ADA Homework 4

許崴棠

December 26, 2023

Problem 0

- **Problem 1:** b11902116 張迪善, b11902046 李明奕, b11902158 黃昱凱
- **Problem 2:** b11902116 張迪善, b11902046 李明奕, b11902084 張閔堯
- **Problem 3:** b11902109 侯欣緯, b11902116 張迪善

Problem 1

(a) To determine if a 2-CNF problem is in 2-SAT, we can regard the 2-CNF formula as a set of clause, take ϕ_1 as an example, we can regard ϕ_1 as

$$\phi_1 = \{(a_1 \lor \neg a_2) \land (a_3 \lor a_1) \land (\neg a_1 \lor \neg a_1)\}$$

and by De Morgan law, we can rewrite ϕ_1 as

$$\phi_1' = \{\neg(\neg a_1 \land a_2) \land \neg(\neg a_3 \land \neg a_1) \land \neg(a_1 \land a_1)\}$$

Consider a directed graph G where $V(G) = \{a_1, a_2, \dots, a_n, \neg a_1, \neg a_2, \dots, \neg a_n\}$. And to make $\neg(x \land y)$ in ϕ_1' true, we can add the edge of $x \to \neg y$ and $y \to \neg x$ since that if x is right, $\neg y$ must be right, if y is right, $\neg x$ must be right. After then, we can find the SCCs by the kosaraju's algorithm, and we can check if a 2-CNF formula is solvable simply by checking if there exist a a_i where a_i and $\neg a_i$ are in the same SCC, since if x is able to reach y, it means that "if x is true, then y must be true ", and if a_i and $\neg a_i$ are in the same SCC, it means that the statement a_i will be true if and only if statement $\neg a_i$ is true, which is obviously wrong. Thus we can know that there will be at least one set of a that can solve the answer, only if there is no a_i and $\neg a_i$ in the same scc.

(b-1) Since we've already knew Max-2-SAT is in NPC, we can prove Max-f-condSAT is in NPC by reduce Max-2-SAT to Max-f-condSAT and prove Max-f-condSAT can be verify in polynomial time.

For ϕ_n in Max-2-SAT, we can use $n^{1000} - n$ unit of time to add $\land(a_{n+1} \lor \neg a_{n+1}) \land (a_{n+2} \lor \neg a_{n+2}) \land \dots \land (a_{n^{1000}} \lor \neg a_{n^{1000}})$ behind the original clause, therefore we can tranlate any clause to Max-f-condSAT in linear time. Thus we can tell Max-f-condSAT is in NP-hard.

And in every unit time we can check if the clause $(a_i \lor a_n)$ is true. Therefore when the number of clause is n, we can tell the number of true clause in n unit of time, so the Max-f-condSAT is in NP.

Since Max-f-condSAT belongs to NP and NP-hard, Max-f-condSAT belongs to NPC.

(b-2) When $f(x) = \log x$, giving a ϕ_n with m clauses, for every literal a_i, we only need to check $\log n$ literals before a_i, since that the best answer before $a_{i-\log n}$ is all set, so when we're checking a_i, we can apply the best answer of $a_{i-\log n-1}$ as template, and enumerate the best answer among $a_{i-\log n} \in \{true, false\} \land a_{i-\log n+1} \in \{true, false\} \land \dots \land a_i \in \{true, false\}$.

Since every time we check we need to spend m unit of time (run through ϕ_n). And for every literal, we had to check if the $\log n$ literals before a_i is true or false, which will take $2^{\log n}$time, so it will take $n \cdot m \cdot 2^{\log n} = m \cdot n^{1+\log 2}$ to check through the whole ϕ_n in order to get the answer, which only take polynomial time to get the answer.

(c-1) For every literal in ϕ_i, we can let $(a_1 = 1 \vee a_2 = 2)$ be $(a_{1(1)} \vee a_{2(2)})$(for example) where $a_{m(n)}$ means that $a_m = n$ and add

$$
\begin{gathered}
(\neg a_{1(1)} \vee \neg a_{1(1)}) \wedge (\neg a_{1(1)} \vee \neg a_{1(2)}) \wedge \cdots \wedge (\neg a_{1(1)} \vee \neg a_{1(c)}) \wedge \\
(\neg a_{1(2)} \vee \neg a_{1(1)}) \wedge (\neg a_{1(2)} \vee \neg a_{1(2)}) \wedge \cdots \wedge (\neg a_{1(2)} \vee \neg a_{1(c)}) \wedge \\
\cdots \wedge \\
(\neg a_{1(c)} \vee \neg a_{1(1)}) \wedge (\neg a_{1(c)} \vee \neg a_{1(2)}) \wedge \cdots \wedge (\neg a_{1(c)} \vee \neg a_{1(c)}) \wedge \\
\cdots \wedge \\
(\neg a_{c(c)} \vee \neg a_{c(1)}) \wedge (\neg a_{c(c)} \vee \neg a_{c(2)}) \wedge \cdots \wedge (\neg a_{c(c)} \vee \neg a_{c(c)})
\end{gathered}
$$

at the end to ensure that a_1 only equals one number. And after conversion, we can reduce a c-GenSetSAT ϕ_n with constraint Tc, k clause and n literals into a 2-SAT problem in $k + n \cdot c^2$ unit of times. And since that 2-SAT is in P, so c-GenSetSAT is in P too.

(c-2) Since that we've already know 3-colorability problem is in NPC, and we can reduce every connected area (a_i, a_j) to

$$\neg(a_i = 1 \wedge a_j = 1) \wedge \neg(a_i = 2 \wedge a_j = 2) \wedge \neg(a_i = 3 \wedge a_j = 3)$$

because we've already knew every connected area can't have same colors. And by De morgan's Law, when $C \geq 3$ we can turn it into

$$
\begin{gathered}
(a_i \in \{1, 2, \ldots, C\} - \{1\} \vee a_j \in \{1, 2, \ldots, C\} - \{1\}) \wedge \\
(a_i \in \{1, 2, \ldots, C\} - \{2\} \vee a_j \in \{1, 2, \ldots, C\} - \{2\}) \wedge \\
(a_i \in \{1, 2, \ldots, C\} - \{3\} \vee a_j \in \{1, 2, \ldots, C\} - \{3\})
\end{gathered}
$$

After we translate every connected area to the upper form, we tell that we've turn a 3-colorability problem to a c-GenSetSAT problem. For every 3-colorability problem with n connections, we can convert it into a c-GenSetSAT problem in $O(3n) = O(n)$ time, and since we can reduce a 3-colorability problem to c-GenSetSAT problem, c-GenSetSAT problem is in NP-hard.

And we can first go through every clauses in c-GenSetSAT, since every clause only got two literals, we can determine whether the clause is true or false in $O(n)$. And check through every clauses only take $O(n)$ unit of time, therefore we've know we can verify the c-GenSetSAT formula in polynomial time, so c-GenSetSAT problem is in NP.

since c-GenSetSAT problem is in NP and NP-hard, c-GenSetSAT problem is NPC.

Problem 2

(a) By observation, we can tell that after we put α items into the current box, we'll had to spend 2α time to make a new box, so when we're putting new items into the box, we can save 7 dollars to the bank per item, therefore, after we put α items into the box with the size 2α, we can directly spend the 4α dollars we've saved in the bank to make a new box with the size of 4α, and spend 2α dollars to move the 2α items in the original box. By doing this, we can say that the time complexity of putting n items into the box are $O(7n) = O(n)$.

(b) It is impossible to amortize the time complexity to $O(n)$ because that when $n = s+1$, we need to spend s^2 time to make a new box, and it would be impossible to amortize a s^2 step to $s+1$ step to make the overall time complexity $O(n)$.

(c) By observation, we can tell that after we put α items into the current box, we'll had to spend α^2 time to make a new box. When the box is full when we put the i_{th} item into the box, we had to make sure the previous $i - \sqrt{i}$ items can afford the price of making new box and move the every items into the new box. So when we're moving n^2 items to a new box, we had to pay $(n^2)^2 + n^2$ dollars, and it should be pay by the $n+1_{th}$ to ${n^2}_{th}$ items. By math, we can know that to put the if we assume every i_{th} item had to pay $\alpha i + \beta$ dollars to bank. Since we already knew the first box (size 2) can be enlarge to the second box if every one put $2n$ dollars into the bank.
We had to calculate the general case, which is

$$(n^2)^2 + n^2 \leq \alpha \frac{(n^2+n+1)\cdot(n^2-n)}{2} + \beta(n^2-n) \; \forall \; n \geq 2$$

where n^2 is the current box size.
By math, we can tell that the tightest α is 2 and the tightest β is 3. Therefore, if every i_{th} insert put $2i+3$ dollars into the bank, we can safely make a n^2 box and move the previous n_{th} items into the new box only by using the $n^2 + n$ the previous $\sqrt{n} + 1_{th}$ to n_{th} items had saved.

(d) First, we assume the worst case, which happens when the manager come and check after every time Nathan put an item into the box.
In the cost section of the table below, it represent the (cost of insert + cost of move)

i	cost	i	cost	i	cost
1	1+0	6	1+1	11	1+1
2	1+1	7	1+1	12	1+1
3	1+3	8	1+1	13	1+1
4	1+1	9	1+9	14	1+1
5	1+5	10	1+1	15	1+1

By the table above, we can find out that the maximum averagetime complexity happens when we put $n = 2^k + 1$ item into the boxes, which can take at most $2n-1+2(2^k-1) = 4n-3$ times, and we can tell the time complexity is $O(4n-3) = O(n)$.

(e) First of all, we can assume the potential function is

$$\phi(D_i) = 2 \cdot (\text{number of boxes filled}) + 9 \cdot (\text{number of packages})$$

validity check:

$$\begin{cases} \Phi(d_0) = 0 \rightarrow \text{ there won't be any box filled initially or recieved any packages} \\ \Phi(d_i) \geq 0 \rightarrow \text{ there won't be negative numbers of boxes or packages} \end{cases}$$

let c_i be the cost of the i_{th} operation and $\hat{c_i} = c_i + d_i - d_{i-1}$ be the amortized cost of the i_{th} operation.

The armortized cost of Arrival:
We knew that every time a box with size 2^i are created, it means that there must have every boxes with size of $2^0, 2^1, \ldots, 2^{i-1}$ are cleared in one operation since there can't be 2 box with same sizes at the same time.
we define $LSBx$ as the smallest empty box, for example, if we had $2^0, 2^1, 2^2, 2^4, 2^5, 2^6$ filled, the $LSBx$ is 3 since the 2^3 box is the smallest box unfilled.
Therefore, we knew the armortized cost of Arrival is

$$\begin{aligned}\hat{c_i} =& c_i + d_i - d_{i-1} \\ =& (1 + 2 \cdot LSBx(i-1)) + (d_{i-1} - 2 \cdot (LSBx(i-1) - 1) + 9) - d_{i-1} \\ =& 1 + 2LSBx(i-1) - 2LSBx(i-1) + 11 = 12\end{aligned}$$

The armortized cost of Deliver:
when we had to deliver k object, if we take every thing out of the box and put $(package\_num - k)$ packages back to the box, we can find out the cost of taking every thing out a box takes $3 \cdot (package\_num) - 2 \cdot (box\_num)$ since take a thing out cost $package\_num$ unit of time and split every box to size $= 1$ takes $2 \cdot (package\_num) - 2 \cdot (box\_num)$
And since putting n things back can be regarde as the reverse movement of taking n packages out of the boxes that stored n packages, which cost $3 \cdot (package\_num - k) - 2 \cdot (new\_box\_num)$.

Deliver(take every thing out):

$$\begin{aligned}\hat{c_i(1)} =& c_i(1) + d_{mid} - d_{i-1}\\ =& \{3(package\_num) - 2(box\_num)\} + \{0\} - \{2(box\_num) + 9(package\_num)\}\\ =& 3(package\_num) - 4(box\_num) - 9(package\_num)\end{aligned}$$

Deliver(put $package\_num - k$ things back):

$$\begin{aligned}\hat{c_i(2)} =& c_i(2) + d_i - d_{mid}\\ =& \{3 \cdot (package\_num - k) - 2 \cdot (new\_box\_num)\} + \{2(new\_box\_num)\\ &+ 9(package\_num - k)\} - \{0\}\\ =& 3(package\_num - k) + 9(package\_num - k)\end{aligned}$$

Therefore we can know the overall armortized cost of Deliver is the sum of take every thing out and put the rest thing back, which is

$$-4(box\_num) + 6package\_num - 12k$$

and since $2k \geq package\_num$, the overall amortized cost is less than $-4(box\_num)$, which is less than zero. Therefore we can know that after amortizing, the cost of every step is constant or less than 0, therefore the overall cost of doing n operation is $O(n)$.