DDA6050 - Assignment 3

Yohandi

November 17, 2023

1 Amortized Analysis.

(a). Assuming we possess two stacks capable of performing push and pop operations at a constant $O(1)$ cost per operation, construct a queue with enqueue and dequeue operations in such a way that the amortized time for each queue operation remains within $O(1)$. Also prove that the implemented queue indeed has the $O(1)$ amortized time per operation. (10 marks)

(a) For convenience purposes, assume that the stack supports an operation `move`(S, T) that `pop` element from S and `push` it into T until S is empty. The following shows how to implement a queue using two stacks, S_1 and S_2.

- `Enqueue`(x): perform `push`(S_1, x)
- `Dequeue`$()$:
  - If S_2 is empty, perform `move`(S_1, S_2) and perform `pop`(S_2).
  - If S_2 is not empty, perform `pop`(S_2).

We will use potential method analysis to prove that the implemented queue indeed has the $\mathcal{O}(1)$ amortized time per operation.

Let ϕ represent the potential function and $\phi(D_i)$ defined as twice the number of elements in S_1 added by the number of elements in S_2, i.e., $\phi(D_i) = 3|S_1| + |S_2|$. For validity check, $\phi(D_0) = 0$ as both stacks are initially empty and $\phi(D_i) \geq 0$ as the number of elements on both stacks is always ≥ 0.

For each operation, we compute the amortized cost. Denote c_i as the actual cost of i-th operation and $\hat{c}_i$ as the amortized cost of the i-th operation. The computation is as follows:

- `Enqueue`(x): $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (3|S_1| + 3 + |S_2|) - (3|S_1| + |S_2|) = 4$
- `Dequeue`$()$ when S_2 is empty: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = (2|S_1| + 1) + (0 + |S_1| - 1) - (3|S_1| + 0) = 0$
- `Dequeue`$()$ when S_2 is not empty: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (3|S_1| + |S_2| - 1) - (3|S_1| + |S_2|) = 0$

All operations have $\mathcal{O}(1)$ amortized cost.

(b). Suppose that we run a program over a sequence of n days. On the i-th day, if $\log_2(i)$ is an integer, then this program costs i units of computation resources to examine the outputs obtained so far. Otherwise, it only costs 1 unit of computation resource this day. Compute the amortized computation cost per day. (10 marks)

(b) We will use aggregate method analysis to compute the amortized computation cost per day.

On days where $\log_2(i)$ is an integer, i is a power of two. Assuming n is a power of two, the total computation resources made is $1 + 2 + 4 + \ldots + n = 2n - 1$.

There are $n - (\log_2(i) + 1)$ days that only cost 1 computation resources.

To summarize up, we have $T(n) = (2n-1)+(n-(\log_2(n)+1)) = 3n-\log_2(n)-2$. Then, by aggregate method, the average computation cost per day is $\frac{T(n)}{n} = \frac{3n-\log_2(n)-2}{n}$. As n goes bigger, both $-\log_2(n)$ and -2 terms no longer affect the cost. Then, the computation cost per day is $3 = \mathcal{O}(1)$.

When n is not a power of two, we can consider n' where $n' \leq 2n$ and n' is a power of two. Then, we will have $T(n') = (3n' - \log_2(n') - 2)$. As $T(n)$ is increasing, this implies that $\frac{T(n)}{n} \leq \frac{T(n')}{n} \leq \frac{6n-\log_2(2n)-2}{n} \leq 6 = \mathcal{O}(1)$.

2 Governing Set Problem.

In this question, we only consider the connected graph. A **governing set** in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that every vertex $v \in V \backslash S$ is adjacent to a vertex $s \in S$. Now, the governing set problem is that, given a graph $G = (V, E)$ and an integer k, we want to determine the existence of a **governing set** $S \subseteq V$ such that $|S| \leq k$. Show that the vertex-cover problem $\leq_p$ governing set problem. (Hint: You may consider the following algorithm. Try to prove the correctness of the following algorithm. This algorithm claims a necessary and sufficient condition, so you must demonstrate both aspects.)

Algorithm 1 Vertex cover problem to governing set reduction

Input: graph $G = (V, E)$ and and integer k.

Output: Graph $H = (V', E')$ such that G has a vertex cover of size k if and only if H has a governing set of size k.

```
Initialize V' ← V, E' ← E.
for every edge {u, v} ∈ E do
    Add vertex w_uv to V'.
    Add edges {u, w_uv}, {v, w_uv} to E'.
end for
return H = (V', E')
```

To show that the vertex-cover problem $\leq_p$ governing set problem, we perform a reduction from vertex-cover to governing set.

1. Create a new graph $G' = (V', E')$ from G. Initially, V' is assigned with V and E' is assigned with E.
2. For edge $\{u, v\}$ in E, we add a new vertex w_{uv} to V'. The purpose of this is to ensure that when either u or v is in the vertex cover in G, w_{uv} will be *governed* in G'.
3. Add both $\{u, w_{uv}\}$ and $\{v, w_{uv}\}$ edges to E'. To cover the edge $\{u, v\}$ in G, either u or v must be chosen in G' as w_{uv} is adjacent to both u and v.

4. The size k remains the same for both the vertex cover in G and the governing set in G', as the condition for the size of the subset is preserved in the reduction.

The reduction is correct as a vertex cover C in G ensures that all edges are covered, meaning each edge $\{u, v\}$ has at least one of its endpoints in C. While in G', set C will also govern all new vertices w_{uv} as w_{uv} is connected to both u and v, for which one of them has been mentioned to be in C, implying C is a governing set in G'. Conversely, a governing set in G' must include all vertices in V, as each w_{uv} is connected only to u or v, making it a vertex cover in G as well.

The reduction is also polynomial as:

1. Graph Creation and Initialization: Constructing the graph G' from G involves initializing V' and E' with V and E, respectively. This step requires linear time relative to the number of vertices and edges in G, which is polynomial.

2. Adding New Vertices for Each Edge: For each edge $\{u, v\}$ in E, a new vertex w_{uv} is added to V'. Since there are at most $O(n^2)$ edges for n vertices in a simple graph, this step is also polynomial in the size of G.

3. Adding New Edges to E': Similarly, adding edges $\{u, w_{uv}\}$ and $\{v, w_{uv}\}$ to E' for each edge in E is a polynomial-time operation, as it only involves iterating over the edges in E.

4. Preservation of Problem Size k: Although the relationship between the sizes of the vertex cover in G and the governing set in G' needs careful definition, the process of preserving or translating this size constraint is inherently polynomial, as it involves no additional computations beyond those required to construct G'.

Thus, each step of the reduction process is polynomial in complexity, ensuring that the reduction itself conforms to the requirements of polynomial-time reducibility. This reinforces the conclusion that the vertex-cover problem is indeed polynomial-time reducible to the governing set problem, highlighting the computational equivalence of these two problems in terms of their complexity classes.

3 NP-complete.

Prove that the following problem is NP-complete. Given a set X and a family $\mathcal{F}$ of subsets of X, whether there are two disjoint sets S_1, S_2 such that $S_1 \cup S_2 = X$ and for any set A in $\mathcal{F}$, A is not a subset of S_1 and S_2. The following is an example. Let $X = \{1, 2, 3, 4, 5\}, \mathcal{F} = \{\{1, 2\}, \{3, 4\}, \{1, 2, 4\}\}$. A possible assignment is $S_1 = \{1, 4\}, S_2 = \{2, 3, 5\}$. The assignment of $S_1 = \{1, 3, 4\}, S_2 = \{2, 5\}$ is infeasible since a subset $\{3, 4\}$ of $\mathcal{F}$ is a subset of S_1. (Hint: consider not-all-equal 3-satisfiability (NAE3SAT, see Wikipedia) which is NP-complete)

To prove that the given problem is NP-complete, we show that:

- The given problem $\in$ NP
- The given problem $\in$ NP-hard (NAE3SAT $\leq_p$ the given problem):
  - NAE3SAT is a known NPC problem
  - Construct a reduction f transforming every NAE3SAT instance to the given problem instance
  - Prove that $x \in$ NAE3SAT `iff` $f(x) \in$ the given problem
  - Prove that f is a polynomial time reduction

For the given problem, we can verify the correctness of S_1 and S_2 in the given problem by:

(1) Checking whether $S_1 \cup S_2 = X$, which is done by merging both S_1 and S_2 then compare every elements (sorted) to X. This takes $\mathcal{O}(|S_1| + |S_2| + |X|) = \mathcal{O}(|X|)$, which is polynomial.

(2) Checking for every A in F whether A is a subset of S_1, which is done by merging both A and S_1 then compare every elements (sorted) to S_1. We deduce A is a subset of S_1 if $A \cup S_1 = S_1$. This takes $\mathcal{O}(|F|(|A| + |S_1| + |S_1|)) = \mathcal{O}(|F||X|)$, which is polynomial.

(3) Similar verification is done with every A in F for S_2, which is computed in $\mathcal{O}(|F||X|)$, another polynomial computation.

As any solution to the given problem can be verified in polynomial time, we confidently state that the given problem is $\in$ NP.

Consider another problem, NAE3SAT, which is a known NPC problem (Wikipedia). By its definition, NAE3SAT requires that three values in each clause are not all equal, i.e., at least one is true, and at least one is false. Later, this will be utilized to split the condition between S_1 and S_2.

To construct the main set X, we put every variable for both itself (v) and its negation ($\tilde{v}$). Then, for each clause, $C_r = (l_1^r, l_2^r, l_3^r)$, we add the corresponding set $\{l_1^r, l_2^r, l_3^r\}$ into F. Now, if there is a satisfying solution to the previous assignments, we build S_1 and S_2 as follows:

- If a variable v is assigned as true, we put the corresponding element into S_1 and its negation into S_2.
- If a variable v is assigned the otherwise, then we put the corresponding element into S_2 and its negation into S_1.

This mapping implies that no set in F (representing each clause) is completely contained within either S_1 or S_2, satisfying the requirement of our problem. We will not violate the $A \in F$ rule as each clause in NAE3SAT ensures that all laterals can't have the same truth value. This proves the $x \in$ NAE3SAT $\Rightarrow f(x) \in$ the given problem.

For a solution to the set-partitioning problem, we can construct a truth assignment for NAE3SAT similarly. If an element is in S_1, assign the corresponding literal with true; otherwise, assign it with false if it is in S_2. As the solution ensures that no set $A \in F$ is completely contained in both S_1 and S_2, this guarantees that not all literals in the same clause have the same truth value. This proves the $x \in$ NAE3SAT $\Leftarrow f(x) \in$ the given problem.

Lastly, we must show that f is a polynomial time reduction. In our construction, we put every variable into set X that is done using a loop that runs in $\mathcal{O}(n)$, where n is the number of variables. For each clause, we add the corresponding set into F, which is also done in $\mathcal{O}(3n) = \mathcal{O}(n)$. f runs in $\mathcal{O}(n)$, making it a polynomial time reduction.

With all the previously mentioned requirements, we conclude that the given problem is NP-complete.

4 Randomized Algorithm.

Suppose we have n servers and m tasks. Each task is independently and uniformly randomly assigned to a server among n of them. In this question, you do not need to rigorously consider if a value is integer. Suppose that $m = 2n \log n$.

(a) We focus on the first server. Show that the probability of it receiving at least $2e \cdot \log n$ tasks is no larger than $1/n^2$. (5 marks)

(a) Since each task is both independent and uniform, then for the first server:

$$\mu = E[x] = \frac{m}{n} = 2 \log n$$

Then, by Chernoff's inequality:

$$\begin{aligned} Pr[x \geq (1+\epsilon)\mu] &\leq (\frac{e^{\epsilon}}{(1+\epsilon)^{(1+\epsilon)}})^{\mu} \\ \Rightarrow Pr[x \geq 2e\log n] &\leq (\frac{e^{e-1}}{e^{e}})^{2\log n} \\ &= \frac{1}{e^{2\log n}} \\ &= \frac{1}{n^2} \end{aligned}$$

Hence, the probability of the first server receiving at least $2e\log n$ tasks is no larger than $\frac{1}{n^2}$ is shown.

(b) Following (a), show that when n is large enough, then with high probability, no server receives at least $2e\log n$ tasks. (5 marks)

(b) By part (a), we can conclude that for any of the n servers, the probability of having at least one server get at least $2e\log n$ tasks is at most:

$$\begin{aligned} 1-(1-\frac{1}{n^2})^n &= 1-(\frac{n^2-1}{n^2})^n \\ &= \frac{n^{2n}-(n^2-1)^n}{n^{2n}} \\ &= \frac{f(x)}{n^{2n}} \end{aligned}$$

, where $f(x)$ is a polynomial function with degree less than $2n-1$ (as the n^{2n} term cancels out).

When n is large, say n goes to ∞, then:

$$\lim_{n\to\infty}\frac{f(x)}{n^{2n}} = 0$$

The probability of having at least one server get at least $2e\log n$ tasks is 0; equivalently, no server receives at least $2e\log n$ tasks.