Assignment 3

Q1. [10 Points] Binomial Trees *(maximum 2 pages)*

Let $\mathcal{B}$ be the family of binomial trees. A *binomial tree of order* k is defined recursively as follows:

- A binomial tree of order zero is a single node, denoted B_0, and is in $\mathcal{B}$.
- For all $k > 0$, a binomial tree of order k consists of two binomial trees of order $k-1$ with the root of one tree connected as a new child of the root of the other and is in $\mathcal{B}$.

See Figure 1 for the first few binomial trees.

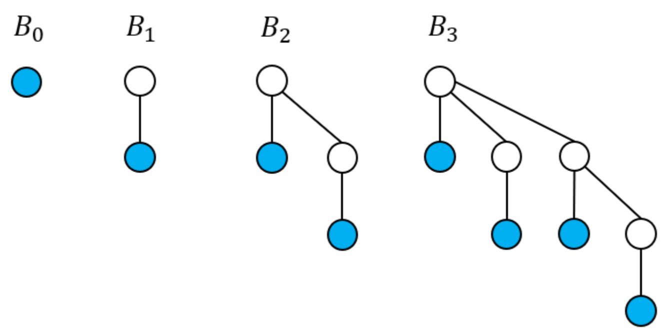

Figure 1: The first four binomial trees. The shaded nodes are leaves.

a. Prove that for all $k \in \mathbb{N}$, a binomial tree of order k has exactly 2^k nodes *using structural induction.*

Proof. For B_n, let our predicate $P(B_n)$ be: B_n has 2^n nodes. We want to show that $P(B)$ is true for all $B \in \mathcal{B}$. Base case. The predicate is true for B_0 which is a single node.

Inductive step. Suppose that $P(B_k)$ is true for some $B_k \in \mathcal{B}$. We want to show that $P(B_{k+1})$ is true. Note that the number of nodes in B_{k+1}, denoted $|V(B_{k+1})|$ is equal to $2|V(B_k)|$ since B_{k+1} is constructed from two copies of B_k. By IH, we have that $|V(B_k)| = 2^k$. Thus $|V(B_{k+1})| = 2^{k+1}$ as required.

By the principle of structural induction, $P(B)$ is true for all $B \in \mathcal{B}$.

[2 Marks] 0.5 for predicate, 0.5 for base case, and 1 for inductive step. □

b. Prove that for all integer $k \geq 1$, attaching a new leaf to every node in a binary tree of order $k-1$ results in a binomial tree of order k *using structural induction.*

Proof. First, we will need a helper predicate. For B_k, our predicate $P(B_k)$ is: the root of B_k has k children and there exists a subtree rooted at a child which is identical to B_0, ..., B_{k-1}.

Base case. When $k = 0$, this is clearly true for the single node.

Inductive step. Note that B_{k+1} is constructed from two copies of B_k with the root of one being the child of the other. By IH, we have that the root of B_k which is also the root of B_{k+1} has subtrees rooted at each of its k children identical to B_0, ..., B_{k-1}. Since we added a copy of B_k as the child of this root, it now has subtrees rooted at each of its $k+1$ children identical to B_0, ..., B_k as required.

Next use $P(B)$ to prove the predicate $Q(B_k)$: B_k is identical to B_{k-1} with a leaf added to each node. We prove that $Q(B_k)$ is true for all $B_k \in \mathcal{B}$ with $k \geq 1$. The base case is clearly true when $k = 1$. B_1 is a root with a single leaf which is one leaf added to the node in B_0.

Inductive step. From the truth of $P(B_k)$ and $P(B_{k+1})$, we know that B_k has a subtree rooted at each of its k children identical to each of B_0, ..., B_{k-1} while B_{k+1} has a subtree rooted at each of its $k+1$ children identical to each of B_0, ..., B_k. By IH, adding a leaf to every node in B_i gets us a tree which is identical to B_{i+1} for all $i = 0, \ldots, k-1$. Further we add a leaf to the root of B_k which is identical to B_0. It follows that by adding a leaf to every node of B_k, we obtain a tree identical to B_{k+1}.

[4 Marks] 1 for predicate, 1 for base case and 2 for inductive step. Generally, if it is not entirely clear what the predicate is and the proof is confusing because of this, take off a mark. □

c. Prove that for all non-negative integers k and d, a binomial tree of order k has exactly $\binom{k}{d}$ nodes at depth d *using structural induction.*

If you use any binomial identities not shown in the lecture, you must prove it.

Proof. For B_k, our predicate $P(B_k)$ is: $\forall d \in \mathbb{N}$, B_k has exactly $\binom{k}{d}$ nodes at depth d. We will prove $P(B)$ is true for all $B \in \mathcal{B}$ using structural induction.

Base case. For B_0, this is true as $\binom{k}{d} = 0$ for all $d > 0$ and $\binom{k}{0} = 1$.

Inductive step. Let $k \geq 0$. We want to show that $P(B_{k+1})$ is true assuming the predicate is true on the two constituent B_k trees which make up B_{k+1}. When $d = 0$, it is still the case that the number of nodes of depth 0 in B_{k+1} is 1, namely the root. For $d \geq 1$ the number of nodes of depth d in B_{k+1} is the number of depth d nodes in the copy of B_k whose root is the root of B_{k+1} and the number of nodes of depth $d-1$ in the copy of B_k whose root is now the child of the other root. By IH, we have that for all d, the number of nodes in B_k of depth d and $d-1$ is $\binom{k}{d}$ and $\binom{k}{d-1}$ respectively. It follows that

$$\begin{aligned}
\binom{k}{d} + \binom{k}{d-1} &= \frac{k!}{d!(k-d)!} + \frac{k!}{(d-1)!(k-d+1)!} \\
&= \frac{k!}{(d-1)!(k-d)!}\left(\frac{1}{d} + \frac{1}{k-d+1}\right) \\
&= \frac{k!}{(d-1)!(k-d)!}\left(\frac{k+1}{d(k-d+1)}\right) = \binom{k+1}{d}
\end{aligned}$$

Thus, by the principle of structural induction, $P(B)$ is true for all $B \in \mathcal{B}$. □

[4 Marks] 1 for predicate, 1 for base case and 2 for inductive step.

Q2. [10 Points] Bipartite Graphs *(maximum 3 pages)*

Let $G = (V, E)$ be a graph with n vertices and m edges. G is *bipartite* if the vertices V can be partition into two disjoint parts A and B such that all edges have one end-point in A and the other in B. More formally, G is bipartite if $\exists A, B \subset V$ such that $\forall v \in V$ either $v \in A$ or $v \in B$ and $A \cap B = \emptyset$ and for all $(u, v) \in E$ either $u \in A$ and $v \in B$ or $u \in B$ and $v \in A$.

Come up with an algorithm which outputs the partition (A, B) **if** G **is bipartite and outputs** None **otherwise.** Assume that $n \geq 2$ and the input to your algorithm is an *adjacency list* i.e. G is a list containing n lists where list i stores the indices of the neighbours of vertex i. For a list L, the time required to check the length of L ($|L|$), add an element to the end of L ($L.add(i)$), remove and return the element at the end of L ($L.pop()$) is $O(1)$. The output should be a pair of sets A and B. Generally, for a set S, the time required to check if an element i is in S ($S.contains(i)$), check the length of S ($|S|$) , add to S ($S.add(i)$), and remove element i from S if it exists ($S.remove(i)$) is $O(1)$.

Your algorithm should run in $O(m + n)$ time.

a. Prove that G is bipartite if and only if G does not have any odd cycles.

Proof. We show that both directions are true.

Necessary. We show that "G is bipartite" implies "G does not have any odd cycles". Suppose for a contra-positive, that G had an odd cycle C. Alternate coloring the nodes of C red and blue. Since C is odd, there must be two adjacent nodes with the same color. All the red nodes *must* be in one part and all the blue nodes *must* be in the other. It follows that *some* edge between adjacent nodes must be in the same part.

Sufficient. Next, show that "G does not have any odd cycles" implies "G is bipartite". Assume that G does not have an odd cycle. Similar to the above, we will red-blue color the nodes of G starting from an arbitrary red node u. The neighbours of u are colored blue, their neighbors are colored red and so on. If some two adjacent nodes v and w are given the same color, then we have an odd walk: $u \rightsquigarrow v \to w \rightsquigarrow u$. By a lemma we saw in-class, if a graph contains an odd closed walk, then it must also contain an odd cycle. Since G does not have any odd cycles, it must be the case that no two adjacent nodes are given the same color. The partition of the vertices by color shows that G is bipartite, i.e., all edges will have one end-point of each color. $\square$

[2 Marks] 1 marks for each direction.

b. Write the pseudo-code of your algorithm in Algorithm 1.

See part d.

c. Proof-of-correctness setup. List the variables use in the proof, and the loop-invariant(s).

Solution. We will define the variables and loop-invariant with the proof in the next section.

See part d.

d. Proof-of-correctness. Prove that your algorithm outputs the desired values.

Algorithm 1 Bipartition(G)

Require: G is a graph with at least two nodes given as an adjacency list.

Ensure: Returns sets A and B such that $A \cap B = \emptyset$, $\forall i \in [n]$ either $i \in A$ or $i \in B$, and for every edge (u, v) in G, one of u or v is in A and the other is in B.

```
V ← {}                                   ▷ empty set V
A ← {}                                   ▷ empty set A
B ← {}                                   ▷ empty set B
C ← []                                   ▷ empty list C
for i ∈ [n] do
    if i ∉ V then
        A.add(i)
        C.append(i)
        while |C| > 0 do
            k ← C.pop()
            if ¬V.contains(k) then
                if k ∈ A then            ▷ add neighbours of a node in A to B
                    for j ∈ G[k] do
                        if j ∈ A then return None
                        end if
                        B.add(j)
                        C.append(j)
                    end for
                else if k ∈ B then       ▷ add neighbours of a node in B to A
                    for j ∈ G[k] do
                        if j ∈ B then return None
                        end if
                        A.add(j)
                        C.append(j)
                    end for
                end if
                V.add(k)
            end if
        end while
    end if
end for
return A, B
```

Proof. There are two nested loops in the algorithm. The for-loop which begins on line 5 and the while-loop which begins on lines 9. The outer for-loop picks a node in a connected component[1] to two-coloring and the while-loop performs the actual two-coloring. V represents the set of nodes which have been *processed*, i.e., it and all its neighbours have been colored. For my solution the while-loop is more important so I will prove it formally. For the for-loop, we observe that it iterates over all nodes so ensures that every node will eventually be in V. Either the if-statement of line 6 is true, and the node is already in V, or it is false, and the node will be added to V on line 27 because it is added to and then removed from C.

It suffices to consider the case where the graph is connected. First we make the following **observation** $(*)$: a node will be in A or B when it is popped from C on line 10. The first time we pop a node from C there is only one element i added in line 8. i is added to A on line 7. Consider some x must be added to C in some later iteration on line 17 or 24. In order for either of those lines to execute so must lines 16 and 23.

Let our variables be V_t, A_t, B_t, and C_t which represent the sets V, A, and B and the list C after the t^{th} iteration of the while-loop. Our loop-invariants will be $Q(t)$: A_t and B_t represent all nodes colored red and blue in the *frontier* of V_t (i.e., every node in $A_t \cup B_t$ is in V_t or adjacent to a node in V_t and every node adjacent to some node in V_t is in $A_t \cup B_t$), C_t represent edges from V_t to their neighbours (at least one end-point of such an edge is in C_t), and the nodes in V_t are given a valid two-coloring (i.e., for adjacent nodes in V, they will be given different colors). Since V will eventually contain all the nodes, either the final two-coloring is valid or the process will fail somewhere in the middle and return None.

We prove $Q(t)$ by induction. In the base case V_0, A_0, B_0, C_0 are all empty so the claim is trivially true. In the inductive step suppose that $Q(0) \wedge \cdots Q(s)$ is true and we want to prove that $Q(s+1)$ is true assume that C_s is non-empty. Let x_{s+1} be the node popped on line 10. If $x_{s+1} \in V_s$ then $Q(s+1)$ is true, instead, suppose $x_{s+1} \notin V_s$ and gets added to V in the $s+1^{\text{th}}$ iteration. By IH and $(*)$ we know that x_{s+1} is in the frontier $A_s \cup B_s$. One of the for-loops on line 13 or 20 will add all of x_{s+1}'s neighbours to $A \cup B$ so $A_{s+1} \cup B_{s+1}$ will contain the new frontier. Like-wise the edges with one end-point x_{s+1} will be represented by the other end-point added to C_s in lines 17 and 24. V_s was a valid coloring by IH and will continue to be a valid coloring after adding x_{s+1} as otherwise line 14 or line 21 returns None. □

[6 Marks] The pseudo-code and proof of correctness should be graded simultaneously. If the algorithm is not correct then the submission can get no more *than 2 marks (depending on how severe the error is). Otherwise it is 2 marks for the predicate, 1 marks for defining the appropriate variables, and 3 marks for the rest of the proof (1 mark base case and 2 marks inductive step).*

e. Running time. Evaluate the running time of your algorithm and justify your answer.

Solution. We consider each connected component separately. For a connected component, the running time is proportional to the maximum length of C. To bound the length, we bound the maximum number of times a node k can appear in C. We claim that the bound is $O(1 + \deg(k))$ times where $\deg(k)$ is the degree of k. This is because, k can only be added to C in line 8 or one of lines 17 or 24. The former case can happen at most once per node. The latter case can happen as many times as k has neighbours. Over all nodes, this bound is $O\left(n + \sum_v \deg(v)\right) = O\left(n + m\right)$.

[1] This is a *maximal* (i.e., the largest possible) subset of the vertices which are connected.

[2 Marks] Justification is the main part. If the running time is not $O(n + m)$, *then the submission must also correctly evaluate the running time.*