Potential method

In computational complexity theory, the **potential method** is a method used to analyze the amortized time and space complexity of a data structure, a measure of its performance over sequences of operations that smooths out the cost of infrequent but expensive operations.[1][2]

Contents

Definition of amortized time

In the potential method, a function Φ is chosen that maps states of the data structure to non-negative numbers. If S is a state of the data structure, $\Phi(S)$ may be thought of intuitively as an amount of potential energy stored in that state;[1][2] alternatively, $\Phi(S)$ may be thought of as representing the amount of disorder in state S or its distance from an ideal state. It represents work that has been accounted for ("paid for") in the amortized analysis, but not yet performed. The potential value prior to the operation of initializing a data structure is defined to be zero.

Let o be any individual operation within a sequence of operations on some data structure, with S_{before} denoting the state of the data structure prior to operation o and S_{after} denoting its state after operation o has completed. Then, once Φ has been chosen, the amortized time for operation o is defined to be

$$T_{\text{amortized}}(o) = T_{\text{actual}}(o) + C \cdot (\Phi(S_{\text{after}}) - \Phi(S_{\text{before}})),$$

where C is a non-negative constant of proportionality (in units of time) that must remain fixed throughout the analysis. That is, the amortized time is defined to be the actual time taken by the operation plus C times the difference in potential caused by the operation.[1][2]

When studying asymptotic computational complexity using big O notation, constant factors are irrelevant and so the constant C is usually omitted.

Relation between amortized and actual time

Despite its artificial appearance, the total amortized time of a sequence of operations provides a valid upper bound on the actual time for the same sequence of operations.

For any sequence of operations $O = o_1, o_2, \ldots, o_n$, define:

- The total amortized time: $T_{\text{amortized}}(O) = \sum_{i=0}^{n} T_{\text{amortized}}(o_i)$,
- The total actual time: $T_{\text{actual}}(O) = \sum_{i=0}^{n} T_{\text{actual}}(o_i)$.

Then:

$$T_{\text{amortized}}(O) = \sum_{i=1}^{n} (T_{\text{actual}}(o_i) + C \cdot (\Phi(S_i) - \Phi(S_{i-1}))) = T_{\text{actual}}(O) + C \cdot (\Phi(S_n) - \Phi(S_0)),$$

where the sequence of potential function values forms a telescoping series in which all terms other than the initial and final potential function values cancel in pairs. Rearranging this, we obtain:

$$T_{\text{actual}}(O) = T_{\text{amortized}}(O) - C \cdot (\Phi(S_n) - \Phi(S_0)).$$

Since $\Phi(S_0) = 0$ and $\Phi(S_n) \geq 0$, $T_{\text{actual}}(O) \leq T_{\text{amortized}}(O)$, so the amortized time can be used to provide an accurate upper bound on the actual time of a sequence of operations, even though the amortized time for an individual operation may vary widely from its actual time.

Amortized analysis of worst-case inputs

Typically, amortized analysis is used in combination with a worst case assumption about the input sequence. With this assumption, if X is a type of operation that may be performed by the data structure, and n is an integer defining the size of the given data structure (for instance, the number of items that it contains), then the amortized time for operations of type X is defined to be the maximum, among all possible sequences of operations on data structures of size n and all operations o_i of type X within the sequence, of the amortized time for operation o_i.

With this definition, the time to perform a sequence of operations may be estimated by multiplying the amortized time for each type of operation in the sequence by the number of operations of that type.

Examples

Dynamic array

A dynamic array is a data structure for maintaining an array of items, allowing both random access to positions within the array and the ability to increase the array size by one. It is available in Java as the "ArrayList" type and in Python as the "list" type.

A dynamic array may be implemented by a data structure consisting of an array A of items, of some length N, together with a number $n \leq N$ representing the positions within the array that have been used so far. With this structure, random accesses to the dynamic array may be implemented by accessing the same cell of the internal array A, and when $n < N$ an operation that increases the dynamic array size may be implemented simply by incrementing n. However, when $n = N$, it is necessary to resize A, and a common strategy for doing so is to double its size, replacing A by a new array of length $2n$.[3]

This structure may be analyzed using the potential function:

$$\Phi = 2n - N$$

Since the resizing strategy always causes A to be at least half-full, this potential function is always non-negative, as desired.

When an increase-size operation does not lead to a resize operation, Φ increases by 2, a constant. Therefore, the constant actual time of the operation and the constant increase in potential combine to give a constant amortized time for an operation of this type.

However, when an increase-size operation causes a resize, the potential value of n decreases to zero after the resize. Allocating a new internal array A and copying all of the values from the old internal array to the new one takes O(n) actual time, but (with an appropriate choice of the constant of proportionality C) this is entirely cancelled by the decrease in the potential function, leaving again a constant total amortized time for the operation.

The other operations of the data structure (reading and writing array cells without changing the array size) do not cause the potential function to change and have the same constant amortized time as their actual time.[2]

Therefore, with this choice of resizing strategy and potential function, the potential method shows that all dynamic array operations take constant amortized time. Combining this with the inequality relating amortized time and actual time over sequences of operations, this shows that any sequence of n dynamic array operations takes O(n) actual time in the worst case, despite the fact that some of the individual operations may themselves take a linear amount of time.[2]

When the dynamic array includes operations that decrease the array size as well as increasing it, the potential function must be modified to prevent it from becoming negative. One way to do this is to replace the formula above for Φ by its absolute value.

Multi-Pop Stack

Consider a stack which supports the following operations:

- Initialize - create an empty stack.
- Push - add a single element on top of the stack, enlarging the stack by 1.
- Pop(k) - remove k elements from the top of the stack, where k is no more than the current stack size

Pop(k) requires O(k) time, but we wish to show that all operations take O(1) amortized time.

This structure may be analyzed using the potential function:

$$\Phi = \text{number-of-elements-in-stack}$$

This number is always non-negative, as required.

A Push operation takes constant time and increases Φ by 1, so its amortized time is constant.

A Pop operation takes time O(k) but also reduces Φ by k, so its amortized time is also constant.

This proves that any sequence of m operations takes O(m) actual time in the worst case.

Binary counter

Consider a counter represented as a binary number and supporting the following operations:

- Initialize: create a counter with value 0.
- Inc: add 1 to the counter.
- Read: return the current counter value.

For this example, we are *not* using the transdichotomous machine model, but instead require one unit of time per bit operation in the increment. We wish to show that Inc takes O(1) amortized time.

This structure may be analyzed using the potential function:

$$\Phi = \text{number-of-bits-equal-to-1} = \text{hammingweight}(\text{counter})$$

This number is always non-negative and starts with 0, as required.

An Inc operation flips the least significant bit. Then, if the LSB were flipped from 1 to 0, then the next bit is also flipped. This goes on until finally a bit is flipped from 0 to 1, at which point the flipping stops. If the counter initially ends in k 1 bits, we flip a total of k+1 bits, taking actual time k+1 and reducing the potential by k−1, so the amortized time is 2. Hence, the actual time for running m Inc operations is O(m).

Applications

The potential function method is commonly used to analyze Fibonacci heaps, a form of priority queue in which removing an item takes logarithmic amortized time, and all other operations take constant amortized time.[4] It may also be used to analyze splay trees, a self-adjusting form of binary search tree with logarithmic amortized time per operation.[5]

References

1. Goodrich, Michael T.; Tamassia, Roberto (2002), "1.5.1 Amortization Techniques", *Algorithm Design: Foundations, Analysis and Internet Examples*, Wiley, pp. 36–38.
2. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. "17.3 The potential method". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. 412–416. ISBN 0-262-03293-7.
3. Goodrich and Tamassia, 1.5.2 Analyzing an Extendable Array Implementation, pp. 139–141; Cormen et al., 17.4 Dynamic tables, pp. 416–424.
4. Cormen et al., Chapter 20, "Fibonacci Heaps", pp. 476–497.
5. Goodrich and Tamassia, Section 3.4, "Splay Trees", pp. 185–194.