| **CSEP521: Algorithms** | October 14, 2022 |
|---|---|
| Homework 1 | |
| *Anup Rao* | Due: October 23, 2022 |

Each problem is worth 10 points:

1. Arrange the following in increasing order of asymptotic growth rate:

   (a) $f_1(n) = 100n^2$
   (b) $f_2(n) = n^3$
   (c) $f_3(n) = 2^{\sqrt{n}}$
   (d) $f_4(n) = n(\log n)^{1000}$
   (e) $f_5(n) = 2^{n \log n}$
   (f) $f_6(n) = 2^{(\log n)^{0.9}}$

   **Solution:** In increasing asymptotic order, the functions are:

   $$2^{(\log n)^{0.9}}, n(\log n)^{1000}, 100n^2, n^3, 2^{\sqrt{n}}, 2^{n \log n}.$$

   This can be shown by comparing the logarithm of the functions, which can be arranged in increasing order as follows:

   $$\log^{0.9} n, \log n + 1000 \log\log n, 2\log n + \log 100, 3 \log n, \sqrt{n}, n \log n.$$

2. A walk of length $k$ in a graph is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $v_i$ is a neighbor of $v_{i+1}$ for $i = 0, 1, 2, \ldots, k-1$. Suppose the product of two $n \times n$ matrices can be computed in time $O(n^\omega)$ for a constant $\omega \geq 2$. Give an algorithm that counts the number of walks of length $k$ in a graph with $n$ vertices in time $O(n^\omega \log k)$. HINT: If $A$ is the adjacency matrix, prove that the $(i,j)'th$ entry of $A^k$ is exactly the number of walks of length $k$ that start at $i$ and end at $j$. Repeatedly square the adjacency matrix to compute $A^k$.

   **Solution.** The algorithm is given below.

   (a) **Proof of correctness:** The key claim is that the $i,j$-th entry of $A^k$ counts the number of $k$-length walks from $i$ to $j$, for all $i, j$. Thus summing over the matrix gives us the total number of $k$-length walks.

   The proof of the claim is via induction. The base case, $k = 1$, follows from the fact that a one length walk between any two vertices corresponds to a (directed) edge. Since the $ij$-th entry of $A$ computes the number of edges from $i$ to $j$, it also computes the number of one length walks between from $i$ to $j$. Now we proceed to the inductive case, assuming the claim holds for some $k \geq 1$. To this end, we will prove that

   $$\#(k+1 \text{ length walks from } i \text{ to } j) = \sum_{j' : A_{j'j} = 1} \#(k \text{ length walks from } i \text{ to } j'). \qquad (1)$$

```
   Input: Adjacency matrix A and a natural number k
   Result: A^k
 1 Result ← I;
 2 while k ≠ 0 do
 3 |   if k mod 2 = 0 then
 4 |   |   k ← k/2;
 5 |   |   A = A * A
 6 |   end
 7 |   else
 8 |   |   k ← k − 1;
 9 |   |   Result ← A * Result;
10 |   end
11 end
12 count ← 0 for i from 1 to n do
13 |   for j from 1 to n do
14 |   |   count ← count + A_{i,j}
15 |   end
16 end
17 return count;
```

Assuming Equation (**??**), we are done because by the inductive hypothesis, the number of $k$ length walks from $i$ to $j'$ is given by $A^k_{ij'}$. Therefore, the number of $k+1$ length walks from $i$ to $j$ would then be $\sum_{j':A_{j'j}=1} A^k_{ij'} = \sum_{j'} A^k_{ij'} A_{j'j} = A^{k+1}_{ij}$.

We prove Equation (**??**) in two parts. First, note that we can form a walk from $i$ to $j$ of length $k+1$ by starting at $i$ and walking to some neighbor of $j$, say $j'$, and then walking on the edge from $j'$ to $j$. Every choice of the length $k$ walk from $i$ to $j'$ followed by the edge from $j'$ to $j$ leads to a distinct length $k+1$ walk from $i$ to $j$. Hence,

$$\#(k+1 \text{ length walks from } i \text{ to } j) \geq \sum_{j':A_{j'j}=1} \#(k \text{ length walks from } i \text{ to } j').$$

Moreover, every $k+1$ length walk from $i$ to $j$, can be decomposed into two parts, the $k$ length walk starting at $i$ and ending at $j$'s neighbor and the edge from the neighbor to $j$. Furthermore, no two $k+1$ length walks can have the same decomposition, for otherwise, the two walks would be identical. Therefore,

$$\#(k+1 \text{ length walks from } i \text{ to } j) \leq \sum_{j':A_{j'j}=1} \#(k \text{ length walks from } i \text{ to } j').$$

(b) **Runtime analysis:** Since each matrix multiplication is of runtime $O(n^w)$, a total of $\log k$ multiplications gives $n^w \log k$. Summing over the matrix is of order $n^2$, making the total runtime of the algorithm $O(n^w \log k)$.

3. In class we discussed an algorithm to color the vertices of an undirected $n$ vertex graph with 2 colors so that every edge gets exactly 2 colors (assuming such a coloring exists). We know

of no such algorithm for finding 3-colorings in polynomial time. Here we'll figure out how to color a 3-colorable graph with $O(\sqrt{n})$ colors.

(a) Give a greedy polynomial time algorithm that can properly color the vertices with $\Delta+1$ colors, as long as every vertex of the graph has degree at most $\Delta$.

**Solution.** Pseudo-Code version:

```
   Input: An undirected graph G = (V, E) with max degree Δ + 1
   Result: Color G using atmost Δ + 1 colors.
1  for v ∈ V do
2  |  Let C be {1, 2, ..., Δ + 1}
3  |  for v' ∈ neighbors of v do
4  |  |  Remove the v'_color from C
5  |  end
6  |  Set v_color to be an arbitrary color from C
7  end
```

**Runtime:** The first loop involves going through every vertex $v$ in the graph, and the second involves going through every neighbor of $v$. There are polynomially many vertices and each vertex has a polynomial number of neighbors, thus the algorithm runs in polynomial time.

**Proof of Correctness:** Regardless of how the neighbors of a node are colored, it is always possible to color a node with one of the $\Delta + 1$ colors. Each node has at most $\Delta$ neighbors, so there are at most $\Delta$ colors that a node *cannot* be colored with, yet $\Delta + 1$ colors are available. Thus we will never run out of color, and thus greedy coloring of the nodes work.

(b) Give a polynomial time algorithm that can properly color the graph with $O(\sqrt{n})$ colors, as long as the input graph is promised to be 3-colorable. HINT: If a vertex $v$ has more than $\sqrt{n}$ neighbors, then argue that the subgraph of the neighbors of $v$ must be bipartite, and use the algorithm from class to color $v$ and its neighbors with 3 new colors. Continue this process until every vertex has less than $\sqrt{n}$ neighbors, and then use the algorithm from part (a).

**Solution.**

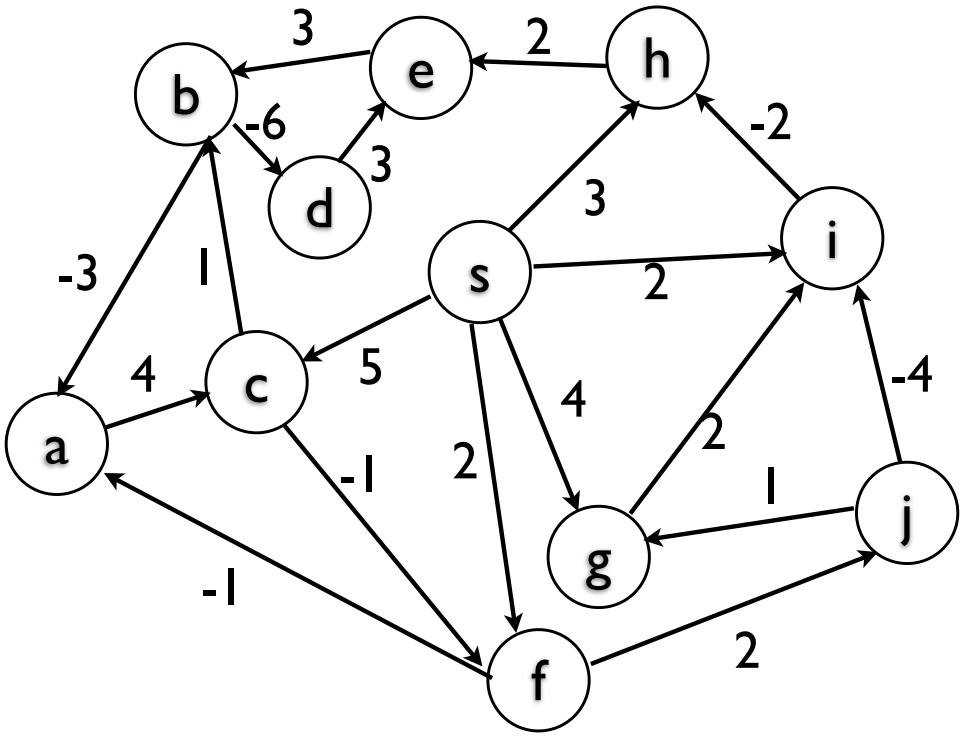
**Input:** An undirected 3-colorable graph $G = (V, E)$.
**Result:** Color $G$ using at most $O(\sqrt{|V|}) = O(\sqrt{n})$ colors.

```
for v ∈ V do
    if v has at least √n uncolored neighbors then
        Pick 3 new colors, c1, c2, c3;
        Color v with c1;
        Color the induced subgraph of v's uncolored neighbors with c2, c3 by
          traversing the subgraph and assign alternating color on the path;
    end
end
Let G' be the induced subgraph of remaining uncolored nodes;
Color G' with algorithm 3(a) using √n new colors;
```

i. **Runtime:** The first for loop involves inspecting each node, and counting the number of colored neighbors, which requires inspecting every edge twice and every node once. Thus, for the first loop overall runs in $O(|E|+|V|)$ (or $O(m+n)$) time. Creating the induced subgraph involves, at most, copying over the original graph, which requires work proportional to the length of the input, followed by $O(m+n)$ to restrict the graph to the pertinent edges and vertices. Coloring that takes $O(m+n)$ time, which follows from the analysis in part (a). Overall, the runtime of this algorithm ss $O(m+n)$.

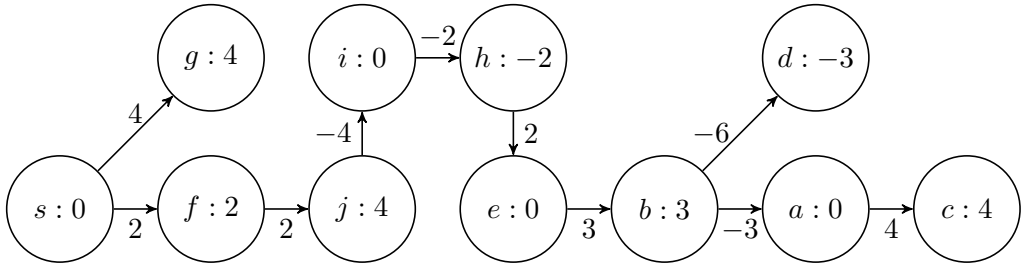
ii. **Proof of Correctness:** In a 3-colorable graph, the neighbors of a single node must be 2-colorable, because they all share a neighbor of the same color. Hence, every vertex with more than $\sqrt{n}$ (uncolored) neighbors with its neighbours, gets assigned 3 new colors each iteration. The outer loop iterates at most $\sqrt{n}$ times, because each time it does so, it colors at least $\sqrt{n}$ uncolored nodes, and there are only $n$ nodes total. Each iteration uses 3 colors, and thus the first loop uses at most $3\sqrt{n}$ colors, which is $O(\sqrt{n})$. The second part correctly colors the induced subgraph with at most $\sqrt{n}$ colors, as the first loop reduces the maximum degree in the induced subgraph to at most $\sqrt{n}-1$. Therefore, the algorithm uses at most $O(\sqrt{n})$ colors overall, as desired.

4. Compute the shortest path tree for the following graph to find all shortest path distances from $s$:




You only need to show the shortest path tree for full credit.

**Solution.**




Note that the numbers inside the node are distances