Efficient Path Finding Over Directed Acyclic Graphs Through Pruning

Kaan Akduman
Department of Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
kxa317@case.edu

Maryam Iqbal
Department of Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
mxi128@case.edu

ABSTRACT

This paper attempts to implement and extend on the FELINE algorithm discussed by Veloso et al. for the use case of efficiently finding a path in Queen Elizabeth II's family tree represented as a directed acyclic tree (DAG) to help simplify social analysis. It is based on the course project topic of making big data small within the theme of making big networks smaller to support faster path search. The pruning algorithm implementation provides results over 96% faster than an unpruned search algorithm in the DAG created for the family tree.

CCS CONCEPTS

~ Information systems

~ Data management systems

~ Database design and models

~ Graph-based database models

~ Network data models

KEYWORDS

Computer Science, Reachability Queries, Graph Indexing, Database Management, Discrete Mathematics, Graph Theory, Graph labeling, Graph algorithms, Path Finding

1 Introduction

Developing scalable methods for the analysis of large sets of graphs, including graphs that model social structures, is a challenging task as seen in recent literature. With the ever-expanding nature of the size of graphs reaching millions of node sizes, efficient algorithms are needed for handling the graphs. A common problem is verifying whether a vertex is reachable from another. [1] This paper adapts that problem for a family tree. In a graph with a set of vertices and edges, a reachability query would ask whether a vertex is connected to another vertex and path of edges required to reach it. When used for a family tree, this helps us find if a relationship exists between two entities; if one is the ancestor of the other. A common approach seen in similar papers in the area is the preprocessing of graphs in order to produce an efficient index structure, allowing fast access to the reachability information of the vertices. However, the majority of these existing methods can not handle very large graphs. [1]

This paper hence builds upon the work of FELINE (Fast rEfined onLINE search), an index building method that creates indexes from the graph and represents them in a two-dimensional plane that provides reachability information in constant time for a significant portion of queries. [1] We have adapted the FELINE algorithm in this paper to help in finding connections between people in the family tree of Queen Elizabeth II. This fits into the first theme of the course project options of making big data small whereby we make big networks smaller to support faster path search.

In section 2, we discuss the background behind directed acyclic graphs, the basics of topological sorting, two common algorithms of topological sorting, and the FELINE algorithm including pseudocode as well as other theoretical concepts pertaining to our work. In section 3 we discuss the differences between our algorithm and FELINE. We also discuss the data collection process, the implementation of our algorithm, and its use case on Queen Elizabeth. In section 4, we discuss and evaluate experimental results regarding the performance values of our algorithm compared to similar algorithms with other levels of pruning. We also show graphs demonstrating the assignment of indices to vertices in the graph. Section 5 includes the conclusion and possible extensions of our current work.

2 Background

In this section, we will discuss the theory behind some of the algorithms we used.

2.1 Directed Acyclic Graphs

Directed acyclic graphs are graphs that only have directed edges and do not contain any cycles. They have vertices and directed such that no cycle exists. They are often used to visualize the flow of values between nodes and to provide optimization techniques on a network of data. [2]

2.2 Topological Sort

Topological sorting is when in a directed graph, there is a linear ordering of vertices such that for every directed edge **ab** from a vertex **a** to vertex **b**, and **a** comes before **b** in the ordering. A topological sort then provides a valid sequence for the tasks in the nodes of the graph. [3]

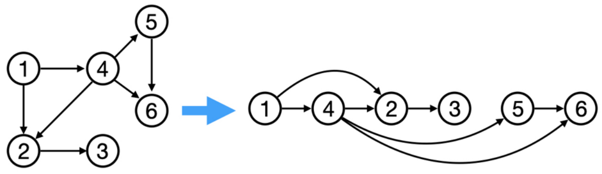


Figure 1: The left side shows a directed acyclic graph and the right side shows one possible topological sorting of the graph. [4]

All DAGs must have at least one topological ordering, but a DAG can have more than one valid topological sorting. [3] For example, in Figure 1 above, there are multiple ways in addition to the sorting shown on the right side for topological sorting of the LHS graph. For example, the 3 could have been moved between the 5 and the 6 and it would still be valid. It also could have been moved after the 6 and also be valid as all conditions are still satisfied. [4]

2.3 Topological Sort Based On Depth First Search

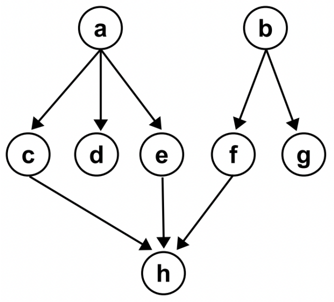
One topological sorting algorithm is based on a depth first search (DFS). In this case, a temporary stack is used where rather than printing vertices immediately after they are reached, we recursively call the topological sorting algorithm on all its adjacent vertices before pushing it to the stack. Then, the stack is topologically sorted with the newest item in the stack being the front of the topological sorting. [3]

2.4 Kahn's Algorithm

Kahn's algorithm is another way to topological sort in a directed acyclic graph. Kahn's algorithm works by assigning an indegree to each node corresponding to the number of incoming edges the node has. It will then go through all nodes with an indegree of 0, move them to the next open position in a list of sorted nodes, and modify the indegree of any of that node's children by subtracting 1. It will then repeat this process until all nodes have been moved to the sorted list. [5]

2.5 FELINE

The FELINE algorithm is the foundation of our adapted path finding algorithm. FELINE works by associating every vertex with a unique ordered pair of natural integers (x, y). Geometrically, (a, b) ⩽ (c, d) means that the coordinate (c, d) is in the upper-right quadrant of the two-dimensional Cartesian system in comparison to (a, b). It also means that the vertex at (a, b) dominates (c, d). [1] A dominance region of a vertex is the rectangular region of the graph where both x and y values of the point are greater than the x and y values of the vertex. [1] Given a DAG *G*, every vertex is associated with a pair (x, y) ∈ N^2 such that, for any two vertices *u, v* ∈ G, if there is a directed path from *u* to *v*, then the x-coordinate of *u* is less than or equal to the x-coordinate of *v* and y-coordinate of *u* is less than or equal to the y-coordinate of *v*. [1] The resulting index can be represented graphically. The pair of integers (x, y) associated with a v. Figure 2 shows an example of a small DAG and its index to establish reachability.



	X	Y
a	1	4
b	5	1
c	2	7
d	3	6
e	4	5
f	6	3
g	8	2
h	7	8

(A) (B)

Figure 2: A DAG and its related index, where each row represents the coordinate of a vertex. [1]

The ⩽ relations between the vertices (or points in the plane) are illustrated in Figure 3, where the dashed lines express the reachability area starting from each vertex. [1]

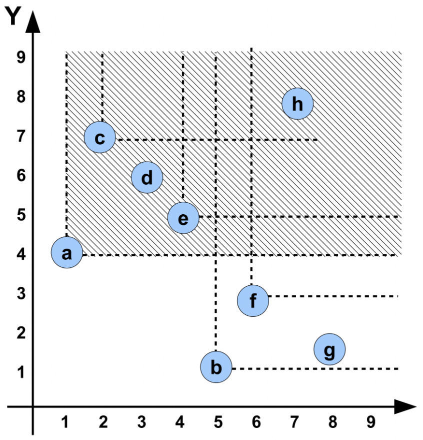


Figure 3: An example of a dominance region. [1]

In Figure 3, even though node h falls within the dominance region of node d, h is not reachable through d. Therefore, FELINE can only be used for pruning nodes and cannot always determine if there h is reachable from d from just comparing dominance regions.

FELINE uses two algorithms, one to build the indexes and the other to establish reachability. The first algorithm, shown in Figure 4 below, generates the coordinates for the index. The x coordinates are determined by an undefined topological ordering algorithm while the y coordinates are calculated using a variation of Kahn's algorithm shown in Figure 4. [1]

```
Algorithm 1: Index construction
Data: $(V, E)$, a directed acyclic graph whose set of vertices, $V$,
      is the set of integers from 1 to $|V|$ and $E \subset V^2$
Result: $(X, Y)$, two vectors of size $|V|$ such that $\forall u \in V$,
        $i(u) = (X_u, Y_u)$
begin
    $X \leftarrow$ TopologicalOrdering$(V, E)$;
    $Y \leftarrow \emptyset$;
    heads $\leftarrow (\emptyset, \dots, \emptyset)$;
    $d \leftarrow (0, \dots, 0)$;
    forall the $(u, v) \in E$ do
        heads$_u \leftarrow$ heads$_u \cup \{v\}$;
        $d_v \leftarrow d_v + 1$;
    roots $\leftarrow \{v \in V \mid d_v = 0\}$;
    while $roots \neq \emptyset$ do
        $u \leftarrow \arg\max_{v \in \text{roots}}(X_v)$;
        // Append $u$ to $Y$
        $Y \leftarrow (Y, u)$;
        // Update $d$ and the roots set
        roots $\leftarrow$ roots $\setminus\{u\}$;
        forall the $v \in heads_u$ do
            $d_v \leftarrow d_v - 1$;
            if $d_v = 0$ then
                roots $\leftarrow$ roots $\cup\{v\}$;
```

Figure 4: FELINE algorithm 1 for index construction. [1]

The second algorithm used in FELINE is in regards to reachability and is detailed below in Figure 5. For two vertices *u* and *v*, it first checks whether *u* is equal to *v*. No search is needed for getting the non-reachability between u and v. Only if *i(u)* $\leqslant$ *i(v)*, FELINE has to explore all vertices inside the region between *u* and *v* recursively via DFS. [1]

```
Algorithm 2: Reachable
Data: $(u, v) \in V^2$, two vertices of the DAG
Result: $r(u, v)$, whether $v$ is reachable from $u$
begin
    if $u = v$ then
        return true;
    if $i(u) \preceq i(v)$ then
        forall the $w \in heads_u$ do
            if $Reachable(w, v)$ then
                return true;
    return false;
```

Figure 5: FELINE algorithm 2 for reachability. [1]

3 Our Implementation

Our implementation was similar to FELINE but had several differences. Most importantly, we added a third set of indices to use for pruning. Additionally, since the FELINE paper did not give specifics on which topological sorting they used to assign values to their x-coordinates, we decided to use a DFS approach to topological ordering. Finally, rather than using the variation of Kahn's algorithm shown in Figure 4, we used a different variation where after removing a vertex from the graph, if any of the vertex's children's indegree was reduced to 0, we would recursively call Kahn's algorithm on those children.

3.1 Data Collection

For our dataset, we chose to use the family of Queen Elizabeth II. Since she is a public figure, much is known about her and her family. Additionally, Wikipedia makes this information easily accessible and even provides hyperlinks to her immediate family members who also have links to their children and parents. Wikipedia, unlike many other sites, permits web scraping. Therefore, we can recursively go through a person's Wikipedia page, connect them to their immediate family members, and recurse on those new members. By doing this, we can construct a family tree of over 7,324 people who are somewhat related to Queen Elizabeth II. To keep this graph acyclic, we decided to define an edge from *u* to *v* as "*u* is a parent of *v*". The graph has 7,324 vertices and 11,451 edges.

3.2 Data Analysis

To analyze our data, we began by distributing three sets of indices to each person in our dataset. The first set of indices was determined through the DFS topological sort described in section 2.3. The second set of indices was determined through the topological sort created by Kahn's Algorithm which was described in section 2.4. The third set of indices was determined through another DFS topological sort. However, we used the TreeMap data structure to iterate through the nodes in the reverse order that we had used when assigning the first set of indices to the coordinates. Thus, this gives us a new unique topological ordering which can be used for additional pruning in the later steps of the algorithm.

After distributing indices to our data, we created a set of 1,000,000 pairings of random people from our graph. Each pairing within the set was analyzed to see if the second item in the pairing could be reached from the first item in the pairing through 4 different approaches. The first approach was a DFS approach that used no pruning. The second approach used just the indices from the DFS topological sort for pruning. The third approach used the indices from the DFS topological sort as well as the indices from the Kahn's algorithm topological sort. The fourth approach used all three third indices for pruning.

For robustness of the scraping algorithm, and to assure that we have no cycles, we added an error checker. One of the features of the error checker prints out any cases where it finds a person who has more than 2 parents. This does happen several times, but we found that this is due to Wikipedia labeling step-parents as real parents. Since this does not cause any cycles in our graph, we did not feel the need to make any changes to this section. Another feature of the error checker is that it asserts that each of the three sorting algorithms is a valid topological ordering. It does this by making sure that for any vertex on the graph, each of the three indices that it was assigned is less than the corresponding indices in all of its children. By repeating this for each vertex on the graph, we can assure that we have a valid topological ordering.

The time it took for each important step in the process was tracked. We recorded the time it took to scrape Wikipedia, assign indices, and answer all 1,000,000 queries using each of the four different approaches. The source code for the data collection and analysis is publicly available on GitHub at https://github.com/akdukaan/royal-family-mapper.

The program then wrote information about each person including their name, indices, and their children to a JSON file which was then graphed using another program we wrote which can be found at https://github.com/akdukaan/royal-family-plotter. We used this second program to plot the data from two different perspectives. The first way was a 3-dimensional scatterplot showing all three indices of each point. The second way was a 2-dimensional scatterplot which shows the indices from the first two sorting algorithms and represents the third index using color.

3.3 Proof of Correctness

Both the DFS topological sort and Kahn's algorithm have been proven to be correct. [3,5] Since our algorithm uses both of these algorithms, it must also provide correct results. Additionally, our error checker described in section 3.2 provides further evidence that each part of the graph is correctly sorted.

3.4 Complexity Analysis

The complexity of Algorithm 1 of FELINE used for index construction is $O(|V|\log|V|+|E|)$ because all edges are enumerated and roots stored as a max-heap structure. Since the algorithm uses a $O(|V|+|E|)$ topological ordering, its final complexity is $O(|V|\log|V|+|E|)$. Our adapted version of FELINE's Algorithm 2 takes time $O(1)$ when for two vertices $u, v \in G$, either u and v belong to the same strongly connected component or the weak dominance relation does not hold. In the worst case, our algorithm may need to traverse the entire graph, in which case the performance complexity of doing so would be $O(|V|+|E|)$. [1] Since our algorithm mirrors FELINE quite closely, the complexity is similar.

3.5 Performance Guarantees

Our algorithm has a bounded performance guarantee since all parts of the algorithm have a bounded performance guarantee. The three parts of our algorithm are the assignment of indices, the pruning of nodes, and the DFS which is performed if the goal node falls within the dominance region of the starting node. All of our index assignment algorithms have a bounded performance guarantee. Since FELINE has a bounded performance guarantee, we know that all parts of our algorithm that were in FELINE also have a bounded performance guarantee. [1] Our algorithm adds another DFS topological sort, and since we already know that DFS topological sorts have bounded performance guarantees, this one must also be bounded. Since all parts of our algorithm have bounded performance guarantees, we know that our algorithm also has a bounded performance guarantee.

4 Results

The total time to construct all three sets of indices was 35ms. This value is very small in comparison to the total time it takes to compute the set of 1,000,000 queries, and would become negligible with a larger quantity of queries.

	Time (s)	**Percentage of original**	**Marginal drop**
No Pruning	60.771	100%	-
1D Pruning	4.291	7.1%	92.9%
2D Pruning	2.447	4.0%	3.1%
3D Pruning	1.624	3.3%	0.7%

Figure 6: A tabular comparison of the time needed to compute 1,000,000 identical queries using varying degrees of pruning.

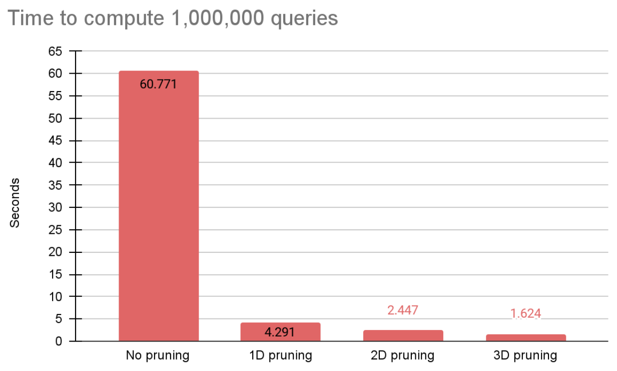


Figure 7: A graphical comparison of the time it takes to compute 1,000,000 identical queries using varying degrees of pruning.

From Figures 6 and 7, we can clearly see that adding additional dimensions of pruning continues to increase the efficiency of querying the graph. The first dimension adds the greatest marginal performance improvement with a 92.9% drop in the query time. Additional dimensions further decrease query time, but the marginal rate decreases as additional dimensions are added. As indicated in Figure 6, the second dimension of pruning shaves 3.1% from the previously pruned version, and the third dimension shaves only 0.7%. Adding any further dimensions could be seen as nonsensical as the marginal performance improvement decreases while the additional marginal storage needed to store the additional set of indices remains constant.

Our algorithm also created two graphs to help visualize the outputted data. Figure 8 is a two-dimensional graph that uses color to show the third dimension. The lighter, more yellow colors depict a higher value for the inverse DFS topological sort indices while the darker, more purple colors depict a lower value for that same index. By using color, we can show all three dimensions of the sorted vertices using a two-dimensional graph. Figure 9 is a three-dimensional graph that shows the same data and color scheme across the z- dimension for easier visualization.

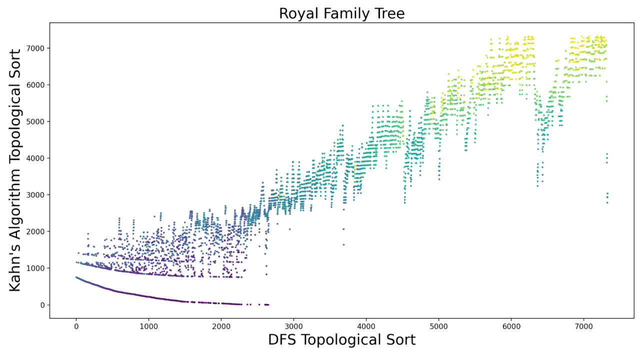


Figure 8: A two-dimensional depiction of the indices of 7,324 vertices on a graph where color represents the values of the third set of indices.

A trend in the data can be observed in figures 8 and 9 regarding a positive correlation between each of the axes. In a graph where there exists an edge pointing from vertex v to vertex u, we know that the topological ordering must always place v before u. This must be true for any topological ordering including the three topological orderings that were used to assign indices to coordinates. Therefore, each edge reduces the range that any point can exist in a topological ordering and therefore, fixes the vertex in the graph toward a specific location in all three axes.

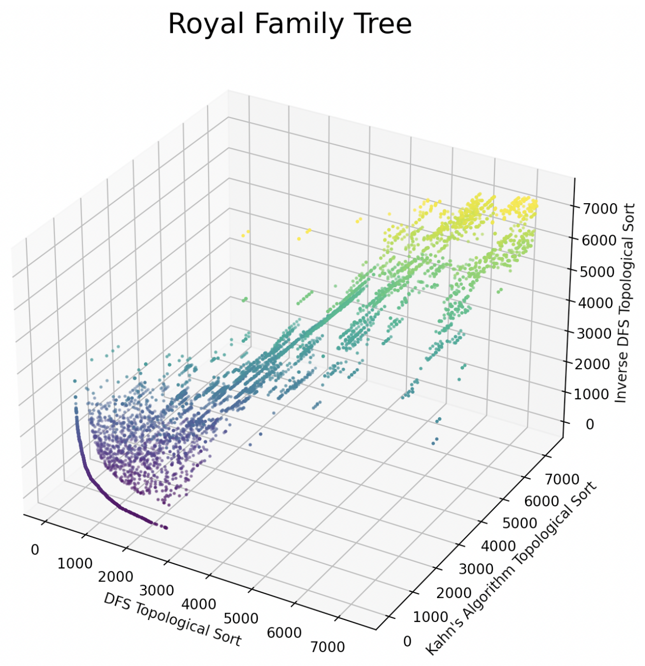


Figure 9: A three-dimensional depiction of the indices of 7,324 vertices on a graph.

5 Conclusion and Future Work

In conclusion, the results show a successful adaptation of the FELINE algorithm for the use case of Queen Elizabeth's family tree. The computational time saved through pruning is quite large and incorporating additional dimensions further improves performance. The robustness and scalability help in its application for further uses.

For future work, it could be important to determine what types of graphs can work best with this algorithm. We would like to consider the effects of graphs of varying size, density, and reachability. The example covered in this paper of Queen Elizabeth II creates a medium sized graph. Running this algorithm on other datasets of smaller and larger sizes could help determine which types of graphs this works best for. We predict that our algorithm would work better for larger graphs as a DFS on a large graph could require a lot of computation time. Our algorithm helps prune nodes from the graph which saves time from having to perform these heavy searches. We would also like to test the effect of varying density and reachability within the graph. The example covered in this paper creates a graph where each node typically has only one or two incoming edges because of the number of parents that people have. Comparing the performance results with graphs of varying density and reachability would help in understanding the relevance of these different types of connections in computational analysis.

Another area of future work could consist of further adding more dimensions for further pruning. It could be interesting to see at what point the efficiency of the graph decreases with additional dimensions due to the needed additional comparisons. Since any given graph has a fixed number of possible topological orderings, it is possible that at one point, the algorithm could be comparing two near-identical orderings which causes a lot of unnecessary computations. We predict that graphs of even larger sizes would benefit more from being topologically sorted in additional ways while smaller-sized graphs would likely be hurt by this. Lastly, comparing all the various conditions including dimensions, density, reachability, size for running the algorithm optimally would help in further refining its use in practical applications.

ACKNOWLEDGMENTS

We would like to acknowledge Professor Yinghui Wu and TA Mengying Wang for their help in formulating our work and providing guidance on our selected topic.

REFERENCES

[1] Veloso, Renê Rodrigues, Loïc Cerf, Wagner Meira Jr, and Mohammed J. Zaki. "Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach." In EDBT, pp. 511-522. 2014 https://openproceedings.org/EDBT/2014/paper_166.pdf

[2] Thulasiraman, K.; Swamy, M.S.N. (1992), "5.7 Acyclic Directed Graphs", Graphs: Theory and Algorithms, John Wiley and Son, p. 118, ISBN 978-0-471-51356-8

[3] “Topological Sorting.” GeeksforGeeks, January 18, 2022. https://www.geeksforgeeks.org/topological-sorting/.

[4] “Introduction to Topological Sort.” Leetcode. Accessed May 1, 2022. https://leetcode.com/discuss/general-discussion/1078072/introduction-to-topological-sort.

[5] “Kahn's Algorithm for Topological Sorting.” GeeksforGeeks, January 21, 2022. https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/.