Algorithm Engineering Project

May 2023

Contents

1 Problem Statement

We are going to be modifying the algorithm of Tarjan-Vishkin to incorporate Union-Find data structure as we add edges to the Graph.

We will check and compare how this works in practice with respect to the other algorithms used for biconnected components such as Tarjan's algorithm.

2 Algorithms

2.1 Tarjan's Algorithm

2.1.1 Introduction

Tarjan's algorithm uses depth-first search (DFS) to traverse the graph and find all biconnected components. The basic idea behind the algorithm is to assign a unique dfs order index to each vertex as it is visited. The algorithm also maintains an array low which stores the lowest index reachable from each vertex.These information along with the observation that an articulation points separate two biconnected components are used to find all biconnected components.

2.1.2 Algorithm

The idea is to run a depth-first search while maintaining the following information:

1. The depth of each vertex in the depth-first-search tree (once it gets visited)
2. For each vertex v, the lowest depth of neighbours of all descendants of v (including v itself) in the depth-first-search tree, called the low-point.

The depth is standard to maintain during a depth-first search. The low point of v can be computed after visiting all descendants of$v$ (i.e., just before v gets popped off the depth-first-search stack) as the minimum of the depth of v, the depth of all neighbours of v(other than the parent of v in the depth-first-search tree) and the low-point of all children of $v$in the depth-first-search tree.

The key fact is that a non root vertex v is a cut vertex (or articulation point) separating two biconnected components if and only if there is a child y of v such that $low_y \geq depth_v$.

This property can be tested once the depth-first search returned from every child of v (i.e., just before v gets popped off the depth-first-search stack), and if true, v separates the graph into different biconnected components.

2.1.3 Time Complexity

The time complexity of Tarjan's algorithm is $O(|V| + |E|)$, where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges in the graph.

2.1.4 Space Complexity

The space complexity of Tarjan's algorithm is $O(|V| + |E|)$, where $|V|$ is the number of vertices in the graph. This space complexity is due to the dfs as we need to store the edges of graph stored in adjacency list.

2.2 Tarjan-Vishkin

2.2.1 Introduction

Tarjan-Vishkin algorithm is a linear-time algorithm for finding all the biconnected components in an undirected graph. This algorithm was proposed by Robert Tarjan and Uzi Vishkin in 1984. The algorithm is an improvement over Tarjan's original algorithm and it achieves better performance on graphs for parallel implementation.
Since it uses spanning tree instead of DFS tree,the whole algorithm can be parallelised which improves its performance.

2.2.2 Algorithm

The Algorithm proceeds as follows:

1. Let `T` be any rooted spanning tree of `G`.
2. Find the preorder number of vertices according to `T`
3. Create an auxiliary graph G' where G' contains one vertex for each edge of G.
   Edges of G' are as follows:
   (a) Case 1 : Edge connecting uw to vw in `G'` whenever there is a tree edge $w \rightarrow u$ with u as the parent of w and a non tree edge $v \rightarrow w$ with $pre_v < pre_w$
   (b) Case 2 : Edge connecting uv to xw in `G'` whenever there is a tree edge $v \rightarrow u$ with u as the parent of v and a tree edge $w \rightarrow x$ with x as the parent of $w$submissions and a non tree edge vw with v and w not having an ancestor-descendant relationship in `T`.
   (c) Case 3 : Edge connecting uv to vw in `G'` whenever there is a tree edge $v \rightarrow u$ with u as the parent of v and a tree edge $w \rightarrow v$ with v as the parent of w and a non tree edge joins a descendant of w to a non-descendant of v in `T`.

   Now to construct the auxiliary graph `G'`, we see that the nodes in the graph `G'` are the edges of graph `G`. The connected components in `G'` are corresponding to the biconnected components of `G`

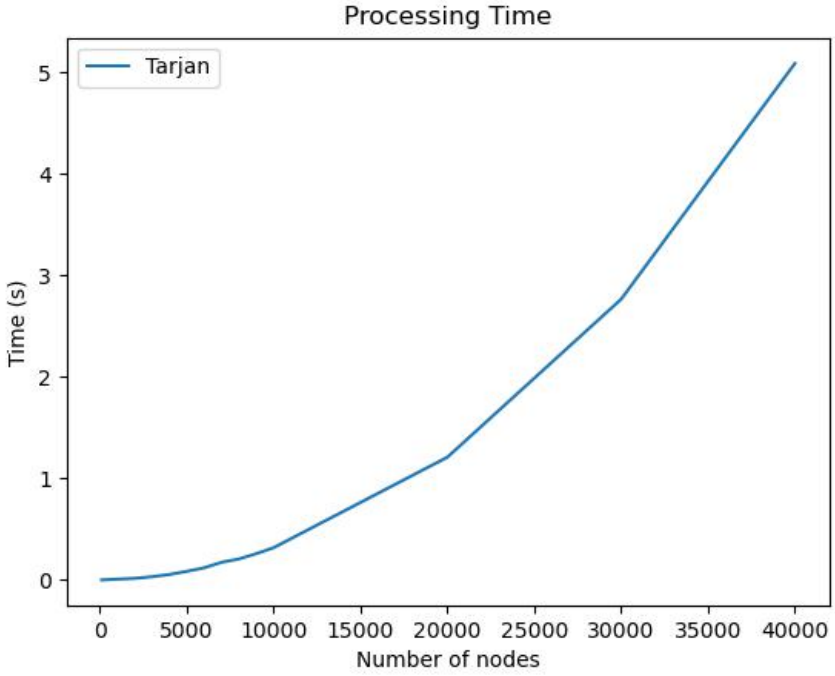
3 Benchmarking

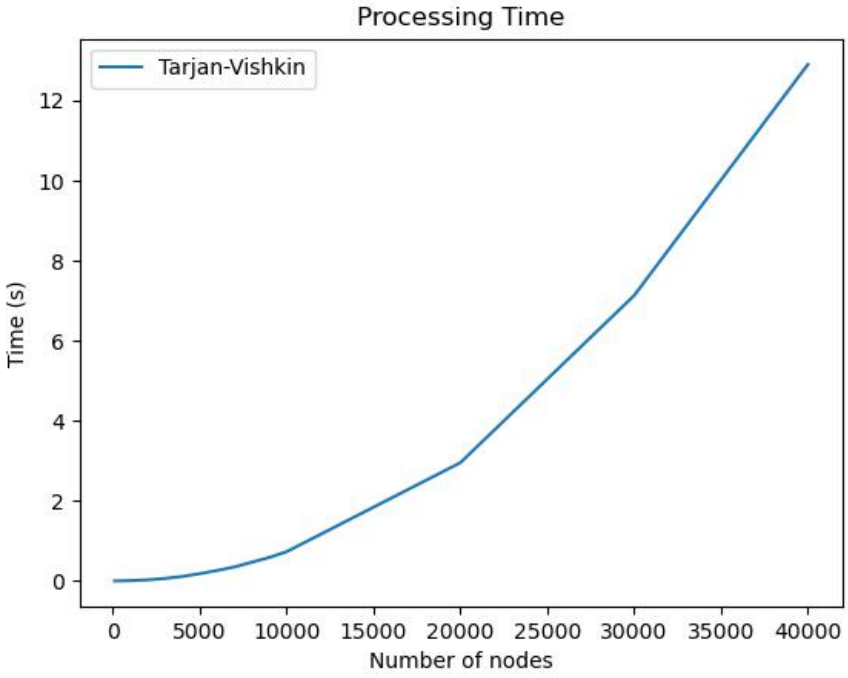
We ran both Tarjan Algorithm and Tarjan-Vishkin Algorithm to identify biconnected components for randomly generated connected graphs.

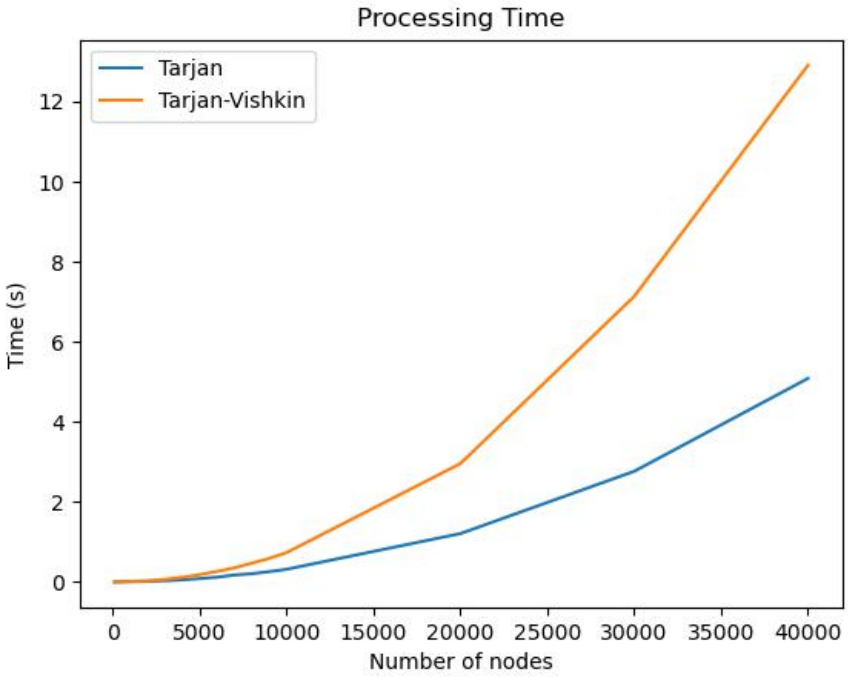
Machine : Macbook Air Apple M1 8GB

The run-times are the following:
$T_1 \rightarrow$ Tarjan's Runtime
$T_2 \rightarrow$ Tarjan-Vishkin's Runtime

Nodes	Edges	T_1	T_2
100	505	0.000	0.002 s
500	12622	0.004	0.006 s
1000	50366	0.008	0.012 s
2000	200287	0.016	0.030 s
3000	450341	0.032	0.066 s
4000	800705	0.054	0.116 s
5000	1252736	0.085	0.184 s
6000	1801436	0.120	0.264 s
7000	2451648	0.174	0.353 s
8000	3201347	0.207	0.471 s
9000	4052086	0.259	0.589 s
10000	5002986	0.318	0.734 s
20000	20007717	1.209	2.957 s
30000	45002838	2.764	7.133 s
40000	80006225	5.085	12.905 s


Processing Time
Tarjan
Time (s)
0
1
2
3
4
5
0
5000
10000
15000
20000
25000
30000
35000
40000
Number of nodes


Processing Time
Tarjan-Vishkin
Time (s)
0
2
4
6
8
10
12
0
5000
10000
15000
20000
25000
30000
35000
40000
Number of nodes


Processing Time
Tarjan
Tarjan-Vishkin
Time (s)
0
2
4
6
8
10
12
0
5000
10000
15000
20000
25000
30000
35000
40000
Number of nodes

4 Observations

We observe that the execution time for both the algorithms increases as the size of the graph increases.This is because of the fact that Tarjan is $O(|V| + |E|)$ for V nodes and E edges.

We also observe that our implementation of Tarjan-Vishkin with DSU doesn't perform better than the standard Tarjan algorithm.This is because of the reason that Tarjan-Vishkin will be optimal for dense graphs if we use parallelisation but for non-parallel implementation of Tarjan-Vishkin ,Tarjan will perform relatively better.The better performance of Tarjan for non-parallel implementation of Tarjan-Vishkin is due to the fact that Tarjan-Vishkin uses DSU and the implementation requires additional helper functions which increases constant factor.

5 Implementation

5.1 Union-Find

```
class UnionFind
{
public :
    vector<int> par;
    vector<int> rnk;
    void init(int n){
        par.resize(n+1);
        rnk.resize(n+1);
        for(int i=0;i<=n;i++){
            par[i]=i;
            rnk[i]=0;
        }
    }
    int get(int x){
        if(x==par[x]){
            return x;
        }
        else{
            return par[x]=get(par[x]);
        }
    }
    int merge(int x,int y){
        int px = get(x);
        int py = get(y);
        if(px==py)
            return 0;
        if(rnk[px]<rnk[py]){
            par[px]=py;
        }
        else if(rnk[px]>rnk[py]){
            par[py]=px;
        }
        else{
            par[px]=py;
            rnk[py]++;
        }
        return 1;
    }
};
```

5.2 Finding Low Time

Here we parallelize the finding of Low-Time to compute efficiently.

```
void initLowtime(int n){
    lowTimeMin.resize(n+1);
    lowTimeMax.resize(n+1);

    for(int i=1;i<=n;i++){
        lowTimeMin[i]=dfsorder[i];
        lowTimeMax[i]=dfsorder[i];
    }

    // Using multiple threads here
    auto lambda_func_parallel = [&](int l, int r){
        for(int i=l;i<=r;i++){
            auto A=edges[i];

            if(nontree[A.second]){
                int u = A.first.first;
                int v = A.first.second;
                int l = getLCA(u,v);
                lcaStore[A.second]=l;

                if(l==u or l==v){
                    if(depth[u]>depth[v]){
                        swap(u,v);
                    }

                    lowTimeMin[v]=min(lowTimeMin[v],dfsorder[u]);
                    lowTimeMax[v]=max(lowTimeMax[v],dfsorder[u]);
                }
                else{
                    lowTimeMin[u]=min(lowTimeMin[u],dfsorder[v]);
                    lowTimeMax[u]=max(lowTimeMax[u],dfsorder[v]);
                    lowTimeMin[v]=min(lowTimeMin[v],dfsorder[u]);
                    lowTimeMax[v]=max(lowTimeMax[v],dfsorder[u]);
                }
            }
        }
    };
```

```
    // Initialising the threads
    thread t[1000];
    int nth=50;

    for(int i=0;i<nth;i++){
        t[i]=std::thread(
                lambda_func_parallel,
                i*edges.size()/nth,
                (i+1)*edges.size()/nth-1);
    }

    for(int i=0;i<nth;i++){
        t[i].join();
    }

    function<void(int)> dfsLowTime = [&](int u){
        for (auto v : tree[u]){
            dfsLowTime(v.first);
            lowTimeMin[u]=min(lowTimeMin[u],lowTimeMin[v.first]);
            lowTimeMax[u]=max(lowTimeMax[u],lowTimeMax[v.first]);
        }
    };

    dfsLowTime(1);
    return;
}
```

5.3 Lowest Common Ancestor

```
void initLCA(int n){
    int lg = log2(n)+1;
    par.resize(n+3);
    LG = lg+1;

    for(int i = 0; i <= n; i++){
        par[i].resize(lg+3);
    }
    par[1][0]=0;

    for(int i = 1; i <= n; i++){
        for(auto A:tree[i]){
            par[A.first][0] = i;
        }
    }

    for(int i = 1; i < LG; i++){
        for(int j = 1; j <= n; j++){
            par[j][i] = par[par[j][i-1]][i-1];
        }
    }
    return;
}

int getLCA(int u, int v){
    if (depth[u] < depth[v])
        swap(u, v);

    for (int i = LG-1; i >= 0; i--){
        if (depth[par[u][i]] >= depth[v]){
            u = par[u][i];
        }
    }

    if (u == v) return u;

    for (int i = LG-1; i >= 0; i--){
        if (par[u][i] != par[v][i]){
            u = par[u][i];
            v = par[v][i];
        }
    }
    return par[u][0];
}
```

5.4 Finding Bi-connected Components

```
void findBiconnectedComponents(int n) {
    UnionFind dsu;
    dsu.init(edges.size());

    for (auto A: edges) {
        if (nontree[A.second]) {
            int u = A.first.first;
            int v = A.first.second;

            int l = lcaStore[A.second];
            if (l == u or l == v) {
                if (l == v) swap(u, v);
                dsu.par[A.second] = dsu.get(parentEdgeSpanningTree[v]);
            }
            else {
                dsu.merge(parentEdgeSpanningTree[u], parentEdgeSpanningTree[v]);
                if (dfsorder[u] > dfsorder[v]) {
                    dsu.merge(parentEdgeSpanningTree[u], A.second);
                } else {
                    dsu.merge(parentEdgeSpanningTree[v], A.second);
                }
            }
        }
        else {
            int u = A.first.first;
            int v = A.first.second;

            if (depth[u] > depth[v])
                swap(u, v);
            if (depth[u] == 0) continue;

            int w = par[u][0];
            if (!(dfsorder[u] <= lowTimeMin[v] &&
                lowTimeMax[v] <= dfsend[u])) {

                dsu.merge(
                    parentEdgeSpanningTree[v],
                    parentEdgeSpanningTree[u]
                );
            }
        }
    }
    return;
}
```