian agencies: the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES-PROEX) – Finance Code 001, the National Council for Scientific and Technological Development (CNPq), and Amazonas State Research Support Foundation – FAPEAM – through the POSGRAD project 2024/2025 and Programa de Desenvolvimento da Pós-Graduação – Parcerias Estratégicas nos Estados III CAPES/FAPEAM (Edital n°038/2022 PDPG/CAPES).

Data availability statement

The research data associated with this article are included in the article.

References

[1] R.K. Ahuja, K. Mehlhorn, J. Orlin and R.E. Tarjan, Faster algorithms for the shortest path problem. *J. ACM* **37** (1990) 213–223.

[2] A. Andersson and M. Thorup, A pragmatic implemention of monotone priority queues (1996).

[3] G.S. Brodal, A survey on priority queues, in Space-Efficient Data Structures, Streams, and Algorithms, edited by A. Brodnik, A. López-Ortiz, V. Raman and A. Viola. Vol. 8066 of *Lecture Notes in Computer Science*. Springer, Heidelberg (2013) 150–163.

[4] B.V. Cherkassky, A.V. Goldberg and C. Silverstein, Buckets, heaps, lists, and monotone priority queues. *SIAM J. Comput.* **28** (1999) 1326–1346.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, Introduction to Algorithms. MIT Press, Cambridge (2022).

[6] E.V. Denardo and B.L. Fox, Shortest-route methods: 1. Reaching, pruning, and buckets. *Oper. Res.* **27** (1979) 161–186.

[7] R. Dial, F. Glover, D. Karney and D. Klingman, A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks* **9** (1979) 215–248.

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

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

[10] E.A. Dinic, Economical algorithms for finding shortest paths in a network. *Transp. Model. Syst.* (1978) 36–44.

[11] P. Emde Boas, R. Kaas and E. Zijlstra, Design and implementation of an efficient priority queue. *Math. Syst. Theory* **10** (1976) 99–127.

[12] D. Ferone, P. Festa, A. Napoletano and T. Pastore, Shortest paths on dynamic graphs: a survey. *Pesquisa Operacional* **37** (2017) 487–508.

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

[14] M.L. Fredman and D.E. Willard, Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.* **47** (1993) 424–436.

[15] J.F. Gilsinn and C. Witzgall, A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees. Vol. 13 of *NBS Technical Note 777*. National Bureau of Standards, Washington, DC (1973).

[16] A.V. Goldberg, Shortest path algorithms: engineering aspects, in Algorithms and Computation, edited by P. Eades and T. Takaoka. Springer, Heidelberg (2001) 502–513.

[17] A.V. Goldberg, A simple shortest path algorithm with linear average time, in Algorithms – ESA 2001, edited by G. Goos, J. Hartmanis, J. Van Leeuwen and F.M.A. Der Heide. Vol. 2161 of *Lecture Notes in Computer Science*. Springer, Heidelberg (2001) 230–241.

[18] A.V. Goldberg, A practical shortest path algorithm with linear expected time. *SIAM J. Comput.* **37** (2008) 1637–1655.

[19] A.V. Goldberg and C. Silverstein, Implementations of Dijkstra's algorithm based on multi-level buckets, in Network Optimization, edited by G. Fandel, W. Trockel, P.M. Pardalos, D.W. Hearn and W.W. Hager. Vol. 450 of *Lecture Notes in Economics and Mathematical Systems*. Springer, Heidelberg (1997) 292–327.

[20] L.E. Hitchner, *A comparative investigation of the computational efficiency of shortest path algorithms*. Operations Research Center, University of California, Berkeley, Report ORC (1968) 68–17.

[21] E.L. Johnson, On shortest paths and sorting, in Proceedings of the ACM Annual Conference on – ACM'72. Vol. 1. ACM Press, Boston (1972) 510.

[22] E.F. Moore, The Shortest Path Through a Maze. *Bell Telephone System*. Technical Publications, Monograph (1959).

[23] C. Prins, Comparaison d'algorithmes de plus courts chemins sur des graphes routiers de grande taille. *RAIRO-Oper. Res.* **30** (1996) 333–357.

[24] R. Raman, Priority queues: small, monotone and trans-dichotomous, in Algorithms – ESA'96, edited by G. Goos, J. Hartmanis, J. Leeuwen, J. Diaz and M. Serna. Vol. 1136 of *Lecture Notes in Computer Science*. Springer, Heidelberg (1996) 121–137.

[25] R. Raman, Recent results on the single-source shortest paths problem. *ACM SIGACT News* **28** (1997) 81–87.

[26] M. Thorup, On RAM priority queues, in Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms. Vol. 81. SIAM (1996) 59.

[27] M. Thorup, Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM* **46** (1999) 362–394.

[28] M. Thorup, Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Syst. Sci.* **69** (2004) 330–353.

[29] M. Thorup, Equivalence between priority queues and sorting. *J. ACM* **54** (2007) 28.

[30] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.* **6** (1977) 80–82.

[31] J. Van Leeuwen, Handbook of Theoretical Computer Science (Vol. A) Algorithms and Complexity. MIT Press, Cambridge (1991).

[32] R.A. Wagner, A shortest path algorithm for edge-sparse graphs. *J. ACM* **23** (1976) 50–57.

[33] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\theta(N)$. *Inf. Process. Lett.* **17** (1983) 81–84.

[34] J.W.J. Williams, Algorithm 232: heapsort. *Commun. ACM* **7** (1964) 347–348.

Please help to maintain this journal in open access!

This journal is currently published in open access under the Subscribe to Open model (S2O). We are thankful to our subscribers and supporters for making it possible to publish this journal in open access in the current year, free of charge for authors and readers.

Check with your library that it subscribes to the journal, or consider making a personal donation to the S2O programme by contacting subscribers@edpsciences.org.

More information, including a list of supporters and financial transparency reports, is available at https://edpsciences.org/en/subscribe-to-open-s2o.