Algorithms I – supervision 6

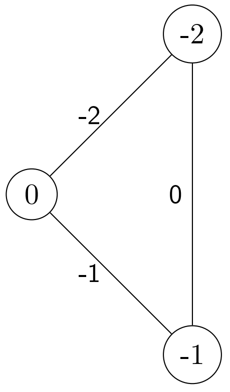
James Wood

April 26, 2015

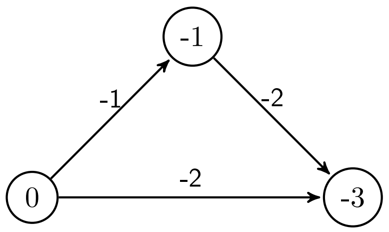
1 Dijkstra

(a) The algorithm starts at a given vertex, whose distance is marked as 0. All edges connected to this vertex are added to a priority queue, sorted by weight in ascending order (lightest first). The first vertex in the queue is then traversed, and the new vertex is given a distance equal to the weight of the edge. Its edges are then added to the queue, and the process is repeated (assigning distances equal to the weight of the edge plus the distance given to the vertex at the other end of the edge). Edges between vertices with distances already calculated are removed from the queue. Assuming a connected graph, the algorithm terminates when all vertices have been assigned a distance, or equivalently when the queue is exhausted.

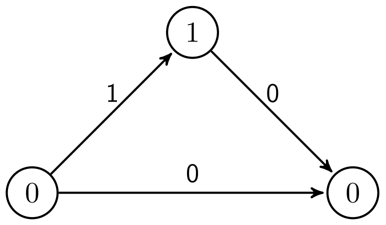
(b)



(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)

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() $\wedge$ 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

When only a single item of an array is put out of order, bubble sort requires only two passes (one to put the element in the correct place, and another to check that the array is sorted).

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 $O\left(|E| \cdot \log|E|\right)$ 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 $O\left(|E| \cdot \log|E|\right)$.