SHORTEST PATHS WITH INTEGER LENGTHS

25.09.2023

---

1 Dijkstra's algorithm

- Dijkstra' algorithm finds the shortest path from starting vertex s to all other vertices in a graph with non-negative edge lengths; it works as follows:
    - each vertex is in one of 3 states: *unreached* (we haven't reached these yet, their distance is still ∞), *tentative* (this is the boundary; we know *some* path to these vertices but it can still improve as we explore different paths), and *done* (we already know the true distance from s)
    - we maintain tentative distance $d(x)$ for each vertex x, such that *some* path from s to x has total length $d(x)$
    - initially s is *tentative* with $d(s) = 0$ and every $x \neq s$ is *unreached* with $d(x) = \infty$
    - select a tentative vertex x such that $d(x)$ is minimum and declare it *done*; for each edge (x, y), if $d(x) + c(x, y) < d(y)$, replace $d(y)$ by $d(x) + c(x, y)$ and declare y tentative
- we can implement this using a priority queue as follows:
    - we maintain the *tentative* vertices in a heap (ordered by d)
    - we select vertex with minimum distance by calling `get_min`
    - when we find a shorter route and replace $d(y)$ by $d(x) + c(x, y)$, we call `decrease_key`
    - when declaring an *unreached* vertex as *tentative* ($d(x)$ changes from ∞ to sth finite), we `insert` it in the heap

- the resulting complexity is $O(n \times (T_{\texttt{insert}} + T_{\texttt{get\_min}}) + m \times T_{\texttt{decrease\_key}})$
    - implementation with a simple array is $O(n \times (1 + n) + m \times 1) = O(n^2)$,
    - with binary heap this is $O(n \times (\log n + \log n) + m \times \log n) = O(m \log n)$,
    - and with Fibonacci heap $O(n \times (1 + \log n) + m \times 1) = O(m + n \log n)$ (but this is impractical)

2 Integer lengths

- now assume that all lengths are integers from the range $[0, 1, \ldots, C]$ – can we implement something better?

- extreme case: if all lengths are 1, we could use BFS, which is $O(m)$
- if we only have very small lengths, we could replace each edge of length c by a path of length c (c unit edges) and then use BFS – the number of edges will grow $\leq C$ times so the complexity will be $O(C \times m)$

- another idea: all distances will be in the range $[0, 1, \ldots, n \times C]$, so we could use a simple array, where we would have all the vertices at distance d in slot d of the array
- we would step through this array from left to right – some fields will be empty, but when we come across non-empty ones, these vertices are at the minimum distance among unprocessed vertices
- all operations are constant time, except that we have to traverse the entire array, so this is $O(m + n \times C)$ time and space

- can we do better?

3 Radix heap

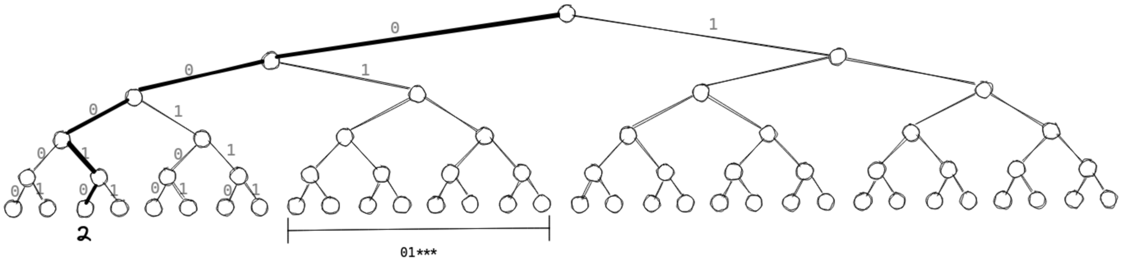
- note two important properties related to how Dijkstra's algorithm uses its heap:
  - `get_min` operations return a non-decreasing sequence of distances – when we process some vertex at distance d, all the other vertices that we will encounter have an even greater distance!
  - let x be the last removed vertex (*done* vertex with max distance); then all the *tentative* vertices have distance in the range $[d(x), d(x)+1, \ldots, d(x)+C]$
  - why? when a vertex turns from *unreached* to *tentative*, the distance is in that range; since then, its distance can only decrease and distance of the last removed vertex can only increase, so all tentative distances remain in the range $[d(x), d(x)+1, \ldots, d(x)+C]$

- two things follow from this:
  - we only need a so-called *monotone heap*, which assumes that successive `get_min` operations return vertices in nondecreasing order and all `insert`s and `decrease-key`s never add an element smaller than the current minimum
  - $O(C)$ memory should be enough – if we maintain the relative distance from the last vertex, then we only need values $[0, \ldots, C]$ and ∞

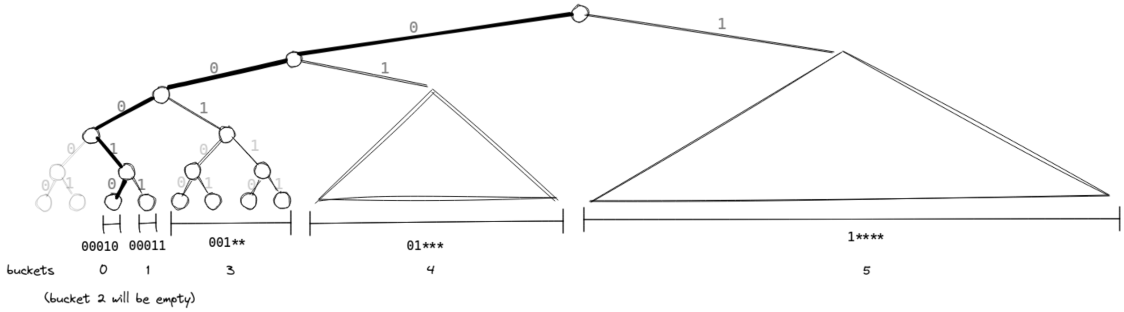
3.1 Idea

- we will have buckets with value ranges of length $1, 1, 2, 4, 8, 16, 32, \ldots$ ($O(\log C)$ buckets are enough), plus we will remember the value of the `last_deleted` element
- the original version of the data structure (`https://kubokovac.eu/ds/mat/sssp.pdf`) split all elements into buckets based on the difference from `last_deleted`; we will describe a *simpler* and *more efficient* variant which is also explained here: `http://ssp.impulsetrain.com/radix-heap.html`

- first, let's start with a toy example and see, what is happening *conceptually*, then we will talk about implementation
- let's assume we are working with 5-bit integers, i.e. numbers in the range `00000` … `11111` in binary
- we can *imagine* these in a form of a binary tree where `0` bit goes left and `1` goes right (NOTE: this is just an imaginary/implicit tree to help us with the explanation, but it is NOT constructed in the actual implementation)



- every 5-bit number corresponds to a single root-to-leaf path in this tree, e.g. $2 =$ `00010` is the bold path above
- a shorter prefix corresponds to a subtree / a range of numbers; range `01000`…`01111` (or `01`$***$; where $*$ is a wildcard denoting 0 or 1) is shown above
- we will split these into buckets based on the *most significant bit* (MSB) where x and `last_deleted` differ
- in our imaginary tree, the MSB corresponds to the first point where the root-to-leaf paths differ; `last_deleted` goes left and x goes right
- example: let `last_deleted` $= 2$; e.g. MSB(2, 13) = MSB($\texttt{0\underline{0}010}$, $\texttt{0\underline{1}101}$) = 4, so 13 goes to bucket #4 (see the range depicted above)

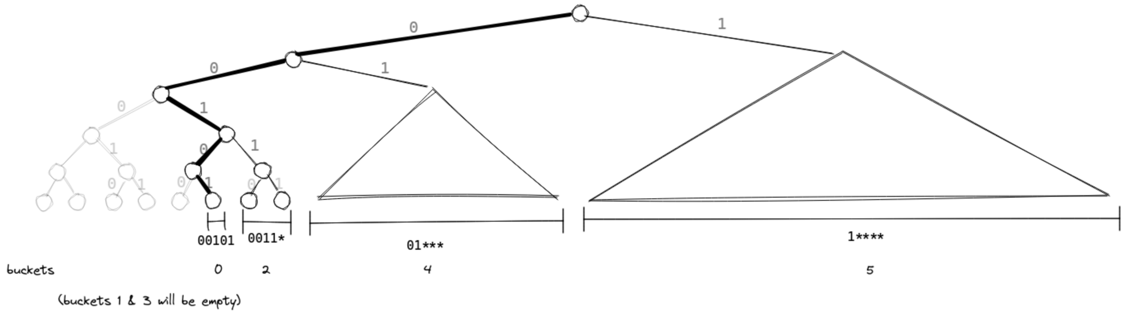


- let `last_deleted` = 2; all numbers in the heap will be ≥ 2 and we will have the following buckets (see Fig. above):
  - 0: `00010` (number 2 itself)
  - 1: `00011` (number 3; differing in the 1st bit from right)
  - 2: —— (numbers differing in the 2nd bit are smaller than 2, so this bucket is empty)
  - 3: `001`∗∗ (numbers 4–7; differing in 3rd bit)
  - 4: `01`∗∗∗ (numbers 8–15; differing in 4th bit)
  - 5: `1`∗∗∗∗ (numbers 16–31; differing in 5th bit)

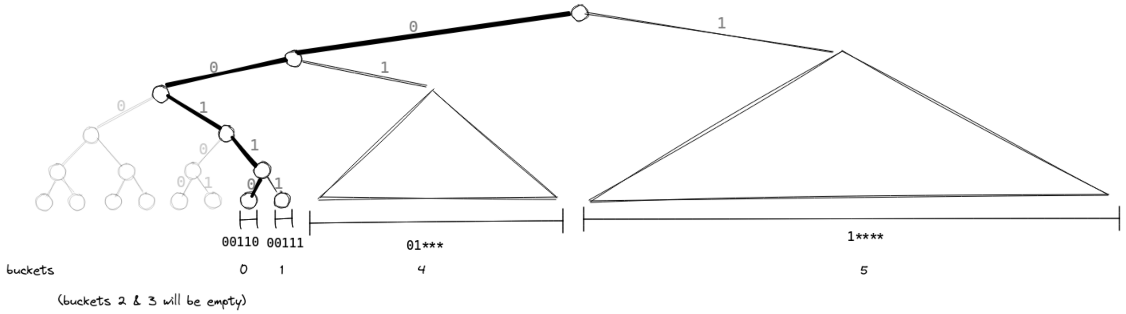
- more generally: for element x, take the binary representation of numbers x and `last_deleted` and let i be the *most significant bit* (MSB) where x and `last_deleted` differ (counting from 1 from right; define $i = 0$ if $x =$ `last_deleted`); then put x into the i-th bucket
- this basically amounts to computing a XOR of x and `last_deleted` and then finding the highest set bit (XOR of two numbers has bits set at positions where they differ and zeros at positions where the corresponding bits are the same)

- if we want to insert a new element, we simply insert it into the correct bucket
- if we want to change the value of an element (while we know its position in the heap), we simply remove it and reinsert it where it belongs
- how does `get_min` work?
  - if the 0th bucket is non-empty, we just remove and return that number
  - otherwise, traverse from left to right until we find the first non-empty bucket (that's at most $O(\log C)$ steps)
  - then go through this entire bucket and find a new minimum (note that numbers in a single bucket do not have to be sorted!)
  - remove this minimum and move all other elements from this bucket to their correct buckets, based on the new value of `last_deleted`

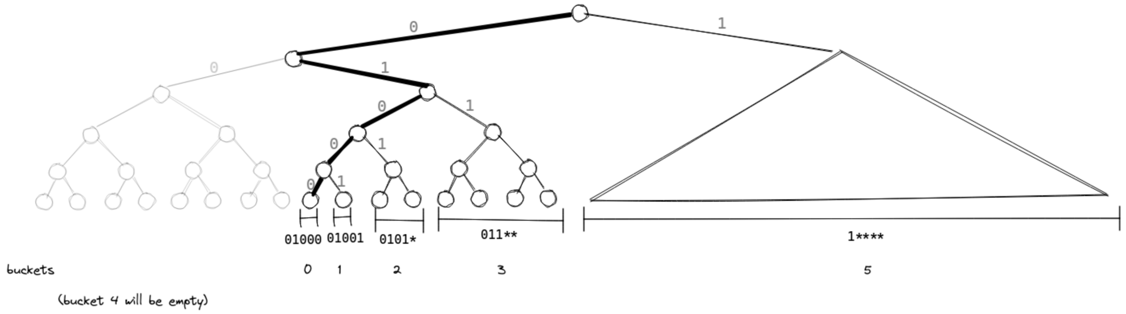
- say the radix heap contains numbers 5, 6 (in bucket 3), 8, 10, 11, 17 (in bucket 4), 18, 27 (in bucket 5) and 2 is the `last_deleted` (see Fig. above)
- if we `get_min`, we scan buckets 0, 1, 2 and find #3 as the first non-empty bucket; we return 5 and move 6 to the 2nd bucket; after the operation, the heap will look like this:



- if we `get_min` again, we return 6 from the 2nd bucket and the heap will look like this:



- finally, if we `get_min` again, bucket 4 is the first non-empty
- `last_deleted` started `00***` but there are no more elements in that left subtree and subtree `01***` is the first nonempty
- we find that 8 is the minimum and redistribute the rest (10, 11, and 17) into buckets 0...3 based on the new value of `last_deleted` = 8 (see Fig. below)
- note that all the elements in bucket 4 start `01***` so they can only differ in bits 1...3
- also note that for the higher buckets (only bucket 5 in this toy example) nothing changes – they *still* differ from the new value of `last_deleted` in the higher bits, so there is no need to move them



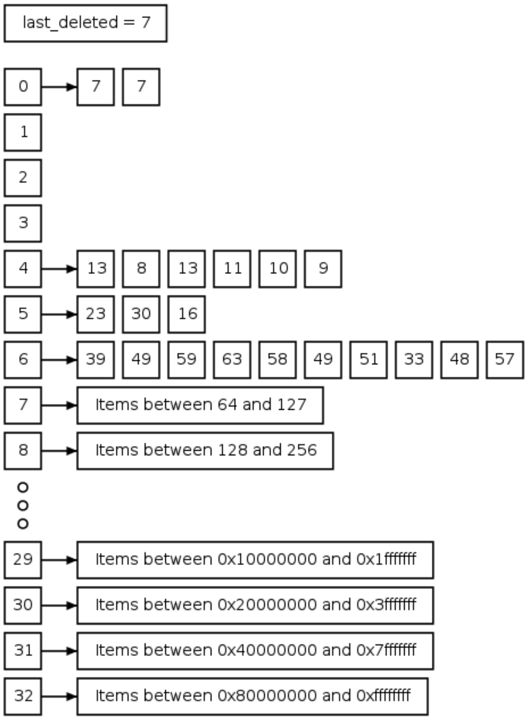
- in general, if k is the first non-empty bucket, it means that the previous `last_deleted` had k-th bit `0`; there are no more elements in that left subtree; so we traverse the right subtree, find the minimum of the k-th bucket and split all the other numbers into buckets 0,…,$k-1$; the higher buckets are unaffected

- how much time do these operations take?
  - `insert` and `decrease_key` is obviously $O(1)$
  - `get_min` is in $O(\log C + B)$, where B is the size of the bucket we process – in the worst case it can be $\Theta(N)$, but amortized it's still $O(\log C)$ – why?
  - let's look at the life cycle of a single element: first, we put it in some bucket and from then on it only moves to the left! `decrease_key` only moves it to the left and each bucket redistribution during `get_min` moves elements only to the left
  - but since we only have $O(\log C)$ buckets, each element can be moved only $O(\log C)$-times
  - we can imagine charging $\lg C\$$ to the `insert` operation and all later moves can be paid from that
  - think about it for a minute: the worst-case complexity of `insert` is $O(1)$, but *if* we make the amortized cost $O(\log C)$ (i.e. much more than $O(1)$), *then* the amortized cost of `get_min` can be also $O(\log C)$; this is a nice example how by over-charging one operation we can pay for another operation

3.2 Implementation

- each bucket will be a simple (unsorted) vector
  - (don't use linked lists due to cache-efficiency)
  - remember that when removing an element from a vector, you definitely *don't* want to use a function that deletes an element and moves all the elements after it – this has linear complexity! (how to do it in $O(1)$?)

- see also `http://ssp.impulsetrain.com/radix-heap.html` for an explanation and a picture with a bigger (32-bit) example:



- the whole heap can be just a simple array of 33 or 65 buckets (depending on whether you want to work with 32-bit or 64-bit numbers)
- how do we find the correct bucket for number x?
  - just compute bitwise XOR of `last_deleted` and x and find the highest set bit
  - e.g. if these were `01001110` and `01011000`, the XOR is `00010110`
  - remember that XOR has `1` on the positions where the numbers differ and `0` elsewhere
  - the highest set bit can be calculated on current CPUs using a single instruction (see BSR – bit scan reverse, `https://c9x.me/x86/html/file_module_x86_id_20.html`)
  - you can google how to do this with your compiler; e.g., if you are compiling with `gcc`, you can compute the highest set bit by `31-__builtin_clz(d)` if d is a 32-bit number; CLZ means "count leading zeros" – see `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html` for a list of accessible built-in functions (these usually compile to simple instructions); e.g., `__builtin_clzll` is a variant for `unsigned long long`
  - you can use `8` $*$ `sizeof(int)` to find how many bits does an `int` have
- note that for the Dijkstra algorithm, you will also need to be able to find a given vertex x in the heap so you will need to maintain and update a table with positions of vertices in the heap