Amortized Analysis: a Summary

Patrice Belleville
Department of Computer Science
University of British-Columbia
February 15, 2011

1 What is amortized analysis?

Amortized is a topics in analysis of algorithms that is unfortunately not covered well in most textbooks. The only text that actually discusses it in details is that by Cormen et al. [1] (chapter 17), but the authors do not spend enough time explaining the intuition behind the potential method (which is the only method that will be discussed in this document). It is also explained very briefly in the book by Goodrich and Tamassia [3]. Other texts use it, but either do not mention its name, or assume that the reader already knows about it. This short document is an attempt at filling in this gap, and giving you an additional source of information beyond the notes you may have written in class. I will refer to the examples we have discussed, or will discuss, in class, tutorials, and on the assignments from time to time.

Let us first talk about what amortized analysis is not.

- It is **not** a technique used to design algorithm. If you are given a problem to solve, you do not start by saying: "I can take the general structure of an amortized analysis algorithm[1] and fill in the blanks to get a solution to my problem". Hence it is not like greedy algorithms, divide-and-conquer or dynamic programming. You can not use amortized analysis to design your solution.
- It is **not** a way to speed up an already designed algorithm. You can not take an algorithm and make it faster using amortized analysis.

Amortized analysis is a technique that we use to *analyze* an algorithm's running time, and obtain a good upper-bound on the worst-case running time of a *sequence* of operations. These operations can be the parts of an algorithm, or simply operations performed in sequence on a data structure. Once again

- Amortized analysis only allows us to obtain a bound on *the worst-case running time of a sequence of operations*. It does not help us get a better bound on the worst-case running time of *one* operation, because it relies on the fact that even if one operation can be expensive, it's not possible to only perform expensive operations.

[1] Whatever that might be.

- Amortized analysis does not change the algorithm in any way. It's just a way to count the running time of a sequence of operations more precisely than by just adding up the worst-case running times of each operation in the sequence.

2 The potential method

As the old adage says, "Time is Money". When thinking about amortized analysis, it helps to think of running time as a cost we have to pay. That is, each step costs us 1$, and the worst-case running time of a sequence of operations is the maximum amount we will need to pay to execute these operations. The idea behind the potential method is the following:

- The i^{th} operation we perform costs us a certain amount of money. This it the *real cost* of operation i, denoted by $cost_{real}(op_i)$.

- To help us analyze the running time of a sequence of operations, we will pay a specified amount of money each time we perform an operation. The amount paid to perform the i^{th} operation is called the *amortized cost* of operation i, and is denoted by $cost_{am}(op_i)$.

  If the amortized cost of operation i is greater than its real cost, then we put the remaining amount $(cost_{am}(op_i) - cost_{real}(op_i))$ in a bank account. The amount accumulated in the account after i operations is called the *potential* of the data structure (or algorithm) at this point, and denoted by $\Phi(D_i)$.

  If the amortized cost of operation i is smaller than its real cost, then we need to withdraw an amount equal to $cost_{real}(op_i) - cost_{am}(op_i)$ from the bank to pay for the difference. That is, the potential will decrease by this amount.

The bank account (potential function) should have three properties in order to be valid.

- The initial balance is 0, since we did not put any money in the bank before we started executing the algorithm or the sequence of operations on the data structure. That is,

$$\Phi(D_0) = 0 \tag{1}$$

- We can never borrow money from the bank, since the bank has no way of predicting when the algorithm will terminate, and hence can not take the chance of ending up with a deficit (this would mean the algorithm did some work without paying for it). That is,

$$\Phi(D_i) \geq 0 \text{ for every } i \geq 0 \tag{2}$$

- Finally, the amount of money in the bank goes up or down according to the difference between the amount we paid to perform an operation, and the amount that operation actually cost, as stated earlier. That is,

$$\Phi(D_i) = \Phi(D_{i-1}) + cost_{am}(op_i) - cost_{real}(op_i) \tag{3}$$

Intuitively, when we use amortized analysis, we prefer to pay a predictable amortized cost for each operation, instead of paying its unpredictable real cost. Assuming the three properties hold, the total amount we will have paid for n operations is at least as large as what they actually cost, which means that we can use the sum of the amortized costs as an upper-bound on the sum of the real costs. Stated mathematically:

$$\begin{aligned}\sum_{i=1}^{n} cost_{am}(op_i) &= \sum_{i=1}^{n} (cost_{real}(op_i) + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} cost_{real}(op_i) + \sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1}))\end{aligned}$$

The second summation on the right-hand side is interesting, because every term except for two will appear twice, once with a positive sign, and once with a negative sign. These cancel out, and so we are left with

$$\sum_{i=1}^{n} cost_{am}(op_i) = \sum_{i=1}^{n} cost_{real}(op_i) + \Phi(D_n) - \Phi(D_0)$$

and since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$, this implies that

$$\sum_{i=1}^{n} cost_{am}(op_i) \geq \sum_{i=1}^{n} cost_{real}(op_i)$$

3 Choosing a potential function

So how do we choose a potential function? This is by far the hardest part of the potential method. Other methods, such as the accounting method, do not use a potential function, but they then need to choose an amortized cost for each operation, which is in my opinion even more difficult since you need to make several non-obvious choices instead of only one. Once the potential function has been chosen correctly, computing amortized costs for the operations is usually somewhat simpler.

When we have a data structure that supports insertions and deletions, we can usually come up with a potential function by considering how much more work the algorithm will need to perform on an element once it has been inserted in the data structure. For instance, consider a stack with operations `PUSH`(x), `POP`() and `MULTIPOP`(j),

where the last operation pops j elements off the stack at once. After an element has been pushed onto the stack, what else remains to be done with it? Well, pop it off, obviously. This means that for every element that we push onto the stack, we should keep an extra \$1 in the bank that will be used to pay for the corresponding pop (if it actually happens; if not we will be left with a positive balance at the end). This suggests that we might want to use

$$\Phi(D_i) = \text{the number of elements of } D_i$$

as our potential function. We saw in class that this choice gave us a constant amortized cost per operation, and hence allowed us to prove that a sequence of n `PUSH`, `POP` and `MULTIPOP` operations runs in $O(n)$ time.

In the binary counter example we discussed in class, we used the number of 1's in the array of bits as potential, because each 1 bit may be turned into a 0 bit later on. Of course, a 0 bit may also be changed to a 1, but one `INCREMENT` operation only changes one 0 bit to 1, whereas it may change many 1 bits to 0. Thus there is no need to worry about 0 turning into 1 in our potential function, because these events are easily predictable.

Something very similar happened with our analysis of the Disjoint Sets data structure: we considered how often a node might be moved upwards in its tree by the path compression operation, and chose the value $\texttt{amount}(N)$ to allow us to pay for each of the times we perform path compression on this node. However, unlike the stack example, and unlike the complete example from Section 5, this amount did not pay for *every* time we move the node upwards in its tree. There remained $\log^* n$ instances that could not be paid for by money previously stored in the bank, and had to be paid when we were actually performing the `find` operation. That is why the amortized cost of `find` ended up being $O(\log^* n)$, and not $O(1)$.

In each of these examples, we can think of the money currently in the bank as being attached to specific objects inside our data structure (which is what the *accounting* method does). There are however situations where it is more convenient not to try to associate an amount with a specific object in this way. For instance, in the amortized analysis of the running time of the operations on the *Fibonacci Heap* data structure[1] (chapter 19), the potential of D_i is equal to twice the number of trees in the data structure, plus the number of "marked" nodes.

4 Computing amortized costs

Computing the amortized cost of an operation is usually relatively straightforward, with one caveat. The operations whose running time does not vary much (like `PUSH` or `POP` in our stack example) are easy to deal with. But what about operations for

which some invocations will execute quickly, whereas others will take much longer? These are after all the reason why we need to use amortized analysis in the first place.

The trick is that there will typically be a parameter that can be used to determine both the real cost of the operation, and the change in potential it causes. For instance, with the `MULTIPOP`(j) operation this parameter was the value of j. For the binary counter example, it was the number of 1 bits that were changed to 0's (that is, the number of 1 bits to the right of the righmost 0 bit in the array). In the problem from Section 5, the parameter is the number of `MERGE` operations that need to be performed in order to complete the insertion of the new element. In our analysis of disjoint set structures, this parameter was the number of times x during path compression that we moved a node whose rank was in the same interval as its parent's rank.

In all cases, we need to write both the real cost and the change in potential as a function of that parameter and, assuming that we chose the right potential function, the parts of the expressions will cancel out when we add them to obtain the amortized cost of the operation (please look through the examples mentioned earlier to see how it worked in each case).

5 Example: a simple data structure that supports fast SEARCH and INSERT operations

We can perform a SEARCH operation on a sorted array with n elements in $O(\log n)$ time using binary search. Unfortunately, the INSERT operation runs in $\Theta(n)$ in the worst-case, as up to n elements may need to be shifted over to make room for the new element. In this exercise, we will discuss a way to improve the time for INSERT significantly by keeping several sorted arrays instead of only one. The worst-case running time of SEARCH will become worse, but not by a lot[2].

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \log_2(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1} n_{k-2} \ldots n_0 \rangle$. We will use k sorted arrays A_0, A_1, $\ldots A_{k-1}$. Array A_i is either empty, or contains 2^i element, depending on whether $n_i = 0$ or $n_i = 1$, respectively. The total number of elements held in all k arrays is therefore $n = \sum_{i=0}^{k-1} n_i 2^i$.

For instance, when $n = 13_{10} = 1101_2$, we would use three sorted arrays: one with 1 elements, one with 4 elements, and one with 8 element (the array with 2 elements would remain empty).

Although each individual array is sorted, there is **no particular relationship** between elements in different arrays.

[2]You most likely already know that balanced binary search trees will achieve this goal; however they are very complicated. Here we are looking at a much simpler solution.

1. First, we describe how to perform the SEARCH operation for this data structure. The idea is simple: we perform a binary search on each of A_0, A_1, A_2, $\ldots A_{k-1}$. If we find the element in one of the arrays, then we return `true` (or a pointer to the element). Otherwise we return `false` or a null pointer.

   There are $O(\log n)$ arrays, and each binary search takes $O(\log n)$ time, and so the SEARCH operation runs in $O(\log^2 n)$ time.

2. Now, let us describe how to insert a new element into this data structure. The insertion algorithm is similar to binary addition, when one of the integers we are adding is always 1. In this code, assume that `merge` is the subroutine used by the `Mergesort` algorithm to merge two sorted lists.

```
Algorithm Insert(x)

  i ← 0
  newA ← new array with x as only element.

  while A_i is not empty do
    newA ← merge(A_i, newA)
    A_i ← empty
    i ← i + 1
  endwhile

  A_i ← newA
```

   The worst-case running time of the `Insert` operation is in $\Theta(n)$, for instance if the insertion is performed when the data structure already contains $n = 2^t - 1$ elements for some non-negative integer t.

3. Finally, let us use the potential method to analyze the worst-case running time of a sequence of n `Insert` operations. We define the potential function Φ as:

$$\Phi(D_i) = \sum_{j=0}^{k-1} (k-j) n_j 2^j$$

   where $k = \lceil \log_2(n+1) \rceil$, and

$$n_j = \begin{cases} 1 & \text{if } A_j \text{ contains elements.} \\ 0 & \text{if } A_j \text{ is empty.} \end{cases}$$

   Why? Because each element currently in array j will potentially need to participate in `MERGE` operations to move into arrays $j+1$, $j+2$, $\ldots$, k. That is,

each of the 2^j elements currently in array j may still be involved in $k-j$ further `MERGE` operations.

To simplify our analysis, we will split the insertion into two stages:

- Adding the element to the data structure as a one-element array.
- Performing `merge` operations until we have at most one array with 2^i elements for each value of i.

The real cost of the first stage is in $\Theta(1)$, and it increases the potential of D_i by k. Its amortized cost is therefore $1 + k$.

Consider now the sequence of `merge` operations. One call to `mergeList` with two lists with 2^j elements takes $2 \cdot 2^j = 2^{j+1}$ units of time (so, constant time per element involved). Since 2^{j+1} elements move from A_j to A_{j+1}, the potential $\Phi(D_i)$ decreases by 2^{j+1}. Hence the amortized cost of each `merge` operation is 0.

Thus the amortized cost of one `Insert` operation is in $O(1 + k)$, which means that n `Insert` operations take time at most $n(1 + k) \in \Theta(n \log n)$.

6 Further examples

A report submitted by a student for a course at Princeton University [2] contains several other examples where amortized analysis is used. Most of the examples are not fully written out, but these will give you an idea of some other situations where amortized analysis can be used.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, third edition, 2009.

[2] Rebecca Fiebrink. Amortized analysis explained. See `http://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf`, April 2007. Student project.

[3] R. Goodrich, M. T. and Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley & Sons, Inc., 2002.