1938. Maximum Genetic Difference Query

Leetcode Link

Problem Explanation

We have a tree with `n` nodes, where each node has a unique genetic value. The genetic difference between two genetic values is the bitwise-XOR of their values. The tree has a root node and the parent for each node is given as an array `parents`. We are also given an array of queries, where each query contains a node and a value. For each query, we want to find the maximum genetic difference between the given value and any node's genetic value on the path between the query node and the root (including the query node and the root). The output should be an array of answers for each query.

Approach and Algorithm

To solve this problem, we'll use a data structure called a binary trie. A binary trie is a tree data structure where a bitwise representation of a number is stored with each bit occupying a node in the trie. It can be used to efficiently find the maximum XOR between a query number and numbers already stored in the trie.

To implement the solution, we'll use the following data structure and functions:

- TrieNode: The trie node structure with two children pointers for 0 and 1 and a count value to store how many nodes are passing through this trie node.
- Trie: The trie class with functions to insert a number with a given value and query the trie for the maximum XOR for a given number.
- dfs: A function to traverse the tree in depth-first search manner, updating the trie as it goes and calculating the answers for each query.

We'll create a tree from the given `parents` array and for each query, we'll associate the queries with their respective tree nodes. Then, we'll traverse the tree using the dfs function. The dfs function will take the trie, tree, nodeToQueries, and ans as arguments.

During the traversal, each node value is inserted in the trie. For each query associated with the current node, we'll calculate the answer using the trie query function and store it in the ans array. Once the traversal of the children of the current node is complete, we will remove the node from the trie.

Example

Let's walk through a simple example:

```

parents = [3, 3, 0, -1]
queries = [[0, 1], [2, 6], [2, 2], [3, 7]]
```

1. The tree structure from the parents array would be:

```

  3
 / \
0   1
 \
  2
```

2. The nodeToQueries map will look like this:

```

{
  0: [(0, 1)],
  2: [(1, 6), (2, 2)],
  3: [(3, 7)]
}
```

3. Now, we will perform a depth-first search starting from the root (node 3).

4. First, we insert the value `3` into the trie:

```

  Trie (for value 3):
   * 1
   / \
null  *
/_\ |_\
```

5. We process the query for node 3: `3 XOR 7 = 4`

6. Now, we traverse to the left child (node 0), and insert the value `0` into the trie:

```

   Trie (for values 0 and 3):
      * 2
     __/ \__
  *       *
 /_\     /_\
```

7. We process the query for node 0: `0 XOR 1 = 1`

8. Now, we traverse to the right child (node 2), and insert the value `2` into the trie:

```

  Trie (for values 0, 2, and 3):
        * 3
     __/    \__
    *         *
   / \__    __/ \
  *     *  *     *
 /_\   /_\/_\   /_\
```

9. We process the queries for node 2: `2 XOR 6 = 4` and `2 XOR 2 = 0`

10. After processing all queries, the final ans array would be `[1, 4, 0, 4]`.

C++ Solution

```

cpp
#include<bits/stdc++.h>
using namespace std;

struct TrieNode {
  vector<shared_ptr<TrieNode>> children;
  int count = 0;
  TrieNode() : children(2) {}
};

class Trie {
 public:
  void update(int num, int val) {
    shared_ptr<TrieNode> node = root;
    for (int i = kHeight; i >= 0; --i) {
      const int bit = (num >> i) & 1;
      if (node->children[bit] == nullptr)
        node->children[bit] = make_shared<TrieNode>();
      node = node->children[bit];
      node->count += val;
    }
  }

  int query(int num) {
    int ans = 0;
    shared_ptr<TrieNode> node = root;
    for (int i = kHeight; i >= 0; --i) {
      const int bit = (num >> i) & 1;
      const int targetBit = bit ^ 1;
      if (node->children[targetBit] && node->children[targetBit]->count) {
        ans += 1 << i;
        node = node->children[targetBit];
      } else {
        node = node->children[targetBit ^ 1];
      }
    }
    return ans;
  }

 private:
  static constexpr int kHeight = 17;
  shared_ptr<TrieNode> root = make_shared<TrieNode>();
};

class Solution {
 public:
  vector<int> maxGeneticDifference(vector<int>& parents, vector<vector<int>>& queries) {
    const int n = parents.size();
    vector<int> ans(queries.size());
    int rootVal = -1;
    vector<vector<int>> tree(n);
    unordered_map<int, vector<pair<int, int>>> nodeToQueries;
    Trie trie;

    for (int i = 0; i < parents.size(); ++i)
      if (parents[i] == -1)
        rootVal = i;
      else
        tree[parents[i]].push_back(i);

    for (int i = 0; i < queries.size(); ++i) {
      const int node = queries[i][0];
      const int val = queries[i][1];
      nodeToQueries[node].emplace_back(i, val);
    }

    dfs(rootVal, trie, tree, nodeToQueries, ans);
    return ans;
  }

 private:
  void dfs(int node, Trie& trie, const vector<vector<int>>& tree,
           const unordered_map<int, vector<pair<int, int>>>& nodeToQueries,
           vector<int>& ans) {
    trie.update(node, 1);

    if (const auto it = nodeToQueries.find(node); it != cend(nodeToQueries))
      for (const auto& [i, val] : it->second)
        ans[i] = trie.query(val);

    for (const int child : tree[node])
      dfs(child, trie, tree, nodeToQueries, ans);

    trie.update(node, -1);
  }
};

int main() {
  vector<int> parents = {3, 3, 0, -1};
  vector<vector<int>> queries = {{0, 1}, {2, 6}, {2, 2}, {3, 7}};
  Solution sol;
  vector<int> results = sol.maxGeneticDifference(parents, queries);
  for (int res : results) {
    cout << res << " ";
  }
  return 0;
}
```

Python Solution

```

python
from typing import List, Tuple

class Trie:

    def __init__(self):
        self.root = {}

    def update(self, num: int, val: int) -> None:
        node = self.root
        for i in range(17, -1, -1):
            bit = (num >> i) & 1
            if bit not in node:
                node[bit] = {"count": 0}
            node = node[bit]
            node["count"] += val

    def query(self, num: int) -> int:
        ans = 0
        node = self.root
        for i in range(17, -1, -1):
            bit = (num >> i) & 1
            target_bit = bit ^ 1
            if target_bit in node and node[target_bit]["count"]:
                ans += 1 << i
                node = node[target_bit]
            else:
                node = node[bit]
        return ans

class Solution:

    def maxGeneticDifference(self, parents: List[int], queries: List[List[int]]) -> List[int]:
        n = len(parents)
        ans = [0] * len(queries)
        root_val = -1
        tree = [[] for _ in range(n)]
        node_to_queries = {}
        trie = Trie()

        for i in range(n):
            if parents[i] == -1:
                root_val = i
            else:
                tree[parents[i]].append(i)

        for i in range(len(queries)):
            node, val = queries[i]
            if node not in node_to_queries:
                node_to_queries[node] = []
            node_to_queries[node].append((i, val))

        def dfs(node: int, trie: Trie, tree: List[List[int]], node_to_queries: dict, ans: List[int]) -> None:
            trie.update(node, 1)

            if node in node_to_queries:
                for i, val in node_to_queries[node]:
                    ans[i] = trie.query(val)

            for child in tree[node]:
                dfs(child, trie, tree, node_to_queries, ans)

            trie.update(node, -1)

        dfs(root_val, trie, tree, node_to_queries, ans)
        return ans

# Test
parents = [3, 3, 0, -1]
queries = [[0, 1], [2, 6], [2, 2], [3, 7]]
sol = Solution()
print(sol.maxGeneticDifference(parents, queries))  # Output should be [1, 4, 0, 4]
```

JavaScript Solution

```

javascript
class Trie {

    constructor() {
        this.root = {};
    }

    update(num, val) {
        let node = this.root;
        for (let i = 17; i >= 0; --i) {
            const bit = (num >> i) & 1;
            if (!(bit in node)) {
                node[bit] = { count: 0 };
            }
            node = node[bit];
            node.count += val;
        }
    }

    query(num) {
        let ans = 0;
        let node = this.root;
        for (let i = 17; i >= 0; --i) {
            const bit = (num >> i) & 1;
            const target_bit = bit ^ 1;
            if (target_bit in node && node[target_bit].count) {
                ans += 1 << i;
                node = node[target_bit];
            } else {
                node = node[bit];
            }
        }
        return ans;
    }

}

function maxGeneticDifference(parents, queries) {
    const n = parents.length;
    const ans = Array(queries.length).fill(0);
    let root_val = -1;
    const tree = Array.from({ length: n }, () => []);
    const node_to_queries = {};
    const trie = new Trie();

    for (let i = 0; i < n; ++i) {
        if (parents[i] === -1) {
            root_val = i;
        } else {
            tree[parents[i]].push(i);
        }
    }

    for (let i = 0; i < queries.length; ++i) {
        const [node, val] = queries[i];
        if (!(node in node_to_queries)) {
            node_to_queries[node] = [];
        }
        node_to_queries[node].push([i, val]);
    }

    function dfs(node, trie, tree, node_to_queries, ans) {
        trie.update(node, 1);

        if (node in node_to_queries) {
            for (const [i, val] of node_to_queries[node]) {
                ans[i] = trie.query(val);
            }
        }

        for (const child of tree[node]) {
            dfs(child, trie, tree, node_to_queries, ans);
        }

        trie.update(node, -1);
    }

    dfs(root_val, trie, tree, node_to_queries, ans);
    return ans;
}

Parents = [3, 3, 0, -1]
queries = [[0, 1], [2, 6], [2, 2], [3, 7]]
console.log(maxGeneticDifference(parents, queries));  // Output should be [1, 4, 0, 4]
```

All three solutions provided are for the problem as originally described using the appropriate Trie data structure and depth-first search traversal in C++, Python, and JavaScript.

Got a question? Ask the Teaching Assistant anything you don't understand.