Advanced Algorithms

Martin Mirakyan - IST194771

Asish Ghosh - IST1954845

Lisbon, May 2020

Problem statement 1: Second MST

Finding the minimum spanning tree (MST) in the graph is a popular problem in computer science. Finding MST could be a very expensive operation if the graph size is very big.
The first part of the project addresses solving the problem of finding the second MST in the tree. A more detailed statement formulation is as follows:

Given a tree with **N** vertices, and given **Q** queries of the form `a b w`, for each input, find the first and second MST formed after adding the `a b w` edge to the tree (queries do not modify the tree). Where a and b are the numbers of the vertices and w is the weight of the connecting edge.

Input:
The first line contains the number of vertices.
Then n-1 lines follow, which have the form of `a b w` meaning the vertex `a` is connected to vertex `b` with a weight `w`.
This is followed by the number of queries `q`.
After which `q` lines describe the queries of the form `a b w`.

Output:
For each query `a b w` output the value of first and second MST separated by a space.

Approach: Heavy Light Decomposition

The implementation is available on Github.
Our implementation is based on heavy-light decomposition. For each query `a b w` we calculate the maximum weight on the path from `a` to `b` and compare it with the `w` after which: if `w` >= `maxWeightOnPath` => first MST remains the same, and second is `MST` - `maxWeightOnPath` + `w`. Otherwise, the results are swapped.

This problem can be solved with many other techniques including the Kruskal's Algorithm and modelling the 2nd MST into an LCA problem [https://cp-algorithms.com/graph/second_best_mst.html], but our aim for the project was to explore the heavy light decomposition for educational purposes and how can it be used in different problems.

Heavy light decomposition is a general technique that aims to run queries on a tree. To understand HLD, let there be a tree G of n vertices, with an arbitrary root.

The essence of the tree decomposition is to split the tree into several paths so that we can reach the root vertex from any v by traversing at most log(n) continuous paths. In addition, none of these paths should intersect with another. That way a query on each separate path can be performed to obtain some value. In our case, we needed the maximum value in the path from `a` to `b`.

It is clear that if we find such a decomposition for any tree, it will allow us to reduce certain single queries of the form “calculate something on the path from a to b” to several queries of the type ”calculate something on the segment [l,r] of the kth path”.

To understand how the algorithm works, below is an example of input and a query.

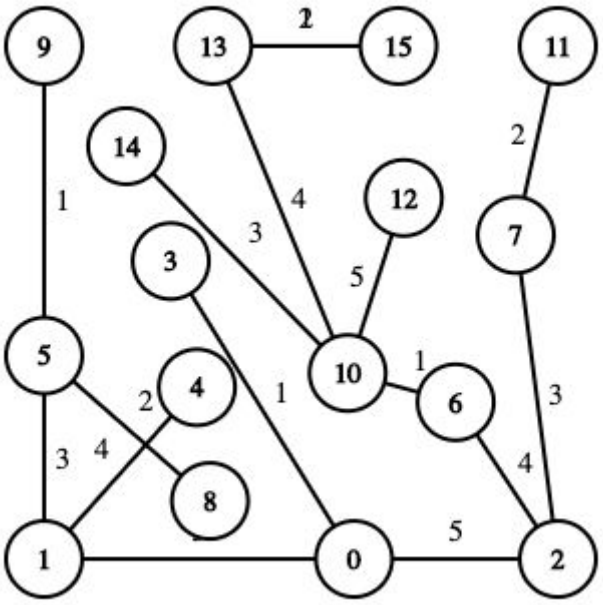


Fig 1: Input example tree

The query is `12 7 4`, meaning, “What would be the first and second MST after adding an edge of weight 4 from node 12 to node 7”.
The output should be `12 13`. As the largest weight in the path from 12 to 7 has a value of 5.

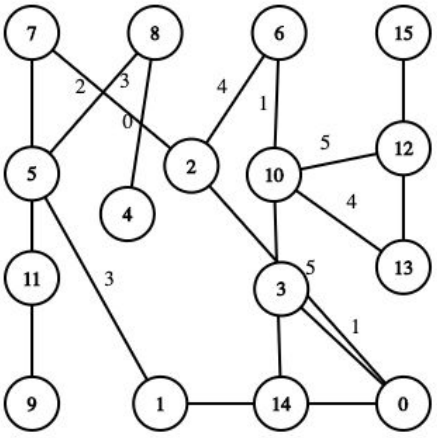


Fig 2: Tree after one input query

For another example, if the input is “8 4 0”, the older connection is removed and a new connection is made as shown in fig2.

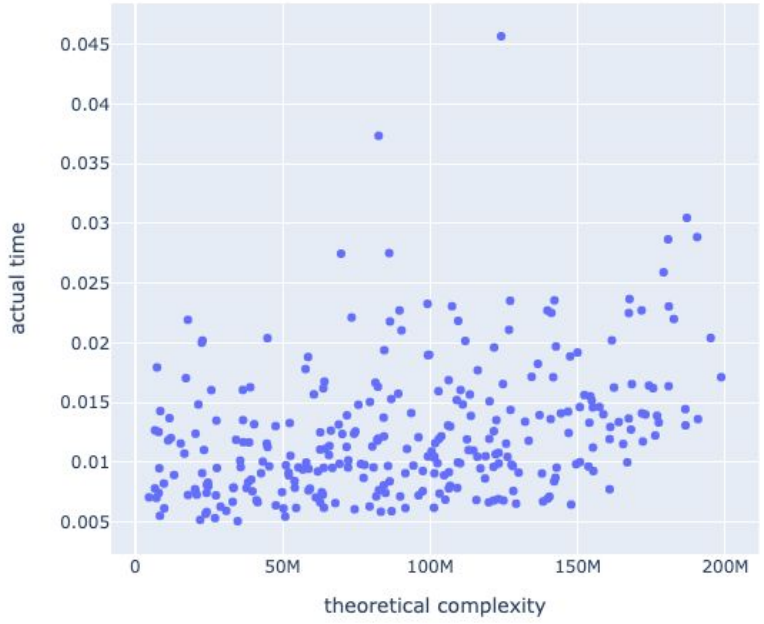
Implementation details

Our implementation is influenced by https://cp-algorithms.com/graph/hld.html. We use only one segment tree instead of keeping a lot of segment trees for each segment. That makes the code way cleaner and easier to implement. We use C++20 standard.

Complexity Analysis

As the construction of the heavy-light decomposition requires O(N*logN) time for setting up the binary lifting of the LCA and O(N) time for constructing the Maximum Segment Tree, while for each query there are at most O(logN) passes from one path to another and on each path, the query takes O(logN) time which results in O(log^2N) total complexity for each query.
Overall the total complexity, when given N nodes and Q queries, is O(N*logN + Q*log^2N).

As in our implementation, we use only one segment tree for the vertices - it takes up O(N) space. Also, each node is responsible for keeping its parent, depth, heavy pointer, head of the segment - that results in O(N) space as well. So having in total O(N) memory complexity.



In the graph of the theoretical complexity vs actual time, the x-axis represents the theoretical complexity of O(N*logN + Q*log^2N) for each random N and Q. The y-axis represents the seconds the program took to compute the second MST for all the Q queries.

As the trees are generated randomly and the queries are random as well, in most of the cases the switches between paths in HLD doesn't happen often thus leading to faster overall time than the theoretical complexity.

Problem statement 2: DrBC

Given a graph, compute relative betweenness-centrality scores for each node. The focus is to compute the relative scores among nodes as opposed to computing the exact betweenness-centrality scores.

Centrality indices are an essential concept in network analysis. For those based on shortest-path distances, the computation is at least quadratic in the number of nodes, since it usually involves solving the single-source shortest-paths problem from every node. Therefore, exact computation is infeasible for many large networks of interest today. Centrality scores can be estimated, however, from a limited number of SSSP computations. To solve this problem, we have implemented DrBC (Deep Ranker for Betweenness Centrality) as suggested in this paper by Fan, Changjun and Zeng, Li and Ding, Yuhui and Chen, Muhao and Sun, Yizhou and Liu, Zhong. We have used TensorFlow 2.x with Keras to implement it.

DrBC Method:

DrBC method an encoder-decoder framework. It consists of two components: 1) the encoder that generates node embeddings, and 2) the decoder that maps the embeddings to scalars for ranking nodes specified by BC values.

In this method, GNNs is used as an encoder, as it is easy to train small scale graphs and test directly in large networks. All neighbours of a node are included for exploration. Weighted sum aggregators are used to aggregate neighbours, as computing the number of shortest paths between two nodes is very expensive in a large network. Then a combine function is used to deal with the combination of the neighbourhood embedding generated by the current layer, and the embedding of the node itself generated by the previous layer. They used GRU as the combine function. With GRU, models can learn to decide how much proportion of the features of distant neighbours should be incorporated into the local feature of each node. In comparison with other COMBINE functions, GRU offers a more flexible way for feature selection and gains a more accurate BC ranking in experiments. Max-pooling aggregator is used to selecting the most informative (using the max operator) layer for each feature coordinate. This mechanism is adaptive and easy to implement, and it introduces no additional parameters to the learning process, which is more effective than the other two-layer aggregators for BC ranking also. The maximum number of layers set to be 5 when training and more layers lead to no improvement.

The decoder is implemented with a two-layered MLP which maps the embedding zv to the approximate BC ranking score yv :
$yv = DEC(zv;ΘDEC) = W5ReLU(W4zv)$
where $ΘDEC = \{W4 \in R, W5 \in R\}$, p denotes the dimension of zv, and q is the number of hidden neurons.

DrBC Complexity:

The training is generally affordable, the training on large networks takes acceptable time, and once trained the same model can be reused.

The time complexity of the inference process is determined by two parts. For the first part, the encoding complexity takes O(L|V |N (·)), where L is the number of propagation steps, and usually is a small constant, V is the number of nodes, N (·) is the average number of node neighbours. But as most matrices are sparse, through the adjacent matrix in this settings the complexity turns to be O(|E|), where E is the number of edges. The second is from the decoding phase. Once the nodes are encoded, we can compute their respective BC ranking score, and return the top-k highest BC nodes. the time complexity of this process mainly comes from the sorting operation, which takes O(|V |). Therefore, the total time complexity for the inference phase should be O(|E| + |V | + |V |log|V |).

Implementation differences from the original DrBC

We've used TensorFlow 2.x (tf-nightly for 2.3 version) with tf.keras to implement the network and the training. We have reused all the C++ and Cython codes while reimplementing the network structure and the training loop from scratch. Therefore, we initialize the weights of the layers with keras-default initializers as opposed to random weights from a truncated normal distribution which was used in the original implementation.

We noticed that having 8 graphs per batch instead of 16 with node count in the range [2000, 3000] instead of [4000, 5000] speeds up the performance a lot and consumes less resources which helped us to experiment more and we used parameters as those for our final model. We use Adam optimizer with default parameters and train the network for at most 100 epochs.

The source code of our implementation can be found on Github.

We decoupled the original code to make it more modular. In our implementation, the network architecture, data generation, training loop, and prediction modules are completely separate from each other which enables for easier experiments and a wider range of use of TensorFlow and keras utilities.

Dataset

The training is done on generated graphs of type `powerlaw` and the training and validation sets are re-generated once every 5 epochs.

Results on real datasets

	ABRA	RK	KADABRA	Node2Vec	Original DrBC	Our DrBC
com-youtube	56.2	13.9	NA	46.2	57.3	**58.2**
amazon	16.3	9.7	NA	44.7	69.3	**71.2**
Dblp	14.3	NA	NA	49.5	**71.9**	69.1
cit-Patents	17.3	15.3	NA	4.0	72.6	**73.7**
com-lj	22.8	NA	NA	35.1	71.3	**75.1**

Table 1: Kendal tau for each dataset and method.

	ABRA	RK	KADABRA	Node2Vec	Original DrBC	Our DrBC
com-youtube	**95.7**	76.0	57.5	12.3	73.6	63.0
amazon	69.2	86.0	47.6	16.7	**86.2**	73.4
Dblp	49.7	NA	35.2	11.5	78.9	**79.4**
cit-Patents	37.0	74.4	23.4	0.04	48.3	**49.4**
com-lj	60.0	54.2*	31.9	3.9	**67.2**	52.3

Table 2: Top 1% accuracy on large real-world networks.

	ABRA	RK	KADABRA	Node2Vec	Original DrBC	Our DrBC
com-youtube	**91.2**	75.8	47.3	18.9	66.7	61.1
amazon	58.0	59.4	56.0	23.2	79.7	**81.6**
Dblp	45.5	NA	42.6	20.2	**72.0**	61.5
cit-Patents	42.4	68.2	25.1	0.29	57.5	**57.9**
com-lj	56.9	NA	39.5	10.35	72.6	**74.3**

Table 3: Top 5% accuracy on large real-world networks.

	ABRA	RK	KADABRA	Node2Vec	Original DrBC	Our DrBC
com-youtube	89.5	**100.0**	44.6	23.6	69.5	63.9
amazon	60.3	**100.0**	56.7	26.6	76.9	77.9
Dblp	**100.0**	NA	50.4	27.7	72.5	59.9
cit-Patents	50.9	53.5	21.6	0.99	64.1	**64.6**
com-lj	63.6	NA	47.6	15.4	74.8	**78.5**

Table 4: Top 10% accuracy on large real-world networks.

	ABRA	RK	KADABRA	Node2Vec	Original DrBC	Our DrBC
com-youtube	72898.7	125651.2	**116.1**	4729.8	402.9	216.6
amazon	5402.3	149680.6	**244.7**	10679.0	449.8	360.3
Dblp	11591.5	NA	**398.1**	17446.9	566.7	738.8

cit-Patents	10704.6	252028.5	**568.0**	11729.1	744.1	668.6
com-lj	34309.6	NA	**612.9**	18253.6	2274.2	1919.5

Table 5: Time for inference in seconds for each dataset and method. The values are copied from the original DrBC paper, while our times are recorded on a dual-core CPU machine with 16GB RAM.

From Table 1 results, our approach outperforms the original DrBC results for all the datasets except Dblp. The top N% tables show that our implementation is sometimes better than the other approaches, but that does not hold for all the datasets.
For the time taken on inference, our implementation is a bit faster than the original DrBC method probably due to better-optimized keras layers.

Conclusion

For approximating the betweenness centrality, we have seen these several methodologies like RK, ABRA, KADABRA, Node2Vec and finally DrBC. Through our experiments, we have seen that the overall performance of DrBC is very competitive even outperforming in some cases the previous works. Our implementation shows promising results without any parameter tuning and we think it can be improved with future experiments and new attributes in the network which can be experimented with like attention or some parts of transformer architecture.