Tarjan-Vishkin With Union Find: Introduction to Algorithms Engineering Project

By - Sriteja Pashya (2021111019)

Keval Jain (2021111030)

Project Ideas

- After applying Tarjan-Vishkin, any connected component in the auxiliary graph is a biconnected component in the original graph. By using Union-Find data structure, we can avoid adding extra edges by which the connectivity of the auxiliary graph does not change.
- At every step of adding an edge, if the two vertices happen to be in the same component, we skip adding this edge.
- We compared results for DFS,BFS spanning trees, also while selecting the root vertex randomly and through 4-sweep methods.

Implementation

- The language we used is C++ due to the data structures available in its STL.
- This implementation works for unconnected graphs
- It has been tested on library checkers

Input:

- The input graph is stored in an adjacency list of *vector<vector<int>>*
  - Although vector<list<int>> would allow for easy addition of neighbours without having to reallocate memory, it has higher overhead to access elements, and since we only have static graphs, vector<vector<int>> gives better performance

Implementation

Computation:

- This is done sequentially for each *connected component* in the original graph
- For each connected component, we store the *root node* in a vector
- This computation is done in 3 steps:
  1. First we build the *graphs for tree-edges and back-edges* and store them in a *vector<vector<int>>*
  2. Then in one DFS traversal we calculate the *preorder number, parent* and *number of descendants*
  3. Then in another DFS traversal we calculate the *low* and *high* values for each node
- Note: The graph for tree edges isn't necessarily connected because the original graph might not be connected

Implementation

Building Auxiliary Graph:

- For each connected component in the graph we run a DFS traversal
- In which for each node-- for each tree edge we check rule (ii) from the paper and for each back-edge we check rules (i) and (iii), to *add an edges* in the auxiliary graph

Add Edges Without DSU:

- We used a *map<pair<int,int>,vector<pair<int,int>>>* for the auxiliary graph, its an adjacency list, with each node representing an edge
- Therefore to add an edge we just simply add a *pair<int,int>* to the list of another node

Add Edges With DSU:

- First we initialise *each edge* in the original graph as an element in the DSU
- Then to add an edge, we call Union on the two elements in the DSU

Implementation

Finding BCCs Without DSU:

- Run DFS on the Auxiliary graph, and all the nodes in each connected component correspond to one BCC,
- To print only the nodes in each BCC, store the nodes encountered each component, in a set<int>
  - Nodes which are not connected to any other node need to printed separately. This can be done from the roots we stored

Finding BCCs With DSU:

- To print all BCCs in order, create a *temporary graph* where each parent in the DSU has a list of elements which are its children. Since each node is *pair<int,int>,* the data structure used is map<pair<int,int>,vector<pair<int,int>>> auxiliary;

Implementation

Finding BCCs With DSU:

- First add all elements in the DSU to the temporary graph
- Then for each parent iterate through all its edges to find the BCCs
- Again To print only the nodes in each BCC, store the nodes encountered each component, in a set<int>
- Nodes which are not connected to any other node need to printed separately. This can be done from the roots we stored

Performance Testing

N	M	T	TV_D_R	TV_B_R	TVUF_D_R	TVUF_B_R
1e4	1e4	0.014024	0.026523	0.032426	0.030229	0.031513
1e4	2.5e4	0.0214	0.062584	0.076521	0.063223	0.076109
1e4	5e4	0.032649	0.123767	0.156185	0.121554	0.147609
1e4	7.5e4	0.042312	0.177779	0.23068	0.173659	0.209432
1e4	1e5	0.053944	0.240193	0.315788	0.236872	0.27015
1e4	2.5e5	0.104638	0.656884	1.07537	0.587502	0.67222
1e4	5e5	0.197214	1.39223	1.83952	1.21525	1.29294
1e4	7.5e5	0.286889	2.37697	2.92697	1.79901	1.98766
1e4	1e6	0.362357	3.41079	4.5328	2.66425	2.77235
1e4	2.5e6	0.890946	9.02061	11.3013	6.55929	6.93398
1e4	5e6	1.78095	19.4747	23.5644	13.7968	13.8937
1e5	1e4	0.028866	0.042806	0.04905	0.039524	1.09897
1e5	1e5	0.095954	0.316961	0.326603	0.328922	0.520348
1e5	1e6	0.711118	4.44724	4.65767	3.28888	3.69083
1e5	1e7	6.14714	66.9624	69.4253	37.6406	37.48
1e5	1e8	70.4698	1151.61	1180.422	454.62	420.665

Notation Used -

General:
N - Number of vertices, M - Number of edges

Algorithm Used:

T - Tarjan, TV - Tarjan Vishkin, TVUF - Tarjan Vishkin with Union Find

Tree Used:

_D - DFS Tree, _B - BFS Tree

Rooted Vertex:

_R - Random rooted vertex, _4S - 4-sweep rooted vertex

General Syntax: AlgoUsed_TreeUsed_RootedVertex

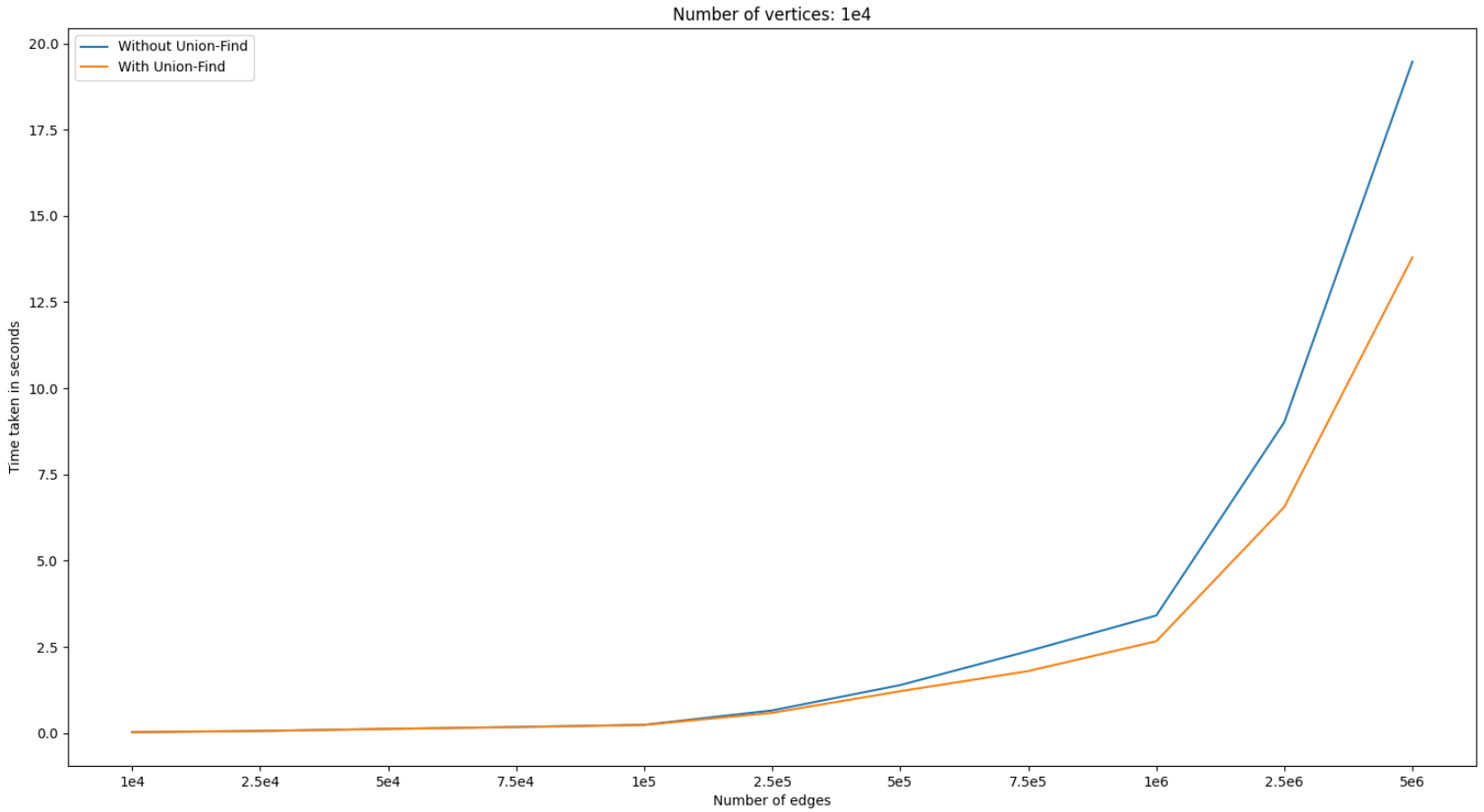
The values in the columns are program duration times in seconds.

- These graphs have been self-generated randomly. The aim is to get a good idea on how the algorithms compare.

Case Study 1: Without Union Find vs With Union Find

- On Sparse Graphs
  - We observed that Tarjan-Vishkin with Union Find is slower for sparse graphs. This is expected as we are not skipping many edges and the cost of maintaining the Union Find data structure would exceed any benefits of such skips.
- On Dense Graphs
  - We observed that Tarjan-Vishkin with Union Find is significantly faster for dense graphs. Many edges are being skipped from being added in auxiliary graph, which overcomes any cost from Union Find.

Case Study 1: Without Union Find vs With Union Find



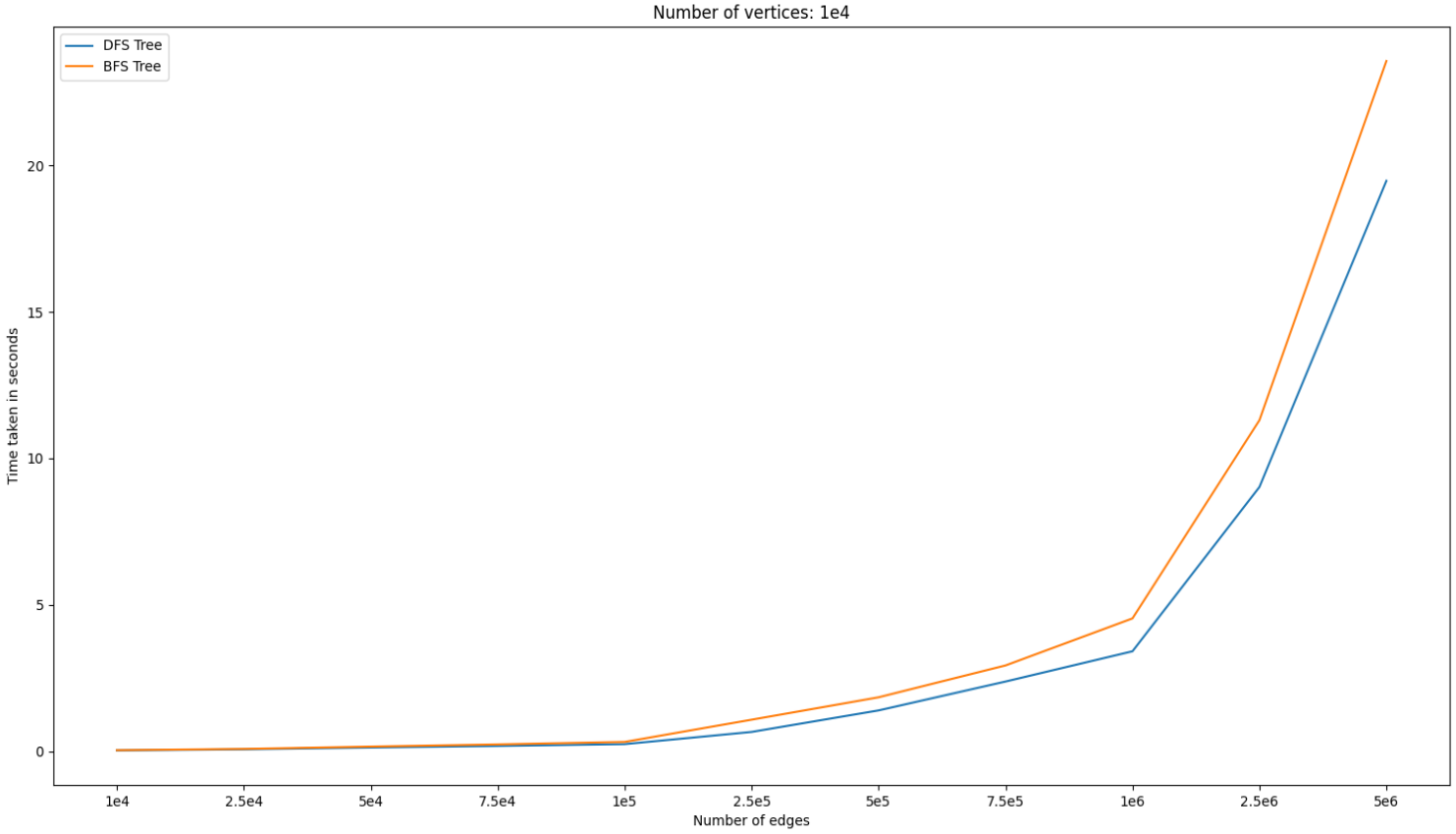
Tarjan-Vishkin without Union Find vs With Union Find for a randomly generated graph with 1e4 vertices and edge count specified in x-axis.
The time is measured in seconds.

As visible, Tarjan Vishkin with Union Find is significantly faster than Tarjan Vishkin without Union Find, on denser graphs

Case Study 2: DFS Tree vs BFS Tree

- The DFS Tree recursion could lead to stack overflow through recursive DFS calls.
- Although the stack limit can be increased, this is will have performance implications.
- We have increased our stack limit to 1GB to allow DFS tree to work on large graphs.
- The general trend can be summarised as:
  - DFS tree implementation is faster than BFS Tree implementation for Tarjan Vishkin without Union Find
  - DFS tree implementation is comparable to BFS Tree implementation for Tarjan Vishkin with Union Find

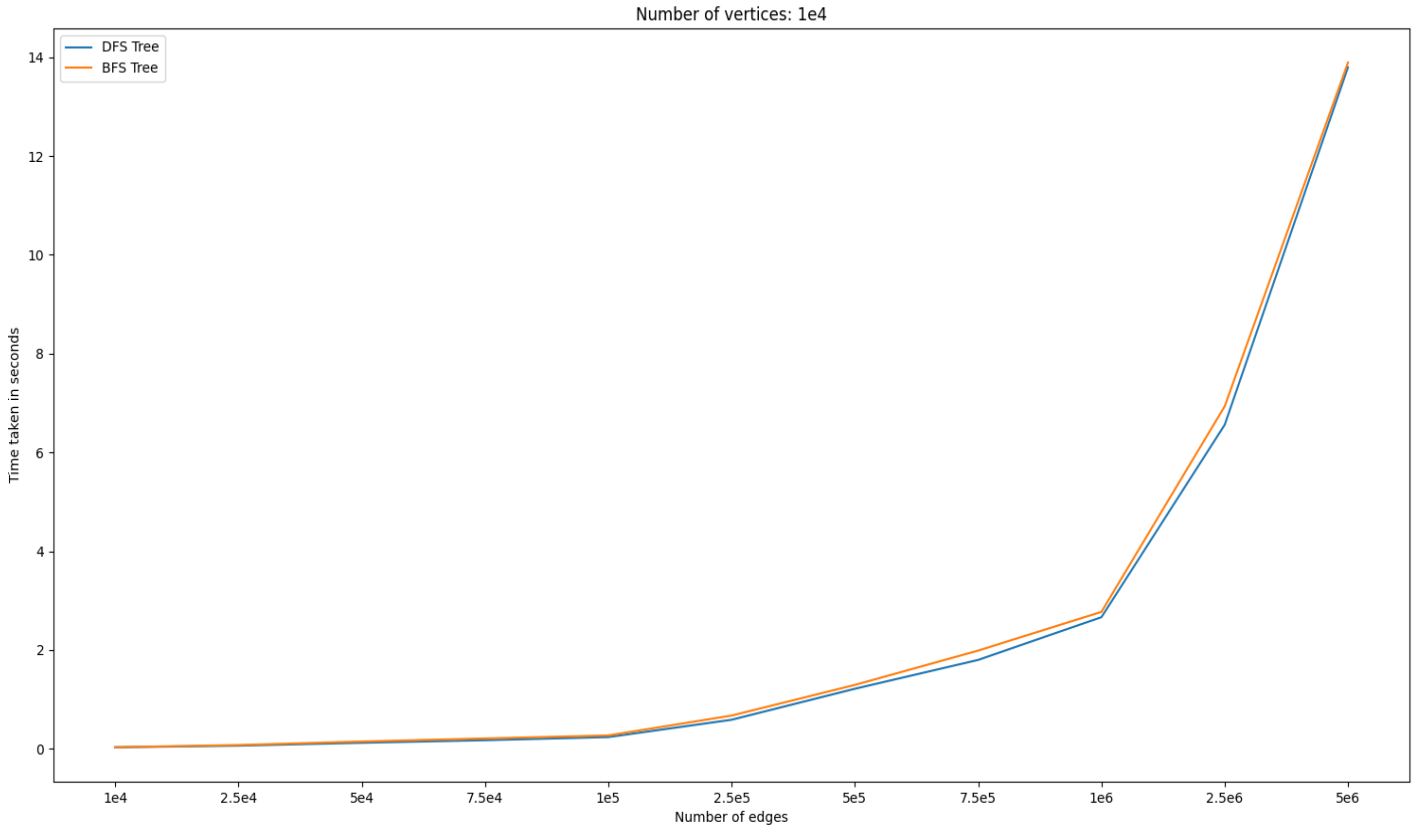
Case Study 2: DFS Tree vs BFS Tree



Tarjan Vishkin without Union Find – A DFS Tree vs BFS Tree for a graph with 1e4 vertices and edge count specified in x-axis. The time is measured in seconds.

As visible DFS tree is significantly faster than BFS tree for Tarjan-Vishkin without Union Find.

Case Study 2: DFS Tree vs BFS Tree



Tarjan Vishkin with Union Find – A DFS Tree vs BFS Tree for a graph with 1e4 vertices and edge count specified in x-axis. The time is measured in seconds.

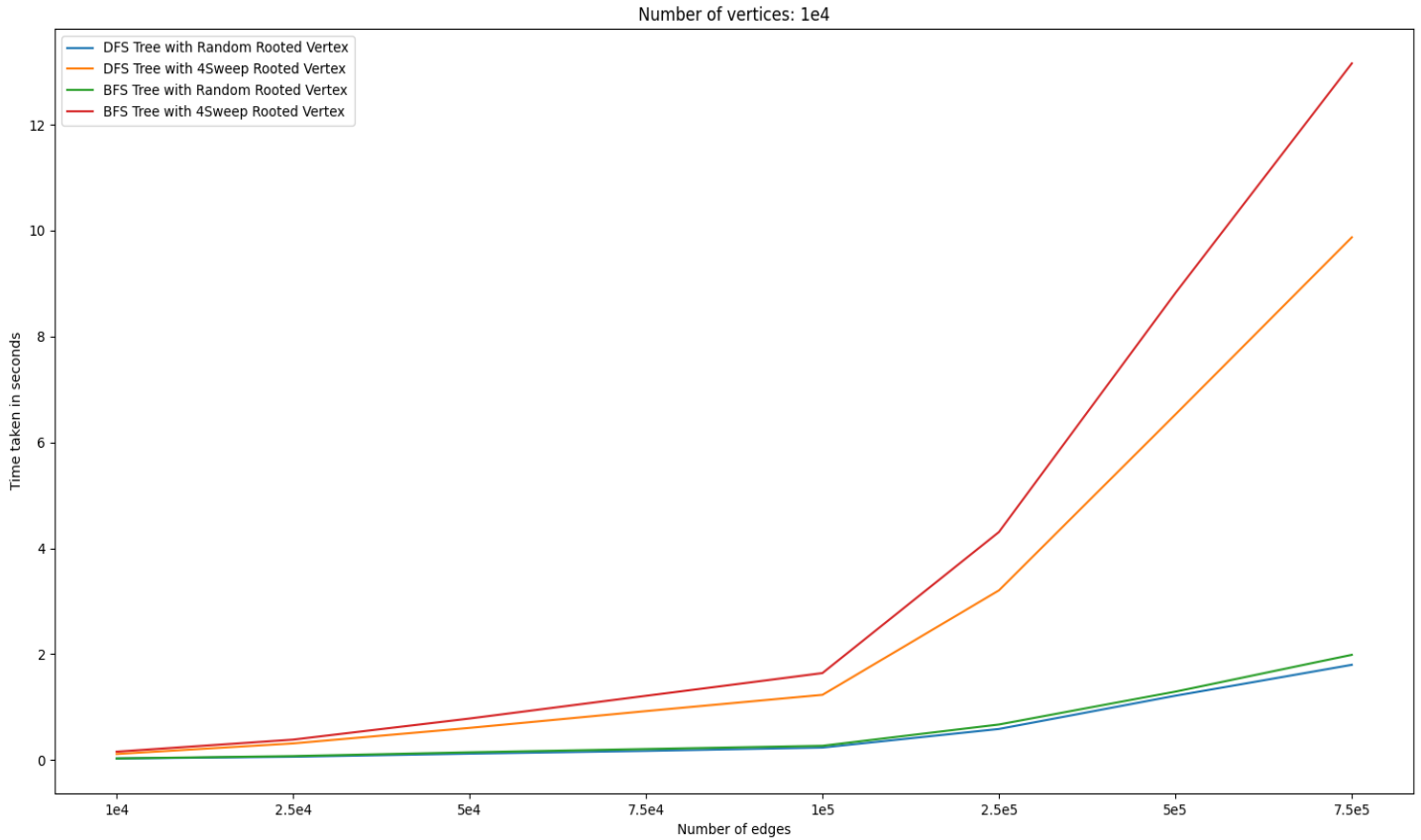
As visible there is not much difference in complexity for DFS tree vs BFS tree for Tarjan-Vishkin with Union Find.

Case Study 3: Choosing the Rooted Vertex

N	M	TVUF_D_R	TVUF_D_4S	TVUF_B_R	TVUF_B_4S
1e4	1e4	0.030229	0.115622	0.031513	0.159256
1e4	2.5e4	0.063223	0.315149	0.076109	0.386802
1e4	5e4	0.121554	0.60786	0.147609	0.784995
1e4	7.5e4	0.173659	0.924847	0.209432	1.21358
1e4	1e5	0.236872	1.23198	0.27015	1.6432
1e4	2.5e5	0.587502	3.2041	0.67222	4.30634
1e4	5e5	1.21525	6.52545	1.29294	8.81841
1e4	7.5e5	1.79901	9.87171	1.98766	13.1558
1e5	1e4	0.039524	0.240244	1.09897	18.3857
1e5	1e5	0.328922	1.32603	0.520348	4.76137
1e5	1e6	3.28888	15.4204	3.69083	20.1549
1e5	1e7	37.9112	163.415	36.8859	217.233

Conclusions Observed:
- It is clearly visible from the data, that taking a random vertex is signicantly faster then running 4 BFS to find the 4Sweep vertex for each connected component.
- The magntiude difference seems to have no affect from the number of vertices, for Ex for 1e5 vertices the ratio of DFS tree times with 4Sweep vertex : DFS tree times with random vertex are - {6.07,4.03,4.68,4.31}

Case Study 3: Choosing the Rooted Vertex



Tarjan Vishkin with Union Find – A DFS Tree vs BFS Tree each comparing a randomly chosen rooted vertex, 4 Sweeped Rooted Vertex for
a graph with
1e4 vertices and edge count specified in x-axis.
The time is measured in seconds.

As visible its much more efficient to take a random root.

Case Study 4: Comparing different steps in DFS, BFS Trees

- Tarjan Vishkin With Union Find for DFS Trees

N	M	Building tree and calculating low and high values	Building auxiliary graph	Finding connected components in auxiliary graph
1e4	1e4	0.010812 (39.61%)	0.010294 (37.71%)	0.006186 (22.66%)
1e4	5e4	0.025562 (20.87%)	0.069587 (56.83%)	0.02729 (22.28%)
1e4	1e5	0.038382 (16.35%)	0.142512 (60.71%)	0.053812 (22.92%)
1e4	5e5	0.115975 (9.47%)	0.842056 (68.80%)	0.265769 (21.71%)
1e5	1e4	0.019738 (49.96%)	0.004259 (10.78%)	0.01551 (39.25%)
1e5	1e5	0.07516 (19.49%)	0.155199 (40.25%)	0.155199 (40.25%)
1e5	1e6	0.487234 (12.49%)	2.57165 (65.96%)	0.839774 (21.54%)

Conclusions Observed -

- For Sparse Graphs majority of the time is utilised in building the spanning tree itself which is followed by finding the connected components in the auxillary graph.

- For Dense Graphs, majority of the time is utilised in building the auxillary graph which is again followed by finding the connected components in the auxillary graph.

Case Study 4: Comparing different steps in DFS, BFS Trees

- Tarjan Vishkin With Union Find for BFS Trees

TVUF_B:

N	M	Building tree and calculating low and high values	Building auxiliary graph	Finding connected components in auxiliary graph
1e4	1e4	0.014808 (25.68%)	0.03585 (62.21%)	0.006995 (12.13%)
1e4	5e4	0.023471 (15.07%)	0.104571 (67.16%)	0.02766 (17.76%)
1e4	1e5	0.03639 (12.31%)	0.205845 (69.68%)	0.053155 (17.99%)
1e4	5e5	0.111007 (7.83%)	1.03657 (73.12%)	0.26997 (19.04%)
1e5	1e4	1.01649 (97.55%)	0.004436 (0.43%)	0.021008 (2.01%)
1e5	1e5	0.255068 (48.37%)	0.18089 (34.30%)	0.091321 (17.31%)
1e5	1e6	0.389736 (9.18%)	3.03636 (71.58%)	0.815768 (19.23%)

Conclusions Observed -

- The conclusions were similar to the DFS tree results, and more magnified here.
- A noticable result was time taken was 97.55% of the time taken for a graph with 1e5 vertices and 1e4 edges were for building the spanning tree but only 9.18% of time for a graph with 1e5 vertices and 1e6 edges.

Case Study 5: Some Special Graphs

A) Extremely Sparse Graphs

Lets define extremely sparse graphs having atmost order of root N edges.

N	M	T	TV_D_R	TVUF_D_R
1e8	1e2	20.5458	28.0128	28.6182
1e8	1e3	20.3549	29.912	28.1915
1e8	1e4	20.4014	28.1354	28.1042

As we can see there is almost no difference between with and without Union Find algorithms for extremely sparse graphs. It is now for the first time almost comparable to normal Tarjan's itself.
This is because this is the amplified version of sparse graphs, where most of the complexity went in just building the spanning tree, a step which is common in all the 3 algorithms.

Case Study 5: Some Special Graphs

- **B) Graphs with Large Cycles**

  By Graphs with Large cycles we mean a cycle of O(n) size.

N	M	T	TV_D_R	TV_B_R	TVUF_D_R	TVUF_B_R
1e4	1e4	0.013763	0.024131	0.027583	0.033282	0.029592
1e4	1e5	0.04992	0.237761	0.310055	0.207144	0.268573
1e4	1e6	0.344664	2.91557	3.96772	2.41956	2.67052
1e5	1e5	0.081256	0.225707	0.286464	0.333665	0.385253
1e5	1e6	0.593984	3.98939	4.24353	2.79438	3.65131
1e5	1e7	5.8255	57.7895	60.42234	35.7286	38.0266

The general trends discussed till now have been followed here as well.
One considerable thing to note is that, these graphs are taking significantly slower on average than normal. The reason could be since there is a large cycle connecting the vertices, the number of connected components of the original graph are much lower.