The Algorithm of Heavy Path Decomposition Based on Segment Tree

Jinpeng Xu[1], Weixing Zhang[2]

[1]School of Software of Zheng Zhou University, Zheng Zhou 450000, Henan Province, China;

[2]School of Software of Zheng Zhou University, Zheng Zhou 450000, Henan Province, China;

Keywords: Heavy Path Decomposition; Segment Tree; data structure; algorithm

Abstract: The time complexity or space complexity of the problem between any two nodes on the classic tree is high, and it is not suitable for the situation where modification and query coexist. The solution in this paper is to discuss a new algorithm- Heavy Path Decomposition, which has good time complexity and wide applicability.

1. Introduction

In the field of computer algorithms, we often encounter the problem of Tree structure. Among them, some types of problems are required to take two points on the tree and perform a series of operations on the path between the two points, others are required to calculate some answers by only one point.

For example, given a tree containing N nodes connected and acyclic, each node contains a value, the following operations need to be supported: the first requirement is to add a number K to the value of all the nodes in the shortest path from the X node to the Y node; The second is to find the sum of the values of all the nodes in the shortest path from X node to the Y node on the tree; The third one is to add a number K to all the node values in the subtree with the X node as the root node; The last requirement is to find the sum of all the node values in the subtree with the X node as the root node.

There are obviously many solutions to this problem. The easiest way to think about it is to scan it from beginning to end every time you do it, but one of the obvious downsides of this approach is that the time complexity is going to be $O(N^2 \times \log(N))$; This is the worst case scenario, so we need to use the appropriate algorithm to minimize the time complexity. At present, the common operation to solve the problem of increasing or decreasing the value of nodes is using "The Difference Tree". This method has a time complexity of O(N+M); For the summation problem, we can also use the method of "the Lowest Common Ancestor", which has a time complexity of O(M*log(N)+N). However, when the problems required in the above examples occur at the same time, the overall project volume will become too large to be completed in a short time. Therefore, a new method named Heavy Path Decomposition is disappearing in the International Collegiate Programming Contest.

Table 1. The difference of those algorithm.

Algorithm	**Characteristic**		
	Volume	***Time Complexity***	***Independent Solution***
Violence	Large	$N^2 \times \log(N)$	Yes
Difference Tree	Middle	N+M	No
Lowest Common Ancestor	Middle	M×log(N)+N	No
Heavy Path Decomposition	Middle	N×log(N)	Yes

  DOI: 10.25236/iwmecs.2020.008

2. Concept

2.1. Definition

Heavy Path Decomposition, as the name implies, it is the tree divided into multiple chains by dividing the light and heavy edges, so as to ensure that each node belongs to and only belongs to one chain. Then, each chain is maintained through data structures such as Segment Tree or Tree-based Array, making the path of tree in the Segment tree in order, so that the efficiency of query modification is greatly improved. There are some basic concepts:

Heavy Son: The node with the largest number of children of the parent node in the neutron tree.

Light Son: The son node of the father node except the heavy son.

Heavy Edge: An edge formed by the father node and the Heavy Son.

Light Edge: An edge formed by a father node and the Light Son.

Heavy Path: A path formed by several heavy edges.

Light Path: A path formed by several light edges.

As shown in illustration, the red edges are Heavy Edges, the chain formed by Heavy Edges and all the nodes connected with the red sides are Heavy Sons, and root node named “0” is the chain head. And it is important to note that there is not necessarily just one Heavy Edge, every subtree has at least one Heavy Edge.

2.2. Properties

1. A tree chain must be the beginning of a Light Son or the root node, with Heavy Edges strung together to form some Heavy Sons.

2. A non-leaf node has only one Heavy Son.

3. A single leaf node is also a tree chain (by property 1).

4. A tree can be divided into several chains (by property 3).

5. If edge (u, v) is a Light Edge, size(v) < size(u) / 2.

Proof: If the size(v) > size(u) / 2, the number of nodes of the subtree with y node as the root must be the large among the sons of u node, which is contradiction with v node as the son node of u node. Therefore, size(v) < size(u) / 2.

6. The number of Light Edges on the path from the root node to any node on the tree does not exceed log(N).

Proof: According to the fifth property, when you go down from the root node, the number of nodes in the subtree with the root node as the root is reduced by at least half for each Light Edge, so the number of Light Edges on the path from the root node to any node on the tree doesn’t exceed O(log(N)).

7. The time complexity of the Heavy Path Decomposition is O (N × log(N)).

3. Implementation Procedure

3.1. The First Pretreatment

In the first pre-processing, we will use Depth First Search to figure out the size of the subtree where each node is located, find its Heavy Sons, and record the father of the node and the depth of this node in the whole tree.

The problem is that there can be a lot of path processing for a node. Therefore, it is impossible to satisfy that the id from the top to the bottom of each chain is increased in turn, only part of the chain can be selected to form a continuous id. Therefore, we are going to try to pick the chain as long as possible so that we can have more points to deal with. It is means that every time using Depth First Search to a node, we mark its son with the largest number of nodes as a Heavy Son.

This is the pseudo-code for this step.

Algorithm 1: The First Pretreatment

```
DFS1(id, fa)    // the number of local node, the number of father node
 1:   father[id] ← fa
 2:   depth[id] ← depth[father[fa]] + 1
 3:   size[id] ← 1
 4:   for i ← 0 to graph[id].size() – 1 do
 5:       v ← graph[id][i]
 6:       if v != fa then
 7:           DFS1 (v, id)
 8:           size[id] ← size[id] + size[v]
 9:           if size[v] > size[son[id]] then
10:               son[id] = v
```

Algorithm 1. The First Pretreatment

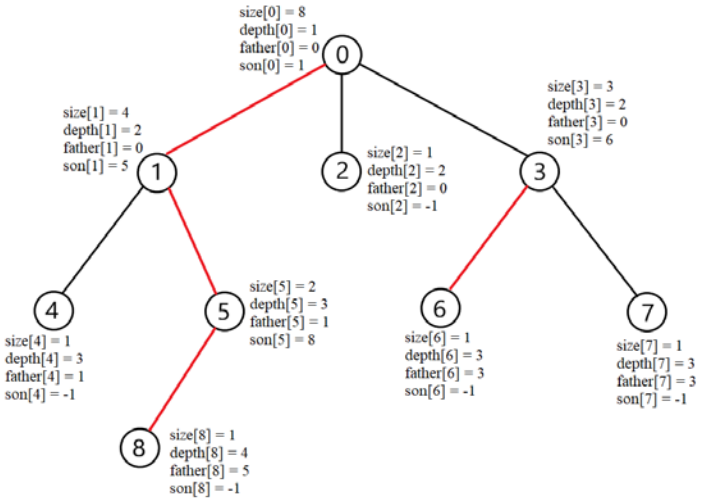

Figure 2. An Example of First Pretreatment

3.2. The Second Pretreatment

Algorithm 2: The Second Pretreatment

```
DFS2(id, t)    // the number of local node, the node at the top of Heavy Path
 1:   ids[id] ← ++ans
 2:   top[id] ← t
 3:   in[ans] ← id
 4:   if son[id] == 0 then
 5:       return
 6:   for i ← 0 to graph[id].size() – 1 do
 7:       v ← graph[id][i]
 8:       if v == father[id] || v == son[id] then
 9:           continue
10:       DFS2 (v, v)
```

Algorithm 2. The Second Pretreatment

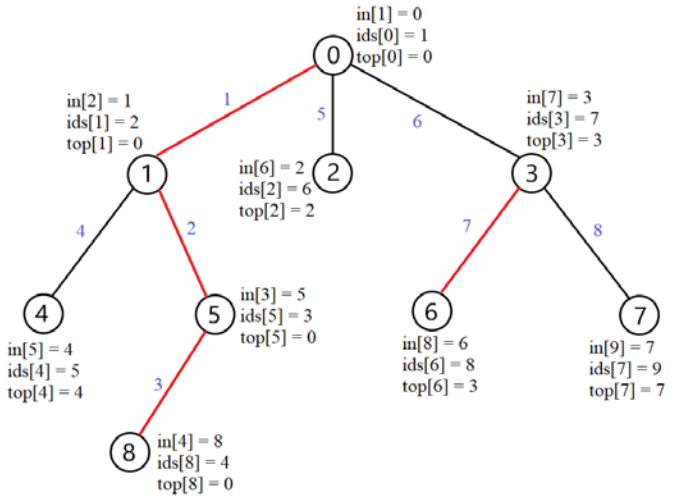

Figure 3. The example in Fig.2 of Second Pretreatment

The second pre-processing is also to use Depth First Search to take the root node as the starting node and extends downward to build the heavy Path. Select the root of the largest subtree to inherit the current Heavy Path. The rest of the nodes take that node as a starting node and pull a Heavy Path down again. At the same time, each node is assigned a position number, and each heavy Path is equivalent to a segment, which is maintained by data structure liking the Segment Tree. Put all the Heavy Paths end to end, into the same data structure, and maintain the whole.

3.3. Build and Update Segment Tree

The third step is to construct the part that creates and updates the Segment Tree. A Segment Tree is a binary tree that divides an interval into "[i, i +1]" cell intervals, and each cell interval corresponds to a leaf node in the Segment Tree. Each node uses a variable named count to record the number of lines that cover that node. So, the time required for each change is O(log^2(n)).

At Frist, constructing a Segment tree to store the tree chain. The array "n" represents the values of each node in the input tree in order. The "tree[id].left" and "tree[id].right" represent the left and right subtree ranges of the current node of the Segment Tree.

Then, we can construct the function needed to update the line segment tree each time, which can be used to modify the data of tree or query the segment tree.

Algorithm 3: Build Segment Tree

```
BUILD_TREE (id, left, right)
 1:  tree[id].left ← left
 2:  tree[id].right ← right
 3:  tree[id].cnt ← 0
 4:  if left == right then
 5:      tree[id].sum = n[in[id]]
 6:      return
 7:  mid ← (left + right) >> 1
 8:  BUILD_TREE (id << 1, left, mid)
 9:  BUILD_TREE (id << 1 | 1, mid + 1, right)
10:  sum1 ← tree[tree[id].left].sum + tree[tree[id].right].sum
11:  sum2 ← tree[id].lazy * (tree[id].right - tree[id].left + 1)
12:  tree[id].sum ← sum1 + sum2
```

Algorithm 4: Update Segment Tree

```
PUSHUP (id)
    len1 ← tree[id<<1].right – tree[id<<1].left + 1
    len2 ← tree[id<<1|1].right – tree[id<<1|1].left + 1
    tree[id<<1].sum ← tree[id<<1].sum + tree[id<<1].lazy*len1
    tree[id<<1|1].sum ← tree[id<<1|1].sum + tree[id<<1|1].lazy*len2
    tree[id<<1].lazy ← tree[id].lazy + tree[id<<1].lazy
    tree[id<<1|1].lazy ←tree[id].lazy + tree[id<<1|1].lazy

UPDATE (id, left, right, value)
    if tree[id].left == left and tree[id].right == right then
        tree[id].sum ← tree[id].sum+(tree[id].right–tree[id].left+1)*value
        tree[id].lazy ← tree[id].lazy + value
        return
    if tree[id].lazy > 0 then
        PUSHUP (id)
    mid ← tree[id].left + tree[id].right >> 1
    if right ≤ mid then
        UPDATE (id << 1, left, right, value)
    if left > mid then
        UPDATE (id << 1 | 1, left, right, value)
    UPDATE (id << 1, left, mid, value)
    UPDATE (id << 1 | 1, mid + 1, right, value)
    tree[id].sum ← tree[tree[id].left].sum + tree[tree[id].right].sum
```

Algorithm 5: QUERY Segment Tree

```
QUERY (id, left, right)
    if tree[id].left ≥ left and tree[id].right ≤ right then
        return tree[id].sum
    if tree[id].lazy > 0 then
        PUSHUP (id)
    mid ← tree[id].left + tree[id].right >> 1
    val ← 0
    if right ≤ mid then
        val ← val + QUERY(id << 1, left, right)
    if left > mid then
        val ← val + QUERY(id << 1 | 1, left, right)
    return val
```

Algorithm 3-5. Build and Maintain Segment Tree

3.4. Modify Data of nodes from X node to Y node

The general idea is to simulate X node and Y node going up, maintaining path information as they go, until X node and Y node are on the same Heavy Edge.

Specific functions are implemented as follows:

1. Set X node as X node and Y node as the node with a greater depth on the Heavy Edge. If not, exchange each.

2. Perform interval modification by adding data K to all nodes between in[X] node and X node. Note that in[top[x]] ≤ in[x] because it recurses from top to bottom.

3. Assign the data of father[top[X]] node to X node, which is equivalent to simulate X node walking through the whole current heavy Edge to reach the bottom of the previous Heavy Edge.

4. Perform 1-3 steps until X node and Y node are on the same Heavy Edge.

5. At this point, let X node be the node with lower depth in X node an Y node; Therefore, in[X] ≤ in[Y], add data K to all nodes between X node and Y node.

Algorithm 6: Modify Data of nodes from X node to Y node

```
MODIFY (x, y, k)
1:  while top[x] != top[y] do
2:      if depth[top[x]] < depth[top[y]] then
3:          SWAP (x, y)
4:      UPDATE (1, in[top[x]], in[x], k)
5:      x ← father[top[x]]
6:  if in[x] > in[y] then
7:      SWAP (x, y)
8:  UPDATE (1, in[x], in[y], k)
```

Algorithm 6. Modify Data of nodes from X node to Y node

Algorithm 7: Sum from X node to Y node

```
SUM (x, y)
1:  sum ← 0
2:  while top[x] != top[y] do
3:      if depth[x] ≥ depth[y] then
4:          sum ←sum + QUERY (1, ids[top[x]], ids[x])
5:          x ← father[top[x]]
6:      else
7:          sum ←sum + QUERY (1, ids[top[y]], ids[y])
8:          y ← father[top[y]]
9:  if ids[x] ≤ ids[y] then
10:     sum ←sum + QUERY (1, ids[x], ids[y])
11: else
12:     sum ← sum + QUERY (1, ids[y], ids[x])
13: return sum
```

Algorithm 7. Sum from X node to Y node

3.5. Sum from X node to Y node

It is similar to the last operation: simulate X node and Y node going up, maintaining path information as they go, until X node and Y node are on the same Heavy Edge.

However, it is worth noting that when the loop ends, the two nodes are on the same Heavy Edge, but they are not the same nodes, so we must count the contributions between them. And the Lowest Common Ancestor is actually used here, and the array top is used for acceleration, because the array top can be used to jump directly to the starting node of the Heavy Edge. We do not need to worry about the Light Edge here, because the top of the node I in the Light Edge is themselves.

4. Application

Heavy Path Decomposition has a very wide range of applications, the following is a few examples of its specific application.

4.1. Package Manager

Linux users and OSX users must be familiar with the package manager. With the package manager, you can install a certain package with a single line of command, and then the package manager will help you download the package from the software source, and automatically resolve all dependencies (that is, downloading other packages that the installation of the package depends on) to complete all configuration.

As a package manager, it is necessary to solve the dependency problem between packages. If package called A depends on package called B, then before installing package A, package B must

be installed first. At the same time, if you want to uninstall the package B, you must uninstall the package A.

Obviously, Heavy Path Decomposition can easily solve this problem when you want to unload multiple software at the same time.

Therefore, for each software, the software he relies is his father, and the two are connected to get a tree chain. So for each node on the tree, a weight of “1” means installed, and “0” means not installed.

When installing the software package, it is first convert to query how many nodes from the node to the root have a weight of “0”, then we query the path from the node to the root, and use the depth path of the node to get the answer, and then update the paths are all set to “1”.

When uninstalling the software package, we first query the sum of the total weights of the subtrees with the node as the root and update all subtrees to “0”.

It is worth noting that for every node, their number must be shifted back one bit, and the number cannot be “0”.

4.2. AI Robot in Tank Battle Game

In such games, AI robots need to maintain the real-time map information that can be obtained. A good solution is to store map information in a tree structure by setting different weight. It is convenient and effective for us to use the Heavy Path Decomposition, because when we scan items or enemies, we need to change the weight on the corresponding path.

4.3. In Algorithm Competition

The following is a question in the 2019 International College Programming Competition China Xian National Invitational Programming Contest:

Ming and Hong are playing a simple game called nim game. They have n piles of stones numbered 1 to n, the i-th pile of stones has ai stones. There are n−1 bidirectional roads in total. For any two piles, there is a unique path from one to another. Then they take turns to pick stones, and each time the current player can take arbitrary number of stones from any pile. Of course, the current player should pick at least one stone. Ming always takes the lead. The one who takes the last stone wins this game. Ming and Hong are smart enough so they will make optimal operations every time.

Obviously, this problem can be transformed into a tree with n nodes, where the root is “1” and there are n-1 edges, each node has a weight. The difficulty is how to maintain the Segment Tree. Let us break it into binary and use the array named num to maintain the number of 1 corresponding to all points in the interval. For the OR operation, if the corresponding bit is 1, the corresponding bits of all points in the interval will be changed to 1, if 0, no operation; for AND operation, if the corresponding bit is 0, the corresponding bits of all points in the interval are all 0, if it is 1, then no operation. And these two operations have an overlay effect, then we can use the lazy flag add[i]=1 to indicate that the i-th bit of the interval is all 1, add[i]=-1 means that the i-th bit of the interval is all 0. During the query operation, the number modulo 2 of all the bits 1 in the interval, that is, the corresponding bits of the XOR sum of the interval, can be compared with the input t for equality.

5. Conclusion

As a new algorithm, Heavy Path Decomposition has good time complexity and wide applicability. It significantly reduces the time complexity of the problem between any two nodes on the tree chain and does not consume too much memory space resources. In this paper, the related concepts and specific examples of tree chain splitting are discussed, and finally the time required for these problems is controlled at $O(N \times \log(N))$, which effectively reduces the time complexity of such problems and lays a foundation for subsequent application promotion basis.

At least, thanks for my algorithm coach and course teacher named Weixing Zhang and my teammates of Algorithm Competition.

References

[1]Wang Xingbo,Zhou Jun. Analytic Criterion and Algorithm for the Lowest Common Ancestor of Two Neighboring Nodes in a Complete Binary Tree[J]. Energy Procedia,2011,11.

[2]Zhou, J., Lan, G., Chen, Z. et al. Fast Smallest Lowest Common Ancestor Computation Based on Stable Match. J. Comput. Sci. Technol. 28, 366–381 (2013).