Algorithms I – supervision 6

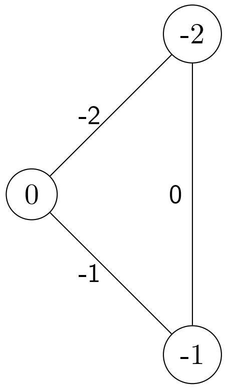
James Wood

April 27, 2015

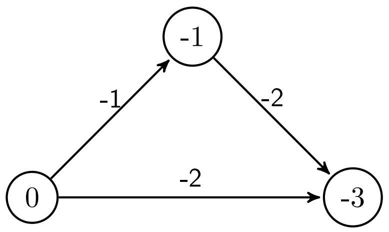
1 Dijkstra

(a) The algorithm starts at a given vertex, whose distance is marked as 0. All nodes directly connected to this vertex are added to a priority queue, sorted by weight of the connecting edge in ascending order (lightest first). The first vertex in the queue is then moved to, and is given a distance equal to the weight associated with it. Vertices directly conected to it without an assigned distance are then added to the queue, with value equal to the distance of the previous vertex plus the weight of the edge used to reach the new vertex, and the process is repeated. Assuming a connected graph, the algorithm terminates when all vertices have been assigned a distance, or equivalently when the queue is exhausted.

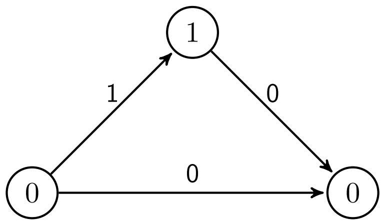
(b)



(c) Consider this graph:



The shortest path from left to right goes via the middle node. Adding 2 to the weight of each arc gives this:



The shortest path from left to right is now to go there directly. Hence, this tactic does not work.

(d) (i) Inserting a vertex into the queue takes $O\,1$ time, and is done $O\,|E|$ times. Extracting a vertex takes $O\,|V|$ time, and this is done $O\,|V|$ times. These give the total complexity $O\,(|E|+|V|^2)$, which is equivalent to just $O\,(|V|^2)$.

(ii) The estimate could be poor. Inserting now takes $O\,|V|$ time, giving a total complexity of $O\,(|V|\cdot|E|+|V|^2)$, which is simplified to $O\,(|V|\cdot|E|)$.

(iii) Inserting takes $O\,(\log|V|)$ time and extracting takes $O\,(\log|V|)$ time, giving complexity $O\,(|E|\cdot\log|V| + |V|\cdot\log|V|)$, which is $O\,(|E|\cdot\log|V|)$.

(iv) Inserting/modifying costs $O\,(\log|V|)$, and extracting costs $O\,(\log|V|)$, giving the same complexity as the binary heap.

(v) Inserting/modifying has $O\,1$ amortized cost, and extracting costs $O\,(\log|V|)$, giving complexity $O\,(|E| + |V|\cdot\log|V|)$.

2 Amortized and aggregate analysis

(a) Amortized analysis involves finding a worst-case sequence of instructions, analyzing the complexity of the sequence and dividing the cost of the sequence by the length of the sequence, yielding an average cost for each instruction. Aggregate analysis is done by creating a measure function for the given data structure, and giving the cost of each operation as its time complexity plus the change it causes in the measure of the data structure.

(b) (i) The `multipush` operation can be run on any stack, and will always have cost n, where n is the length of the list of items being pushed. Hence, its amortized cost is $O\,n$.

(ii)

```
class Queue
    Stack front
    Stack back

    void enqueue(Item x)
        back.push(x)

    Item dequeue()
        if front.isEmpty() then
            until back.isEmpty()
                front.push(back.pop())
        return front.pop()

    Boolean isEmpty()
        return front.isEmpty() ∧ back.isEmpty()
```

(iii) Let ϕ be a function measuring the length of `back`. `enqueue` has time cost 1 plus potential cost 1. `dequeue` has either time cost 1 plus potential cost 0, or time cost $n+1$ plus potential cost $-n$. `isEmpty` has time cost 1 plus potential cost 0. Each of these possibilites give linear aggregate cost, so the amortized running time of each operation is linear.

3 Linear cost

In the operation of the queue, it is always the case that either all of the items are in the correct place, or exactly one item is too high in priority for its position. The latter situation occurs after either an insertion to the start of the array or a call to `decreaseKey`. Keeping high-priority (low-key) items at the right, the out-of-place item is too far left. Bubble sort will move it to the correct position in one pass. If a standard bubble sort algorithm is used, there will also be another pass to check the array, but this doesn't affect asymptotic complexity.

4 Disjoint-sets, linked lists and trees

(a) The disjoint-set data structure represents a collection of sets, with no two of those sets sharing any elements. Items can be queried as to which set they are in, and sets can be merged together (preserving all elements, since they are disjoint to each other).

(b) Each subset is stored as a linked list, and these are all stored inside a linked list. This takes $O\,n$ time to create, $O\,n$ time to find elements and $O\,(s+l)$ time to merge two subsets (where s is the number of subsets and l is the size of the first subset of the two being merged).

(c) In Kruskal's algorithm, data representing each edge of the graph are put into a priority queue with lighter edges first. Also, data representing vertices are put into a disjoint-set data structure. Edges are taken from the priority queue and added to the output tree iff they connect vertices from different subsets. When an edge is added, the sets containing its vertices are merged. The algorithm terminates when there is only one disjoint set left.

(d) At most $|E|$ edges are chosen. If the edges are given in a sorted list (which took $O\,(|E| \cdot \log |E|)$ time to create), the edge is taken from the priority queue in constant time. The resulting computation consists of two `find`s and a `merge`, which combined can take $O\,|V|$ time. In a connected graph, $|V| \leq 2 \cdot |E|$, so this is equivalent to $O\,|E|$. Therefore, the total running time is $O\,(|E| \cdot \log |E|)$.

5 Recurrence relations

(a) Observe:

$$\begin{aligned} T\,u &\leq 2 \cdot T\,(u^{\frac{1}{2}}) + c && \text{for some constant } c \\ &\leq 2^k \cdot T\left(u^{\frac{1}{2^k}}\right) + \left(2^k - 1\right) \cdot c && \text{for any } k \text{ small enough} \end{aligned}$$

This can be verified by induction. Take the implicit base case to be $T\,2 = 1$, and choose k such that:

$$\begin{aligned} u^{\frac{1}{2^k}} &= 2 \\ u &= 2^{2^k} \\ \lg u &= 2^k \end{aligned}$$

Substituting this in, we get:

$$\begin{aligned} T\,u &\leq \lg u \cdot T\,2 + (\lg u - 1) \cdot c \\ &\leq (c+1) \cdot \lg u - c \\ &\in O\left(\lg u\right) \quad \text{QED} \end{aligned}$$

(b) Similarly, we find that:

$$\begin{aligned} T\,u &\leq 2^k \cdot T\left(u^{\frac{1}{2^k}}\right) + c \cdot \lg\left(u^{\frac{k}{2}}\right) \qquad \text{for some } c \\ &\leq 2^k \cdot T\left(u^{\frac{1}{2^k}}\right) + \frac{c}{2} \cdot k \cdot \lg u \end{aligned}$$

Again, try:

$$\begin{aligned} u^{\frac{1}{2^k}} &= 2 \\ u &= 2^{2^k} \\ \lg u &= 2^k \\ \lg\left(\lg u\right) &= k \end{aligned}$$

Using the last two results and the base case $T\,2 = 1$, we rewrite the inequality as:

$$\begin{aligned} T\,u &= \lg u + \frac{c}{2} \cdot \lg\left(\lg u\right) \cdot \lg u \\ &\in O\left(\lg u \cdot \lg\left(\lg u\right)\right) \quad \text{QED} \end{aligned}$$

6 k-Anonymity

(a) Yes. There are an odd number of elements, so there had to be some group of three, which has a cost of at least 3. Pairings have a cost of at least 1. This solution incurred only those costs, so is optimal.

(b) Let the length of the list be $d \cdot k + r$, where $r \in [0..k)$. Produce all possible partitionings of the list where the size of each partition is in $[k..2k)$. In each partition, increase all of the elements to become equal to the largest element, keeping track of the cost. Sum the costs from all partitions in each partitioning, and choose the partitioning with the lowest cost. This can be reässembled to give a valid output.

This seems to have factorial cost, since almost all possible partitionings have to be checked exhaustively.

(c) Some improvement can be gained by memoization, though this doesn't reach linear running time.