Solutions for Homework Assignment #3/4

Answer to Question 1.

a. This question is essentially using disjoint sets ADT. The most efficient way is to implement a disjoint forest with union-by-rank and path compression.

However, your auxiliary (dictionary) data structure storing the individual clones can affect your running time analysis. To get full marks, you are required to include these details. One possible auxiliary data structure would be to store all the clone ids in an AVL tree.

- NEW(I): First do a Search to ensure I is not already in the tree. Do a Make-Set(I) then do an AVL-Insert to put I (and a reference to the disjoint set node for clone I) in the tree.
- OVERLAP(A, B): Do Search on the AVL-tree to find the references for A and B to the appropriate nodes in our disjoint set (if either search fails, return immediately). Do a Union on these two nodes (which consists of two Finds and a Link).

b. The most number of unique OVERLAP data items, given we have n NEW data items, is $n(n-1)/2$. However, we can have at most $n-1$ Union operations. (If two clones are already in a contig, performing Union does not have an effect.) Based on CLRS chapter 21, after processing n NEWs and m OVERLAPs, the worst case sequence has a running time in $O(m\alpha(n)+n)$ (CLRS) or $O(m\log^* n+n)$ (lecture). Hence, the amortized cost is $O(\alpha(n))$ (or $O(\log^* n)$).

However, your auxiliary (dictionary) data structure storing the individual clones can affect your running time analysis. For full marks, you are required to provide another analysis for your auxiliary data structure.

Suppose we store all the clones in one AVL tree. Upon seeing a NEW data item, at most one AVL-Insert is performed. Upon seeing an OVERLAP(A, B) data item, two Searches are performed. For an AVL-tree with k nodes, one Insert takes $O(\log k)$. Similarly, one Search takes $O(\log k)$.

In this case, the worst case sequence is that all the n NEW data items appear before any of the n OVERLAP data items. In this case, the time for the n Inserts is $O(\sum_{i=1}^{n}\log i) = O(\sum_{i=1}^{n}\log n) = O(n\log n)$ and the time for $2m$ Searches is in $O(m\log n)$.

Hence, the AVL-tree amortized cost for either NEW or OVERLAP takes $O(\log n)$.

Combining both running time analyses, to process either NEW or OVERLAP, the amortized cost is $O(\alpha(n)+\log n) = O(\log n)$.

Note that you can achieve this run time, even if you did not choose the best disjoint set ADT, since the amortized running time processing NEW and OVERLAP is affected by the Insert and Search operation on your auxiliary data structure. That is, processing one NEW data item = 1 Make-Set + 1 Insert; processing one OVERLAP data item = 1 Union = 1 Link + 2 Find-Sets + 2 Searches. As long as you choose an efficient auxiliary dictionary data structure (e.g., logarithmic in the number of clones) for Insert and Search, and the amortized cost for Link and Find-Set does not exceed the amortized cost for Insert and Search, the combined ADT is efficient.

Answer to Question 2.

a. Let I_n denote the set of bit positions in the binary representation of n that contain the value 1. Then the worst case time to perform SEARCH is in $\Theta(\sum_{i\in I_n} i)$.

In the worst case, binary search may have to be performed on each array in the linked list. If an array has size 2^i, then binary search on that array takes $O(i)$ time (for $i>0$). There is an array in the linked list of size 2^i if and only if $i \in I_n$. Thus, the worst case time complexity for SEARCH is in $O(\sum_{i\in I_n} i)$.

When the element being sought is not in the list, binary search will be performed on each array in the list and will take time proportional to the logarithm of the size of the array. Thus, the worst case time complexity for SEARCH is in $\Theta(\sum_{i\in I_n} i)$.

In the case that $n = 2^k - 1$ for some integer $k \geq 1$, $I_n = \{0, \ldots, k-1\}$ and the total time taken is in $\Theta(\sum_{i=0}^{k-1} i) = \Theta(k^2) = \Theta((\log n)^2)$.

b. Let v_n denote the largest power of 2 that divides $n+1$. Then the linked list contains arrays of size $1, 2, 4, 8, \ldots, 2^{v_n-1}$, but no array of size 2^{v_n}.

When INSERT is performed on a set of size n, the while loop is performed $v_n - 1$ times. The k'th time through the loop, we merge an array of size 2^k containing the newly created inserted element with an existing array of the same size. This takes $\Theta(2^k)$ time. Thus the total time is in $\Theta(\sum_{i=0}^{v_n-1} 2^i) = \Theta(2^{v_n})$.

In particular, when $n = 2^k - 1$, for some integer $k \geq 1$, then $v_n = k$, so the total time taken is in $\Omega(2^k) = \Omega(n)$.

c. Let us charge 1 credit for each write of an element into an array. The credit invariant is that the number of credits in each array in the list is at least the number of elements in all smaller arrays in the list. For example, if there is an array of size 1, an array of size 2, and an array of size 8, the array of size 2 contains at least 1 credit and the array of size 8 contains at least 3 credits.

Initially, the credit invariant is vacuously true, since there are no nonempty arrays.

Let the amortized cost of each insertion be $2 + \lfloor \log_2 n \rfloor$. The cost to create a new array of size 1 containing x is 1. Allocate 1 credit to each of the other nonempty arrays. Since there are at most $1 + \lfloor \log_2 n \rfloor$ existing arrays, there are sufficient credits to do this.

If the array containing x is merged with an array A of length 2^i, then, before the insertion, there were $2^i - 1$ elements in smaller arrays. This implies that A contains 2^i credits, one for each of these elements in smaller arrays, plus one from the insertion of x. These credits can be used to pay for the merge of A with the 2^i element array containing x. There are no elements in any smaller arrays after this merge, so the invariant is vacuously true for the resulting array.

Each array that is not involved in a merge during this insertion also satisfies the invariant, since it was given one additional credit and only one element was added to the set of elements in smaller arrays.

Thus the credit invariant is true after each INSERT operation. It follows that the amortized insertion time in a sequence of n insertions, starting with an initially empty set, is $O(\log n)$.

Answer to Question 3.

a. To represent an undirected multigraph using adjacency lists, we simply allow a neighbouring vertex to appear multiple times within an adjacency list. The total length of all adjacency lists will be $2m$, where m is the number of edges in the graph.

Alternatively, each entry in an adjacency list could store the adjacent vertex and the multiplicity of these edges (when the multiplicity is at least 1). Thus, to insert an edge (u, v) would require traversing the adjacency list for vertex u looking for vertex v: if v was found, the associated multiplicity would be incremented by one; otherwise, a new entry would be added to the adjacency list containing vertex v and a multiplicity of 1 (this process would then be repeated for v's adjacency list). The total length of all adjacency lists would be no more than $n^2 - n$.

To represent an undirected multigraph using an adjacency matrix, we will store at $A[i, j]$ the multiplicity of the edges between vertex i and vertex j (recall that for a graph we stored either 1 or 0 depending on whether the edge (i, j) was present or not, for multigraphs we store the integer number of edges (i, j) in the graph). The size of the matrix (hence the memory requirement) will be n^2 as before.

b. One way to represent a weighted undirected multigraph using adjacency lists would be to use the first representation we mentioned in part (a). Each adjacency list would contain one entry for each incident edge. Each entry would contain the adjacent vertex and the weight of the edge. Searching for an edge (u, v) of weight x would mean traversing the list for vertex u looking for v, and each time vertex v was found, checking whether this edge had weight x. Since there are m edges in the multigraph (and, in worst case, all of them may have v as an endpoint), the length of this list may be at most m, so the worst-case complexity of this operation would be in $\Theta(m)$.

Using the second representation mention in part (a), to the entry containing the multiplicity of an edge (u, v) we would need to add a list of weights of these edges. If this list is unordered (thus permitting an efficient addEdge operation), in worst case the entire list may be searched looking for an edge (u, v) of weight x. If all edges in the graph are of the form (u, v), the complexity of this search would be in $\Theta(m)$. This could be improved to $O(n + \log m)$ using a sorted list of weights, but this would be at the expense of less efficient insert and delete operations. Many other solutions also exist with various complexities.

Answer to Question 4.

a. Let $v_1, v_2, \ldots, v_k$ be a cycle in a bipartite graph. Consecutive vertices on the cycle must be in different parts of the bipartition, so that all v_i with odd i are in one set, and those with even i are in the other. Since (v_1, v_k) is an edge, v_k is not in the same part as v_1 and therefore k must be even. Thus bipartite graphs do not contain odd cycles.

On the other hand suppose that G contains no odd cycles. We will show that every component of G (and therefore G itself) is bipartite. Let s be any vertex of G, let $X = \{u \in V(G) : d(u,s) \text{ is even}\}$, and let $Y = \{u \in V(G) : d(u,s) \text{ is odd}\}$. $X \cup Y$ contains all the vertices in the component containing s. We will see that X and Y form a bipartition of this component. Suppose that (u, v) is an edge between vertices in, say, X. Let P_u and P_v be shortest paths from s to u and v respectively. Consider the last vertex on P_v that is also on P_u and call it w. The cycle formed by going from u to w along P_u, then to v along P_v and back to u via (u, v) has length

$$d(u,s) - d(w,s) + d(s,v) - d(s,w) + 1 = 1 + d(s,u) + d(s,v) - 2d(s,w).$$

This length would be odd if u and v were in the same part, but that can't happen since there are no odd cycles.

b. We implement the idea of the proof above as follows. We modify the DFS algorithm given in lecture to assign each vertex v to either V_1 or V_2 by setting $part[v]$ to 1 or 2.

When we initialize $p[v]$, we'll initialize $part[v] = 1$. Whenever we set $p[v] = u$, we'll set $part[v] = 3 - part[u]$ (if there is an edge between u and v, then they should be in different partitions). Finally, within the for loop exploring incident edges, we can check if $colour[v] == gray$ (v is discovered) AND $part[v] == part[u]$. If that check is true, then the input graph is not bipartite: this back edge completes an odd cycle. We will print u, $p[u]$, $p[p[u]]$, and so on until we reach v.

The running time of DFS is $\Theta(n+m)$. We added only a constant amount of work each measured step (setting one more variable whenever $p[v]$ is set). If the graph is bipartite, we simply return the computed *part* array. If the graph is not bipartite, we do an extra $O(n)$ work to return the odd cycle. Therefore the algorithm runs in total time $\Theta(n+m)$.