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 $S_{out}$
function Push($x$)
$S_{in}$.push($x$)
end function
function Get()
if $S_{out}$ is empty then
    while $S_{in}$ is not empty do
        $S_{out}$.push($S_{in}$.pop())
    end while
end if
return $S_{out}$.pop()
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 $2^k \le n < 2^{k+1}$, we have:

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

Therefore:

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

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
 1: Let $m = c\lceil\sqrt{n}\rceil$
 2: Randomly sample $m$ distinct indices uniformly from $[1..n]$
 3: Among the sampled indices, find $i^*$ that minimizes $|X[i] - x|$
 4: Follow $next$ pointers from $i^*$ for at most $m$ steps
 5: for $k = 1$ to $m$ do
 6:    if $X[i^*] = x$ then
 7:       return True
 8:    end if
 9:    $i^* = next[i^*]$
10: end for
11: 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** appears to be difficult (to me), I designed another very similar algorithm that can easily prove its 3-approximation property.

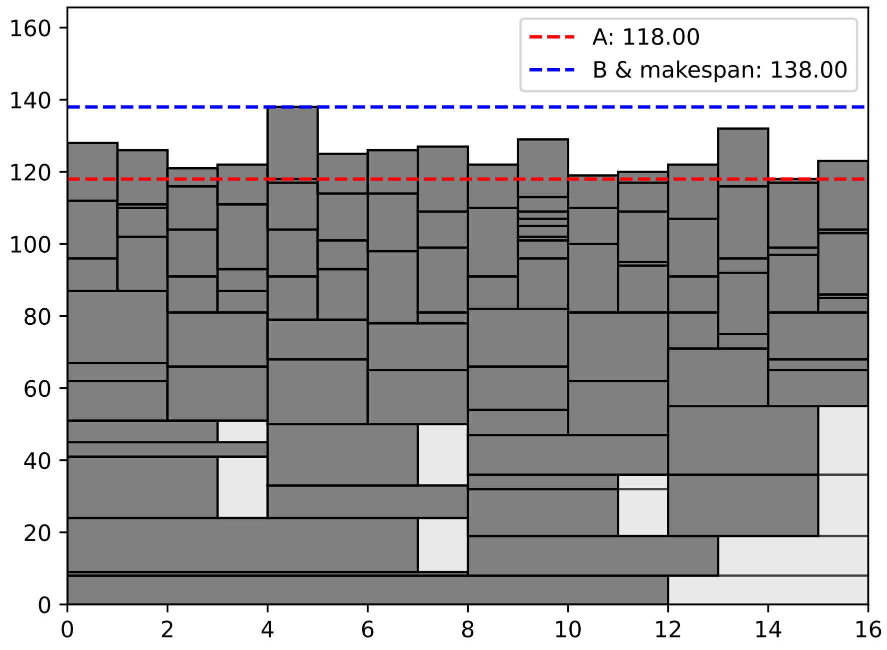

Figure 3.1: An example of the scheduling algorithm, where we have $p = 2^k = 16$ processors, and $n = 100$ jobs. The grey area are processor requirements of each job, while white area are additionally assigned (by rounding up) processors for the job. The vloume below the red line can be considered "A", the volume between the red and blue line considered "B".

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 (different from the original question):

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

where:

- A: earliest end time among all processors
- B: the time between the earliest end time and the latest end time (i.e., the makespan)

A visualization is shown in Figure 3.2. “A” can be considered the height below the red line, while “B” can be considered the height between the red and blue line. Next we will prove that $A \leq 2 \cdot \text{OPT}$ and $B \leq \text{OPT}$.

Correctness of Algorithm

The only concern about whether this algorithm can generate a valid schedule is whether, when assigning processors to each job, all of these $j \sim j + 2^i - 1$ processors are actually available.

The justification is that, say that we are assigning processors for job i, since we process all jobs in reverse order of their category (processor requirements), the jobs assigned before job i must have equal or smaller processor requirements. We can follow a simple rule when assigning processors to each job, that is, only choose those processor with index $j \bmod 2^i$. It can be easily proved that, such rule can guarantee that the end time of these $j \sim j + 2^i - 1$ processors are identical.

Proof of $A \leq 2 \cdot$ OPT

According to the definition of A, it is the time of earliest end time among all processors. That is, all p processors are fully occupied before time A. Let I_A be the set of jobs that start before time A, P' be the rounded up processor requirements, T' be the truncated job durations (at time A), then we have:

$$\begin{aligned}
A \times p &= \sum_{i \in I_A} P'[i] \times T'[i] \\
&\leq \sum_{i \leq n} P'[i]T[i] \\
&\leq \sum_{i \leq n} 2P[i]T[i] \\
A &\leq 2\frac{\sum_{i \leq n} P[i]T[i]}{p} \leq 2 \cdot M^* = 2 \cdot \mathrm{OPT}
\end{aligned}$$

Proof of $B \leq$ OPT

We can prove the bound in 2 steps:

Proposition 1. *Every job that appears in B must have started at time A or earlier.*

Proof. We can use proof by contradiction. Suppose there is a job in B that starts after time A. When assigning a processor to this job, we could have assigned it to one of the processors that contributed to bound A. This would violate the previous rule of the algorithm which states that jobs are always assigned to the processor with the earliest finishing time. Therefore, the proposition holds. □

Proposition 2. *The height of B is at most the duration of the longest job* $\max_{i=1}^{n} T[i]$

Proof. The proof is straightforward: consider the job that contributed to bound B, let it be job j, it must have started at time A or earlier, according to the previous proposition. Therefore, the duration of this job is at least the height of B, that is,

$$B \leq T[j] \leq \max_{i=1}^{n} T[i] \leq M^* = \mathrm{OPT}$$

□

Randomized Algorithm

We consider the problem where n servers and $m = 2n \log n$ tasks. Each task is independently and uniformly randomly assigned to one of the n servers.

4.1 Probability Bound for First Server

Let X be the number of tasks assigned to the first server. Since each task is independently assigned to the first server with probability $1/n$, X follows a binomial distribution:

$$X \sim \text{Bin}(m, 1/n)$$

where $m = 2n \log n$.

The expected value is:

$$\mathbb{E}[X] = \frac{m}{n} = \frac{2n \log n}{n} = 2 \log n$$

We want to show that:

$$\mathbb{P}(X \geq 2e \cdot \log n) \leq \frac{1}{n^2}$$

Using the Chernoff bound for the upper tail of a binomial distribution, we have:

$$\mathbb{P}(X \geq (1+\delta)\mu) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}} \right)^\mu$$

where $\mu = \mathbb{E}[X] = 2 \log n$.

We want $(1+\delta)\mu = 2e \cdot \log n$, which gives:

$$1 + \delta = e \quad \Rightarrow \quad \delta = e - 1$$

Substituting into the Chernoff bound:

$$\begin{aligned}
\mathbb{P}(X \geq 2e \cdot \log n) &\leq \left(\frac{e^{e-1}}{e^e} \right)^{2 \log n} \\
&= \left(\frac{e^{e-1}}{e^e} \right)^{2 \log n} \\
&= \left(\frac{1}{e} \right)^{2 \log n} \\
&= e^{-2 \log n} \\
&= \frac{1}{e^{2 \log n}} \\
&= \frac{1}{n^2}
\end{aligned}$$

Therefore, $\mathbb{P}(X \geq 2e \cdot \log n) \leq 1/n^2$.

4.2 High Probability Bound for All Servers

We now show that with high probability, no server receives at least $2e \log n$ tasks.

Let A_i be the event that server i receives at least $2e \log n$ tasks. By the union bound:

$$\begin{aligned}
\mathbb{P}(\text{at least one server receives } \geq 2e\log n \text{ tasks}) &= \mathbb{P}\left(\bigcup_{i=1}^{n} A_i\right) \\
&\leq \sum_{i=1}^{n} \mathbb{P}(A_i) \\
&= n \cdot \mathbb{P}(A_1) \quad \text{(by symmetry)} \\
&\leq n \cdot \frac{1}{n^2} \quad \text{(from part 1)} \\
&= \frac{1}{n}
\end{aligned}$$

Therefore:

$$\mathbb{P}(\text{no server receives } \geq 2e\log n \text{ tasks}) \geq 1 - \frac{1}{n}$$

As $n \to \infty$, this probability approaches 1, which means with high probability, no server receives at least $2e \log n$ tasks.