AMORTIZATION and SELF-ADJUSTMENT

In the recent years a powerful technique in the complexity analysis of data structures called *"amortization"* , or averaging over time, has emerged. Amortized running time is a realistic but robust complexity measure for which we can obtain surprisingly tight upper and lower bounds on a variety of algorithms. By following the principle of designing algorithms whose amortized complexity is low, we obtain *"self-adjusting"* data structures that are simple, flexible and efficient.

Introduction

Webster's dictionary defines "amortize" as "to put money aside at intervals, as in a sinking fund, for gradual payment of (a debt, etc.)." We shall adapt this term to computational complexity, meaning by it "to average over time" or, more precisely, "to average the running times of operations in a sequence over the sequence." The following observation motivates our study of amortization: In many uses of data structures, a sequence of operations, rather than just a single operation, is performed, and we are interested in the total time of the sequence, rather than in the times of the individual operations. A worst-case analysis, in which we sum the worst-case times of the individual operations, may be unduly pessimistic, because it ignores correlated effects of the operations on the data structure. On the other hand, an average-case analysis may be inaccurate, since the probabilistic assumptions needed to carry out the analysis may be false. In such a situation, an amortized analysis, in which we average the running time per operation over a (worst-case) sequence of operations, can yield an answer that is both realistic and robust.

To make the idea of amortization and the motivation behind it more concrete, let us consider the following simple example first:

Example 1: Consider the manipulation of a stack by a sequence of the following type of operations (the first two are considered primitive operations):

$push(x, S)$:	Add item x to the top of stack S.
pop(S):	Remove and return the top item from stack S.
$flush(k, x, S)$:	Remove the top k items of stack S (or until S becomes empty, in case it originally contained fewer than k items) and then push item x on top of the stack. (In other words, perform $\min(k, \lvert S \rvert)$ pops followed by a push. Here $\lvert S \rvert$ stands for the number of items in stack S prior to the operation.)

(There are, of course, other primitive operations to initialize the stack and check the stack for emptyness, but we choose to ignore them in this example.)

Suppose we start with an empty stack and carry out an arbitrary sequence of m such operations (intermixed in arbitrary order). Let us assume that each of the primitive operations *push* and *pop* takes one unit of time. As a measure of running time let us count the number of push and pop operations performed in the sequence. A single *flush* operation

makes $\min(k, |S|)+1$ pop and push operations and this can be as high as m. For instance, this can happen if each of the first $m-1$ operations in the sequence is a push operation and the last operation is $flush(k, x, S)$, where x is some arbitrary item and $k \geq m-1$. We may, somewhat pessimistically, conclude that the total running time of the sequence is $O(m^2)$, since the sequence consists of m operations and the worst-case time of an operation is $O(m)$. However, after a more careful look, we see that altogether the m operations in the sequence can perform at most $2m$ pushes and pops, since there are at most m pushes and each pop must correspond to an earlier push. In other words, the amortized time of each operation (in a worst-case sequence) is 2 units of time. □

This example may seem too simple to be useful, but such stack manipulation indeed occurs in applications as diverse as graph planarity testing and linear-time string matching algorithms. Not only does amortized running time provide a more exact way to measure the running time of known algorithms, but it suggests that there may be new algorithms efficient in an amortized rather than a worst-case sense. As we shall see in this course, such algorithms do exist, and they are simpler, more efficient, and more flexible than their worst-case cousins.

Amortized Analysis

In order to analyze the amortized running time of operations on a data structure, we need a technique for averaging over time. In general, on data structures of low amortized complexity, the running times of successive operations can fluctuate considerably, but only in such a way that the average running time of an operation in a sequence is small. To analyze such a situation, we must be able to bound the fluctuations.

Our view of amortization is analogous to what a *physicist* might have. We define a *potential function* Φ that maps any configuration D of the data structure into a real number $\Phi(D)$ called the *potential* of D. We define the *amortized time* of an operation to be

$$a := t + \Phi(D') - \Phi(D),$$

where t is the actual time of the operation and D and D' are the configurations of the data structure before and after the operation, respectively. With this definition we have the following equality for any sequence of m operations:

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i - \Phi_i + \Phi_{i-1}) = \Phi_0 - \Phi_m + \sum_{i=1}^{m} a_i,$$

where t_i and a_i are the actual and amortized times of the i^{th} operation, respectively, Φ_i is the potential after the i^{th} operation, and Φ_0 is the potential before the first operation. That is, the total time of the operations equals the sum of their amortized times plus the net decrease in potential function from the initial to the final configuration.

We are free to choose the potential function in any way we wish; the more astute the choice, the more informative the amortized times will be. In most cases of interest, the initial potential is zero and the potential is always nonnegative. In such a situation the total amortized time is an upper bound on the total (actual) time. In other words, if $\Phi_0 = 0$ and $\Phi_m \geq 0$, then $\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i$.

Example 2: To fit the stack manipulation of Example 1 into the physicist's framework, we define the potential of a stack to be the number of items it contains. Then a flush operation that performs k pops followed by a push on stack S has an amortized time of $(k+1)+(|S|-k+1)-|S| = 2$. The amortized time of a pop is $1+(|S|-1)-|S| = 0$. The amortized time of a push is $1+(|S|+1)-|S| = 2$. Therefore, the amortized time of each operation is at most 2. The initial potential is zero and the potential is always nonnegative, so m operations take at most $2m$ pushes and pops. □

We shall see other, important examples of amortized analysis in this course.

Self-Adjusting Data Structures

Standard kinds of data structures, such as the many varieties of balanced trees, are specifically designed so that the worst-case time per operation is small. Such efficiency is achieved by imposing an explicit structural constraint that must be maintained during updates, at a cost of both running time and storage space, and tends to produce complicated updating algorithms with many cases.

If we are content with a data structure that is efficient in only an amortized sense, there is another way to obtain efficiency. Instead of imposing any explicit structural constraint, we allow the data structure to be in an arbitrary state, but we design the access and update algorithms to adjust the structure in a simple, uniform way, so that the efficiency of future operations is improved. Such a data structure is called *self-adjusting*.

Self-adjusting data structures have the following possible advantages over explicitly balanced structures:

- They need less space, since no balance information is kept.
- Their access and update algorithms are easy to understand and to implement.

- In an amortized sense, ignoring constant factors, they are as efficient as balanced structures. Indeed, on some sequences of operations they can be more efficient, because they adjust to fit the usage pattern.

Self-adjusting structures have two possible disadvantages:

- More local adjustments take place than in the corresponding balanced structure, especially during accesses. (In a balanced structure, adjustments usually take place only during updates, not during accesses.) This can cause inefficiency if local adjustments are expensive.
- Individual operations within a sequence can be very expensive. Although expensive operations are likely to be rare, this can be a drawback in real-time applications.

In this course we shall see the design and amortized analysis of a number of self-adjusting data structures in the range of *lists*, *priority queues*, *dictionaries*, and *union-find*.

Bibliography

For more details, see chapter 18 of [CLR91], chapter 11 of [Wei95], chapter 6 of [Meh84a], and the paper [Tar85].

[Meh84a] K. Mehlhorn, "*Data Structures and Algorithms 1: Sorting and Searching*," Springer-Verlag, 1984.

[Tar85] R.E. Tarjan, *"Amortized computational complexity,"* SIAM Journal on Algebraic and Discrete Methods, 6(2), 1985, pp. 306-318.