We begin by proving that the Node-Labeling Decision Problem is NP-complete. Let this problem be denoted by L. Then, do the following:

Show that $L \in$ NP:

- Given any YES-instance of L, which is a graph $G' = (V', E')$, a set of labels $K = \{0,1,\cdots,k-1\}$ with $k \leq |V'|$, and a non-negative integer R, there must exist some certificate that can be used to verify that the solution to the instance is indeed YES in polynomial time. The certificate here is a labeling that assigns a label $c(v) \in K$ to every node $v \in V'$, such that every node can find the k distinct labels within a neighborhood of radius R.
- The verification can be done as follows. For each $v \in V'$ do the following. First find all the nodes that are at most R hops away from v (including v itself). One way to achieve this is to find the shortest distance from v to itself and every other node in V', and then picking all the nodes whose shortest distance from v is no greater than R. This can be done in $O(|V'||E'|)$ time. Second, check if each label in K is assigned to at least one of the nodes found in the first step. This can be done in $O(k|V'|)$ time. Note that the second step must give a YES answer for each $v \in V'$. The total time complexity of this verification procedure is $O(|V'|(|V'||E'| + k|V'|)) = O(|V'|^2(|E'| + k))$ Hence, a polynomial-time verification exists.

Note that the store allocation problem in homework 8 is essentially a 3-label problem, where given an undirected graph $G = (V, E)$ and three labels $\{0,1,2\}$, we ask whether there exists a labeling of the nodes in V such that each node is assigned one of the three labels, and no two adjacent nodes shares the same label. We show a polynomial-time mapping from the 3-label problem which was proved to be NP-complete, to the problem L, and show that the mapping preserves the "YES/NO" answer.

- The mapping is as follows. Suppose we have some instance in the 3-label problem, which is given by $G = (V, E)$ and labels $\{0,1,2\}$. We then create a corresponding instance in L, given by $G' = (V', E')$, $K = \{0,1,2\}$, $R = 1$. Clearly, K and R can be defined in polynomial time. Now, we show how G' can be defined. First, create a copy of G. This can be done in $O(|V| + |E|)$ time. Next, for each edge $e = (u, v) \in E$, define a third node w and add two additional edges $(w, u), (w, v)$ to create a triangle subgraph with nodes u, v, w in G', as illustrated in Fig.1. This can be done in $O(|V| + |E|) * O(1) = O(|V| + |E|)$ time. Next, for every node $u \in V$ with no edges incident on it, define two additional nodes v, w, and add three additional edges $(u, v), (u, w), (v, w)$, to create a triangle subgraph like before, with nodes u, v, w in G' which is also illustrated in Fig.1. This can also be done in $O(|V| + |E|) * O(1) = O(|V| + |E|)$ time. This completes the mapping procedure. Clearly, the mapping takes polynomial time. For better clarity of the mapping procedure, an illustration of an example instance is presented in Fig.2.

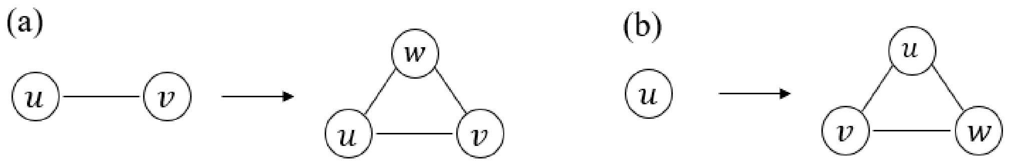

Fig. 1. Mapping transforms: (a) Edges in G to triangles in G'. (b) Disconnected nodes in G to triangles in G'.

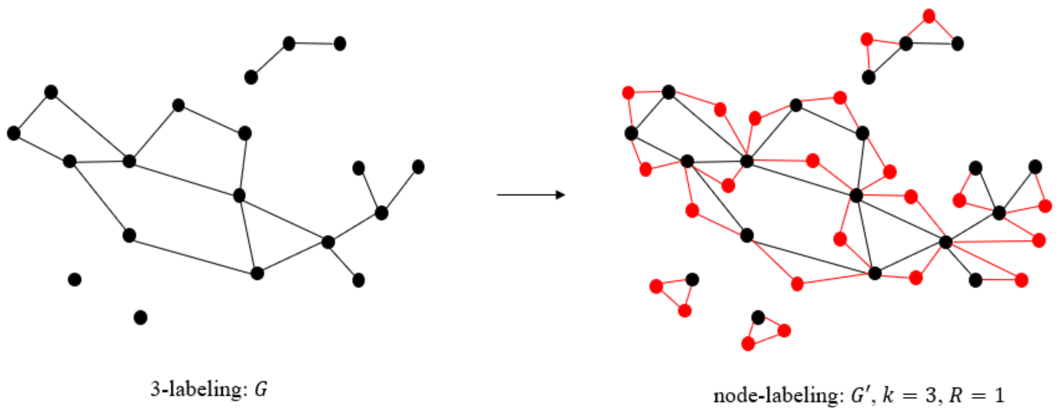

Fig. 2. Polynomial-time mapping from an instance in 3-labeling to a corresponding instance in node-labeling.

- If the answer to the 3-label instance is YES, then we can assign labels to the nodes such that for any edge $(u, v) \in E$, nodes u, v will have two distinct labels from $\{0,1,2\}$. Also, any node with no edges incident will have one of the three distinct labels. Now, consider the L instance. Corresponding to each edge (u, v) in G, we now have a triangle subgraph with nodes u, v, w in G'. Assign the u, v nodes in all these triangle subgraphs in G', with the same labels as the ones assigned to u, v in the 3-label instance. Clearly, u, v will then have distinct labels in K. Now, assign the third distinct label to w. Since u, v, w are $R = 1$ hop away from each other, and they have distinct labels in K, all the nodes in these triangles can find $k = 3$ distinct labels within radius $R = 1$. Now, consider the triangle subgraphs with nodes u, v, w in G', corresponding to nodes u with no incident edges in G. Simply assign u, v, w with the three distinct labels in K, and all the nodes in these triangles can find k distinct neighbors within radius R. Hence, the answer to the L instance is YES.
- Suppose that the answer to the L instance is YES, and we have some YES labeling. Then for any triangle subgraph with nodes u, v, w in G' that corresponds to some edge (u, v) in G, nodes u, v must have distinct labels. This is because node w is only adjacent to nodes u and v, and the YES labeling forces node w to find 3 distinct labels within radius 1. Now, consider the 3-label instance. For each edge (u, v) in G, assign u and v with the same labels that u, v nodes in the corresponding triangle subgraph in G' were assigned in the YES labeling. Clearly, u, v has distinct labels in $\{0,1,2\}$. Now, for each node u in G with no incident edges, assign one of the labels from $\{0,1,2\}$. Hence, the answer to the 3-label instance is YES.

Main Idea

This section discusses the main idea of an algorithm proposed for a special case of the optimization version of the node labeling problem, where the graph is a tree. A greedy strategy is used here. The algorithm first runs a BFS on the given tree, and stores in a list, the nodes in the order in which they were visited. Then, for each node v in the list (we follow the order of the list), the algorithm finds k closest nodes from v (including v) using a BFS-like search, and from these k nodes, labels all the unlabeled nodes with distinct labels in K, that were not used by the labeled nodes. The key result from this strategy is that for each v, once the labeling is done, all the k closest nodes selected will have distinct labels in K.

Pseudocode and Time Complexity

This section discusses the pseudocode and time complexity of the proposed algorithm. First, we present the pseudocode of the BFS-like search that takes the tree $G = (V, E)$, a source node $s \in V$, k, and some labeling $c(v)$ for each node $v \in V$ (If node v is unlabeled, $c(v) = -1$. Otherwise, $c(v) \in K$) and returns k closest neighbors of s (including s), as well as the labels that were assigned to labeled nodes amongst these k nodes. We shall call this algorithm $closest\_k\_neighbors$.

Input: $G = (V, E)$, s, k, c

```
1.  for $v \in V$:
2.     $v.visited \leftarrow FALSE$ // No node is initially visited.
3.  $used\_labels \leftarrow [\,]$ // Store labels used by labeled nodes amongst $k$ closest neighbors.
4.  $counter \leftarrow 1$ // First neighbor is the source node itself.
5.  $k\_closest \leftarrow [s]$ // Store the $k$ closest neighbors.
6.  $Q \leftarrow [s]$ // Initialize a queue for BFS and add source node.
7.  $s.visited \leftarrow TRUE$
8.  if $c(s) \neq -1$:
9.     $used\_labels.insert(c(s))$
10. while $Q \neq \emptyset$ and $counter < k$:
11.    $u \leftarrow Q.pop\_front()$
12.    for $v \in G.adj(u)$:
13.       if $v.visited = FALSE$:
14.          if $counter < k$:
15.             $counter \leftarrow counter + 1$
16.             $k\_closest.insert(v)$
17.             $Q.push\_back(v)$
18.             $v.visited \leftarrow TRUE$
19.             if $c(v) \neq -1$:
20.                $used\_labels.insert(c(v))$
21.          else:
22.             break
23. return $k\_closest$, $used\_labels$
```

Now we present the pseudocode of the node labeling algorithm. Here, the input r represents the root node which is defined by the user.

Input: $G = (V, E), r, k$

```
1.  for $v \in V$: $O(|V|) * O(1)$
2.    $c(v) \leftarrow$ -1 // No nodes are assigned a label yet. $O(1)$
3.  $visited \leftarrow$ BFS$(G, r)$ // Store the nodes in the order they were visited by BFS. $O(|V|)$
4.  for $v \in visited$: $O(|V|) * (O(|V|) + O(1) + O(k^2) + O(k))$
5.    $k\_closest, used\_labels \leftarrow closest\_k\_neighbors(G, v, k, c)$ $O(|V|)$
6.    $allowed\_labels \leftarrow [\,]$ // A queue that stores labels not used by any of the $k$ nodes. $O(1)$
7.    for $i = 0$ to $k - 1$: $O(k) * O(k)$
8.      if $i \notin used\_labels$: $O(k) * O(1)$
9.        $allowed\_labels.insert(i)$ $O(1)$
10.   for $u \in k\_closest$: $O(k) * O(1)$
11.     if $c(u) =$ -1: $O(1)$
12.       $c(u) \leftarrow allowed\_labels.pop\_front(\,)$ $O(1)$
13. return $c$ $O(1)$
```

Now, we analyze the time-complexity. First we evaluate the complexity of $closest\_k\_neighbors$. Lines 1-2 takes $O(|V|)$ time, then lines 3-9 takes $O(1)$ time. Now, consider the while-loop in line 10. This loop repeats $O(|V|)$ times, and inside this loop, line 11 takes $O(1)$ time, and the remaining lines 12-22 takes $O(|E_u|) * O(1) = O(|E_u|)$ time where E_u is the set of nodes adjacent to node u. Since the while loop in the worst-case repeats for all the $u \in V$, lines 10-22 together must have a time-complexity of $O(\sum_{u \in V}(1 + |E_u|)) = O(|V| + |E|)$. The $closest\_k\_neighbors$ algorithm then clearly has a time complexity of $O(|V| + |E|) = O(|V|)$ (since G here is a tree) which is to be expected since it is a BFS-like search. Note that a more careful complexity analysis from lines 10-22 shows that the total number of operations in these lines can be a tighter $O(k)$ estimate since G is a tree, and exactly k nodes are visited, with the rest of the operations for each of these k nodes being $O(1)$. However, since lines 1-2 has complexity $O(|V|)$, the total complexity is still $O(|V|)$.

Now, let us evaluate the time-complexity of the node labeling algorithm. From the complexity analysis performed alongside the pseudocode, we can easily see how the algorithm runs in $O(|V|) * \left(O(|V|) + O(1) + O(k^2) + O(k)\right) = O(|V|^2 + |V|k^2)$ time. Note that line 8 takes $O(k)$ time since in the worst case, $used\_labels$ can have k entries, and in this case, finding whether $i \notin used\_labels$ may require scanning through k entries.

Proof of Proximity Ratio

In this section, we will prove that the proposed algorithm has a $\rho = 1$ proximity ratio. To show that this is true, it is sufficient to show that for each node $v \in V$ (in the order they are in $visited$), once we run BFS with v as the root node until we have exactly k nodes (including v) and do the labeling on these k nodes as previously explained, all the k nodes will have distinct labels in K. Proving this fact, however, requires proving a few intermediate lemmas. Before we start the proof, let us define some terms that can make the presentation of the proof easier.

- k-node BFS tree (kBFST) from source node: A BFS tree starting at some source node and ending when exactly k distinct nodes (including source node) are found.
- Suppose we have a parent node, and a child node in a tree. Then we define the following.
    - Descendants of a child/parent: Children of child/parent or children of descendants of child/parent.
    - Other nodes in tree: Every node in the tree that is not the child or its descendants.

We now start with the following lemma.

Lemma 1: Consider some child node in a tree, and its parent node. The nodes present in the kBFST from child, but not present in the kBFST from parent, must be descendants of the child.

Proof: Child and its descendants are 1 hop closer to the child than it is from the parent, and the other nodes in tree are 1 hop farther from the child than it is from the parent. We know that the BFS explores nodes that are 0 hops away, then 1 hop away, then 2 hops away, and so on. Hence, the kBFST from parent first explores other nodes in tree that are i hops away from the parent, before the ith hop descendants of child, for some i. Similarly, the kBFST from child first explores the ith hop descendants of child, before other nodes in the tree that are i hops away from the parent, for some i. Consequently, the subset of other nodes explored by the kBFST from child will also be explored by the kBFST from parent, and the subset of descendants of child explored by the kBFST from parent will also be explored by the kBFST from child. For better clarity, we illustrate an example of this in Fig. 3. Hence, if a node is present in the kBFST from child, but not present in the kBFST from parent, it cannot be other node in tree. Hence, it must be a descendant of the child.

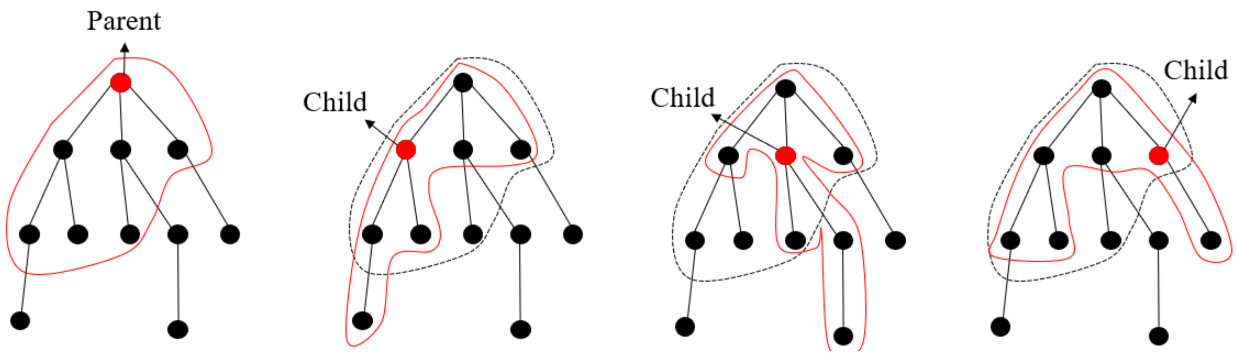

Fig .3. In this example, $k = 7$. Observe how subset of other nodes explored by child kBFST are also explored by parent kBFST and the subset of descendants of child explored by parent kBFST are also explored by child kBFST.

Let us from now on call nodes present in the kBFST from child, but not present in the kBFST from parent, as remainder nodes from child.

Lemma 2: In the proposed algorithm, when finding the kBFST from some child node, the remainder nodes from that child will be unlabeled.

Proof: Suppose that a remainder node from child is labeled. Then, we must have previously found the kBFST from some source node, and the remainder node from child was included in that tree.

Since the algorithm picks source nodes in the order of BFS on the entire tree, we pick the root node, then layer 1 nodes, then layer 2 nodes, and so on. Hence, the potential source nodes from which we found kBFST previously are either the siblings of the child, or the parent of the child, or some other nodes in the previous layers in the tree. However, the source node cannot be the parent node, by definition of remainder node from child. The source node cannot also be siblings because by lemma 1, the remainder node from child must be a descendant of the child, and remainder nodes from sibling must be descendants of siblings, and for a tree, sibling nodes cannot share descendants. Now, suppose that the source node is in some previous layers. Then, since the only way for a kBFST from source node to explore a remainder node from child is by exploring the child node first, the kBFST must contain the child node. However, note that the parent is the closest to the child node since it is only 1 hop away from it. Any other potential source node in previous layers will be farther from the child than the parent. Because of this and the nature of BFS exploring closer nodes first, since the kBFST from the parent node does not contain the remainder node from child, the kBFST from any other potential source node in previous layers cannot contain the remainder node from child. This then leads to a contradiction of our initial assumption, and thereby concludes the proof.

Theorem 1: When the algorithm finds the kBFST from each node $v \in V$ (in the order they are in $visited$), and assigns labels to these nodes as previously explained, all the k nodes in this sub-tree will have distinct labels in K.

Proof: We can prove this using an inductive type reasoning. First, note that when the algorithm finds the kBFST from the root node at the beginning, all the nodes in this sub-tree are unlabeled, and the algorithm labels these k nodes with distinct labels in K. Now, assume that the algorithm assigns the nodes in the kBFST from some parent node $v \in V$, with distinct labels in K. Then, by lemma 2, when the algorithm finds the kBFST from a child node of that parent node, the remainder nodes from that child will be unlabeled. Also, note that the nodes in kBFST from child node that are not remainder nodes will have distinct labels in K since these nodes are shared with the kBFST from the parent node. The algorithm simply labels the unlabeled remainder nodes from child, with distinct labels in K that were not assigned to the rest of the labeled nodes in kBFST from child node. Consequently, all the nodes in kBFST from child node will have distinct labels in K. Hence, the theorem should then be true by mode of induction.

Theorem 2: The proposed algorithm has a $\rho = 1$ proximity ratio.

Proof: This easily follows from theorem 1, since the algorithm ensures that for any node $v \in V$ in a tree, some k closest nodes to that node are assigned distinct labels in K. Hence $\frac{r(v)}{m(v)} = 1 \; \forall \; v \in V$