Algorithm Design and Analysis (H) Amortized Analysis

SHAN CHEN

(SLIDES EDITED FROM THOSE CREATED BY YAO ZHAO)

Amortized Analysis（摊还分析）

In computer science, amortized analysis is a method for analyzing a given algorithm's complexity, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time per operation, rather than per algorithm, can be too pessimistic.

While certain operations for a given algorithm may have a significant cost in resources, other operations may not be as costly. Amortized analysis considers both the costly and less costly operations together over the whole series of operations of the algorithm. This may include accounting for different types of input, length of the input, and other factors that affect its performance.

https://en.wikipedia.org/wiki/Amortized_analysis

Methods

- Aggregate method (聚合分析)
- Accounting method (核算法)
- Potential method (势能法)

All above are correct. The choice of which to use depends on which is most convenient for a particular situation.

Example： ArrayList (JAVA)

每超过一定阈值，就把数据移到更大的内存块

DEFAULT_CAPACITY = 10: first 10 elements →O(1)

加上原始内存的一半（右移）

NEW_CAPACITY = OLD_CAPACITY + OLD_CAPACITY >> 1

The 11th element →O(n)
12th ~ 15th element →O(1)

只有临界点才有O（n）

The 16th element →O(n)　17th ~ 22th element →O(1)

Obviously, worst case operation cost O(n).
But we shouldn't consider that insert n elements will cost $O(n^2)$.

Aggregate Method（聚合法）

- The aggregate method determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n) / n$.
  *(*Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.*)*

Aggregate Method: ArrayList

- $T(n) = \sum_{i=1}^{n} c_i$

$$= \sum_{i=1, i-1 \neq 10 \times 1.5^k}^{n} c_i + \sum_{i=1, i-1 = 10 \times 1.5^k}^{n} c_i$$

$$\leq \sum_{i=1, i-1 \neq 10 \times 1.5^k}^{n} 1 + \sum_{i=1, i-1 = 10 \times 1.5^k}^{n} (10 \times 1.5^k + 1)$$

$$= n + \sum_{k=0}^{\log_{1.5}\frac{n}{10}} 10 \times 1.5^k = n + 10 * \left(\frac{1 - 1.5^{\log_{1.5}\frac{n}{10}+1}}{1-1.5} \right) = n + 10 * \left(\frac{1 - 1.5 * \frac{n}{10}}{-0.5} \right)$$

$$= n + 3n - 20 = 4n - 20$$

- $\frac{T(n)}{n} = O(1)$
- Pushing an element onto the dynamic array takes constant time when the worst case operation's cost amortize to other operations.

Accounting Method（核算法）

- The accounting method is a form of aggregate analysis which assigns to each operation an amortized cost which may differ from its actual cost. Early operations have an amortized cost higher than their actual cost, which accumulates a saved "credit" that pays for later operations having an amortized cost lower than their actual cost. Because the credit begins at zero, the actual cost of a sequence of operations equals the amortized cost minus the accumulated credit. Because the credit is required to be non-negative, the amortized cost is an upper bound on the actual cost. Usually, many short-running operations accumulate such a credit in small increments, while rare long-running operations decrease it drastically. (Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.)

Accounting Method: ArrayList (JAVA)

Amortized cost for each element: 4

10 Credit upon 1st expansion = 40 – 10 + 4 ≥ 10 + 1

NEW_CAPACITY = OLD_CAPACITY + OLD_CAPACITY >> 1

15

早期的操作可能会有一个高于实际成本的摊销成本，用以为后续的操作储存一个“信用”，以支付那些摊销成本较低的操作

Credit upon 2nd expansion ≥ 3 * 5 + 4 ≥ 15 + 1

22

Credit upon 3rd expansion ≥ 3 * 7 + 4 > 22 + 1

The saved credit is always sufficient to accommodate the next O(n) cost.
So, the amortized running time for pushing each new element is ≤ 4.

Potential Method（勢能法）

- The potential method is a form of the accounting method where the saved credit is computed as a function (the "potential") of the state of the data structure. The amortized cost is the immediate cost plus the change in potential.
  (Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.)

Function Φ

Φ : maps states of the data structure to non-negative numbers

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

where C is a non-negative constant of proportionality (in units of time) that must remain fixed throughout the analysis. constant C is usually omitted when using big O notation.

Define Φ such that $T_{\text{amortized}}(o)$ is the same (e.g., constant) for each operation o.

Define Φ

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:

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

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

Note that $\Phi(S_0) = 0$ and $\Phi(S_n) \geq 0$, $T_{\text{actual}}(O) \leq T_{\text{amortized}}(O)$

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.

Potential Method: ArrayList (JAVA)

- Define $\Phi(S_i) = 3(n_i - 1) - 2N_i$, where n_i is the number of elements and N_i is the capacity.
  - *S_0 is the state of $n_0 = 11$, $N_0 = 15$.* Then, we have $\Phi(S_0) = 3(11 - 1) - 2 * 15 = 0$. *(ArrayList has capacity 10 when there are ≤ 10 elements. So, we do our analysis with $n_0 = 11$, $N_0 = 15$* to ensure $\Phi \geq 0$.)
  - When $n_i = N_{i-1} + 1$ *(right upon enlarging the capacity)*, we have $\Phi(S_i) = 3N_{i-1} - 2N_i \geq 0$.
- When we insert an element n_i, but not enlarge the table (i.e., $N_i = N_{i-1}$):

$$T_{amortized}(o_i) = T_{actural}(o_i) + \phi(S_i) - \phi(S_{i-1})$$
$$= 1 + (3 * n_i - 2N_i) - (3 * n_{i-1} - 2N_{i-1}) = 1 + 3 * (n_i - n_{i-1}) = 4$$

- When we insert an element i, and enlarge the table (i.e., $N_i = N_{i-1} + N_{i-1} >> 1 \approx N_{i-1} * 3/2$):

$$T_{amortized}(o_i) = T_{actural}(o_i) + \phi(S_i) - \phi(S_{i-1})$$
$$\approx n_i + \left(3 * n_i - 2 * N_{i-1} * \frac{3}{2}\right) - (3 * n_{i-1} - 2N_{i-1}) = i - N_{i-1} + 3 = 4$$

- Together, $\sum_{i=1}^{n} T_{amortized}(o_i) \approx 4n$.

This shows that any sequence of *n* dynamic array operations takes *O(n)* time.

Reference

- Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.