Homework #5
Algorithms I
600.463
Spring 2017

Due on: Saturday, March 18th, 11:59pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: On Gradescope, under HW5
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

April 11, 2017

Problem 1 (20 point)

Let $G = G(V, E)$ be a directed graph represented by an adjacency list. G is a bipartite graph if it is possible to partition the vertices of G into two disjoint sets, i.e. $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$ such that there are no edges between vertices in the V_1 and there are no edges between vertices in the V_2. Design an efficient algorithm that works in $O(|E| + |V|)$ time and checks if G is bipartite. Prove the correctness of your algorithm and analyze the running time.

Answer:
First, if we want to determine whether a graph G is a bipartite graph or not, we may try to make use of a classic result for bipartite graph.

Lemma 0.1. *Every bipartite graph has chromatic number ≤ 2 (a.k.a 2-colorable). Any 2-colorable graph is bipartite.*

Proof. By definition, the chromatic number of a graph G is the smallest number of colors needed to color the vertices of G, so that there is no two adjacent vertices share the same color. From the definition of bipartite graph, V can be divided into

two disjoint sets V_1 and V_2 and there are no edges between the vertices in V_1 and between the vertices in V_2. So if there is no edge (a, b) where $a \in V_1$ and $b \in V_2$, we can just use one color to color all the vertices. If there is at least one such edge as (a, b), two colors are needs. This is because we need one color for V_1 and the other color for V_2 to avoid that two adjacent vertices share the same color.
If a graph is colored by only one color, vertex set can be divided into any two subsets that meet the definition. If a graph is colored by two colors, divide the vertices into two subsets based on color. By the coloring method mentioned above, any pair of vertices in the subsets are not adjacent.

One observation is that we can convert the directed graph G to undirected graph G^* and determine if G^* is bipartite. Let's provide a BFS coloring algorithm to decide whether G^* is bipartite.

1. Choose any uncolored vertex s and color s RED.
2. Color the neighbors of s BLACK. (use adjacency list)
3. Color the neighbors of all neighbor's RED. (use adjacency list)
4. If all vertices are colored, return TRUE, else repeat the steps starting from 1. While all the coloring steps, if a neighbor of the current vertex is already colored the same color as the current vertex, return FALSE. In this way, a proper 2-coloring is failed and graph G is not bipartite.
5. return TRUE.

Running Time:
The procedures take the same number of vertex search as BFS, which takes $O(|V^*|+|E^*|)$ time by using adjacency list. Since $|E^*| \leq |E|$, $O(|V^*| + |E^*|) = O(|V| + |E|)$.
Correctness:
By running the algorithm, we can use at most two colors to color all the vertices. If the coloring succeed finally without returning FALSE, by lemma 0.1, the graph is bipartite.

Problem 2 (20 points)

Problem 2.1 (10 points)

Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n $INCREMENT$ and $RESET$ operations takes time $O(n)$ on an initially zero counter. (Hint: Keep a pointer to the high-order 1.)

Answer:

We introduce a new field $max[A]$ to hold the index of the high-order 1 in A. Initially, $max[A]$ is set to 1, since the low-order bit of A is at index 0, and there are initially no 1's in A. The value of $max[A]$ is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of A must be looked at to reset it. By controlling the cost of $RESET$ in this way, we can limit it to an amount that can be covered by credit from earlier $INCREMENTs$.

```
$INCREMENT(A)$
$i \leftarrow 0$
**while** $i < length[A]$ *and* $A[i] = 1$ **do**
    $A[i] \leftarrow 0$
    $i \leftarrow i + 1$
**end**
**if** $i < length[A]$ **then**
    $A[i] \leftarrow 1$
    **if** $i > max[A]$ **then**
        $max[A] \leftarrow i$
    **end**
**end**
**else**
    $max[A] \leftarrow -1$
**end**
```

```
$RESET(A)$
**for** $i \leftarrow 0$ *to* $max[A]$ **do**
    $A[i] \leftarrow 0$
**end**
$max[A] \leftarrow -1$
```

As for the counter in the book, we assume that it costs \$1 to flip a bit. In addition, we assume it costs \$1 to update $max[A]$.

Setting and resetting of bits by $INCREMENT$ will work exactly as for the original counter in the book: \$1 will pay to set one bit to 1; \$1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing.

In addition, we will use \$1 to pay to update max, and if max increases, we will place an additional \$1 of credit on the new high-order 1. (If max doesn't increase, we can just waste that \$1— it won't be needed.) Since $RESET$ manipulates bits at positions only up to $max[A]$, and since each bit up to there must have become the high-order 1 at some time before the high-order 1 got up to $max[A]$, every bit seen be $RESET$ has \$1 credit on it. So the zeroing of bits of A by $RESET$ can be completely paid for by the credit stored on the bits. We just need \$1 to pay for resetting max.

Thus, charging \$4 for each $INCREMENT$ and \$1 for each $RESET$ is sufficient, so the sequence of n $INCREMENT$ and $RESET$ operations takes $O(n)$ time.

Problem 2.2 (10 points)

Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

$INSERT(S, x)$ inserts x into S.

$DELETE\text{-}LARGER\text{-}HALF(S)$ deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m $INSERT$ and $DELETE\text{-}LARGER\text{-}HALF$ operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

Answer:

We can use a simple array A to be our underlying data structure. Array A contains a set S of integers. The operations are as follows.

- $INSERT(S, x)$: The element can be inserted to the last element in A.
- $DELETE\text{-}LARGER\text{-}HALF(S)$: First, find the median mid of S using array A. We then partition S into two subsets, S_1 and S_2, where all the elements in S_1 are less than mid and the elements in S_2 are larger than and equal to mid by comparing each element in A with mid. Last, we delete S_2 and set $S = S_1$.

Suppose there are m operations performed on S. For each $i = 1, 2, \ldots, n$, let c_i be the actual cost of the ith operation and A_i be the data structure that results after applying the ith operation to data structure A_{i-1}. Since we use the linear time algorithm for finding the median, the running time of the algorithm for discarding the larger half is $O(m)$ for an m-element array. Suppose the actual time for discarding the larger half is km for some constant k. We define the *potential function* Φ on each data structure A_i to be

$$\Phi(A_i) = 2k \times |A_i|,$$

where $|A_i|$ is the number of elements in array A_i. For an operation i, we analyze the cost by considering the following two cases.

$INSERT(S,x)$: The actual cost c_i for insertion is 1. Then, the amortized cost

$$\widehat{c_i} = c_i + \Phi(A_i) - \Phi(A_{i-1}) = 1 + 2k \times |A_i| - 2k \times |A_{i-1}| = 1 + 2k,$$

since $|A_i| = |A_{i-1}| + 1$ when inserting an element to A_{i-1}. The amortized cost therefore is $O(1)$.

$DELETE\text{-}LARGER\text{-}HALF(S)$: As mentioned above, the actual cost for deleting the larger half is km if there are m elements. Then, the amortized cost

$$\begin{aligned}
\widehat{c_i} &= c_i + \Phi(A_i) - \Phi(A_{i-1}) \\
&= k \times |A_{i-1}| + 2k \times |A_i| - 2k \times |A_{i-1}| \\
&= k \times |A_{i-1}| + 2k \times \frac{1}{2}|A_{i-1}| - 2k \times |A_{i-1}| \\
&= k \times |A_{i-1}| + k \times |A_{i-1}| - 2k \times |A_{i-1}| = 0
\end{aligned}$$

since $|A_i| = \frac{1}{2}|A_{i-1}|$ when deleting the larger half of A_{i-1}. The amortized cost therefore is 0.

Therefore, the m operations run in $O(m)$ time.