ICCS208: Assignment 8

Theeradon Sarawek
theeradon.sar@student.mahidol.edu
6 December 2024

1: Breadth-First Search Running Time

I will annotate this BFS code to determine its runtime:

```
// total for nbrsExcluding - see below
Set<Integer> nbrsExcluding(UndirectedGraph<Integer> G, Set<Integer>
    vtxes, Set<Integer> excl) {
     Set<Integer> union = new TreeSet<>(); // not HashMap
     for (Integer src : vtxes) { // V
          for (Integer dst : G.adj(src)) // d (average degree of a
              vertex)
               if (!excl.contains(dst)) { union.add(dst); } // O(logS)
                  and O(logE) for exclusion set
     }
     return union;
}

Set<Integer> bfs(UndirectedGraph<Integer> G, int s) {
     // let V -> visited
     // let F -> frontier
     Set<Integer> frontier = new TreeSet<>(Arrays.asList(s)); //
        O(logF)
     Set<Integer> visited = new TreeSet<>(Arrays.asList(s)); //
        O(logV)
     while (!frontier.isEmpty()) {
          frontier = nbrsExcluding(G, frontier, visited); //
             O(F*d*logV)
          visited.addAll(frontier); // the i-th position is what's
             reached at i hops ---- O(FlogV)
     }
     return visited; // O(1)
}
```

Here are some additional notes for context (based on my own understanding of things):

1. I'd assume that the size of the graph G should be the same as the number of vertices. Thus, if the adjacency table stores non-duplicate edges meaning that say, the list of adjacent vertexes to vertex A is vertex B, but vertex A is not in the list of adjacent vertexes for vertex B.
2. If all vertices in the tree are connected, then $m \geq n$ in theory.
3. We use the textbook presumption that n = number of vertices and m = number of edges

For *nbrsExcluding*, here are some considerations for the total runtime:

- V is the size of the set vtxes
- d is the average size of the degree of each vertex
- *contains* and *add* both take logarithmic time, but I'd want to assume that *excl* would be larger than *union* so the dominant time is $O(logE)$

Therefore, the total runtime for nbrsExcluding should be $O(V \cdot d \cdot logE)$

As for bfs:

- Initializing the sets would in theory take logarithmic time, where since s is only one element, we won't need to do any additional multiplication
- One iteration of the while loop takes $O(FdlogV + FlogV)$ which is $O(d + 1 \cdot FlogV)$
- In BFS, every edge is considered only once, so d would in theory approach m.
- Likewise, we must go through every vertex, so V should be of size n by the end.
- Each vertex enters F once. This means that F sums up to n in the end.

Consider the sum of iterations:

$$\sum d + 1 \cdot F \cdot logV \tag{1}$$

In theory, the most dominant variant of this would be in the very end of the while-loop, where as said in the previous two bullet points, we'd have the worst case sizes of d, V, and thus F. I do this since the rest of the iteration is irrelevant when it comes to finding the dominant term that ends up being the runtime.

$$(nm + n) \cdot logn \tag{2}$$
$$= (m + n) \cdot logn \tag{3}$$

Thus, we can conclude that the runtime of this TreeMap / TreeSet based BFS is $O((m+n) \cdot logn)$ which is slower than the textbooks' HashSet implementation of BFS but it does make sense since TreeSet operations take $O(logn)$ per operation as opposed to HashSets that, on average, take $O(1)$ per operation.

2: Random Permutations

Consider the following code:

```
int minSoFar = Integer.MAX_VALUE;
int numUpdate = 0;
for (int i=1;i<=n;i++) {
    if (p(i) < minSoFar) {
        minSoFar = p(i);
        numUpdate++;
    }
}
```

Given that $p : [n] \to [n]$, one thing to note is that the probability of $p(i)$ being the new minimum depends on all $p(i-1)...p(2), p(1)$ before it. If p is truly random, then every permutation has an equal chance of being achieved. Because of this, we'd want to assume that the chances of us getting a new minimum each iteration would be $\frac{1}{i}$

We can give an iterator random variable R_i to *numUpdate* such that:

$$R_i \to \begin{cases} 1 & \text{p(i) is less than minSoFar} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Therefore, per the expected value formula,

$$\text{E[numUpdate]} = \sum_{i=1}^{n} R_i \cdot P(i) \tag{5}$$

Since numUpdate only updates if we find a new minimum (since if we don't find the minimum, $R_i = 0$), and knowing that the chances of finding said new minimum is $\frac{1}{i}$, we actually find that the formula ends up looking like:

$$\text{E[numUpdate]} = \sum_{i=1}^{n} \frac{1}{i} \tag{6}$$

This looks a lot like the Harmonic series. Thus, finding the lower and upper bounds of this would look something like (using $x = i$ for simplicity), and also taking stuff from Wikipedia's page on the harmonic series:

$$\text{Lower bound} \tag{7}$$

$$\int_1^{n+1} \frac{1}{x} dx \tag{8}$$

$$= [ln(n+1) - ln(1)] \tag{9}$$

$$= ln(n+1) \tag{10}$$

$$\text{Upper bound} \tag{11}$$

$$1 + \int_1^{n} \frac{1}{x} dx \tag{12}$$

$$= 1 + [ln(n) - ln(1)] = 1 + ln(n) \tag{13}$$

Therefore,

$$ln(n+1) \leq \text{E[numUpdate]} \leq 1 + ln(n) \tag{14}$$

3: HackerRank Problems

My username on HackerRank is @theeradon.sar