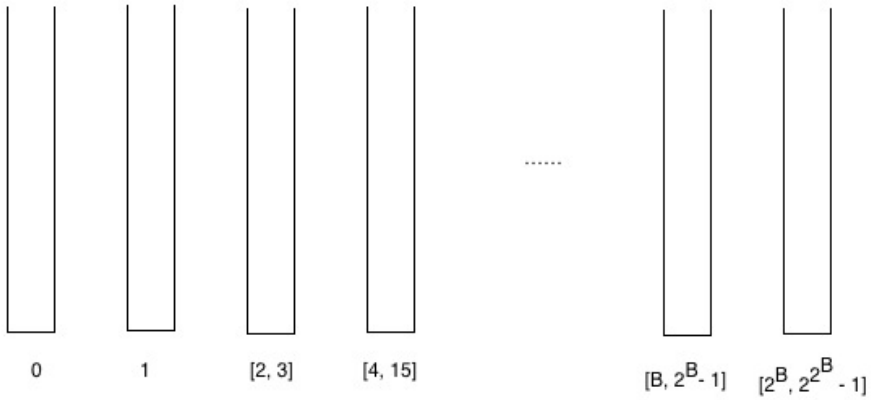
Analysis of Algorithms – HW3

Li-Hen Chen

Department of Computer Science & Engineering, Texas A&M University

1. If m operations, either Union or Find, are applied to n elements, the total run time is $\boldsymbol{O(m \log^* n)}$, $where \log^*$ is the iterated logarithm.

   A bucket is a set that contains vertices with particular ranks as the following figure.



   We can make two observations about the buckets.

   1. The total number of buckets is at most log*n

   2. The maximum number of elements in bucket $[B, 2B - 1]$ is at most $2n/2B$

   Let F represent the list of "find" operations performed, and

   The total cost of m find is $T = T_1 + T_2 + T_3$

   $T_1 = \sum(link\ to\ the\ root)$

   $T_2 = \sum(number\ of\ links\ traversed\ where\ the\ buckets\ are\ different)$

   $T_3 = \sum_F(number\ of\ links\ traversed\ where\ the\ buckets\ are\ the\ same)$

   Since each find operation makes exactly one traversal that leads to a root, we have $T_1 = O(m)$.

   Also, from the bound above on the number of buckets, we have

   $T_2 = O(mlog^*n)$.

   For T3, suppose we are traversing an edge from u to v, where u and v have rank in the bucket [B, 2B − 1] and v is not the root (at the time of this traversing, otherwise the traversal would be accounted for in T1). Fix u and consider the sequence v1,v2,...,vk that take the role of v in different find operations. Because of path compression and not accounting for the edge to a root, this sequence contains only different nodes and because we know that the ranks of the nodes in this sequence are strictly increasing. By both of the nodes being in the bucket we

can conclude that the length k of the sequence (the number of times node u is attached to a different root in the same bucket) is at most the number of ranks in the buckets B.

Therefore, $T_3 \leq \sum_{[B,2^B-1]} \sum_u 2^B$

From Observations 1 and 2, we can conclude that $T_3 \leq \sum_B 2^B \frac{2n}{2^B} \leq 2nlog^*n$

Therefore, $\boldsymbol{T_1 + T_2 + T_3 = O(mlog^*n)}$

2. Initially, we will make each vertex have a D value of ∞. Then we relax along each edge V-1 times. Then, we do a final relaxation. If any value changes, indicated the existence of a negative cycle. At the end, $v.d$ is what we are solving, the value δ for every vertices v. This algorithm is modified from Bell-Ford Algorithm. Apparently, through the statement that we relax each edges (E) V-1 times and according to the Pseudo code below we can conclude that this algorithm runs O(nm)-time.

Pseudo code

```
for each v ∈ G.V
    v.d = Infinity

for i = 1 to |G.V| − 1
   for each edge(u, v) ∈ G.E
      RELAX(u, v, w)

RELAX(u, v, w)
  min = w(u, v) + ((u.d < 0) ? u.d : 0)
  if v.d > min
    v.d = min
```

3. We have a graph G, which is given in an adjacency list G[n] and allowed multiple edges in a weighted edge. Now we have to design a linear-time algorithm for only keeping the maximum edges in the G, and remove the other multiple edges.

HW2 Li-Hen Chen 928003907
Department of Computer Science & Engineering, Texas A&M University

Algorithm Design:

First of all, initialize an adjacency matrix represent the graph G as a sparse array. Run through the edges of each vertex. If matrix[v, u] is empty, then set matrix[v, u] = adjacency list[u, u]. If matrix[v, u] is not empty, then compare the value of adjacency list[u, u] to the matrix[v, u] and set the matrix[v, u] with the larger value. Finally we obtain the graph without multiple edges between every 2 vertices.

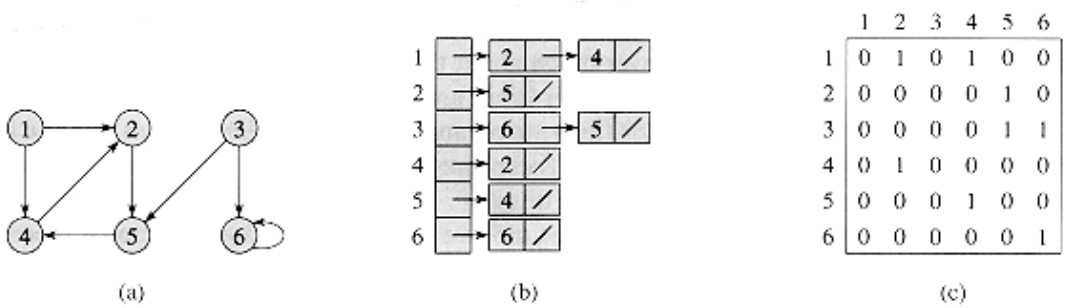


Fig. Adjacency List vs Adjacency Matrix

Pseudo code:

Initialize an adjacency matrix

```
for each V:
    for each E:
        if matrix[v,u]==null:
            matrix[v,u] = edge[v,u]
        else if matrix[v,u] <edge[v,u]:
            matrix[v,u] = edge[v,u]
```

Time Complexity:

At this case, no matter the graph is directed or undirected and no matter the graph has negative edges or not, we are able to use the above algorithm to solve the question. Although, we inspect the above code, we may conclude that the algorithm runs $O(EV)$. In fact, it takes $O(|V| + |E|)$ because we are running on adjacency list the whole process can be told from the above fig which is V+E. To sum up, this algorithm for removing multiple edges in graph takes $\boldsymbol{O(|V| + |E|)}$ time, which is linear.

4. First, Let's talk about the most basic quicksort, which chooses pivot randomly. The following is the Pseudo code.

```
quicksort(arr[], lo, hi)
    if lo < hi
        p = partition_r(arr, lo, hi)    // here partition with the random pivot and sort
                                        // left and right part separately.
        quicksort(arr, p-1, hi)
        quicksort(arr, p+1, hi)
```

The worst case time complexity of a typical implementation of QuickSort is $O(n^2)$. The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.

The following is the pseudo code of the modified Quicksort using median as pivot.

```
quicksort_median (int arr[], int l, int h)
    if (l < h)
        int n = h-l+1; // Find size of current subarray
        step 1. int med = kthSmallest(arr, l, h, n/2); // Find median of arr[].
        // Partition the array around median
        int p = partition(arr, l, h, med);
        step 2. quickSort(arr, l, p - 1);
        step 3. quickSort(arr, p + 1, h);
```

We can analyze the quicksort_median algorithm to prove that the worst case is $O(\log n)$.

Lemma: Median Finding algorithm is linear time.

At the step 1 we can find the median of the array in linear time. So we find the median first, then partition the array around the median element. This prevent the worst case of quick sort that picked pivot is always an extreme and make sure the

picked pivot is always median. So we are guaranteed to have the following recurrence relation.

$$\mathrm{T(n)} = O(n) + 2T(\frac{n}{2})$$

The master theorem for divide-and-conquer recurrences tells us that$\mathrm{T(n)} = \mathrm{O(n \log} n)$.

Although worst case time complexity of the above approach is $\mathrm{O(n \log} n)$, it is never used in practical implementations. The hidden constants in this approach are high compared to normal Quicksort due to the fact that in order to find median we have to operate $\mathrm{O(n)}$ comparison each partition. That's the reason why in practice it would be slower than normal Quicksort algorithm.