University of Zurich^UZH

Department of Informatics

Andrianos Michail

Masters Basic Module : Identifying desirable spanning trees for dynamic connectivity

July 2021

University of Zurich
Department of Informatics (IFI)
Binzmuhlestrasse 14, CH-8050 Zurich, Switzerland

A. Michail:
July 2021
DBTG
Department of Informatics (IFI)
University of Zurich
Binzmuhlestrasse 14, CH-8050 Zurich, Switzerland
URL: https://www.ifi.uzh.ch/en/dbtg.html

Introduction

Graphs are important in a lot of applications domains such as transportation systems[1], communication networks [2] and social networks. A fundamental operation on graphs are Connectivity Queries(CQ), i.e, given two nodes, we are requested to know whether a path exists between them.

For static graphs, all the query pairs can be precomputed and it is easy to answer this query efficiently. However, the graphs in the real world are fully dynamic and require constant updates, i.e Social networks where we constantly add and remove friends. Updating the precomputed pairs for dynamic large graphs is prohibitively expensive.

Consequently, it is necessary to maintain a data structure that can answer these connectivity queries. Spanning Trees are an adequate data for representing connected components in a graph. By using Spanning Trees, a connectivity query then is simplified to checking whether the two nodes are part of the same Spanning Tree and therefore, Connected Component. To be precise, a rooted spanning tree is used to complete the process fast. The procedure consists of traveling from both vertices to the root, if the root is the same, then they are part of the same spanning tree and therefore connected.

The most expensive operation is the deletion of an edge that is used in the spanning tree because it is costly to find a replacement edge if one exists so that we can try to reconnect the two components. If no replacement edge is available, we now have two different connected components to represent the same graph. We look into ways to make this search cheaper by constructing the Spanning Tree in an optimized way.

Background

Preliminaries

Within this research we restrict the domain to undirected unweighted simple graphs that are fully dynamic.

[Graph(G)] A Graph $G(V, E)$ is defined by a set of vertices V and a set E of Edges $\subseteq V \times V$ and it is simple iff there is at most one edge $(u, v) \in E$ that connects a pair of vertices $u, v \in V$. A graph is fully dynamic iff edges can be deleted and inserted at any point in time. Common usages of graphs are to represent friendships(E) between people(V) in Social Media, paths(E) between physical locations(V) and more. [Connected Component(CC)] A Connected Component is a maximal subgraph $CC(V', E')$ of a graph $G(V, E)$ with $V' \subseteq V, E' \subseteq E$, in which all pairs of nodes are connected via a path.

[Spanning Tree (ST)] A spanning tree $ST = (V', E_T)$ for a connected component $CC(V', E')$ is a rooted tree containing all the vertices of CC and $|E_T| = |V'| - 1$ edges of CC ($E_T \subseteq E'$). For graphs with more than one component, a spanning forest that consists of a spanning tree for each component is used. For clarity, the vertices and edges of the spanning tree are called Nodes and Arcs, respectively. To formally define this, $SpanningTree(ST) = (N, A)$

The amount of unique possible spanning trees of a connected component becomes exponentially larger the more vertices and edges the connected component contains.

Dynamic Connectivity

A Dynamic Connectivity structure of a graph maintains information about the components of the graph. At any given time, the structure can perform a connectivity query or update in the graph, whilst maintaining the properties.

Amongst all the spanning trees of the input graph, some spanning trees yield better performance in connectivity queries and updates over the rest.

Average Cut Number(ACN)

When deletion of a tree edge is performed, we need to find a replacement edge to try to reconnect the two disconnected components. To achieve this we can perform a search algorithm for the two components in parallel. The runtime for the search is equal to the size of each component.

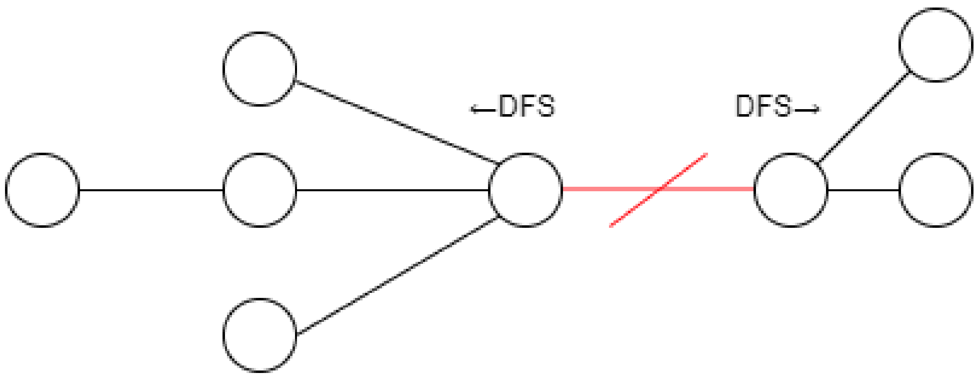

Figure 1: Tree Edge deletion reconnection procedure

Both searches can be performed in parallel. As soon as one of the two searches is complete, we can terminate the procedure as we have already found a replacement edge if one exists. It is clear that by minimizing the size of the smaller component, we reduce the maximum runtime.

This shows that for connectivity queries, the best-performing spanning trees for deletion of a tree edge are characterized by uneven splits. Let us introduce this formally as a measure:

The cut number of an arc, e, in the spanning tree, is denoted as $c(e)$. After the removal of an arc e of the spanning tree, two spanning sub-trees are generated $tree_1$ and $tree_2$; $c(e) = min(size(tree_1), size(tree_2))$, where $size(tree_1)$ and $size(tree_2)$ are the numbers of nodes in $tree_1$ and $tree_2$, respectively. The set of arcs of the spanning tree st, is denoted as E_T. The metric is formally defined as:

$$Average\ Cut\ Number = \frac{1}{|E_T|} \sum_{e \in E_T} c(e)$$

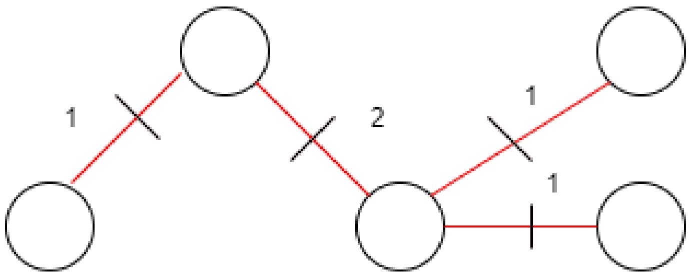

Figure 2: The total cut number is $1 + 2 + 1 + 1$ therefore ACN $= \frac{5}{4}$

The smaller the Average Cut Number, the better the performance of searching for a replacement edge in a spanning tree for deletion queries.

Tree Diameter and Height

Tree Diameter(TD) in a spanning tree is defined as the longest distance between any two nodes within the tree, defined by:

$$Tree\ Diameter = max(dist(i, j))\ \forall i, j \in V'$$

Where the number of nodes to travel from i to j on the shortest path is denoted as $dist(i, j)$.

Maximum Tree Height(MTH) in a rooted ST is defined as the longest shortest path from the root to the leaves. In a dynamic connectivity structure, the optimally rooted spanning tree is stored to answer connectivity queries.

Maximum Tree Height can also be calculated as followed:

$$Maximum\ Tree\ Height = \lfloor (Tree\ Diameter/2) \rfloor$$

The smaller the Maximum Tree Height the better the asymptotic complexity of the connectivity query.

Kirchhoff's Theorem

Laplacian Matrix (L): The Laplacian Matrix is an unweighted matrix representation of a graph defined as:

$$L_{ij} := \begin{cases} \deg(u_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } u_i \text{ is adjacent to } u_j \\ 0 & \text{otherwise} \end{cases}$$

Kirchhoff's Theorem states that the number of possible unique spanning trees in a graph is equal to any co-factor of the Laplacian Matrix [3]. Let us see this in an example:

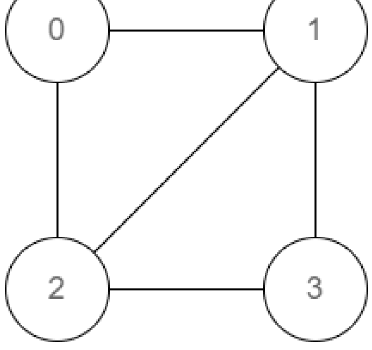

Figure 3: Small example graph

The graph illustrated at Figure 3 yields the following Laplacian Matrix:

$$\begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & 3 & 2 \end{bmatrix}$$

By calculating the co-factor of the Matrix the graph in Figure 3 we can show that there are 8 different unique spanning trees within the graph. The number of possible trees grows exponentially with the size of the graph, for example in a graph with $|V| = 10$ and $|E| = 19$ with the same method we found out that there are 11394 unique possible spanning trees.

Problem Definition

Within this research, we are searching for the spanning tree of a dynamic graph such that both ACN and MTH are minimized. Generating all spanning trees of Connected Components and choosing the best one is extremely costly in larger graphs. Instead, we look into finding polynomial algorithms that generate a near-optimal ST that can perform the updates and queries at a similar speed.

The lower bound of the average cut number is 1 which occurs when every arc cut generates a component of size 1. This condition is satisfied in graphs that allow spanning trees with a structure similar to the sunflower type spanning tree shown in Figure 4.

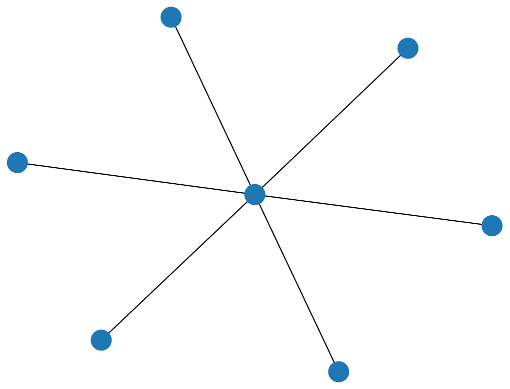

Figure 4: Optimal Spanning Tree with ACN = 1

The first thought that comes to mind when seeing this spanning tree is that it could be generated by running BFS. Within this research, we expand this hypothesis further to see whether a BFS inspired algorithm could lead to the consistent generation of low ACN spanning trees.

Solutions and Algorithms

To construct the optimal or near-optimal spanning tree, multiple algorithms and ideas were developed and tested. Not all algorithms were complete but regardless there is merit to all of the ideas and all are presented.

[Degree] Given a connected graph $G = (V, E)$, the degree of a vertex v, $deg_G(v)$, is the number of edges that are directedly connected to v. Note that the definition of degree applies to nodes in spanning trees. Given a spanning tree T and a node v, the degree of node v, $deg_T(v)$, is the number of arcs that are directedly connected to v.

[Degree Outside of Spanning Tree (DoST)] Given a connected graph $G = (V, E)$ and a spanning tree $ST = (N, A)$ where $N' \subseteq V$ and $A \subseteq E$. The Degree Outside of Spanning Tree of a vertex v, DoST(v) = $deg_C C(v)$ - $deg_S T(v)$.

Intuitively, DoST refers to the degree of a vertex when only considering vertices that are not in the tree.

Following we examine greedy algorithms that generate a spanning tree of a connected component for connectivity queries.

Greedy Connected BFS

Algorithmus 1: Greedy Connected BFS

```
input  : A Graph $G(V, E)$
output: A Spanning Tree $ST$
1 st = $\emptyset$ , visited = $\emptyset$
2 while $|st| < (|V| - 1)$ do
      verticeLargestDoST = 0
      verticesToCheck = if visited = $\emptyset$ then V else visited
      for $vertice \in verticesToCheck$ do
          verticeLargestDoST = max(largestDoST, doSTVertice(G, vertice,
          spanningTreeArcs))
      currentVertice = vertice with verticeLargestDoST
      for $neighbour \in G.AdjList(currentVertice) \notin visited$ do
          st = st ∪ (currentVertice, neighbour)
          visited = visited ∪ neighbour
3 return st
```

This BFS style greedy algorithm starts with the highest degree node and adds all of its neighbors to the spanning tree. Afterward, it repeatedly chooses the vertices with the highest *DoST* that are already visited (in case of a draw, we arbitrarily choose the one with the higher degree) and adds those edges to the tree until it forms a complete spanning tree.

This algorithm is based on a BFS which has a $\mathcal{O}(|V|)$ complexity to generate a spanning tree. Additionally, it requires performance of a simple linear ($|V|$) method to select the next node to iterate from. The while loop is expected to be executed significantly less than $|V|$ times, hence the average complexity is expected to be much lower.

The Asymptotic Complexity of this algorithm is: $\mathcal{O}(|V|^2)$

Greedy Disconnected BFS

Similar to the previous algorithm, Greedy Disconnected BFS performs the same procedure without the restriction of having to choose a visited vertice for each iteration. Depending on the graph, this can lead to the generation of disconnected spanning trees instead of a single connected spanning tree. To use this algorithm across all graphs, further work is required to create a form of bridge between the disconnected components.

The Asymptotic Complexity of this algorithm is also: $\mathcal{O}(|V|^2)$

Greedy Edge Based Disconnected Algorithm

Algorithmus 2: Greedy Edge Based Algorithm

```
   input  : A Graph $G(V, E)$
   output: A Spanning Tree $ST$
1  st = $\emptyset$
2  while $|st| < (|V| - 1)$ do
       edgeLargestDoST = 0
       edgesToCheck = E $\notin$ $spanningTreeArcs$
       for $edge \in edgesToCheck$ do
           edgeLargestDoST = max(largestDoST, doSTVertice(G, vertice,
           spanningTreeArcs))
       edgeToAdd = edge with edgeLargestDost
       st = st = st $\cup$ (edgeToAdd)
3  return st
```

Additionally, an edge based algorithm was developed. The edge based algorithm repeatedly adds the edge that connects the two vertices with the highest total $DoST$ with a vertex already visited, until it has added $|V| - 1$ edges. Depending on the graph, this can lead to generation of disconnected components instead of a connected spanning tree. Similar to the last one, to use this algorithm further work is required to create a form of bridge between the disconnected components.

The runtime of this algorithm depends on the choice of an important implementation detail: whether we use a dynamic data structure that gets updated during the addition of edges of the spanning tree., or trivially performing a linear search on all of the edges when adding every edge. Regardless, the asymptotic complexity remains to be the same in both cases.

Asymptotic Complexity: $\mathcal{O}(|V| \cdot |E|)$

Within this project, the dynamic data structure of this algorithm was developed as it is expected to slightly improve runtime.

Further work with this algorithm would be to allow initialization with the highest total *DoST* edge and afterward only selecting edges that have exactly 1 end in the spanning tree resulting in the creation of a connected spanning tree.

Identifying Disconnected Components

To identify how many components are formed with the arcs in the spanning tree, the following formula is used:

$$|components| = |visited| - |arcs|$$

By actively knowing how many components the tree contains, there exists a foundation so that the two greedy algorithms can be tuned and guarantee the creation of a connected spanning tree. This would be done by selecting points to force the algorithm to find bridges as replacement edges to reconnect the disconnected components. These ideas are aimed to be explored in further work.

Best Spanning Tree from a Set

Alternatively to directly generating the optimal spanning tree, what if we would instead generate multiple spanning trees through the a Randomized BFS procedure described in the following section and select the one with the best metrics.

We assume correct usage of hashing of sets and repetitions to be fixed to $|V|^2$. Taking into account the cost of generation of a spanning tree though BFS to be $|V|$ and that it is repeated over $|V|$ vertices this entails that for each repetition the cost is $|V|^2$.

Running the aforementioned procedure over $|V|^2$ repetitions would entail that the generation algorithm costs $|V|^2 \cdot |V|^2$ hence $\mathcal{O}(|V|^4)$.

The evaluation of the ACN and Tree Diameter of a spanning tree takes computation of $\mathcal{O}(|V|^2)$ as we have to perform a search algorithm for $\mathcal{O}(|V|)$ for all the nodes in the spanning tree.

To summarize, the asymptotic complexity process of finding an optimal ST through random BFS generation is $|V|^4 + |V|^2 -> \mathcal{O}(|V|^4)$.

This is a polynomial algorithm and we expect it to be feasible to run in relatively large graphs.

Experimental Setup

To allow the empirical assessment of spanning tree generation algorithms, we need to compare them amongst multiple spanning trees of the same graph. To achieve this, an algorithm to enumerate all possible unique spanning trees in small graphs is used.

Graphs used in Experiments

All of the graphs used within this research are randomly generated graphs ($|V| = 20|E| = 80$) and the results need to be further investigated before applied in real world graphs.

Exhaustive Enumeration

Within this research, the simplest exhaustive enumeration algorithm was developed. The algorithm works by recursively constructing new spanning trees until all combinations are made. Further details about the algorithm are not analysed due to it's complexity. The asymptotic runtime of this algorithm can be thought of as the cost of creating a spanning tree ($|V|$) multiplied by the maximum possible spanning trees in the graph.

The maximum amount of possible spanning trees occurs in a complete graph and it can be calculated by Cayley's formula [4] as $|V|^{|V|-2}$

Therefore, the worst case Asymptotic Complexity is $|V|^{|V|-2} \cdot (|V| - 1)) -> \mathcal{O}(|V|^{|V|-1})$.

Utilising the outcome of Kirchoff's Theorem, it is possible to validate that the exhaustive enumeration algorithm has generated all of the unique spanning trees.

Exhaustively enumeration of all spanning trees is feasible for small graphs but becomes more expensive as the number of vertices and edges grows. Faster enumeration algorithms have been presented in [5] [6]. The faster enumeration techniques remain too computationally expensive and therefore alternatives methods have to be investigated. In the next subsection, an alternative approach is examined that instead generates a subset of spanning trees in polynomial time that is what we use as a comparison set.

Fast Random desired trees enumeration

To generate multiple spanning trees in polynomial time, a randomized procedure was developed.

The procedure consists of performing DFS or BFS starting from all the vertices whilst randomizing the order of the adjacency list at every step. After performing this for multiple amount of repetitions, the unique spanning trees are collected and stored.

To assess the quality of the generated spanning trees, a histogram comparison of the spanning trees ACN metric is visualised in a small graph ($|V| = 10, |E| = 20$).

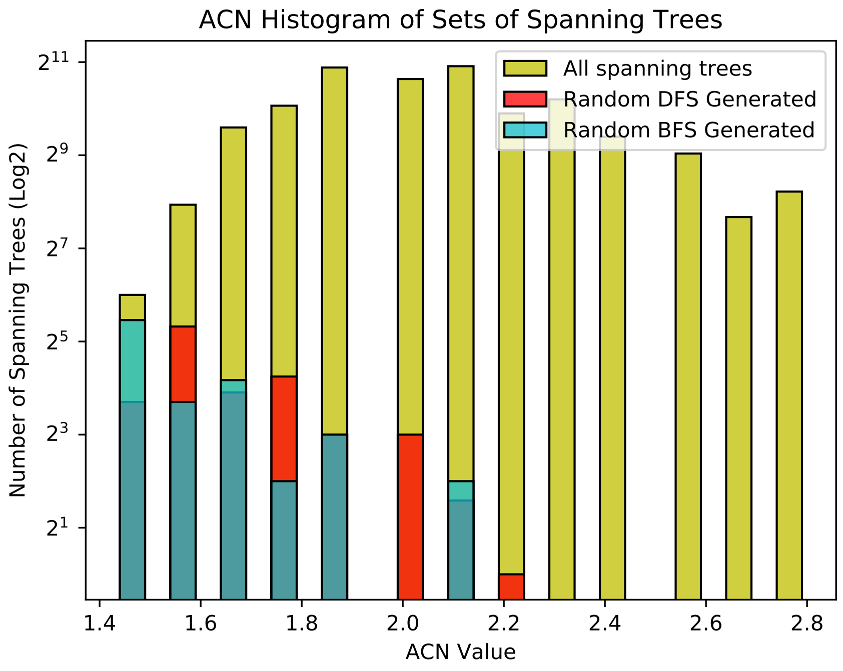

Figure 5: Histogram of ACN of different generation techniques. Note: Y axis is $\log_2$

Within this experiment, the total amount of unique spanning trees in this graph accounts for 11394. From the histogram, the Random BFS Procedure generated a total of 91 spanning trees all within the top 50% of the spanning trees when compared by ACN. However, 56% of the trees generated by Random BFS, actually belong to the best 2.5% of all possible spanning trees.

On the other hand, Random DFS generates 107 unique Spanning trees but with all-around higher ACN metrics.

Random BFS generating a near-optimal ST is expected, as it naturally generates central nodes, which allows an ST to have a low ACN metric.

To empirically prove this claim more experiments should be conducted but due to time constraints and computational complexity of exhaustive enumeration on multiple graphs, it is not feasible as part of this project.

The assumption is made that the spanning trees generated by the Randomized BFS approach is a sufficiently large performing set of spanning trees and therefore a good benchmark to compare the greedy generation algorithms against.

For this algorithm, we have investigated the number of optimal repetitions of the randomized BFS generation technique.

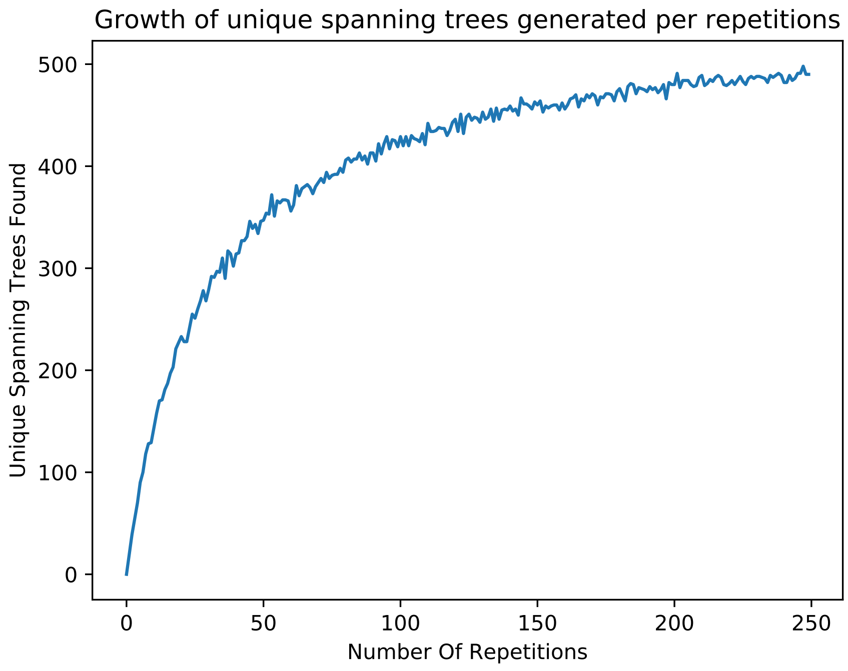

Figure 6: Growth of Unique Spanning Trees over amount of repetitions

As it can be seen from Figure 6 that the growth of the unique spanning trees is logarithmic. To further investigate this, the same experiment is run averaged over 5 different graphs of the same $|V|$ and $|E|$.

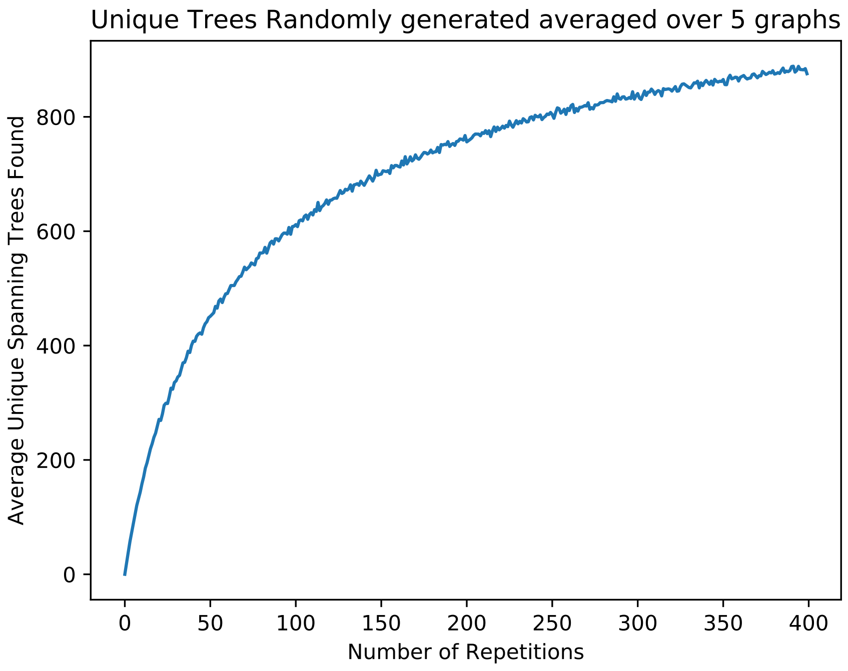

Figure 7: Growth of Unique Spanning Trees over amount of repetition Averaged over 5 graphs

Figure 7 shows the average growth of spanning trees over 5 different random graphs (of identical cardinality of vertices and edges). It is observed that after a certain amount of repetitions, not many new spanning trees are found. A satisfactory amount of repetitions is hypothesized to be at $|V|^2$ as seen to be optimal in Figure 7.

Within our experiments, this appears optimal but other factors to consider before generalizing to more graphs is that the topology and size of the graph is a big factor.

Results

Minimizing both ACN and MTH

Before conducting the initial experiments, we hypothesized that minimizing ACN also minimizes MTH. However, empirically it was shown that it is not always the case.

Both measurements seem to rely on similar characteristics of the ST, lower ACN spanning trees seems to come hand in hand with low MTH but different STs with the same ACN can have different MTH. Therefore, generating a ST with optimal ACN does not guarantee the lowest MTH, but it is expected to be a near-optimal MTH.

Empirical Patterns Emerging

Through Empirical studies, the lowest ACN achievable on the graphs test ranges between $1.5 - 2$. This is expected, as an ACN of 1.5 would entail that approximately half the arcs result in a cut number of 1 and the other half a cut number of 2.

In larger and more complex graphs, a spanning tree similar to the sunflower Figure 4 may not be possible. Therefore, if a direct sunflower shape is not possible, any value near 1 ACN is unrealistic. Within our experiments, the best ACN values for larger graphs appear to be in the 1.5-2 range.

How do the optimal or near-optimal ST's look like in bigger graphs? Let us observe the figures of the best spanning trees we have found in larger graphs.

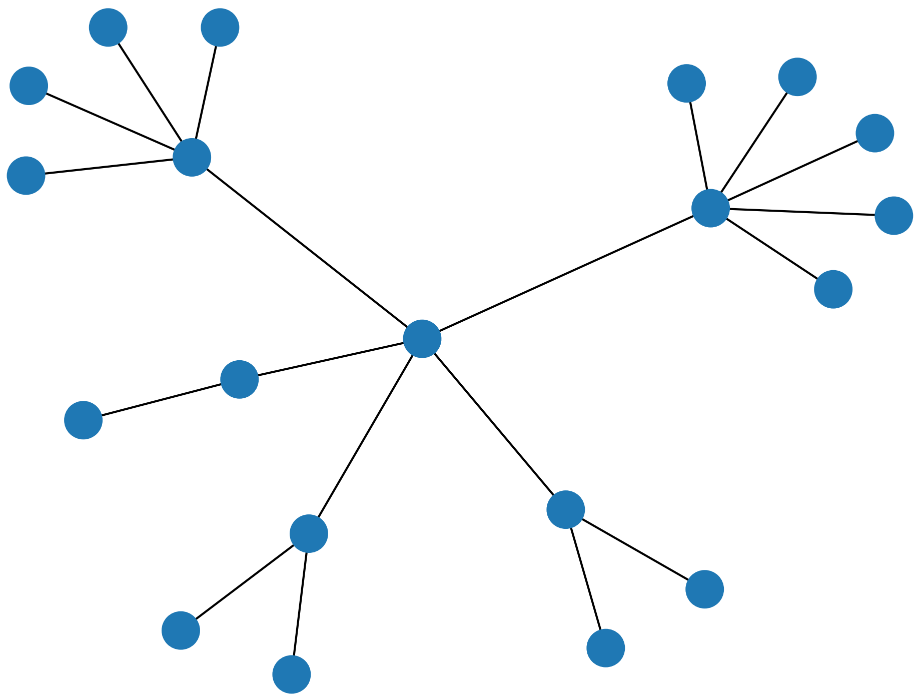

Figure 8: Optimal or near Optimal ST with $|V| = 20$ and $|E| = 80$ ACN equal to 1.74

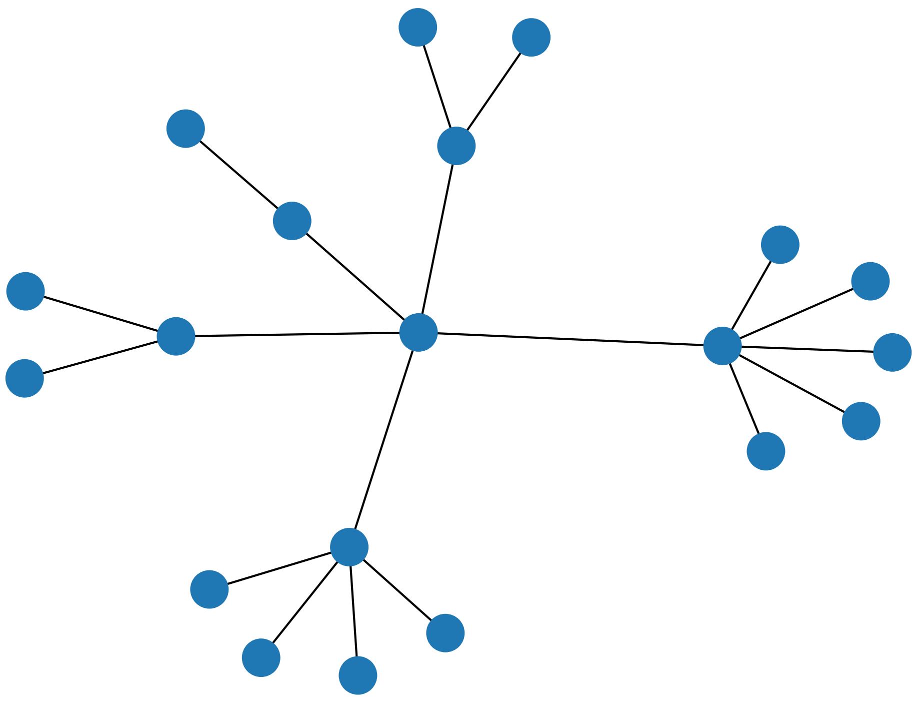

Figure 9: Optimal or near Optimal ST with $|V| = 20$ and $|E| = 80$ and ACN equal to 1.68

From Figures 8 and 9 we observe that the sunflower shape remains. However, the property is slightly different in this case. The petals in this case are ACN optimal sub-spanning trees whilst the core remains to be a node.

Figures 8 and 9 demonstrate that the Greedy Connected BFS algorithm would be unable to produce an optimal solution for those graphs. Greedy Connected BFS would instead try to build the ST using the highest degree vertice as a core instead of the optimal vertice.

Evaluation of Heuristics

Within multiple different random graphs, the ACN value of the Greedy Connected BFS ST is compared to the Random BFS generated Spanning trees with a Histogram. The best and worst performance histogram is visualized:

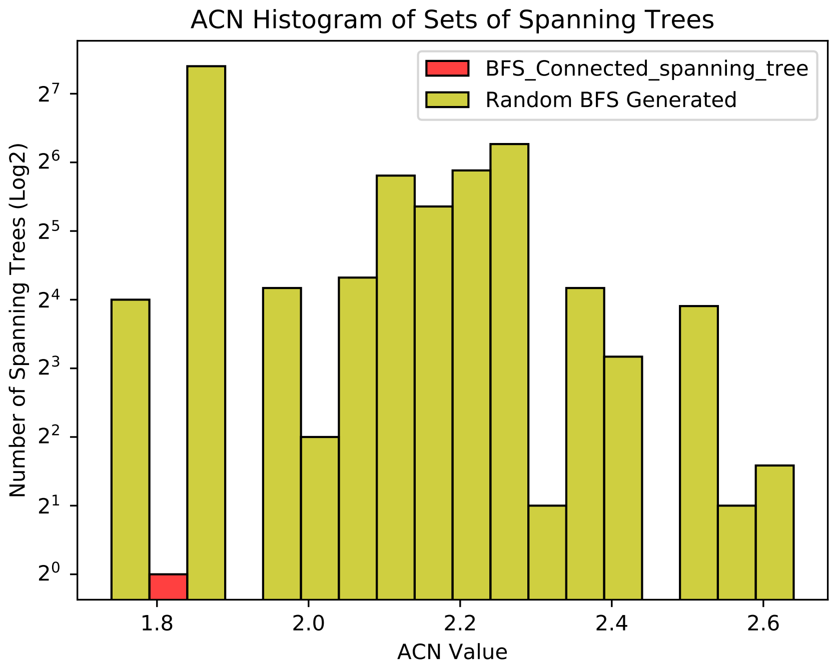

Figure 10: Best performing graph. Greedy algo at (ACN 1.63) and Random BFS with best ST at (ACN 1.58). Y axis is $\log_2$

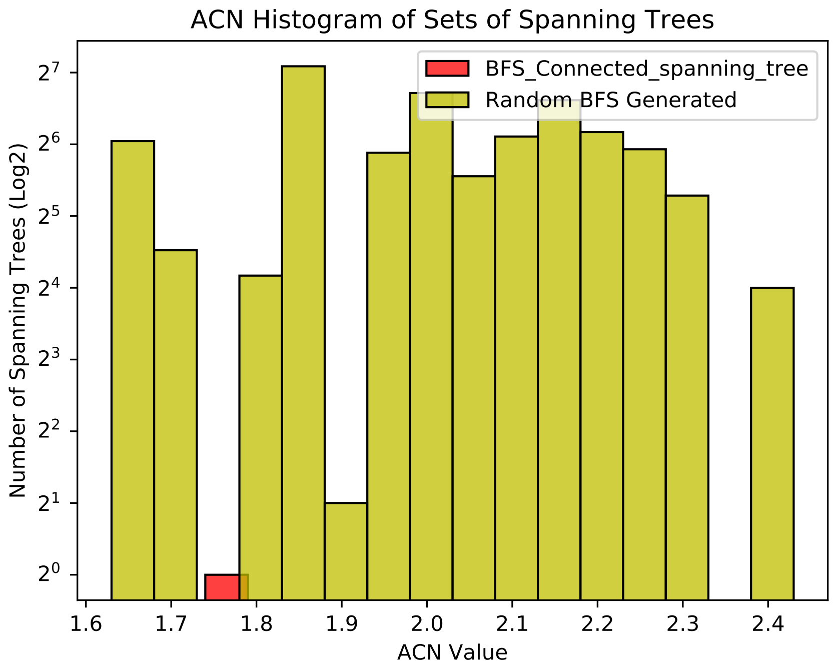

Figure 11: Worst performing Graph. Greedy BFS algo at (ACN 1.79) and Random BFS with best ST at (ACN 1.63). Note: Y axis is $\log_2$

Observing Figures 10 and 11 it can be seen that the greedy algorithm generates one of the best performing spanning trees even in the worst performing graph. This is a good validation of the hypothesis that greedy heuristics can generate a near-optimal spanning tree for connectivity queries.

Combining Heuristics

Further to using just one heuristic generation, it is also possible to use multiple and select the best performing one. Benefits to combining results from heuristics are increased robustness and the ability to create multiple algorithms to handle a different kinds of graphs.

Asymptotic complexity wise, this would be cost equal to the most expensive operation. To identify the final ST, we calculate the ACN and MTH for all generated ST's and select the one with the best metrics.

To see how this would work, to find the best performing ST we generate all spanning trees through Random BFS in $\mathcal{O}(|V|^4)$ and also the single ST by greedy BFS in $\mathcal{O}(|V|^2)$. Afterwards, we select the best performing in $\mathcal{O}(|V|^2)$. This would entail that the total asymptotic complexity is $\mathcal{O}(|V|^4)$.

Combining Heuristic procedures allows the generation of multiple spanning trees that are estimated to be amongst the best ones. This can also act as an extra robustness guarantee for the heuristics and it maximizes the chance to return a ST with good metrics.

Proposed Methods

Algorithm	**Complexity**	**Always Valid ST**
Greedy Disconnected BFS	$\mathcal{O}(\|V\|^2)$	False
Greedy Edge Based Algorithm	$\mathcal{O}(\|V\|^2)$	False
Greedy Connected BFS	$\mathcal{O}(\|V\|^2)$	True
Select {Random BFS}	$\mathcal{O}(\|V\|^4)$	True
Select {Random BFS, GC-BFS}	$\mathcal{O}(\|V\|^4)$	True

Table 1: Asymptotic Complexities of proposed methods

Table 1 shows the algorithms proposed within this research. The first two do not always produce a valid ST therefore we do not consider them.

Greedy Connected BFS appears to be very well performing as seen in Figures 10 and 11. Additionally, the fourth and fifth method that selects an ST from a larger set that also includes Randomly generated ST's guarantees a higher quality result. As we can see from the complexities, it appears to be a tradeoff between speed and guarantee of performance

Further Work & Conclusions

The optimal ACN spanning tree appears to be a very difficult problem, maybe even NP and therefore we investigated heuristic solutions. It is interesting to further examine this from a theoretical perspective and analyze whether a guaranteed procedure to generate the optimal ACN spanning tree is possible in polynomial time.

Within this project, some properties of the best ACN trees were observed and multiple heuristics had an initial suitability investigation. Heuristics are estimated to be able to provide a ST in the top 2.5% of all spanning trees within scalable complexity. Further work is required to validate whether this spanning tree can be maintained and whether the empirical results of this project generalize to real world graphs. It would be possible to enhance this project with the addition of further heuristics to the optimal spanning tree procedure suggested to increase the robustness of producing the optimal spanning tree.

Bibliography

[1] S. Guze, "Graph theory approach to transportation systems design and optimization," *TransNav: International Journal on Marine Navigation and Safety of Sea Transportation*, vol. 8, 2014.

[2] R. P. Singh, "Application of graph theory in computer science and engineering," *International Journal of Computer Applications*, vol. 104, no. 1, 2014.

[3] S. Chaiken and D. Kleitman, "Matrix tree theorems," *Journal of Combinatorial Theory, Series A*, vol. 24, no. 3, pp. 377–381, 1978.

[4] A. Cayley, "A theorem on trees," *Quart. J. Math.*, vol. 23, pp. 376–378, 1889.

[5] S. Kapoor and H. Ramesh, "Algorithms for enumerating all spanning trees of undirected and weighted graphs," *SIAM Journal on Computing*, vol. 24, no. 2, pp. 247–265, 1995.

[6] R. Hariharan, S. Kapoor, and V. Kumar, "Faster enumeration of all spanning trees of a directed graph," in *Workshop on Algorithms and Data Structures*, pp. 428–439, Springer, 1995.