Analysis of Algorithms - Assignment 3

Zhenrui Zheng

Chinese University of Hong Kong, Shenzhen
225040512@link.cuhk.edu.cn

Contents

Amortized Analysis

1.1 Queue Implementation with Two Stacks

Implementation

We implement a queue using two stacks by maintaining:

- **Input Stack** (S_{in}): For enqueue operations
- **Output Stack** (S_{out}): For dequeue operations

Algorithm 1 Queue Operations with Two Stacks

Require: Two stacks S_{in} and S_{out}

```
1: function Push(x)
2: S_in.push(x)
3: end function
4: function Get()
5: if S_out is empty then
6:    while S_in is not empty do
7:       S_out.push(S_in.pop())
8:    end while
9: end if
10: return S_out.pop()
11: end function
```

Proof of O(1) Amortized Time

We use the *accounting method* to prove the amortized cost analysis.

Assigning Costs: We assign an amortized cost of 4 for each Push operation:

- 1 unit pays for the actual Push into S_{in}
- 3 unit is stored as credit on the element for its future operations

For Get operations, we assign an amortized cost of 0:

- When S_{out} is non-empty: the actual cost is 1 (one Pop), and we use 1 credit from the poped element (in S_{out})
- When S_{out} is empty: we transfer all elements from S_{in} to S_{out}
  - Each transfer requires one Pop from S_{in} (cost 1) and one Push to S_{out} (cost 1), paid by the credit on the element
  - After transfer, each element in S_{out} has 1 credit left, which is enough to cover the cost of the future Pops. Then, we can simply Pop 1 element from S_{out} with cost 1, covered by the credit on the element
  - Total cost: $2k + 1$ for transferring k elements and one final Pop, covered by $2k + 1$ credits

Invariant: At any point, the total credit stored equals $3\,|S_{in}| + |S_{out}|$, since each element in S_{in} has 3 units of credit from its Push operation, and each element in S_{out} has 1 credit left after transferring from S_{in}, which cost 2 credits.

Conclusion: Each Push has amortized cost $O(1)$, each Get has amortized cost $O(1)$, and the total amortized cost over any sequence of n operations is $O(n)$, giving $O(1)$ amortized cost per operation.

1.2 Amortized Computation Cost Analysis

Total Cost Calculation:

Let $k = \lfloor \log_2(n) \rfloor$. The total cost is:

$$\begin{aligned} T(n) &= \sum_{j=0}^{k} 2^j + (n - (k+1)) \cdot 1 \\ &= \sum_{j=0}^{k} 2^j + n - k - 1 \end{aligned}$$

We know that $\sum_{j=0}^{k} 2^j = 2^{k+1} - 1 = 2 \cdot 2^k - 1$.

Since $2^k \leq n < 2^{k+1}$, we have:

$$\begin{aligned} 2^k &\leq n \\ 2^{k+1} &\leq 2n \end{aligned}$$

Therefore:

$$\begin{aligned} T(n) &= 2^{k+1} - 1 + n - k - 1 \\ &\leq 2n - 1 + n - k - 1 \\ &= 3n - k - 2 \\ &\leq 3n \\ \frac{T(n)}{n} &\leq \frac{3n}{n} = 3 = O(1) \end{aligned}$$

Thus, the amortized computation cost per day is $O(1)$.

Element Selection Algorithm

2.1 Randomized Algorithm

Algorithm Description

Algorithm 2 Simplified Randomized Search

Require: Arrays $X[1..n]$, $next[1..n]$, $c > 0$ and value x
Ensure: True if $x \in X$, False otherwise

```
Let m = c⌈√n⌉
Randomly sample m distinct indices uniformly from [1..n]
Among the sampled indices, find i* that minimizes |X[i] − x|
Follow next pointers from i* for at most m steps
for k = 1 to m do
    if X[i*] = x then
        return True
    end if
    i* = next[i*]
end for
return False
```

This requires $O(\sqrt{n})$ time complexity as expected.

2.2 Success Probability Analysis

Let X' be the sorted version of X, and x be the target element. Let I be the set of indices within distance $m = c\sqrt{n}$ from x in X', that is, $I = \{i \in [1..n] \mid \mathit{lowerbound}(x) - i \leq m \pmod n\}$. The probability that, if we take m samples uniformly from $[1..n]$, at least one of them is in I is a hypergeometric distribution, which is given by:

$$\begin{aligned}
&P(\text{at least one sample among } m \text{ in } I) \\
&= 1 - P(\text{all } m \text{ samples in } \overline{I}) \\
&= 1 - \frac{\binom{m}{0}\binom{n-m}{m}}{\binom{n}{m}} \\
&= 1 - \frac{(n-m)!(n-m)!}{n!(n-2m)!} \quad (m! \text{ cancels out})
\end{aligned}$$

We want this to be at least 0.99:

$$\begin{aligned}
1 - \frac{(n-m)!(n-m)!}{n!(n-2m)!} &\geq 0.99 \\
\frac{(n-m)!(n-m)!}{n!(n-2m)!} &\leq 0.01 \\
m &\leq 2.15\sqrt{n}
\end{aligned}$$

So $m = \lceil 2.15\sqrt{n} \rceil$ suffices for 99% success probability. Similarly, minimum m for 99.9% and 99.99% success probabilities are $\lceil 2.63\sqrt{n} \rceil$ and $\lceil 3.04\sqrt{n} \rceil$, respectively.

Shelf Scheduling Algorithm

3.1 NP-Hardness Proof

We prove that the parallel scheduling problem is NP-hard by reduction from the Subset-sum problem under a special case, where $p = 2$.

Reduction: Given an instance of subset-sum with a set of positive integers $\{a_1, a_2, \ldots, a_n\}$ and target sum S, we construct a 2-processor parallel scheduling instance as follows:

- occupy time: $T[1, ..., n] = \{a_1, a_2, ..., a_n\}$
- Processor: $P[1, ..., n], P[i] = 1$

The question becomes: Can we schedule these n jobs on 2 processors such that all jobs finish at time $t = S$? This is equivalent to asking whether we can partition the n elements into two subsets, with one of them sum up to S.

The reduction is given within polynomial time, and since Subset-sum is NP-complete, the parallel scheduling problem is NP-hard.

3.2 3-Approximation Algorithm

We present a scheduling algorithm that achieves a 3-approximation guarantee. Since the performance bound of the algorithm given in the hint is difficult to prove, I designed another very similar algorithm that can easily prove its 3-approximation property.

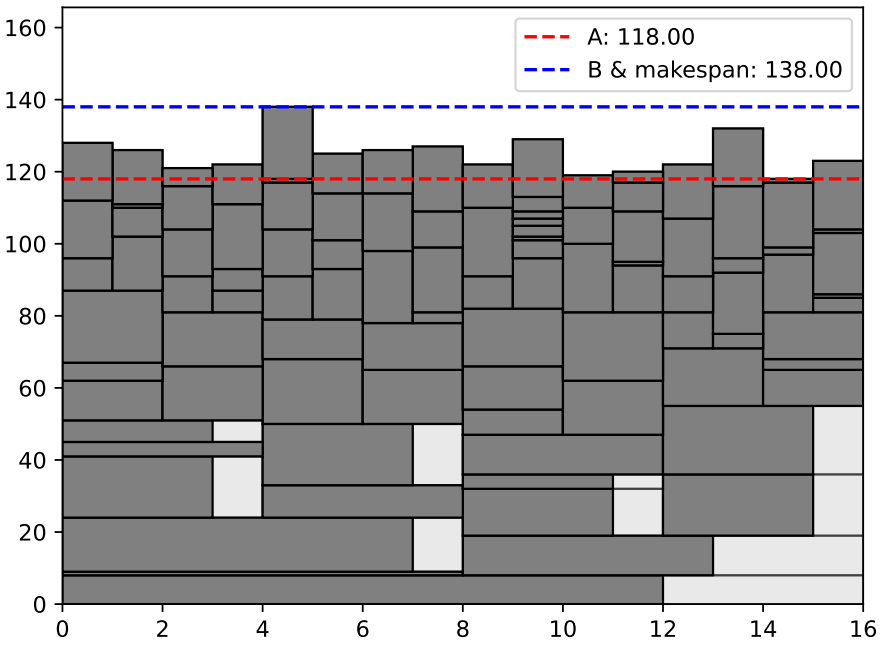

Figure 3.1: An example of the scheduling algorithm

Algorithm Description:

Since $p = 2^k$ for some integer k, there are only $\log_2 p$ distinct scales of processor requirements. We partition jobs into categories based on their processor requirements:

- Category i: jobs with $P[j] \in (2^{i-1}, 2^i]$ for $i = 0, \ldots, k$

The algorithm proceeds as follows:

1. categorize all jobs by processor requirements $P[j]$ into k categories
2. For each category i from k to 0:
    - Round up the processor requirements of all jobs in this category to 2^i
    - For each job in this category, find the processor with the earliest end time among all current processors, for example, processor j, and assign $p[j] \sim p[j + 2^i - 1]$ to this job. Such assignment would push the end time of these 2^i processors to the future.

Time Complexity:

Since there are in total n jobs, and each requires searching the earliest end time among p processors, which can be done in $O(\log p)$ time, the total time complexity is $O(n \log p)$.

3.3 Lower Bounds for Optimal Makespan

Let M^* be the optimal makespan. We establish two lower bounds:

Bound 1: $M^* \geq \max_{i=1}^{n} T[i]$
This is trivial: any job i must run for at least $T[i]$ time units. Therefore, the makespan cannot be smaller than the longest job duration.

Bound 2: $M^* \geq \frac{\sum_{i=1}^{n} T[i] \cdot P[i]}{p}$
This inequality represents the relationship between total computation capacity and job requirements, regardless of scheduling. The total computation capacity can be considered a large rectangle box, which is $p \times M^*$, need to fit in all jobs, each can be considered a small rectangle box, which is $P[i] \times T[i]$.

Thus, it is obvious that the volume of the large box is at least the sum of the volumes of all small boxes, that is, $p \times M^* \geq \sum_{i=1}^{n} T[i] \cdot P[i]$, which gives $M^* \geq \frac{\sum_{i=1}^{n} T[i] \cdot P[i]}{p}$.

3.4 3-Approximation Proof

We decompose the schedule produced by our algorithm into two parts:

$$\text{ALG} = A + B$$

where:

- A: total height of all non-last shelves across all categories
- B: total height of the last shelf in each category

$B \leq$ OPT

Consider the last shelf in each category, which contains the largest duration job in that category,

Randomized Algorithm