Report

Schmidt’s Algorithm

Schmidt’s algorithm, is a self-contained linear time algorithm that can be used to test the bi-connectivity of a graph. The algorithm, like most in this field, is based on a Depth First Search (DFS) traversal, followed by basic path based approach based on the results of the search, to obtain the chain decomposition of the graph, commenting on the bi-connectivity of the graph. The algorithm’s methodology and the implementation details have been discussed below.

Methodology

The algorithm consists of three steps, as highlighted below -

DFS

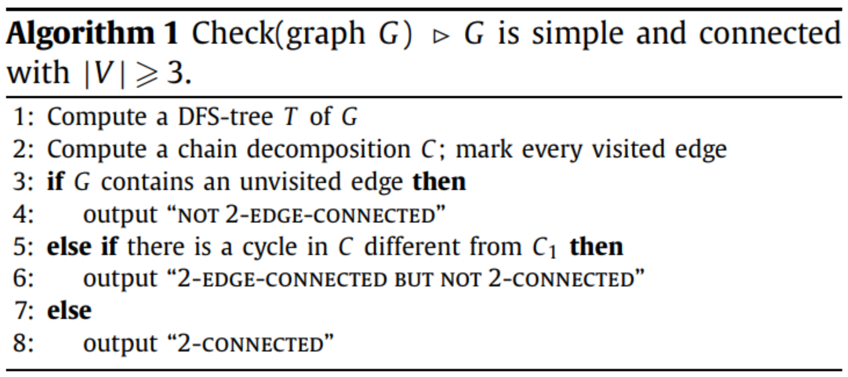
Firstly, a rooted DFS tree of the graph is being computed, assuming anyone of the nodes as the root. During this process, the parent of each node is being stored and simultaneously, the nodes are being ordered according to their discovery times in the DFS.

Chain Decomposition

Chain decomposition of the graph is being calculated, the concept behind which was referred in the research paper attached at the end of the report. Basically, we search for the back-edges in the rooted tree, of the nodes in increasing order of discovery time. Then, we trace a path through this back=edge and tree to obtain a chain. A chain terminates as soon as a visited node is reached, of which an array is being maintained earlier. This is done till there are no more traversable paths, i.e., the back-edges have exhausted. Using this technique all chains are being generated for the graph and the rooted tree.

Commenting on bi-connectivity

The final comment on bi-connectivity is made by using the below algorithm, taken directly from the research paper.



Algorithm 1 Check(graph G) $\triangleright$ G is simple and connected with $|V| \geqslant 3$.

```
1: Compute a DFS-tree $T$ of $G$
2: Compute a chain decomposition $C$; mark every visited edge
3: if $G$ contains an unvisited edge then
4:     output "NOT 2-EDGE-CONNECTED"
5: else if there is a cycle in $C$ different from $C_1$ then
6:     output "2-EDGE-CONNECTED BUT NOT 2-CONNECTED"
7: else
8:     output "2-CONNECTED"
```

Here, $C_1, C_2.....C_n$ are the chains obtained during the decomposition.

It is a direct consequence of the following theorems -

1. Let C be a chain decomposition of a simple connected graph G. Then G is 2-edge-connected if and only if the
chains in C partition E
2. Let C be a chain decomposition of a simple 2-edge connected graph G. Then G is 2-connected if and only if C1 is
the only cycle in C

The prove of the theorems is fairly obvious with respect to ear decomposition and bi-connectivity, discussed in the class.

Implementation

The implementation of the algorithm has been done in the following steps, as one can see from the snippets attached and explained

Input

To take the input, we take in the file having the graph, and after iterating through the comments and ignoring them, we take input to get the number of nodes, edges and the edges itself. During this, and adjacency list is being created storing the complete graph as shown.

```
ios_base::sync_with_stdio(false);
cin.tie(0);

int k;
int n, m;
char fname[100], graph_type[250];
string line, pname;
vector<string> nbrs;
vector<vector<ll>> adj;

if (argc != 2)
{
  printf("\nInappropriate arguments! \n\nExpected <graph_name> as input!!\n");
  return 1;
}

strcpy(fname, argv[1]);
ifstream fin(fname);

nbrs = split(string(fname), '/');
pname = nbrs[nbrs.size() - 1];

k = 0;
bool take_edges_now = false;

while (getline(fin, line))
{
  if (!k)
  {
    strcpy(graph_type, line.c_str());
    k = 1;
  }

  if (line[0] == '%')
    continue;

  nbrs = split(line, ' ');
  int a = atoi(nbrs[0].c_str()), b = atoi(nbrs.back().c_str());
  if (!take_edges_now)
  {
    n = a, m = b;
    take_edges_now = true;
    adj.resize(n);
  }
  else
  {
    if (nbrs.size() == 3)
      b = atoi(nbrs[1].c_str());
    a--;
    b--;
    if (a == b)
      continue;
    adj[a].push_back(b);
    adj[b].push_back(a);
  }
}
```

DFS

Next, a very standard implementation of the DFS algorithm has been done, to get the rooted DFS tree, simultaneously, we check for the graph being connected, rejecting the possibility of bi-connectivity immediately if found to be disconnected.

```
void dfs(vector<vector<ll>> &adj, vector<bool> &dfs_visited, vector<ll> &dfs_queue, vector
<ll> &dfs_parent, ll u)
{
  dfs_visited[u] = true;
  dfs_queue.push_back(u);

  for (auto v : adj[u])
  {
    if (!dfs_visited[v])
    {
      dfs_parent[v] = u;
      dfs(adj, dfs_visited, dfs_queue, dfs_parent, v);
    }
  }
}
```

```
  ll dfs_root = 0;
  vector<bool> dfs_visited(n, false);
  vector<ll> dfs_queue;
  vector<ll> dfs_parent(n, -1);
  dfs(adj, dfs_visited, dfs_queue, dfs_parent, dfs_root);

  if (!check_nodes_visited(dfs_visited))
  {
    cout << "NO" << endl;
    return 0;
  }
```

Chain Decomposition and commenting

Finally, the chain decomposition is being computed, side by side checking for the chain to be a cycle and maintaining a total count for the number of cycles found. This is done by simply computing the ear decomposition, by exploring non - tree edges, or in cases of ascendant - descendant relationships as shown in the snippet below. Finally, we just check whether all nodes are visited and number of cycles found is equal to 1 only. The cyclicity is checked by just checking whether the ending node of a chain is not the parent.

```
void ear_decomposition(vector<bool> &ear_visited, vector<ll> &dfs_queue, vector<ll> dfs_pa
rent, vector<vector<ll>> &adj)
{
  for (auto u : dfs_queue)
  {
    for (auto v : adj[u])
    {
      if (dfs_parent[u] == v || dfs_parent[v] == u)
      {
        continue;
      }

      ear_visited[u] = true;
      bool is_cycle = false;

      for (ll x = v;; x = dfs_parent[x])
      {
        if (x == u)
          is_cycle = true;

        if (ear_visited[x])
          break;
        ear_visited[x] = true;
      }

      cycle_chain_count += is_cycle;
    }
  }
}
```

```
vector<vector<ll>> ears;
  vector<bool> ear_visited(n, false);

  ear_decomposition(ear_visited, dfs_queue, dfs_parent, adj);

  if (check_nodes_visited(ear_visited) && cycle_chain_count == 1)
  {
    cout << "YES" << endl;
  }
  else
  {
    cout << "NO" << endl;
  }
```

Tarjan’s Algorithm

Tarjan's algorithm is a popular algorithm used to identify articulation points in a graph. It is a depth-first search-based algorithm that works by assigning a unique discovery time and low value to each vertex in the graph. An articulation point is then identified if and only if removing it from the graph causes the graph to become disconnected or increase the number of connected components. Tarjan's algorithm has a time complexity of O(V + E), making it efficient for identifying articulation points in large graphs.

If and only if the algorithm doesn’t identify any articulation points, the code outputs that the graph is biconnected.

Methodology

The algorithm requires a single DFS, with some additional bookkeeping to maintain ‘low’ values for each node. In the resulting DFS tree, the low value of node p is the least discovery time t such that there exists a node in the subtree of p which has a back-edge (non-tree edge) connecting it to the node with discovery time t.

In other words, the low value of a node p is the earliest discovery time of any node that can be reached from that p through a path that does not include p’s parent. The low value for each node is computed using the low values of its children, and this process is carried out recursively by the relation:

$$\mathrm{low}[p] = \min(\mathrm{tim}[p], \min_{c \text{ is a child of } p} \mathrm{tim}[c])$$

A non-root node p is an articulation point if and only if it has a child c such that $\mathrm{low}[c] \geq \mathrm{tim}[p]$. In case of the root, it is an articulation point if it has more than one children.

Implementation

Input

We use the same implementation (as in Schmidt’s) to take input from a file.

Relevant Computed Data / Variables

During the DFS, we maintain the following information.

- $\text{vis}[i]$: Whether node i has been visited already. This helps identify which back-edges.
- $\text{tin}[i]$: The discovery time of node i.
- $\text{low}[i]$: The computed low value of node i.
- tim: The current time.

DFS

The DFS is implemented as a lambda function. As soon as we start out DFS for node i, we mark it to be visited. Using this, we can differentiate whether the edges encountered in the future belong to the tree or are back-edges. We also note down the discovery time of the current node in $tin[i]$. As an optimisation, we can also choose to skip noting this down in an array, and just store a local variable inside the DFS function.

We also keep track of how many children we encounter in the DFS tree. This is because, if the current node is a root, we need this information about the number of children to judge whether it is an articulation point.

We iterate over all the adjacent edges, and make updates as necessary, the details of which have been described in the methodology section.

```
vector<bool> vis(n);
vi tin(n, -1), low(n, -1);
int tim = 0;
bool biconn = true;

function<void(int, int)> dfs = [&](int v, int p) {
  if (!biconn) return;

  vis[v] = true;
  tin[v] = low[v] = tim++;

  int children = 0;

  for (auto i: adj[v]) {
    if (!biconn) return;

    if (i != p) {
      if (vis[i]) {
        low[v] = min(low[v], tin[i]);
        continue;
      }
      dfs(i, v);

      if (!biconn) return;

      low[v] = min(low[v], low[i]);

      if (low[i] >= tin[v] and p != -1) {
        biconn = false;
        return;
      }

      children++;
    }
  }

  // root should have a single child
  if (p == -1 && children > 1) biconn = false;
};

dfs(0, -1);

for_(i, 0, n) if (!vis[i]) {
  biconn = false;
  break;
}
```

Heuristic (Early Exit)

In our specific problem, our end goal isn't to identify all articulation points. We just need to know whether there exists atleast one, to be able to make a fully informed statement about the biconnectivity of the supplied graph. As such, we exit the DFS as soon as we find the first articulation point. This requires a few changes. Specifically, whenever we encounter a child which passes the condition of having the low value greater than or equal to the current node's discovery time, we set a global flag to denote that the graph is biconnected. Additionally, we stop our process of iterating over edges in the current DFS function call as well as in all ancestors.

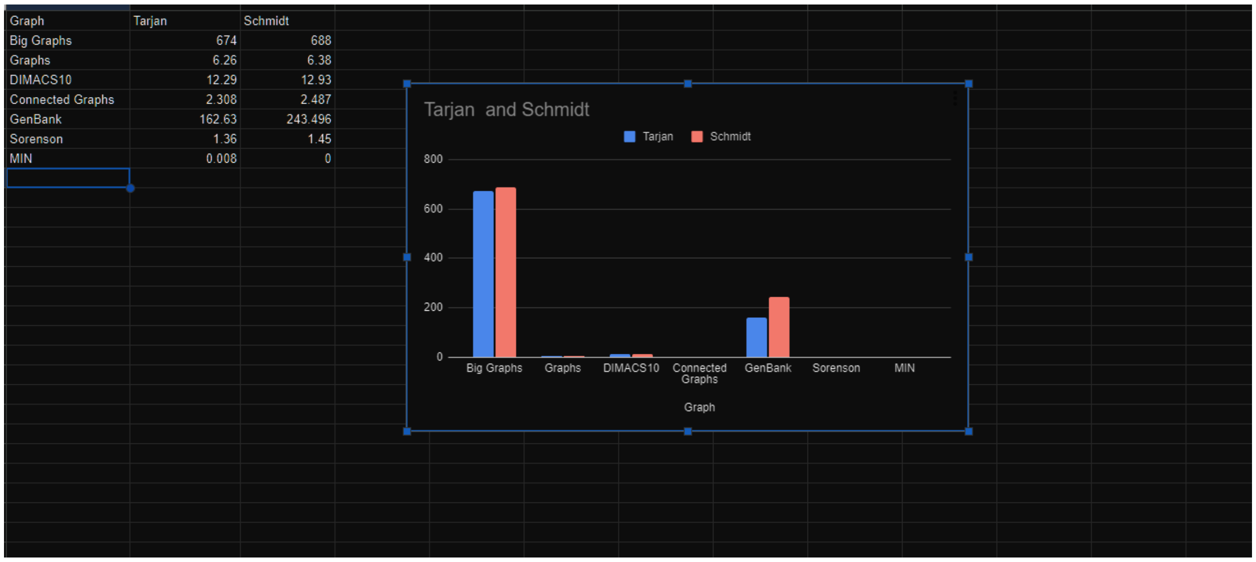
In the same vein, we only begin the DFS from a single node, and not start one for each unvisited node. Instead, we just check that is some node wasn't visited in the initial DFS, then it doesn't belong to the same connected component - which also implies that there exist atleast two biconnected components.

References

https://www.sciencedirect.com/science/article/pii/S0020019013000288

https://dl.acm.org/doi/10.1145/362248.362272

Analysis



Graph	Tarjan	Schmidt
Big Graphs	674	688
Graphs	6.26	6.38
DIMACS10	12.29	12.93
Connected Graphs	2.308	2.487
GenBank	162.63	243.496
Sorenson	1.36	1.45
MIN	0.008	0

The above graph summarises the run times of both algorithms for different set of graphs in seconds for both the algorithms.