

LECTURE NOTES FOR CMSC 651
DIJKSTRA'S ALGORITHM WITH BOUND ON THE WEIGHTS

1 DIJKSTRA'S ALGORITHM: BOUNDED EDGE WEIGHTS

By using Fib-heaps as our data structure we obtained an algorithm for the Single Source Shortest Path (SPPP) problem that runs in time

$$O(V \log V + E).$$

Can we do better? Dijkstra's algorithm can be used to SORT numbers (this is an easy exercise). Hence, Dijkstra's algorithm cannot do better than $O(V \log V)$ time (the lower bound for sorting). But what if all the weights are (non-negative) integers bounded by a constant C ?

We **can** do better! We will exhibit an algorithm that runs in time

$$O((\log C)V + E).$$

We will not change the algorithm, just the the data structure that supports it.

This algorithm has been used at Bell Labs as a subroutine in an algorithm for min-cost flow which assigned clients to nearby copies of websites.

Key Observation: Assume C bounds the edge weights. Assume that Dijkstra's algorithm is run. Assume that DELETMIN removes vertex v which has value $d(v)$. Let w be the very next vertex that is removed by DELTETMIN. Then

$$d(v) \leq d(w) \leq d(v) + C.$$

Minor Adjustment: We need one minor adjustment in Dijkstra's algorithms. Note that since the bound on the weights is C , and all paths are of length $\leq n$ we will obtain $d(v) \leq Cn$. In the usual Dijkstra's algorithm we initially set the $d(v)$'s to be ∞ . We will instead set them equal to $Cn + 1$. Note that the largest integer ever encountered for a value of $d()$ will be $Cn+1$.

We will describe a data structure that supports INSERT, FINDMIN, DELETMIN, DECKEY.

- We *will not* bound the cost of these operations.
- We *will not* bound the amortized cost of these operations.
- We *will* bound the amortized cost **provided that the DELETTEMINS operate as in Dijkstra's algorithm with bounded weights**. That is, provided that if the DELETTEMINS delete v_1, v_2, \dots, v_n in that order then,

$$(\forall i)[d(v_i) \leq d(v_{i+1}) \leq d(v_i) + C].$$

The data structure itself depends on the number C . We will assume C is a power of 2. Let $L = \lg C$.

We describe the data structure.

1. At every stage there will be a set of the form $\{\text{dmin}, \dots, nC + 1\}$. All keys are in this set. Initially $\text{dmin} = 0$.
2. The numbers $\{\text{dmin}, \dots, nC + 1\}$ are divided into $L + 3$ buckets of contiguous numbers. Initially the buckets are

$$\begin{aligned} B_1 &= \{0\} \\ B_2 &= \{1\} \\ B_3 &= \{2, 3\} \\ B_4 &= \{4, 5, 6, 7\} \\ B_5 &= \{8, \dots, 15\} \\ &\vdots \\ B_i &= \{2^{i-2}, \dots, 2^{i-1} - 1\} \text{ (for } 2 \leq i \leq L + 2\text{)} \\ &\vdots \\ B_{L+1} &= \{2^{L-1}, \dots, 2^L - 1\} \\ B_{L+2} &= \{2^L, \dots, 2^{L+1} - 1\} \\ B_{L+3} &= \{2^{L+1}, \dots, nC + 1\} \end{aligned}$$

In general we will denote the endpoints of B_i by $[s_i, m_i]$ where $s_i = m_{i-1} + 1$. (s_i stands for Smallest in bucket i , m_i stands for Max in bucket i .)

3. The sizes of the buckets will be bounded:

$$\begin{aligned} |B_1| &\leq 1 \\ |B_i| &\leq 2^{i-2} \text{ for } 2 \leq i \leq L + 2 \\ |B_{L+3}| &\leq nC + 1 - 2^{L+1} + 1 \end{aligned}$$

The ranges of the buckets will change. For example, if we do a DELETEMIN and remove 89 then we will now partition $[89, nC + 1]$. Note that the size of a bucket does not mean how many entries a bucket can hold. The size refers to how large the range for that bucket can be. So bucket B_1 can hold 10 elements as long as each one has key $dmin$. More generally, B_i may hold more than 2^{i-2} elements since some of the elements may have the same key.

4. The partition will always work as follows: The first $L + 2$ buckets partition

$$\{dmin, \dots, dmin + C\}.$$

The $(L + 3)$ rd bucket will be

$$\{dmin + C + 1, \dots, nC + 1\}.$$

KEY: by **Key Observation** above the next element picked (that is, the element with minimal key value) will always be in one of the first $L + 2$ buckets.

5. There is an array $H[1..L + 2]$ such that $H[i]$ is how many elements are in B_i .
6. Each bucket will have a linked list of elements that are in it.
7. The elements in B_i are in $[s_i, m_i]$.
8. Every element knows what bucket it is in.
9. We define a POTENTIAL FUNCTION. Let D be the structure.

$$\Phi(D) = \alpha \left(\sum_{x \in S} \text{index of bucket that } x \text{ is in} \right)$$

where we will determine α later.

We describe INSERT and DECKEY which will be easy.

1.1 INSERT

INSERT(x):

Do the comparisons $x < s_i$ for decreasing i until you get a YES. Now you have $s_1, s_2, \dots, s_{i-1} \leq x < s_i$. We now know that x should go into B_{i-1} .

Link x to the linked list B_{i-1} . (Formally we actually link (x, i) so that x ‘knows’ what bucket its in.)

$$H[i - 1] = H[i - 1] + 1.$$

END OF ALGORITHM FOR INSERT

The actual cost is dominated by the cost for the comparisons and is $O(\log L)$. (We could have used binary search to speed it up to $O(\log \log(L))$ but this will not help the analysis.) The most the potential function can increase occurs when we add the element to bucket $L + 3$. In this case the change in potential is $O(\log C)$. So the amortized cost is $O(\log C)$.

Note 1.1 There are two ways to decrease the amortized cost of INSERT. Unfortunately they still result in $O(\log C)$.

1. Instead of using sequential search use binary search to find where to insert. This cuts the actual cost from $O(\log C)$ to $O(\log \log C)$.
2. In our particular application (Dijstras algorithm) we only insert one element s with $d(s) = 0$ and all the rest of the elements v have $d(v) = nC + 1$. Hence we do not have to search for which bucket to put them into. Thus the actual cost is $O(1)$ per insert.

The increase in potential is still $O(\log L)$ per insert, so the amortized cost is still $O(\log L)$.

1.2 DECKEY

DECKEY(x, x')

Let i be such that $x \in B_i$ (this takes $O(1)$ to find since all elements know the bucket they are in).

Do the comparisons $x' \geq s_i$? $x' \geq s_{i-1}$? etc until you get a YES. If $x' \geq s_i$ then $x \in B_i$ then just replace x by x' . If $x' \in B[j]$, $j < i$, then

Remove x from $B[i]$.

$$H[i] = H[i] - 1$$

Place x' into $B[j]$.

$$H[j] = H[j] + 1.$$

END OF ALGORITHM FOR DECKEY

The number of comparisons made to find out which bucket to put x' into is $i - j$. So the actual cost is $O(i - j)$. Since x is removed from B_i and placed into B_j , the potential is decreased by $\alpha(i - j)$. Hence the amortized cost is

$$O(i - j) - \alpha(i - j).$$

Pick α so that this is $O(1)$.

1.3 DELETEMIN

We now look at DELETEMIN which will be more complicated. Every time DELETEMIN is called we will change the intervals of the buckets and (more important) if the min is found in any bucket other than B_1 then *every* element in the bucket where the min was found will be moved into a bucket of lower index.

DELETEMIN

Ask $H[1] \neq 0?$, $H[2] \neq 0?$... until a YES is encountered. Let j be the least numbers such that $H[j] \neq 0$. We now know that the MIN is in B_j . By **KEY** we know that $j \leq L + 2$.

Find the MIN of all the elements in B_j . (This may be expensive.)

Let $dmin$ be the MIN.

Remove $dmin$ from B_j .

$H[j] = H[j] - 1$.

Reset the buckets (see how after the algorithm)

Place the elements of B_j into their new correct buckets.'

END OF DELETEMIN ALGORITHM

We first show how to reset buckets in a way that is not quite right. We will then comment on how to make it right. Reset Buckets:

$$\begin{aligned} B_1 &= \{dmin\} \\ B_2 &= \{dmin + 1\} \\ B_3 &= \{dmin + 2, dmin + 3\} \\ B_4 &= \{dmin + 4, \dots, dmin + 7\} \\ B_5 &= \{dmin + 8, \dots, dmin + 15\} \\ B_6 &= \{dmin + 16, \dots, dmin + 31\} \\ &\vdots \\ B_i &= \{dmin + 2^{i-2}, \dots, dmin + 2^{i-1} - 1\} \text{ (for } 2 \leq i \leq j) \\ &\vdots \\ B_j &= \{dmin + 2^{j-2}, \dots, dmin + 2^{j-1} - 1\} \\ B_{j+1} &= \text{old } B_{j+1} \\ &\vdots \\ B_{L+1} &= \text{old } B_{L+1} \\ B_{L+2} &= \text{old } B_{L+2} \\ B_{L+3} &= \text{old } B_{L+3} \end{aligned}$$

There is one thing right and two things wrong with this bucket resetting. We discuss all of these and then we say how to adjust the buckets in a way

that makes the wrong things not a problem, while maintaining the right thing.

1. RIGHT THING: *Every* element of the old B_j now belongs in a lower bucket. The old B_j had minimal element dmin and had a range of 2^{j-2} elements. Hence old $B_j \subseteq \{\text{dmin}, \dots, \text{dmin} + 2^{j-2} - 1\}$. The new B_j has least element $\text{dmin} + 2^{j-2}$. Hence *every* element from the old B_j belongs in a lower indexed bucket. Note that it is crucial that $j \neq L + 3$, which comes from our particular application to Dijkstra's algorithm with bound C on the edges which yields **Key Observation** above.
2. WRONG THING: What if $j = 10$ and $B_{10} = \{\text{dmin}, \dots, \text{dmin} + 9\}$. The buckets as defined above do not work. You could have B_1, B_2, B_3, B_4 as defined, but the new B_5 would have an element from the old B_{11} in it. The way around this is to make the max endpoint of B_i for $i \leq j$ be the max of the value written and m_j . With this we need another convention- if $s_i > m_i$ then $B_i = \emptyset$.
3. WRONG THING: What if the new B_j overlaps with B_{j+1} ? It is important that we DO NOT change B_{j+1}, B_{j+2}, \dots since that would be extra work. We will upper bound B_j with $s_{j+1} - 1$.

We can now define the buckets correctly:

$$\begin{aligned}
B_1 &= \{\text{dmin}\} \\
B_2 &= \{\min\{\text{dmin} + 1, m_j\}\} \\
B_3 &= \{\text{dmin} + 2, \min\{\text{dmin} + 3, m_j\}\} \\
B_4 &= \{\text{dmin} + 4, \dots, \min\{\text{dmin} + 7, m_j\}\} \\
B_5 &= \{\text{dmin} + 8, \dots, \min\{\text{dmin} + 15, m_j\}\} \\
B_6 &= \{\text{dmin} + 16, \dots, \min\{\text{dmin} + 31, m_j\}\} \\
&\vdots \\
B_i &= \{\text{dmin} + 2^{i-2}, \dots, \min\{\text{dmin} + 2^{i-1} - 1, m_j\}\} \text{ (for } 2 \leq i \leq j) \\
&\vdots \\
B_j &= \{\text{dmin} + 2^{j-2}, \dots, \min\{\text{dmin} + 2^{j-1} - 1, s_{j+1} - 1\}\} \\
B_{j+1} &= \text{old } B_{j+1} \\
&\vdots \\
B_{L+1} &= \text{old } B_{L+1} \\
B_{L+2} &= \text{old } B_{L+2} \\
B_{L+3} &= \text{old } B_{L+3}
\end{aligned}$$

Assume that B_j has b elements in it. Amortized cost:

1. j questions to find that $H[j] \neq \emptyset$. $O(j)$ actual cost.
2. b elements had to be compared to find the minimum. $O(b)$ actual cost.
3. j adjustments to the bucket intervals. $O(j)$ actual cost.

The last part of the amortized cost is the possible moving of the $b - 1$ elements in B_j into lower buckets. First note that every one of them *will* go into a lower indexed bucket: The old B_j had minimal element d_{\min} and had $\leq 2^{j-2}$ elements in it. Hence the old B_j is contained in

$$\{d_{\min}, \dots, d_{\min} + 2^{j-2} - 1\}.$$

The new B_j has least element $d_{\min} + 2^{j-2}$. Hence *every* element from the old B_j will get moved to a lower indexed bucket. Note that it is crucial that $j \neq L + 3$, which comes from our particular application to Dijkstra's algorithm with bound C on the edges.

For each of the $b - 1$ elements in B_j (that are not d_{\min}) we ask a number of questions and then place it in a lower indexed bucket. We count this carefully since we are balancing two things:

- the more questions you have to ask the more steps you are taking (BAD NEWS)
- the more questions you have to ask the lower the index of the bucket you place the elements, and hence the more you lower the potential function (GOOD NEWS).

Let $x \in B_j$ (the old B_j). If we have to make q questions to find out which new bucket it is in, then it is in bucket B_{j-q} . Hence the potential decreases by αq . The potential change of just dealing with x is $q - \alpha q = (1 - \alpha)q$. Since we will be taking $\alpha > 1$ the amortized cost is highest when q is lowest. Hence we assume the worst case where every element is placed into B_{j-1} . Then the potential change is at at most $(b - 1)(1 - \alpha)$ (which is negative).

The total amortized cost is

$$O(j + b + b(1 - \alpha)).$$

We take α large enough so that this is $O(j) = O(L) = O(\log C)$.

1.4 SUMMING UP

In summary

INSERT has amortized cost $O(\log C)$

DECKEY has amortized cost $O(1)$

DELETEMIN has amortized cost $O(\log C)$.

Hence DIJKSTRA'S algorithm will have total cost

$$O(E + (\log C)V).$$

1.5 WHAT ELSE IS KNOWN

There is an algorithm that runs in time

$$O\left(E + \frac{\log C}{\log \log C} V\right)$$

that is not so hard to understand (it may be on a later homework).

There is also an algorithm that runs in time

$$O(E + (\sqrt{\log C})V)$$

which is more difficult.