Design and Analysis of Algorithms - Project Report

IMT2018043 Mohith Sathyanarayanan
IMT2019064 Pratik Ahirrao
IMT2019084 Shrey Tripathi

17 March, 2021

1 Problem

Given a tree T, give an algorithm that either correctly computes a dominating set for T such that every vertex in T has exactly one vertex from the dominating set in its open neighbourhood (The open neighbourhood of a vertex contains all its neighbours) or decides that such a dominating set does not exist.

2 About the problem

This problem is an **NP-complete decision** problem, and hence there may not be an efficient algorithm, and only approximate algorithms can be found.
Please note that at multiple stages we could be having different valid nodes to select and the algorithm would have to decide which node to select. This is the reason why it is also a decision problem.

3 Approach

- Start from the leaf node and add its parent (having 2 or more leaf nodes) to the **Dominating Set**.
- Given a tree, to ensure that we have a node in the dominating set from the neighbourhood of each node we could do one of the following:
  1. We can select the 3rd node from the current node.
  2. We can select the next node.
  3. We can select the 4th node from the current node.
- Note that if there exists a node A such that more than 1 of its neighbouring nodes (B, C) have more than 1 leaf nodes then the solution does not exist as in that case we will be required to select both B and C (to cover the neighbourhood of the leaf nodes) and that would mean we are selecting 2 nodes from the neighbourhood of A, which is not feasible.
- If we don't get a solution by proceeding through the 1st step mentioned above, we then add the leaf node itself to the dominating set and proceed to our algorithm further through the 2nd step mentioned above.

4 Pseudo code

```
find_neighbours(node) {   // using Breadth First Search
    Return list of all neighbouring nodes
}

// for checking, if the neighbours of a node have already been covered or not
list_searcher(node_neighbours[ ], covered_neighbours_set) {
    if (no element in node_neighbours[] is present in covered_neighbours_set)
        Return true
    Else
        Return false
}

Dominating_set = [ ]
covered_neighbours_set = [ ]

Start from any leaf node whose parent has two or more leaf nodes:
Add the parent of this leaf node to Dominating_set.
Add all the neighbours of this node to covered_neighbours_set.

V = parent node
For V not in covered_neighbours_set
{
    if (list_searcher(find_neighbours($V_{J+3}$), covered_neighbours_set))
        $DS = DS + V_{J+3}$
        $V = V_{J+3}$
        continue
    Else if (list_searcher(find_neighbours($V_{J+1}$), covered_neighbours_set))
        $DS = DS + V_{J+1}$
        $V = V_{J+1}$
        continue
    Else if (list_searcher(find_neighbours($V_{J+4}$), covered_neighbours_set))
        $DS = DS + V_{J+4}$
        $V = V_{J+4}$
        continue

    If Dominating_set.length() + covered_neighbours_set.length() == Total number of Nodes:
        print Dominating_set
        break
}

If Dominating_set.length() + covered_neighbours_set.length() != Total number of Nodes:
    print "Does not exist"
```

5 Recurrence Relation

Let $DS = \{........v_i\}$ be the **dominating set** where v_i is the last node added to the dominating set.

So, the required dominating set DS_{final} will become:

$$DS = \begin{cases} DS + v_{i+3}, & \text{or} \\ DS + v_{i+1}, & \text{or} \\ DS + v_{i+4} \end{cases}$$

6 Time Complexity

Let the total number of searches that we will have to do be k searches in the *covered_neighbours_set*, each of which takes $O(n)$ time (using linear search).
So, the running time is nk.
Now for the BFS in the function $find\_neighbours$, we will have a complexity contribution of $O(n + m)$.
So, the total time complexity is:

$$n(n + m) + n2^{logk}$$

Please note that k here is not a constant and is in-fact a variable.
So, the running time is actually $(2^{logk})n$ (exactly like in 01 Knapsack problem) since the first part can be ignored in comparison to the second part of the equation.

This running time is **pseudo-polynomial** running time.
This also shows that the problem is **NP-Complete**.

7 Proof of correctness

The algorithm can be proved using the proof for recurrence relation.

Claim: Let $DS = \{........v_i\}$ be the **Dominating Set** where v_i be the last node added to the Dominating Set. We add either v_{i+3} or v_{i+1} or v_{i+4} to the Dominating Set.

Proof: We can prove the claim by using Contradiction.

Suppose v_{i+k} is the node added in the partial solution where $k = \{N - \{1, 3, 4\}\}$
Consider two cases:-

Case 1: When $k = 2$
If v_{i+2} is added to the dominating set, it implies we have selected more than one node to the Dominating set from the neighbourhood of v_{i+1}.
This does not give the required solution.

Case 2: When $k > 4$
For all $k > 4$, We get at least one vertex without selecting the vertex from its neighbourhood.
It implies this is a contradiction.
Also, the node selected can be the leaf node whose parent can have another (one or more) leaf nodes.
The neighbourhood of the other leaf nodes contains only their parent. So, the required condition will not be satisfied, as their parent is not selected in the dominating set.

To help decide between different possible nodes, we will be assigning priority to them. That is, we will first try v_{j+3}, then v_{j+1} and only then v_{j+4}, the reasoning behind this being:

- If we are at a node, the farthest node we can go to such that exactly one neighbour of each of the nodes on the path between them lies in the dominating set is v_{j+3}.
- If we cannot add v_{j+3} to the dominating set, we go to v_{j+1} because that is the second farthest we can go to satisfy the required condition.
- If that is also not possible, we go to the $4th$ node.

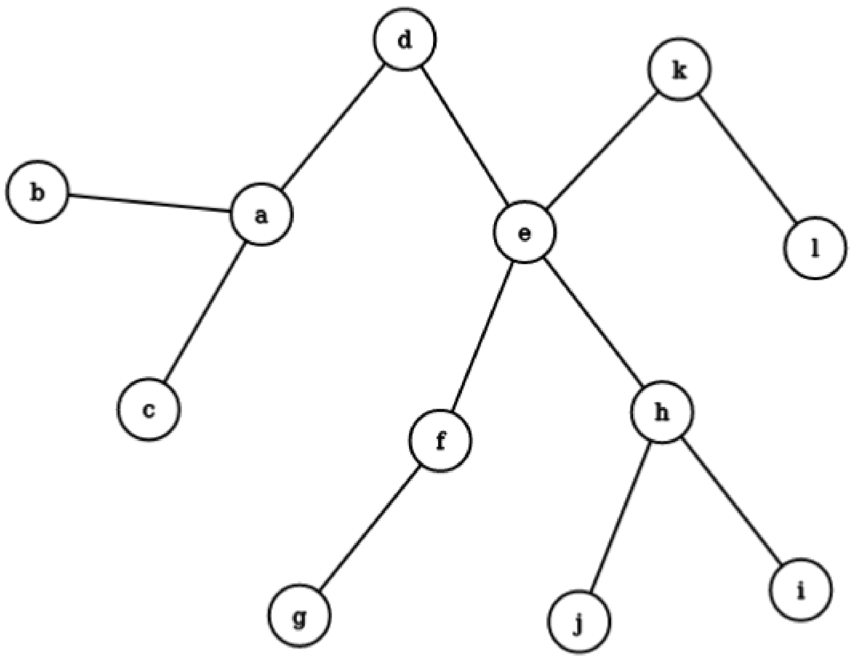
Condition for **NO** solution:

Lemma: If at a given point of time, we encounter a node such that it has more than one neighbouring node each having more than one leaf nodes, then the solution does not exist.

Proof: We can prove this by using contradiction.
Assuming that solution exists, then it would mean that we have to select both the neighbouring nodes to cover all the neighbourhoods, but that would be a violation of our condition to have only one node from each neighbourhood.
Hence, this is a contradiction, implying that our assumption was wrong. Hence, the lemma was correct.

8 Example

The workings of the algorithm can be understood through an example.



Here, let us start from the node j.

So, we make node h a part of the dominating set and add all its neighbours to the covered neighbours list. Hence,

$$covered\_neighbours\_set = \{j, i, e\}$$

$$dominating\_set = \{h\}$$

Now, on going to v_{j+3}, we can select any of the nodes l, g, d.
Since we already have a node to select from v_{j+3}, we do not try other cases.
Let us say we select l (we could select any of the 3). We add the neighbours of l to the list and l to the dominating set list.

$$covered\_neighbours\_set = \{j, i, e, k\}$$

$$dominating\_set = \{h, l\}$$

Now, we go to v_{j+3} again, and our options are $\{f, d\}$, but both are neighbours of e, whose neighbourhood is already covered.
So, we now try v_{j+1}, which gives us k, which is a neighbour of e.
So, we cannot select k either. Now we go to v_{j+4}, which gives us 2 options a and g. Since neither of the neighbours are covered, we can select either of them.
Let us say we select g. So,

$$covered\_neighbours\_set = \{j, i, e, k, f\}$$

$$dominating\_set = \{h, l, g\}$$

Now, we go to v_{j+3}, but similar to last time, we cannot select anything,and same is the case for v_{j+1}. So, we go to v_{j+4} and we see that we can select the node a. So,

$$covered\_neighbours\_set = \{j, i, e, k, f, b, c, d\}$$

$$dominating\_set = \{h, l, g, a\}$$

Here we see that

$$len(Dominating\_set) + len(covered\_neighbours\_set) = total\ nodes\ in\ the\ tree$$

Hence, the for loop exits here and we have our solution.