Single-Source Shortest Paths with Integer Weights

Ruijie Fang

Introduction. This was an exercise proposed by Tarjan during his fall '20 graph algorithms seminar at Princeton. We are not aiming for crazily fast time bounds like Thorup's algorithm here, instead, we focus on improving the bucketting scheme of Dial's algorithm. The desired solution uses either an R-heap or a Fibonacci heap and achieves $O(n \log c + m)$ running time where c is the maximum integer valued weight. This note describes a simple formulation that achieves $O((n+m)\log(c))$-time. Our algorithm, which is based on a refined bucketting approach, runs in time $O((n+m)\log c)$ and $O(n+m+c)$ space. It is faster than Dial's algorithm (a trivial application of bucketting using $O(nc)$ buckets) but slower than Thorup's algorithm, which can solve integer-weighted SSSP in linear time.

Base idea: $O(nc+m)$-time algorithm using nc buckets. We can get an algorithm (also called Dial's algorithm in some literature) that runs in $O(nc+m)$-time using $O(nc+m)$-space by directly replacing the priority queue in Dijkstra's algorithm with a bucket queue of nc buckets, with the k-th bucket storing all vertices of distance k from source. The algorithm is as follows:

Algorithm Bucket-SSSP(G, $s \in V$, $w : V \times V \to \mathbb{Z}^+$) { // Solves SSSP from source s with weight function w.

1. Create an array $B[1...nc]$ of nc buckets and initialize all buckets to empty.
2. Let $d_s(v)$ denote the distance from s to a vertex v; initially let $d_s(v) = \infty$ for all $v \in V$.
3. For each outgoing edge $(s, v) \in E$, insert v into $B[w(s, v)]$.
4. While B contains non-empty buckets do
   (a) Scan through $B[1...nc]$ and pick the minimum nonempty bucket index i.
   (b) For each $u \in B[i]$ do
      i. Delete u from $B[i]$.
      ii. Let $d(s, u) := i$.
      iii. For each outgoing edge $(u, x) \in E$, if $d(s, u) + w(u, x) < d(s, x)$ then insert x into $B[d(s, u) + w(u, x)]$.
   (c) enddo
5. enddo

6. report $d_s(\cdot)$.

}

The algorithm is correct, since it is just Dijkstra's algorithm with a different queue implementation. As for the running time, we spend $O(nc)$ in step 4a and $O(m)$ total time to perform the edge relaxations. With a careful implementation, insertion and deletion from a bucket takes constant time (we can maintain a linked list in each bucket, for example). Each edge is relaxed at most once, hence the overall running time is $O(nc + m)$.

Speedup 1: Alternative implementation using interval trees. We define an interval tree to be a complete binary tree of size nc (similar to a tournament tree). Its root node represents the interval $[1, n]$, with left child and right child representing intervals $[1, nc/2]$ and $[nc/2 + 1, nc]$ correspondingly. Since the interval tree is a complete binary tree with size n, it has height $O(\log nc)$. We can use the interval tree to maintain the bucket list, so that we can determine, in $O(\log nc)$-time, the minimum non-empty bucket index. More specifically, we maintain one extra bit per node in the interval tree; the bit is set whenever a bucket in the interval the node represents is non-empty. A parent's bit is set if any of its children's bits is set to 1. (Again, this is similar to a tournament tree to anser range minimum/maximum queries). For a more detailed description of the implementation, please refer to footnote 1.

Swapping bucket queue with an interval tree in the above algorithm results in an algorithm that works in $O((n + m) \log nc)$-time, since insertion into the buckets take $O(\log nc)$-time (now we have to maintain the interval tree as well).[1]

Speedup 2: Bounding the number of active buckets. Observe the following two facts:

Fact. *The index of the buckets we visit in the main loop (step 4) of the Bucket-SSSP algorithm above is non-decreasing.*

The above fact comes from the greedy property of Dijkstra's algorithm: We iteratively relax edges, and always visit vertices that are closer in distance to the source first. Next, observe that

Fact. *At a vertex of distance k, the total number of reachable buckets is of size at most c.*

This comes from the definition of c as the maximum edge weight. Therefore, since the vertices we visit are monotone non-decreasing in distance, we don't need to consider any

[1]Notably, let tree T denote the interval tree, and let it support operations `insert`(T, x), `findMin`(T) and `delete`(T, x). All implementations work in $O(\log nc)$-time since we represent the interval $[0, nc]$. For `delete`$(\cdot)$ to work effortlessly, we store a binary value {0,1} on the leaves of the tree, with 0 denoting the bucket at leaf i is empty, and 1 denoting the bucket at leaf i contains content. We similarly maintain an additional bit in every internal node, where the bit of the parent is set to 1 iff one of its children has a bit set to 1 (a leaf has the bit set iff it is non-empty). We can change the Bucket-SSSP algorithm step 4a to a single `insert` operation, step 4b(i) to a `delete` operation, and step 4b(iii) to an `insert` operation. The main reason behind the $m \log nc$-term is now (due to the interval tree maintenance) we take $O(\log nc)$-time per edge relaxation.

vertex in buckets with indices less than k when visiting vertices of distance k. In other words, at any stage of the algorithm, when we consider vertices of distance k, only buckets in range $B[k...k+c]$ are relevant; hence, there are only $c+1$ active buckets (c additional buckets plus the current one) at any stage of the above algorithm.

Given the analysis above, it suffices to reduce the number of buckets we maintain from nc to c, and we can now simply index vertices at distance k into bucket $k \mod (c+1)$; since we know that, by the time we visit a vertex at distance k, all vertices of distance $< k$ would be removed from the buckets. Since there are now only $c+1$ buckets in total, we can use the interval tree structure discussed in the section above to maintain these $c+1$ buckets with operations taking time

- `insert`(T, x): Make bucket x in interval $[0...c+1]$ non-empty. Works in $O(\log c)$-time.[2]
- `delete`(T, x): Make bucket x in interval $[0...c+1]$ empty again. Works in $O(\log c)$-time.
- `findMin`(T, x): Find a minimum non-empty bucket index in range $[0...c+1]$. Works in $O(\log c)$-time.

Overall, the running time of the algorithm becomes $O((n+m)\log c)$.

A closing thought. There are other ways of reaching a similar time bound. We now briefly sketch the folklore technique of using a radix queue. A radix queue is a list of buckets of length $O(\log nc)$, with bucket i representing all vertices in range $[2^i, 2^{i+1}]$ (similar to the way we construct a binomial heap or a sparse table for range minimum queries). Since the range represented by each bucket grows exponentially, we get a logarithmic amount of buckets, and a linear search to find non-empty buckets now works in time $O(\log nc)$, making the entire SSSP algorithm run in $O(n \log nc + m)$ (note that, since insertion into a bucket is $O(1)$ and we take constant time to decide which bucket a vertex must go in, we spend $O(1)$ per relaxation, so the m term is separate from the $n \log nc$ term, unlike in our interval trees implementation).

It is also possible to replace the interval tree by a heap storing the indices of non-empty buckets, and the resulting algorithm would run in the same time and space bounds. In fact, this is the original algorithm we discussed during lecture, initially proposed by Dimitry Paramonov.

[2]Note that by inserting and deleting, we aren't really inserting and deleting nodes inside the tree. As footnote 1 discussed, we are only maintaining an extra bit per node in the tree that denotes whether any buckets in the current interval is non-empty. So "insert" and "delete" really should be "make-available" and "make-empty".