An Introduction to Amortized and Competitive Analyses

K. Subramani
LDCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Introduction

In this lecture, we shall study two novel techniques of analysis, viz. Amortized Analysis and Competitive Analysis. These techniques are not only useful from the perspective of analyzing Online algorithms, but are interesting in their own right as well. In Section §2, we shall discuss 3 techniques for amortized analysis, while competitive analysis is introduced in Section §3.

2 Amortized Analysis

Amortized analysis is used to show that the average cost of an operation over some data structure is small, if one averages over a sequence of operations, although a single operation within the sequence may be expensive. Amortized analysis is fundamentally different from average-case analysis in that probability is not involved; the analysis guarantees the average performance of each operation in the worst case. We illustrate the use of amortized analysis through two problems.

The stack data structure is one that is used extensively in applications. It usually supports two operations, viz. *Push* which pushes a single item on top of the stack and *Pop*, which deletes a single item from the top of the stack. Let us augment the standard stack with an extra operation called *Multi-Pop(k)* which pops k items from the top of the stack. Our cost model is defined as : \$1 to push a plate onto the stack and \$1 to pop a plate from the stack. Now consider the following problem:

$\mathbf{P_1}$: *What is the worst-case cost of n Push, Pop and Multi-Pop operations?*

A naive analysis would argue that the maximum cost of an operation is n, i.e. the cost of *Multi-Pop(n)*. Therefore, the cost of a sequence of n operations cannot exceed $O(n^2)$. While this analysis is correct, we shall show that the bounds on the worst-case cost can be significantly tightened in a deterministic sense.

A second data structure that we consider is the *k-bit counter*. The counter is initialized to 0 and at each step, it increments itself. In our cost model, we charge \$1 for each bit that is flipped from 0 to 1 or from 1 to 0. We are interested in the following problem:

$\mathbf{P_2}$: *What is the worst-case cost of incrementing the counter n times?*

Once again a naive analysis would argue that if all k bits are set to 1, then incrementing the counter causes all the bits to be reset, resulting in a cost of k; hence a sequence of n bits costs $O(n \cdot k)$ in the worst case.

There are 3 principal techniques of amortized analysis; Section §2.1 is concerned with the Aggregate method, Section §2.2 elaborates on the Accounting method and Section §2.3 discusses the Potential method.

2.1 The Aggregate method

In the aggregate method, we determine the total cost $T(n)$ on a sequence of n operations and then assign the same amortized cost $\frac{T(n)}{n}$ to each operation.

2.1.1 Stack

The principal observation that we make is that an item can be popped from the stack using a *Pop* or a *Multi-Pop* operation only after it has been pushed onto it first using a *Push* operation. Thus, the number of times that *Pop* (including the calls within *Multi-Pop*) can be called is at most the number of times *Push* is called which is at most n. Thus the total cost of a sequence of n operations cannot exceed $O(n)$ and hence the amortized cost of all operations is $\frac{O(n)}{n} = O(1)$.

2.1.2 Counter

For problem $\mathbf{P_2}$, we observe that not all bits are flipped in each increment operation; for instance, the least significant bit B_0 is flipped at each step, the next bit B_1 is flipped every other step, the bit to its left viz. B_2 is flipped every fourth step and so on. Thus bit B_i is flipped once every 2^i increment steps for a total of $\frac{n}{2^i}$ flips. Accordingly, the total cost over n increments is :$\sum_{i=1}^{\log n} \frac{n}{2^i} \leq 2 \cdot n = O(n)$.

2.2 The Accounting method

In the accounting method, we associate a charge with each operation; this cost is called the *amortized cost* for that operation. The amortized cost of an operation may be larger than its actual cost in some cases, and less than the actual cost in other cases. In the case when the amortized cost exceeds the actual cost of an operation, the difference is called the credit and it is assigned to specific objects in the data structure. This credit is used up later to pay for other operations wherein the amortized cost is less than the actual cost of the operation. *This method is different from the aggregate method in which all operations have the same amortized cost.*

In order for the accounting method to provide a true upper bound on the actual costs of operations, the amortized costs must be chosen carefully. Let $\hat{c_i}$ denote the amortized cost of the i^{th} operation and c_i denote its actual cost. Then, we must have

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i \tag{1}$$

over all sequences of n operations!

2.2.1 Stack

To apply the accounting method to problem $\mathbf{P_1}$, we assign the following amortized costs: \$2 for *Push*, \$0 for *Pop* and *Multi-Pop*. Recall that the actual costs are \$1 for a *Push* or *Pop* and \$$k$ for a *Multi-Pop(k)* operation.

We now argue that any sequence of amortized costs can be paid for using the amortized costs. When an item is pushed into the stack for the first time, we charge the operation an amortized cost of \$2; \$1 is used to pay for the operation itself, while the remaining \$1 is used as credit and stored on the item. When this item is popped, the \$1 credit that is stored on it, is used to pay for the *Pop* operation. Thus although the amortized cost is \$0 for the *Pop* operation, the total amortized cost assigned to the *Push* and *Pop* operations is at least as large as the actual costs. The same argument can be used to explain the amortized cost of the *Multi-Pop* operation as well. The key point is that whenever an item is to be popped, it already has a dollar of credit on it, that can be used to pay for the popping. Thus the total cost over a sequence of n Stack operations is at most $2 \cdot n = O(n)$.

2.2.2 Counter

We assign an amortized cost of \$2 to a bit when it is flipped from 0 to 1 and a cost of \$0 when it is flipped from 1 to 0. Since the actual cost of flipping a bit from 0 to 1 is \$1, the amortized cost of \$2 represents an overcharge

of \$1; this credit of \$1 is stored on the bit and used to pay for the cost of flipping it from 1 to 0! At any point in time, every bit that is 1, already has the credit to pay for the flip back to 0 and hence we do not have to pay anything extra for this flip. *Further note that in an increment operation (at any step), precisely one bit is set to* 1*; a number of bits may be set to* 0 *though!* Consequently, the amortized cost for a single step is at most 2 and hence the total amortized cost over n operations is at most $2 \cdot n = O(n)$.

2.3 The Potential method

In the accounting method, we associated credit with individual items; in the potential method credit is associated with the data structure as a whole. We start with an initial data structure D_0, which is transformed through the sequence of n operations into data structure D_n; more specifically operation i transforms D_{i-1} into D_i. A potential function ϕ maps each D_i to a real number $\phi(D_i)$. The amortized cost $\hat{c_i}$ associated with i^{th} operation is defined as:

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) \tag{2}$$

In other words, the amortized cost is defined as the actual cost plus the increase in potential due to the operation. It follows that the total amortized cost over n operations is:

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1})) \tag{3}$$

$$= \sum_{i=1}^{n} c_i + (\phi(D_n) - \phi(D_0) \tag{4}$$

Equation (4) follows from Equation (3) on account of the telescoping property of $\sum_{i=1}^{n} (\phi(D_i) - \phi(D_{i-1}))$!

For the amortized cost to be an upper bound on the actual cost, we must have $\phi(D_n) \geq \phi(D_0)$. Since we do not know the value of n in advance, we need that $\phi(D_i) \geq \phi(D_0)$, $\forall i$. Typically, we define $\phi(D_0) = 0$ and show that $\phi(D_i) \geq 0$, $\forall i$.

The quantity $\phi(D_i) - \phi(D_{i-1})$ is called the potential difference of the i^{th} operation; intuitively, if this difference is positive, then the amortized cost of the i^{th} operation represents an overcharge and the potential of the data structure increases; likewise if the potential difference is negative, the amortized cost is an undercharge to the i^{th} operation and the actual cost is paid by the decrease in the potential of the data structure.

Remark: 2.1 *The actual amortized costs depend upon the nature of the potential function; it is possible for different potential functions to provide different bounds for the same problem. Choosing a tight potential function is a creative task and there do not exist algorithms for the same!*

2.3.1 Stack

Returning to problem $\mathbf{P_1}$, we define the potential of the stack to be the number of items in the stack. Assuming that the stack is initially empty, $\phi(D_0) = 0$; further since the number of items on the stack is never negative, we have $\phi(D_i) \geq 0$, $\forall i$. Thus our choice of potential function satisfies the requirements outlined above, for the amortized cost to be an upper bound on the actual cost.

Let us now calculate the amortized cost of each operation:

1. *Push* - When an item is pushed onto the stack, the potential increases by 1; accordingly, the amortized cost is $\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 1 = 2$;

2. *Pop* - When an item popped from the stack, the potential decreases by 1; accordingly, the amortized cost is $\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 - 1 = 0$;

3. *Multi-Pop(k)* - Since k items are popped, the potential decreases by k; accordingly, the amortized cost is $\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = k - k = 0$.

The amortized cost of each of these operations is $O(1)$ and hence the amortized cost of a sequence of n operations is $O(n)$, the exact result that we obtained with the other two methods!

Exercise: 2.1 *Analyze problem* $\mathbf{P_2}$ *using the Potential Method.*

For more details on Amortized analysis, see [CLR92].

3 Competitive Analysis

In a typical algorithms course, techniques are presented to design and analyze algorithms for problems in which the input is fixed, prior to the commencement of the algorithm; for instance, the MERGE-SORT algorithm sorts an array of n elements in $O(n \log n)$ time; it is understood that *all elements are read into the computer's main memory, before the algorithm starts.* Such a problem and the accompanying algorithm are called *offline*; however, in real-world situations, it is rarely the case that we encounter offline problems. Indeed, a vast majority of problems that we need to confront are inherently online, in that decisions need to be made without complete knowledge of the input data. Consider the examples discussed below:

1. The Post Office Problem - A constant stream of packets arrive at the Post office and these need to be routed to their respective destinations. The Post office has 2 trucks at its disposal to service 5 locations. If a packet to a destination arrives immediately after the trucks have departed, then it will be delayed till such time as the trucks are scheduled for the same destinations. Thus there is a reduction in the *quality of service* defined as the reciprocal of the time taken by a packet to reach its destination. (You may want to think of FedEx and its 24−hour guarantee!) Thus, the question is: How to schedule the packets on trucks, so as to provide some guarantee of quality of service for all packets?

2. Load Balancing - Consider a cluster of m identical processors, which services jobs of varying sizes. Consider a sequence of n jobs $\{J_1, J_2, \ldots, J_n\}$ with processing times $\{p_1, p_2, \ldots, p_n\}$ respectively. The goal is to assign these jobs to the m processors, so that there exists some sort of balance on the load factor of each processor. This problem is called the Load Balancing problem and is known to be `NP-complete`, even when there are at most 2 processors [GJ79]. Observe that load balancing criterion is achieved by *minimizing the maximum completion time of all jobs.*

3. Bin Packing - Assume that a stream of items $a_1, a_2 \ldots, a_n$, $0 < a_i \leq 1$ need to packed into a set of unit capacity bins. The question is: How do we pack the items so as to minimize the number of bins used? This problem is called the *One-dimensional Bin-Packing* problem and is known to be `NP-complete` [GJ79].

4. Internet Server - Consider a News server such as "*cnn.com*"; at any time there will be requests for service from a number of places which are separated in distance. In order to minimize response time, a request needs to be served by the server which is geographically the closest. On the other hand, a sequence of requests from the same place could cause a load imbalance and cause performance degradation. The question then becomes how to service these online requests, without degrading performance.

In none of the examples above, can we wait for *all* of the input data, before formulating an optimal solution or even a solution for that matter; indeed decisions need to be made on the fly, if they are to be practical. The question then becomes: How do we evaluate a given strategy? Competitive analysis has proved to be very uesful in answering this question.

Let us build the formal framework required to analyze online algorithms. A minimization problem $\mathcal{P}$ [1] consists of a set of valid inputs $\mathcal{I}$ and a cost function $\mathcal{C}$. Associated with every input $I \in \mathcal{I}$ is a set of feasible outputs $F(I)$ and associated with each feasible output $O \in F(I)$ is a positive real $C(I, O)$ representing the cost of output O with respect to the input I. Given a legal input I for problem $\mathcal{P}$, an algorithm Alg computes a feasible solution $Alg[I] \in F(I)$. The cost associated with this output is denoted by $Alg(I) = C(I, Alg[I])$. An optimal algorithm OPT is such that for all legal inputs I, we have

$$OPT(I) = \min_{O \in F(I)} C(I, O) \tag{5}$$

[1] For most of this course, we shall focus exclusively on minimization problems. The framework and techniques that we develop can be used with some modification for maximization problems as well.

An algorithm Alg is a $\mathbf{c}-$approximation algorithm for a minimization problem $\mathcal{P}$, if there is a constant $\alpha \geq 0$, such that

$$Alg(I) - c \cdot OPT(I) \leq \alpha, \tag{6}$$

for all legal inputs I [2].

The theory of approximation algorithms is meaningful mostly in the *offline* setting, i.e. the algorithm Alg gets all the input and then proceeds to compute a solution which costs at most c times the optimal solution.

The corresponding analogue in the *online* setting is $\mathbf{c}-$competitiveness.

Definition: 3.1 *An online algorithm Alg is said to be $c-$competitive, if there is a constant α, such that for all finite input sequences I,*

$$Alg(I) = c \cdot OPT(I) + \alpha \tag{7}$$

When the additive constant $\alpha \leq 0$, we say that Alg is strictly $c-$competitive. Alternatively, we can say that a $c-$competitive algorithm is a $c-$approximation algorithm with the added restriction that it must compute online. c can be a function of the problem parameters, but it must be independent of the input I.

We now analyze competitive strategies for the Load Balancing and Bin Packing problems.

3.1 Load Balancing

We use the following online strategy:

LS: *Assign the next job to the processor with the smallest load*

The rule **LS** (for List Scheduling) was proposed for the first time by Graham (see [Hoc96]) and it is a historical fact that it is both the first $2-$approximation algorithm and the first $2-$competitive algorithm for this problem or any other problem for that matter!

We now analyze this algorithm and show that it is $2-$competitive. Consider the input I composed of the sequence of n jobs $\{J_1, J_2, \ldots, J_n\}$, with processing times $\{p_1, p_2, \ldots, p_n\}$ as discussed above. Let $OPT(I)$ denote the time at which the last job finishes execution under the schedule of the optimal offline algorithm OPT and let $LS(I)$ denote the time at which the last job finishes under the schedule of Algorithm **LS**. Note that

1. We do not know and do not really need to know how OPT works,
2. The job that determines the completion time of all jobs need not the job J_n; for instance, in the case where there are 3 jobs and 2 processor, let $p_1 = 10$ and $p_2 = 1, p_3 = 1$, the optimal scheduler will assign J_1 to machine 1 and $\{J_2, J_3\}$ to machine 2, i.e. J_1 is the job that determines the value of $OPT(I)$.

Let $W = \sum_{i=1}^{n} p_i$; note that we must have $OPT(I) \geq \frac{W}{m}$, where m denotes the number of processors, since the best possible case for OPT is when $OPT(I) = \frac{W}{m}$. Let job J_h with processing time p_h be the job that determines the values of $LS(I)$, also assume without loss of generality that it was assigned to machine k. Observe that all machines have load at least $LS(I) - p_k$; if not, there is a machine (say l) that has load lesser than $LS(I) - p_k$ and Algorithm **LS** would have assigned J_h to that machine. So the total load on all machines is at least $m \cdot (LS(I) - p_k) + p_k$. Thus, we have

$$W \geq m \cdot (LS(I) - p_k) + p_k \tag{8}$$

which gives

$$\frac{W}{m} \quad \geq \quad (LS(I) - p_k) + \frac{p_k}{m} \tag{9}$$

[2] Technically speaking, the term approximation algorithm is used only if $\alpha = 0$; otherwise, the algorithm is called an asymptotic $c-$approximation algorithm

$$\Rightarrow LS(I) \leq \frac{W}{m} + p_k(1 - \frac{1}{m}) \quad (10)$$

$$\Rightarrow LS(I) \leq OPT(I) + p_k(1 - \frac{1}{m}) \quad (11)$$

$$(12)$$

Finally, note that $OPT(I) \geq p_k$, to get

$$LS(I) \leq OPT(I) + OPT(I)(1 - \frac{1}{m}) \quad (13)$$

$$\Rightarrow LS(I) \leq (2 - \frac{1}{m})OPT(I) \quad (14)$$

In other words, Algorithm **LS** is 2−competitive!

3.2 Bin-packing

We use the following online strategy:

> **NF**: *Assign the next item to the currently open bin, if it fits; otherwise close the current bin, open a new bin and assign the item to the new bin*

Let $NF(I)$ represent the number of bins used by Algorithm **NF** (for Next-Fit) and let $OPT(I)$ represent the number of bins used by Algorithm OPT, on input I; further let $A = \lceil \sum_{i=1}^{n} a_i \rceil$. Clearly, $OPT(I) \geq A$, since the optimal strategy has to use at least that many bins. The key observation is that the sum of the contents of any two adjacent bins using strategy **NF** is at least 1; otherwise **NF** would not have opened a new bin! Thus we know that every 2 bins used by **NF** reduce A by at least 1, i.e. the $A \geq \frac{1}{2} \cdot NF(I)$. We can thus conclude that $NF(I) \leq 2 \cdot OPT(I)$, i.e. **NF** is 2−competitive.

References

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman Company, San Francisco, 1979.

[Hoc96] Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, 1996.