CSC263 Assignment 4

Zikun Chen, Yue Li, Yueen Ma

March 16, 2017

Question 1

(Written by Yueen Ma, Read by Zikun Chen, Yue Li)

Algorithm:

General idea:
Instead of removing the edges starting from the first edge in the linked list to the last one, we do the reverse of it. i.e., we start with all vertices disconnected. From the last edge in the linked list to the first one, CONNECT the graph one edge at a time.
That is, with all vertices disconnected at the beginning, we first connect edge e_m, then e_{m-1}, then e_{m-2}, ..., finally e_1.
This is like rewinding the time. The benefit of doing so is that we can make use of disjoint set to keep track of the components. Every time an edge connects two components, we check the size of the new component, if it exceeds $\lfloor \frac{n}{4} \rfloor$(size $> \lfloor \frac{n}{4} \rfloor$), then this edge is the edge we are looking for.
Assumptions:

- 1) The linked list of edges is a doubly linked list.
- 2) The graph is given using an adjacency-list

Data structure:

- We use disjoint set(disjoint-set forest) to maintain connected components, we do union operation using Weighted Union. So the size of each set(the number of vertices) is also available to us.

Algorithm:

Step 1: For each vertex v in the graph(i.e. each vertex in the adjacency-list), we call MAKE-SET(v). Set the size of each set to 1.

Step 2: We initialize a pointer called *cur_edge* to point to the tail of the linked list of edges

Step 3: While *cur_edge* is not NIL(the previous node of the head node of the linked list has value NIL), we do:(execute each sub-step in order if we do not jump)
(1) Let v1 and v2 be the two vertices to be connected by *cur_edge*, i.e. *cur_edge* = (v1, v2);
(2) Call FIND-SET on both vertices, if FIND-SET(v1) != FIND-SET(v2)(i.e., not in the same set, which mean that they are not in the same component), we continue to do (3); else, jump to (5)
(3) Call UNION(v1, v2), when doing UNION, we also update the size: we sum the size of the two sets, and assign the new size to the new set object.
(4) If the new size exceeds $\lfloor \frac{n}{4} \rfloor$(new size > $\lfloor \frac{n}{4} \rfloor$), we break the loop, and return *cur_edge* as the result; else, continue to do (5)
(5) Let *cur_edge* = *cur_edge.previous*(let *cur_edge* point to the previous node in the linked list). Jump to (1)

Time complexity:

Step 1: Because each MAKE-SET operation takes $O(1)$ time, so n MAKE-SET operation will take $n \times O(1) = O(n)$ time.
Step 2: The initialization process takes $O(1)$ time.
Step 3: Let the edge we are looking for be e_i(the same e_i as we have in the handout), where $1 \leq i \leq m$. Each iteration executes:

- (1): takes $O(1)$;
- (2): Each FIND-SET takes $O(logn)$, so (2) takes $O(2logn)$, which is the same as $O(logn)$;
- (3): Since not every time the UNION will be called, we can analyze the upper bound by assuming that every time UNION would be called. Each UNION takes $O(1)$ time, so (3) takes $O(1)$ time.
- (4) and (5): Checking the conditions will also only take $O(1)$ time; Moving the *cur_edge* will cost $O(1)$ time; Therefore, in total, (4) and (5) takes $O(1)$ time.

By summing them up, each iteration takes $O(1) + O(logn) + O(1) + O(1) = O(logn)$. The number of iterations depends on the number of edges that come after e_i in the linked list(because we move from the back to the front in the linked list). So the number of iterations is equal to $m - i + 1$, which is $m - i + 1 \leq m$. Therefore, Step 3 takes $m \times O(logn) = O(mlogn)$

Then, in summary, the time cost of all three Steps is $O(n)+O(1)+O(mlogn) = O(n) + O(mlogn)$. because a graph with n vertices has at least n-1 edges to connect them, then $m \geq n - 1$. Thus, $O(n) + O(mlogn) \leq O(1) + O(n - 1) + O(mlogn) \leq O(1) + O(m) + O(mlogn) \leq O(1) + O(m(logn + 1)) \leq O(1) + O(mlogn) \leq O(mlogn)$.
Therefore, the algorithm takes $O(mlogn)$ time.

Improve the complexity:
When doing the UNION, we use both Weighted Union(WU) and Path compression(PC).
Then Step 3 can be improved: Step 3 uses 2m FIND-SET and at most n UNION(other $O(1)$ can also be counted into time complexity of Union because they are all constants). Step 3 takes $O(m\alpha^{-1}(m,n))$.
Therefore, the total cost of all 3 steps is $O(n) + O(1) + O(m\alpha^{-1}(m,n)) \leq O(1) + O(m(\alpha^{-1}(m,n)+1)) \leq O(m\alpha^{-1}(m,n))$ as $m \geq n-1$. Then the time complexity of the algorithm is $O(m\alpha^{-1}(m,n))$.

Question 2

(Written by Yue Li, Read by Yueen Ma, Zikun Chen)
Claim: The smallest cost B in the given list is: $c + \frac{3}{2}d$

(a) **Using aggregate method**:
Let n be the number of times incrementing the display.
Let $B = c + td$ be the amortized cost for performing n successive display increments.
Since c and d are fixed costs whenever display is updated, B(n) depends on t, which is the average number of digits need to change when n successive display increments are performed.

Now, find t:

Let N(n) be the total number of digits need to change when n successive display increments are performed.

Then $m' = \frac{N(n)}{n}$
Note: m' is the average number of digits that must be changed.

Since the counter in this scenario is of base 3, when n successive display increments are performed, by the property of base 3 notation:
The 1st digit in representation changes n times,
The 2nd digit in representation changes $\left\lfloor \frac{n}{3^1} \right\rfloor$ times,
The 3rd digit in representation changes $\left\lfloor \frac{n}{3^2} \right\rfloor$ times,
.
.
.
follow this pattern
The ith digit in representation changes $\left\lfloor \frac{n}{3^{i-1}} \right\rfloor$ times, Therefore we get N(n) as follows:

$$
\begin{aligned}
N(n) &= \sum_{i=1}^{\lfloor log_3 n \rfloor + 1} \lfloor \frac{n}{3^{i-1}} \rfloor \\
&\leq n \sum_{i=1}^{\infty} \frac{1}{3^{i-1}} \\
&= n \frac{1}{1 - \frac{1}{3}} \\
&= \frac{3}{2} n \\
\text{Then}, m^{'} &= \frac{N(n)}{n} \leq \frac{3}{2} = t
\end{aligned}
$$

Then $B = c + \frac{3}{2}d$
Therefore, $A(n) \leq B = c + \frac{3}{2}d$, for $n \geq 1$

(b) **Using accounting method:**

Charging Scheme:

Charge: \$1.5 each time the counter needs to be incremented.
Cost: \$1 for each display digit needs to be changed.

Want to show the above charging scheme is enough to support n successive display changes.

1st change: counter start from 0 change to 1, charge \$1.5 and cost \$1, the counter results in 1 with $\$0.5 = 1.5 - 1$ left

2nd change: counter change from 1 to 2, charge \$1.5 and cost \$1, the counter results in 2 with $\$1 = 0.5 + 1.5 - 1$ left

3rd change: counter change from 2 to 10, 2 digits need to be changed, charge \$1.5 and cost \$1, the counter results in 10 with $\$\ 0.5 = 1 + 1.5 - 2$ left

.
.
.

Follow this pattern, The credit invariant is as follows:

digit that is 0 always has \$0;
digit that is 1 always has \$0.5;
digit that is 2 always has \$1;

Proof:

We denote the number as $|d_n d_{n-1} ... d_2 d_1|$ where the d's are the digits. e.g. the trit number $|d_3 d_2 d_1| = 120$(decimal 15), $d_3 = 1, d_2 = 2, d_1 = 0$
To show the credit invariant is true, we consider the follow two steps.

Step 1 For d_1(the least significant digit), it is charged \$1.5 each increment, and costs \$1 each increment because it is changed on each increment. So the credit increases by \$0.5 on each increment. Starting from 0 with \$0 credit, then 1 gets \$0.5 credit and then 2 gets \$1 credit. Then we have a carry, the digit is 2 and it has \$1 credit. After a carry, 2 becomes 0 and the credit is increased by \$0.5(charge \$1.5, cost \$1), then we have \$1.5 credit AFTER d_1 is changed. This extra \$1.5 is transfered to d_2(the second least significant digit). d_1 becomes 0 and has \$0 credit again. So, The credit invariant holds for d_1.
Step 2 Assume for any positive natural number $k = 2, 3, 4 \ldots$, d_k receives \$1.5 whenever there is a carry from d_{k-1}.
d_k gets a carry, 1 needs to be added to the d_k and it receives \$1.5 from d_{k-1} by the assumption. The cost of changing the digit is \$1, so the credit of d_k is also increased by \$0.5. Starts from 0 with \$0 credit, then 1 with \$ 0.5, then 2 with \$1. Then we have a carry, d_k is 2 with \$1 credit, then it is changed to 0 and the credit is increased to \$1.5. This credit is transfered to d_{k+1}. Then d_k is changed back to 0 with \$0 credit. The credit invariant holds for d_k.
(1)By Step 1, we know that d_1 satisfies the credit invariant, and the assumption in Step 2 for $k = 2$ holds;
(2)Then, we know that the credit invariant holds for $k = 2(d_2)$ because the assumption holds(by (1)), and also the assumption in Step2 holds for $k = 3(d_3)$;
(3)Then, the credit invariant holds for $k = 3(d_3)$ because the assumption holds(by (2)), and also the assumption in Step2 holds for $k = 4(d_4)$;
(4)Then, the credit invariant holds for $k = 4(d_4)$ because the assumption holds(by (3)), and also the assumption in Step2 holds for $k = 5(d_5)$;
... so on.
Therefore, every digit satisfies the credit invariant.

The following worst-case example could show that charge scheme stated above can always support the next update,
Consider the display that is 222....2222, say, n successive '2'. In this case, the next display update will cost most, since it will change all n digit plus a carry out digit, therefore (n+1) digits need to change. According to our charge scheme, the cost for next updating is:

$$\begin{aligned} C &= \$1.5 + n \times \$1 - (n+1) \times \$1 \\ &= \$1.5 + \$n - \$n - \$1 \\ &= \$0.5 \\ &> 0 \end{aligned}$$

We have shown the charge scheme stated above is enough to support n successive display changes.

(c) **Show $c + \frac{3}{2}d$ the smallest upper bound for A(n)**

We can find the largest bound in given list smaller than $c + \frac{3}{2}d$, in this case, is $c + \frac{17}{12}d$, and show it is not enough to support n successive display changes where $n \geq 1$:

Let n = 18, Then we get average cost for each update as follows:

Decimal	Trit
0	000
1	00$\underline{1}$
2	00$\underline{2}$
3	0$\underline{1}$0
4	0$\underline{1}$1
5	0$\underline{1}$2
6	0$\underline{2}$0
7	0$\underline{2}$1
8	0$\underline{2}$2
9	$\underline{1}$00
10	$\underline{1}$01
11	$\underline{1}$02
12	$\underline{1}$10
13	$\underline{1}$11
14	$\underline{1}$12
15	$\underline{1}$20
16	$\underline{1}$21
17	$\underline{1}$22
18	$\underline{2}$00

Note: The digits that changed each time are highlighted by underline.
From the table above, by counting the number of underlines of each digit, we can conclude that:

-The 1st digit changes 18 times in total;

-The 2nd digit changes 6 times in total;

-The 3rd digit changes 2 times in total.

Then we can get m' as follows:

$$\begin{aligned} m' &= \frac{N(n)}{n} \\ &= \frac{(18+6+2)}{18} \\ &= \frac{26}{18} \\ &\approx 1.44444 > \frac{17}{12} \approx 1.417 \end{aligned}$$

Which means that $\frac{17}{12}$ is not enough to support n display update **on average** when $n = 18$,

Therefore $c + \frac{3}{2}d$ is the smallest upper bound for A(n) **in the given list**.

Question 3

(Written by Zikun Chen, Read by Yueen Ma, Yue Li)

(a) Charge:

INSERT(S,x): Charge the new element x with \$12

DIMINISH(S): No charge.

We pay \$1 whenever there is a pairwise comparison as indicated in the question.

(b) Credit Invariant (CI): Each element in S has a credit of at least \$12 at any point in the sequence of INSERT and DIMINISH operations.

Proof: Let the number of operations in the arbitrary sequence be m.

There is no pairwise comparison in INSERT. And there is at most 6N pariwise comparisons for DIMINISH(S) where n = N is the number of elements in S. Because in the algorithm for DIMINISH, i) calls MED(S) which takes at most 5N comparisons. And ii) loops over all the elements in S and compare each with m, which takes N comparisons.

Induction on m:

Base Case:

When m = 0, S is empty and there is no element in S so CI is vacuously true.

At this point, we cannot DIMINISH, so we can INSERT a new element and charge it \$12. Now m = 1, and the only element has \$12 charge to it. So CI is true when m = 1.

Induction Hypothesis:

Assume CI holds for all m $\geq$ 1, we want to show CI also holds for m + 1 The size of the list at this point is at most m (when all m operations are INSERT). And each element has \$12 charge.

Case 1: If we INSERT an new element, then the new element will be charged \$12 and no comparison will be made. CI holds.

Case 2: If we DIMINISH, then there will be at most 6m pairwise comparisons we have to pay for. Since CI holds for m, then we use half of the \$12m total credit saved up in the elements to pay for the pair comparisons. We keep the rest of the \$6m credit to the new set which has size at most $\frac{m}{2}$ ($\lfloor \frac{m}{2} \rfloor \leq \frac{m}{2}$ for any $m = 0, 1, 2, \cdots$ and when m is odd, the size will be even smaller.). Now, each elements has credit at least $\frac{6m}{m/2} = 12$ dollars. CI holds.

Therefore the credit invariant holds for any m sequence of INSERT and DIMINISH operations.

(c) By the credit invariant, there is \$12 surplus in each element of S at any point in the sequence of INSERT and DIMINISH operations.

An there is either no elements in S (empty) or some elements in S.

In the worst case, the sequence should have as many DIMINISH operations as possible because it contains all the pairwise comparisons, which are relevant to the time complexity. And when we perform N consecutive INSERT operations, the size of S is now n = N, we can at most have log(N) DINIMISH because each time DINIMISH at least cuts the size of S in half. ($\lfloor \frac{N}{2} \rfloor \leq \frac{N}{2}$ for any $N = 0, 1, 2, \cdots$ and when N is odd, the size will be even smaller.)

Therefore, in the worst case, the number of total operations in the sequence is $k = N + log(N)$.

Therefore, surplus = Total Credit - Total Cost (worst case sequence cost) ≥ 0

Amortized Cost = $\frac{\text{WCSC}}{k} \leq \frac{\text{Total Credit}}{k}$

And Total Credit = $12N$ when we have N INSERT and log(N) DIMINISH (The worst case sequence).

$$\begin{aligned} \text{Amortized Cost} &\leq \frac{12N}{k} = \frac{12N}{N + log(N)} \\ &\leq \frac{12N + 12log(N)}{N + log(N)} = 12 \in O(1) \end{aligned}$$