Algorithms – CMSC 37000

Amortized analysis

Instructor: László Babai   Ry-164   e-mail: `laci@cs.uchicago.edu`
Last updated: February 2, 2009

"Amortized analysis" is an elegant method to analyze the cost of a sequence of requests made to a data structure. The method is particularly useful if the cost of the execution of a request varies greatly depending on the current configuration.

Suppose a data structure maintains a list L of data and serves requests $R_1, \dots, R_k$. Upon request R_j, the list L is updated to the new list $R_j(L)$; this operation incurs cost $c(R_j, L)$. In "amortized analysis," we assign a possibly different value, $a(R_j, L)$, called the "amortized cost," to this update. Given a (finite or infinite) sequence $S = (T_0, T_1, \dots)$ of requests ($T_i \in \{R_1, \dots, R_k\}$), let L_0 be the empty list and $L_{i+1} = T_i(L_i)$. Let $S_t = (T_0, \dots, T_{t-1})$ be the first t requests. The *(actual) cost of* S_t is $c(S_t) = \sum_{i=0}^{t-1} c(T_i, L_i)$, while the *amortized cost of* S_t is $a(S_t) = \sum_{i=0}^{t-1} a(T_i, L_i)$. We say that the assignment $a(R_j, L)$ of "amortized costs" is *correct* if for all sequences S and all t,

$$a(S_t) \geq c(S_t). \qquad (1)$$

This is usually proved by an accounting trick. We set up an "escrow account" with initial balance zero. At step t we "charge" $a(T_{t-1}, L_{t-1})$, but our actual cost is $c(T_{t-1}, L_{t-1})$. If the amount charged exceeds the actual cost, the surplus goes into the escrow account; if the amount charged is less than the actual cost then the difference has to be made up from the escrow account. The escrow account is not permitted to slip into the red (must always keep nonnegative balance).

Formally, we define $e(S_t) := a(S_t) - c(S_t)$; this is the escrow balance after step t. The balance condition on the escrow account then says that for all S and t,

$$e(S_t) \geq 0. \qquad (2)$$

This statement is usually formalized as a loop invariant and is verified by induction on t.

On the next two pages, two toy examples follow, one of them with solution. An example of great elegance and significance is given in the analysis of the "Fibonacci heap" data structure by Fredman and Tarjan. This data structure maintains a set of real numbers and supports the following requests: INSERT, EXTRACT_MIN, DECREASE_KEY, with amortized costs $O(1)$ for INSERT, $O(\log n)$ for EXTRACT_MIN (where n is the number of data currently stored), and, startlingly, $O(1)$ for DECREASE_KEY. So the total cost of a sequence of r INSERTs, s EXTRACT_MINs, and t DECREASE_KEYs (starting from the empty set of data) is $O(r+t+s\log r)$.

Remark. The Fibonacci heap data structure also supports INCREASE_KEY and DELETE at $O(\log n)$ amortized costs. Note the asymmetry between the cost of DECREASE_KEY and INCREASE_KEY. The savings in DECREASE_KEY results in optimal implementations of Dijkstra's algorithm for min cost paths and Jarník's (a.k.a. Prim's) algorithm for min cost spanning trees.

The best explanation of the Fibonacci heap data structure that I am aware of is in the orginal article,

M. L. FREDMAN, R. E. TARJAN: Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* **34** (1987), 596-615.

Another good description is in the text

T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, C. STEIN: *Introduction to Algorithms.* MIT Press and McGraw Hill, 2001. ISBN 0-262-03293-7. Chapter 20: Fibonacci Heaps, pp. 476-497.

Wikipedia also has a pretty good article on it, which, however, avoids the financial terminology (they call the escrow balance a "potential function," an expression borrowed from physics and also used frequently in the analysis of algorithms); the drawback is that the term "amortized" becomes disconnected from the rest of the terminology.

---

1. (Increment and double data structure) A data structure maintains a nonnegative integer X as a linked list of bits and serves two requests: INCREMENT ($X := X + 1$) and DOUBLE ($X := 2X$). The costs are as follows: unit cost for DOUBLE (appending a zero at the end); and $k + 1$ units of cost for INCREMENT, where k is the number of trailing ones in X_{old} (which is the same as the number of trailing zeros in $X_{new} = X_{old} + 1$. So $k + 1$ is the number of consecutive bits to be switched in executing INCREMENT. For instance, the cost of adding 1 to $X = 23$ is 4 units (23 in binary is 10111, 24 is 11000, so the last four bits are switched).

Initially, $X = 0$, represented by the empty string. Prove that the cost of any sequence of n requests is $O(n)$. Use amortized analysis.

Solution. Our goal is to assign amortized costs of $O(1)$ to each request; then the total amortized cost will be $O(n)$ and therefor the actual cost will also be $O(n)$.

We need to give specific values to the amortized costs ($O(1)$ is not specific). Let us charge 1 unit for DOUBLE and 2 units for INCREMENT.

Let us consider the following statement.

- The escrow balance is the number of 1s in the binary expansion of X.

Lemma. This statement is a "loop invariant" in the sense that if it is true before executing a request, it remains true after the execution.

Proof. If the request is "DOUBLE" then the actual cost (1 unit) is precisely covered by the charge, so the escrow balance is unaffected.

If the request is "INCREMENT," suppose X has k trailing 1s before the execution of the command and b digits "1" total. After the execution of

INCREMENT, the new X will have $b-k+1$ digits “1.” The actual cost is $k+1$, the charge is 2, so the escrow balance goes down by $(k+1)-2 = k-1$. If before the request the escrow balance was b (as required), now it is $b-(k-1) = b-k+1$, as required. QED

It now follows by induction on t that for all t, the escrow balance is the number of 1s in the binary expansion of X (base case: $t=0$, $X=0$, balance zero; inductive step: the Lemma). Consequenctly the escrow balance remains nonnegative at all times. QED

It follows that the (actual) cost of any sequence of n requests is $\le 2n$ (because the amortized cost is $\le 2n$).

---

2. (Queue via stacks) We simulate queue Q (a FIFO list) using two stacks R and S as follows:

- simulate enqueue(Q,x) by push(R,x)
- simulate dequeue(Q) by the following program:

```
**if** $S$ empty **then** clear$(R,S)$
**end**(**if**)
$x :=$ pop$(S)$
**return** $x$
```

where clear(R,S) is the procedure

```
**while** $R$ not empty
    $x :=$ pop$(R)$
    push$(S,x)$
**end**(**while**)
```

(a) Verify that this is a correct simulation. State a simple invariant which can be used to verify correctness.

(b) The actual cost of “pop” and “push” is one unit each. Now we add the operation “k-dequeue,” meaning that we dequeue Q k times or until the queue becomes empty. Add this infinite set of operations (k-dequeue for all k) to the simulation by two stacks.

Prove that the cost of any sequence of n requests is $O(n)$. Use amortized analysis. Prove that the following amortized cost table works: enqueue: $O(1)$; k-dequeue: free. State a specific value of the charge for enqueue (how many units).