Lydia Feng

CS323

Writ #3: Mostly Graphs

1. Bloom
    a. m = 8n; k = 6; p = 0.02158

        m = 16n; k = 11; p = 4.6e-4

        m = 24n; k = 17; p = 9.8e-6

    b. p = 0.06222 <= 2^(-4); 5.78 bits per item; k = 4

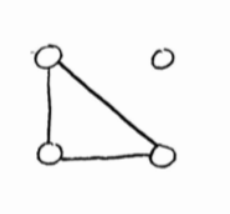
        p = 0.00389 <= 2^(-8); 11.55 bits per item; k = 8

        p = 2.4324e-4 <= 2^(-12); 17.32 bits per item; k = 12

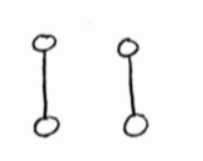
2. TopoSort2
    a. If the final G' is empty, then the output vertex ordering is a topological ordering of G. Consider arbitrary vertices ***v*** and ***u***, if there is a directed edge from ***v*** to ***u***, then ***v*** comes before ***u*** in topological ordering, and ***v*** is deleted before ***u*** in TopoSort2. This is because every pass through the while loop deletes the vertex ***v*** that does not have an edge pointing to it. The while loop also deletes all the edges out of that vertex, so on the next pass through the while loop, the next vertex without an edge pointing to it ***u***, is deleted. This means that ***v*** pointed to ***u***, and they are outputted in that order. If the final G' is empty, then the while loop eventually deletes every single vertex in the order that they are pointed to by the preceding vertex. This is topological ordering.
    b. If the final G' is not empty, then G has a cycle. This is because if G' is not empty, then there is some vertex ***v*** in G that is never deleted by the while loop. This means that vertex ***v*** has an edge pointing to it that cannot be deleted by the while loop. The only way that the edge pointing to ***v*** cannot be deleted is if the edge originates from an arbitrary vertex ***u*** that comes "after" ***v***. In other words, the vertex ***u*** points to its ancestor, vertex ***v***. The edge ***uv*** is a back edge and creates a cycle. In this way, the edge ***uv*** never gets deleted, and the final G' is not empty.
    c. If G is given in adjacency list form, TopoSort2 can be implemented in O(V+E) time. This is because each vertex is checked only one time. This is where the V term comes from. From a vertex, all of its adjacent vertices are checked. The sum of adjacent vertices across all vertices is equal to the number of edges in the digraph. This is where the E term comes from. Adding those together, TopoSort2 is implemented in O(V+E) time

3. Formulas

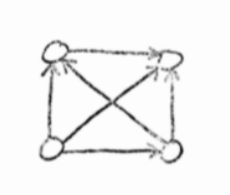
   a. Max # of edges = $\binom{v-1}{2} = \frac{(v-1)!}{2!(v-3)!}$



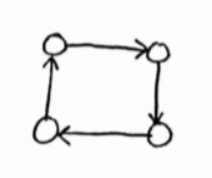
   b. Min # of edges = $\lceil \frac{v}{2} \rceil$



   c. Max # of edges = (v-1) + (v-2) + (v-3) + … + 1 + 0 = $\sum_{i=1}^{v}(v-i)$

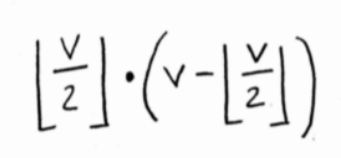


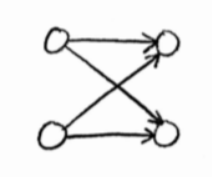
   d. Min # of edges = v



   e. Max # of edges = $\lfloor \frac{v}{2} \rfloor \times (v - \lfloor \frac{v}{2} \rfloor)$

   This formula was difficult to type so here is a hand-written version:





4. DFS Edge Types

```
public String edgeType(int v, int w) {

    if ((beg[w] > beg[v]) && (edgeTo[w] != v) && (end[v] > end[w])) {
        return "forward";
    }

    if (edgeTo[w] == v) {
        return "tree";
    }

    if ((beg[v] > beg[w]) && (end[w] > end[v])) {
        return "back";
    }

    if ((beg[v] > beg[w]) && (end[w] < end[v])) {
        return "cross";
    }

    else {
        return "no edge type identified";
    }

}
```

5. Red-Black Spanning Trees:

   Consider a connected graph G with V vertices and E edges. Each edge of G is either red or black. We can find a spanning tree T with k red edges where 1 <= k <= V.

   a. Assign each edge one of two weights. For example, if the edge is red, have its weight be 1, and if the edge is black, have its weight be much larger, E+1. Create a minimum spanning tree using Prim's algorithm (eager) and a binary heap, and call this tree T1. Keep track of the number of black edges and the number of red edges in T1 as they are added to the MST.

   b. If the number of red edges = k:

      return T1 as T.

   c. If the number of red edges < k:

      No such tree T exists. Prim's algorithm prioritizes lightly weighted paths, and since one black edge is weighted more than all the red edges combined, it should never choose a black edge unless there is no other way to create the MST. Therefore, T1 has all the red edges possible, and if this number is still less than k, then a tree with k red edges does not exist.

   d. If the number of red edges > k:

      Find a black edge ***e*** that is not in T1 and add it to T1. This creates a cycle. Find a red edge ***f*** that is in the cycle in T1 and remove it. Now T1 has one less red edge. Continue this process of adding black edges and removing red edges until the number of red edges in T1 = k. Return T1 as tree T. If there are not enough black edges and the number of red edges in T1 is still greater than k, then the tree T does not exist.

e. This algorithm finds T (or announces that T does not exist) in O(ElogV + 2(E -k)) time. This is because Prim's algorithm with a binary heap uses O(ElogV) time to create the initial MST. Then, if red edges need to be removed, it will take O(1) time, and to replace it with a black edge will take O(1) time. This replacement will need to be done at most E-k times, so the total time for this algorithm is O(ElogV + 2(E-k)).