Amortized Analysis

Andres Mendez-Vazquez

February 26, 2018

Contents

1 Introduction

The amortized analysis method originally emerged as aggregated analysis (Ullmanm, Aho and Hopcroft used a version of it to analyze set representations) to analyze basic set operation in binary trees and union operations. The technique was first formally introduced by Robert Tarjan in his paper *Amortized Computational Complexity*. It was used to study balanced binary trees and the operation for set representations.

2 The Basic Methods

We have the following methods:

- Aggregate analysis determines the upper bound T(n) on the total cost of a sequence of n operations. Then calculates the amortized cost to be $T(n)/n$.
- The accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations.
- The potential method is like the accounting method, but overcharges operations early to compensate for undercharges later.

It is noticeable that the last one was introduced by Cormen et al.

2.1 Aggregate Analysis Basics

The classic example of the for this analysis is the stack with operations:

1. POP

   This remove stuff from the top of the stack

2. PUSH

   Thus put stuff at the top of the stack

3. MULTIPOP

   This pops multiple times using the POP as part of it:

```
Multipops(S,k)
      while not Stack-Empty(S) and k>0
            POP(S)
            k = k-1
```

Then, we have two cases

1. Worst Case:

   Multipops is bounded by $\min(s, k)$. Therefore, in the worst case, we could have at most $n-1$ pushes and one multipop with $k = n-1$. Thus, the worst complexity is $O(n)$. Then for n operations, we have that the complexity is $O(n^2)$.

2. The Amortized analysis

   Even when the Multipop operation can be expensive, it is clear that depends on the pops and pushes done before it. Therefore, any sequence of n push, pops and multipop operations on an initially empty stack cost at most $O(n)$ because of the very nature of multipops. It is more, the number of times a pop or a multipop can be called on a non-empty stack is at most the number of push operations.

   The average cost is then $\frac{O(n)}{n} = O(1)$, thus each of the operations has an amortized cost of $O(1)$.

2.1.1 Binary Counter

The Binary Counter is an array of bits to be used as a counter:

0	0	0	..	..	0	0
a_n	a_{n-1}	a_{n-2}			a_1	a_0

Where

$$x = \sum_{i=0}^{n} a_i 2^i \tag{1}$$

So we have the following algorithm to count

Algorithm 1 Binary Counter

1. Increment(A)
2. $i = 0$
3. while $i < A.length$ and $A[i] == 1$
4. $A[i] = 0$
5. $i = i + 1$
6. if $i < A.length$
7. $A[i] = 1$

Thus:

1. At the start of each iteration of the while loop in lines 2–4, we wish to add a 1 into position i.

2. If $A[i] == 1$, then adding 1 flips the bit to 0 in position i and a carry of 1 for $i+1$ on the next iteration of the loop.

3. If $A[i] == 0$ stop.

4. If i $<$ A.length, we know that $A[i] == 0$, so flip to a 1.

The cost of each INCREMENT operation is linear in the number of bits flipped. The worst case is $\Theta(k)$ in the worst case!!! Thus, for n operations we have $O(kn)$.

So look at this

1st Count	0	0	0	0	0	0	0	0	1
2nd Count	0	0	0	0	0	0	0	1	0
3rd Count	0	0	0	0	0	0	0	1	1
4th Count	0	0	0	0	0	0	1	0	0
5th	0	0	0	0	0	0	1	0	1
6th	0	0	0	0	0	0	1	1	0
7th	0	0	0	0	0	0	1	1	1
8th	0	0	0	0	0	1	0	0	0

Thus, we have that

1. $A[0]$ flips $\lfloor n/2^0 \rfloor$ time

2. $A[1]$ flips $\lfloor n/2^1 \rfloor$ time

3. etc

Thus, we have for $i = 0, 1, 2, ..., k-1$ $A[i]$ flips $\lfloor n/2^i \rfloor$ for $i > k$ no flips at all. Then, the total work is

$$\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n \tag{2}$$

2.2 Accounting Method Basics

In the accounting method, we charge each operation a certain *credit* using the following idea:

- When an operation, with an **amortized cost** $\widehat{c}_i$, exceeds its actual cost, we give the difference to a *credit* (To be stored in the data structure) in order to pay for later operations whose amortized cost is less than their actual cost.

Here is where the analysis becomes some what complex because we need to keep the following property:

- We must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence, i.e. :

$$\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i$$

Therefore, the credit stored in the data structure is:

$$\sum_{i=1}^{n} \widehat{c}_i - \sum_{i=1}^{n} c_i.$$

2.2.1 The Binary Counter

For the amortized analysis, we have the following cost distribution:

1. We charge 2 units of cost to flip a bit to 1. One to pay the setting of the bit to 1, and 1 to pay to flip the bit to 0.
2. We do not charge nothing to flip the bit to zero because we use the credit stored at the bit for this.

The cost of resetting the bits within the while loop in the code is paid for by the credit stored at the reseted bits. In addition, the **increment** procedure set at most one bit in the last two lines, thus the amortized cost of the **increment** operation is at most 2 units. Finally:

- The numbers of 1 at the bit counter never becomes negative.
- The Binary counter never charges a 0 flip if the bit was never changed to 1.

Therefore, the credit never becomes negative, or $\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i$. Thus the amortized cost of n operations is $O(n)$.

2.3 Potential Method Basics

We have the following steps for the potential method:

1. n operations are performed in initial data structure D_0.
2. c_i be the actual cost of the ith operation and D_i the data structure resulting of that operation for $i = 1, 2, ..., n$, when $c_i$is applied to D_{i-1}.
3. We have a potential function $\Phi : \{D_0, D_1, ..., D_n\} \to \mathbb{R}$ that describe the potential energy on each data structure D_i, for $i = 1, 2, ..., n$.
4. Then, we have an **amortized cost**: $\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Then, the total amortized cost of the n operations is

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} \left(c_i + \Phi\left(D_i\right) - \Phi\left(D_{i-1}\right)\right) \\ &= \sum_{i=1}^{n} c_i + \Phi\left(D_n\right) - \Phi\left(D_0\right).\end{aligned}$$

Important

If we can define a potential function Φ such that $\Phi\left(D_n\right) \geq \Phi\left(D_0\right)$, then the total amortized cost is such that $\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i$. However, we do not know how many operations might be performed. Therefore, we will require that

- $\Phi(D_i) \geq \Phi\left(D_0\right)$ for all i or if $\Phi\left(D_0\right) = 0$ then $\Phi\left(D_i\right) \geq 0$.

Note

If the potential difference $\Phi\left(D_i\right) - \Phi\left(D_{i-1}\right)$ is positive, then the amortized cost $\widehat{c}_i$ represents an overcharge to the ith operation.

2.3.1 Stack Operations

We will define the potential function Φ on stack as the number of elements in the stack. For an empty stack, $\Phi\left(D_0\right) = 0$, and because that number is never negative

$$\Phi\left(D_i\right) \geq 0 = \Phi\left(D_0\right).$$

We have three cases to analyze:

Case "Push"

Thus, if the ith operation on a stack containing s objects is a push, then the potential difference is

$$\begin{aligned}\Phi\left(D_i\right) - \Phi\left(D_{i-1}\right) &= (s+1) - s \\ &= 1.\end{aligned}$$

Then:

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi\left(D_i\right) - \Phi\left(D_{i-1}\right) \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

Case "Multipop"

Here, the ith operation on the stack is a multipop, thus $k' = \min(k, s)$. Then, the actual potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = -k'$. Then, the amortized cost of the multipop is

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0. \end{aligned}$$

Case "Pop"

It is similar to mulipop, 0.

The amortized cost for all the three operations is $O(1)$. Therefore, the total amortized cost of n operations is an upper bound of the total cost. The worst-case cost of n operations is therefore $O(n)$.

3 Example: Dynamic Tables

In the Slides.