Algorithm HW#3

B02901178 江誠敏

December 17, 2016

Collaborators: None

1 Problem 1.

We define our day structure to be a pair of Fibonacci heap (G, H) such that

1. G is a max Fibonacci heap.
2. H is a min Fibonacci heap.
3. The size of G, H satisfied $|G| = |H|$ or $|G| = |H| + 1$.
4. For all $g \in G, h \in H$, $g \leq h$.

Let n be the number of elements, i.e., $n = |G| + |H|$, g^* be the maximum of G, and h^* be the minimum of H. It is easy to see that if $k = \lceil n/2 \rceil$, g^* is the k-th smallest element, and h^* is the $k+1$-th smallest element, so

1. If $2 \mid n$, then $(g^* + h^*)/2$ is the median. (Or we shall define g^* to be the median in this case.)
2. If $2 \nmid n$, then g^* is the median.

Now recall that the Fibonacci heap has the preformances list in below.

Lemma 1. *In a Fibonacci heap, the following operations cost*

- `insert`*:* $\mathcal{O}(1)$*.*
- `extract_min`*:* $\mathcal{O}(\log n)$*.*
- `delete`*:* $\mathcal{O}(\log n)$*.*
- `merge`*:* $\mathcal{O}(1)$*.*

We define a new procedure `adjust` as following.

- If $|G| = |H|$ or $|G| = |H| + 1$, do nothing.
- If $|G| < |H|$, preform `extract_min` on H and get x, then preform `insert` x on G. After the procedure, $|G| \leftarrow |G| + 1, |H| \leftarrow |H| - 1$.

- If $|G| > |H| + 1$, preform `extract_min` on G and get x, then preform `insert` x on G. After the procedure, $|G| \leftarrow |G| - 1, |H| \leftarrow |H| + 1$.

Notice that each `adjust` cost $\mathcal{O}(\log n)$. And after the procedure $\Big||G| - |H|\Big|$ decreases until $|G| = |H|$ or $|G| = |H| + 1$.

Then we give the procedure of the operations.

- `ins`(x): If $k \leq g^*$ then `insert`(G, x), else `insert`(H, x). The cost is $\mathcal{O}(1)$.
- `del`(x): If $x \in G$ then `delete`(G, x), else `delete`(H, x). The cost is $\mathcal{O}(logn)$.
- `extract_median`: Simply delete g^*. It cost $\mathcal{O}(\log n)$.
- `union`: Let $(G_1, H_1), (G_2, H_2)$ be the two we shall union. By the statement we only need to merge them if there median are the same. Let g^* be their median, then we know that $x \in G_1 \cup G_2 \implies x \leq g^*$. So we could simply call $G' \leftarrow$ `merge`$(G_1, G_2), H' \leftarrow$ `merge`(H_1, H_2). And (G', H') is the desired result. The cost is $\mathcal{O}(1)$ since they are fibonacci heaps.

Let (G', H') be what we get after a single operation. Since $\Big||G| - |H|\Big| \leq 1$ before the operation, and each operation will delete/insert at most 1 element in G or in H. So $\Big||G'| - |H'|\Big| \leq 2$. (In the `union` operation, $\Big||G_1| - |H_1|\Big| \leq 1$ and $\Big||G_2| - |H_2|\Big| \leq 1$ so still $\Big||G'| - |H'|\Big| \leq 2$.) Hence after at most 2 time of `adjust`, which cost $\mathcal{O}(\log n)$, the two heap "recovered", that is, $|G'| = |H'|$ or $|G'| = |H'| + 1$ again. Hence each operation cost $\mathcal{O}(\log n)$.

2 Problem 2.

We know that "create a new array with double size $2n$, transfer the old n items" takes linear time, so let β be the constant such that the operation above cost no more than βn.

Let n be the current size of the array, and m is the current number of elements inside the dynamic array. We shall define the potential function to be $\Phi = \beta(2m - n + 1)$.

Notice that since if $m \geq 1$, then $n \leq 2m$ by statement since every time we enlarge the array size to at most twice the number of elements when the array is full. And if $m = 0$ then $n = 1$ and $2m - n + 1 = 0$, hence $\Phi \geq 0$.

Now we analyze the amortized cost.

- Insertion with no resizing cost $\mathcal{O}(1)$, and the potential difference is

$$\Phi' - \Phi = \beta(2(m + 1) - n + 1) - \beta(2m - n + 1) = 2\beta.$$

  Hence the amortized cost is $\hat{c} = \mathcal{O}(1) + 2\beta = \mathcal{O}(1)$.

- Insertion with resizing cost $\beta n + \mathcal{O}(1)$, and the potential difference is

$$\Phi' - \Phi = \beta(2(m+1) - 2n + 1) - \beta(2m - n + 1) = 2\beta - \beta n.$$

Hence the amortized cost is $\hat{c} = \beta n + \mathcal{O}(1) + 2\beta - \beta n = \mathcal{O}(1) + 2\beta = \mathcal{O}(1)$.

So we conclude that the amortized cost is $\mathcal{O}(1)$.

3 Problem 3

We first state a lemma.

Lemma 2. *Given a undirected graph $G = (V, E)$. Finding all vertices that could be reach from a vertex s could be done in $\mathcal{O}(V + E)$ time.*

Proof. Simply runs any linear traversal algorithm such as `DFS` start from s, which runs in $\mathcal{O}(V + E)$ time. The set of all vertices which is visited in the process is the answer. □

Now construct a new graph $G' = (V', E')$ from G by the following rules.

- $V' = \left\{ v'_{i,k} : 1 \leq i \leq |V'|, 0 \leq k \leq 2 \right\}$.
- If (v_i, v_j) is an edge in G with cost c, then $(v'_{i,k}, v'_{j,k'}) \in E'$ where $k' = (k+c) \mod 3$, for $k = \{1, 2, 3\}$. We say that these three edges correspond to the edge (v_i, v_j) in G.

We could see that every path from $s \triangleq v_i$ to v_j correspond to a path from $s' \triangleq v_{i,0}$ to $v_{j,k}$ for a k, and vice versa. Also, if $c_1, c_2, \cdots, c_m$ is the cost of the edge in the path, and let $c = \sum c_i$, then $k \equiv 0 + c_1 + c_2 + \cdots + c_m \equiv c \pmod 3$ by definition. So if c is a multiple of 3, $k = 0$. Hence we turn the problem to finding which vertices $\{v_{j,0}\}$ could be reached from $s' = v_{i,0}$. Since G' has $3|V|$ vertices and $3|E|$ edges, the algorithm runs in $\mathcal{O}(3|V| + 3|E|) = \mathcal{O}(V + E)$ time.

4 Problem 4

Let $G = (V, E)$ be a directed graph, where $V = \{v_0, v_1, \cdots, v_n\}, E = \{(v_0, v_i) : 1 \leq i \leq n\} \cup \{(v_i, v_j) : \text{course } j \text{ requires course } i\}$. The statement "you can complete all courses in some orderings" guarantees that the induced subgraph $V \setminus \{v_0\}$ is a DAG, and after adding v_0 by our construction the graph is still a DAG. So a topological ordering exists (which could be calculate in $\mathcal{O}(V + E)$). And thus we could define

$$d(v_i) = \max_{(v_j, v_i) \in E} d(v_j) + 1, \quad d(v_0) = 0.$$

The function d is well define, and could be calculate in $\mathcal{O}(V + E) = \mathcal{O}(V)$ (since the indegree of each vertices is less then $3 + 1 = 4$, [1] thus $|E| \leq 4|V|$.), since we only need

[1] The 1 is from v_0.

to calculate them by definition in the topological order, and calculate one node cost $\mathcal{O}(\deg v_i + 1)$.

Lemma 3. *The solution of the problem, m^*, equals $m \triangleq \max_{v_i \in V} d(v_i)$.*

Proof. First we proof that $m^* \leq m$ by giving a solution with size m.
Let $A_k = \{v_i : d(v_i) = k\}$. Then in the k-th semester we take all the courses in A_k. The configuration is consistent, since

$$d(v_i) = \max_{(v_j, v_i) \in E} d(v_j) + 1 \implies d(v_i) > d(v_j),\ \forall (v_j, v_i) \in E.$$

So for a course in A_k, all its prerequisites v_j is in $A_{k'}$ with $k' < k$, which we've already taken. And since $m \triangleq \max d(v_i)$, we only needs m semester. Hence $m^* \leq m$.

Now, let us proof that there is a path $v_0, v_1, v_2, \cdots, v_k$ with $k = m$ by induction. Choose v_k so that $d(v_k) = 1$. By definition, exists v^* such that $d(v^*) = k - 1$ and $(v^*, v_k) \in E$. By induction there is a path $v_0, v_1, v_2, \cdots, v_{k-1} \triangleq v^*$, and thus we could extend it to a path $v_0, v_1, \cdots, v_{k-1}, v_k$.

Now in this path, v_2 requires v_1, v_3 requires v_2, ..., v_k requires v_{k-1} and hence we need at least k semester to finish course v_k. Hence $m^* \geq m$ and we conclude $m^* = m$. □

By lemma, since we could compute all $d(v_i)$ in $\mathcal{O}(V)$ time, and finding the maximum cost $\mathcal{O}(V)$ time. So the time complexity is $\mathcal{O}(V) = \mathcal{O}(n)$.

5 Problem 5

Let $p(h, w)$ be the shortest path from u to w using at most h edges. Then $p(h, w)$ is either

- Contains less then h edges, then $p(h, w) = p(h - 1, w)$.
- Contains exactly h edges, then $p(h, w) = \langle u, v_1, \cdots, v_{h-1}, w \rangle$, with some v_{h-1} satisfied $(v_{h-1}, w) \in E$ and $\langle u, v_1, \cdots, v_{h-1} \rangle = p(h - 1, v_{h-1})$.

If we let $f(h, w)$ be the length of the shortest path from u to w using at most h edges , then according to the above, we could write down the recursive formula by

$$f(h, w) = \min_{(w', w) \in E} f(h - 1, w') + c(w', w)$$

Where $c(x, y)$ is the cost of the edge from x to y. And we know that $f(0, u) = 0$, $f(0, w) = \infty, \forall w \neq u$.

Notice that the recursive formula above is well ordered, since (h, w) uses only $(h - 1, \cdot)$ and eventually $h - 1 = 0$ which falls into the base case. So we could use dynamic programming to calculate all the $f(h, w)$ for all $0 \leq h \leq k, w \in V$. Calculating each $f(h, w)$ costs $\mathcal{O}(\deg w + 1)$, [2] so if we fix h and calculate all $f(h, w)$ for all w costs $\sum_{w \in V} \mathcal{O}(\deg w + 1) =$

[2]The 1 is just because even when $\deg w = 0$, we would still visit w which cost a constant time

$\mathcal{O}(E+V)$. Hence calculate all the values cost $\mathcal{O}(E+V) \cdot k = \mathcal{O}(k(E+V))$. By memorizing which w' is the optimal one for $f(h,w)$ (i.e., $w' = \arg\min f(h-1,w') + c(w',w)$), we could recover the path $p(h,w)$ for an (h,w) in $\mathcal{O}(k)$ time. So the total run-time is still $\mathcal{O}(k(E+V))$.

Finally, by definition, the solution is simply $p(v,k)$ (or $f(v,k)$ if you just need the length).

6 Problem 6

Since the graph is a DAG, it has a topological order $v_1, v_2, \cdots, v_n$ which could be calculated in $\mathcal{O}(V+E)$ time. The topological order satisfied there all the edges $(e_i, e_j) \in E$ satisfied $i < j$. Assume $u \triangleq v_k$. We use $\langle a_1, a_2, \cdots, a_m \rangle$ to be the path $v_{a_1}, v_{a_2}, \cdots, v_{a_m}$. Consider the longest path $\langle k, a_2, a_2, \cdots, a_{m-1}, j \rangle$ from u to v_j. Then we know that $\langle k, a_2, a_2, \cdots, a_{m-1} \rangle$ must be a longest path from k to $v_{a_{m-1}}$ and $(v_{a_{m-1}}, v_j) \in E$. So let $f(v_j)$ be the length of the longest path from u to v_j, then we could write the recursive formula as following, if $v_j \neq u$

$$f(v_j) = \begin{cases} \max\limits_{(v_{j'}, v_j) \in E} f(v_{j'}) + c(v_{j'}, v_j) & \text{if exists } v_{j'} \text{ satisfied } (v_{j'}, v_j) \in E \\ \infty & \text{otherwise} \end{cases}$$

Where $c(x,y)$ is the cost of the edge from x to y, and $f(u) = 0$. Now, calculate $f(v_j)$ in topological order, then when calculating $f(v_j)$, all the $f(v_{j'})$ with $(v_{j'}, v_j) \in E$ is calculated since $j' < j$. Hence the recursive formula is well ordered. Calculating one node cost $\mathcal{O}(\deg v_j + 1)$, and hence the total cost is $\sum \mathcal{O}(\deg v_j + 1) = \mathcal{O}(V+E)$.

The length of the longest path from u to v is then $f(v)$. Similar to Problem 5., we could memorize $\arg\max\limits_{v_{j'}} f(v_{j'}) + c(v_{j'}, v_j)$ for each vertex v_j. Then backtrace from $v \triangleq v_{k'}$, we could recover the longest path from u to v in no more than $\mathcal{O}(V)$ time. So the total run-time is $\mathcal{O}(V+E)$.