Identifying Citation Rings Using Connected Components Algorithm

About the Dataset

- Dataset: *cit-HepTh* (Citation Network from High-Energy Physics Theory)
- Link to Data Set: https://snap.stanford.edu/data/cit-HepTh.html
- Features:
  - Nodes: 27770
  - Edges: 352807
  - If a paper *i* cites paper *j*, the graph contains a directed edge from *i* to *j*. If a paper cites, or is cited by, a paper outside the dataset, the graph does not contain any information about this.

Methodology

The objective of this project is to identify potential “citation rings” from within a citation network. Citation rings refers to when “multiple scholars or journals agree to cite each other excessively”(Chawla, 2023) in order to artificially boost the exposure of their work. In order to identify potential rings, the algorithm designed searches for communities of nodes with in the network that are of abnormally high density. In order to identify these communities, the algorithm first removed the weakly connected nodes to reveal a denser underlying structure, then identified the connected components using a depth-first-search algorithm. Once the connected components were identified, their density were calculated(as number of edges over number of nodes) and the ten most densely connected components that raise suspicion of being citation rings were returned.

Output

```
Top 10 Densest Components:
Component 411: Density 5.1333, Vertices: [3594, 5531, 6509, 9529, 10214, 10438, 10937, 11794, 11959, 11960, 13359, 15567, 15927, 17115, 18582]
Component 529: Density 3.4444, Vertices: [4746, 8244, 8832, 10096, 11529, 14223, 14224, 14280, 15343]
Component 1828: Density 2.8571, Vertices: [12365, 14291, 14330, 16659, 16726, 18021, 19537]
Component 9: Density 2.8125, Vertices: [36, 213, 243, 438, 577, 650, 690, 740, 891, 898, 1170, 1361, 1576, 1584, 1592, 1700, 1701, 2318, 3596, 3607, 3846, 4137, 4183, 4469, 5892, 6303, 6739, 6752, 7230, 8856, 11556, 13292]
Component 743: Density 2.5000, Vertices: [6299, 8877, 10130, 10276, 11647, 13301]
Component 18: Density 2.4082, Vertices: [67, 200, 229, 352, 500, 737, 894, 948, 1025, 1363, 1500, 1582, 1855, 1982, 1983, 2303, 2464, 2659, 2748, 2847, 3092, 4763, 5181, 5488, 5490, 5539, 6002, 7961, 8394, 8503, 9628, 9652, 10280, 10308, 10325, 11681, 12750, 13235, 14276, 15808, 16891, 17071, 17077, 19767, 19768, 19811, 19812, 20030, 20035]
Component 21: Density 2.3103, Vertices: [74, 421, 3413, 4239, 4556, 4895, 5660, 9074, 9127, 9690, 9705, 10364, 10407, 11066, 11810, 12583, 12641, 12659, 12695, 13606, 13607, 13801, 14592, 14596, 15841, 16767, 16903, 16958, 17180]
Component 178: Density 2.2759, Vertices: [1476, 2177, 3623, 5604, 5638, 7816, 7840, 8512, 10300, 10372, 11068, 11710, 11728, 11939, 12482, 12588, 12647, 14516, 15494, 16728, 16729, 16779, 17114, 17176, 18179, 18298, 19572, 20136, 20142]
Component 243: Density 2.2500, Vertices: [1992, 2169, 2881, 3131, 3615, 4175, 4177, 4778, 4867, 5574, 6379, 6472, 9013, 10315, 10887, 10936, 11680, 13645, 14508, 14863, 16893, 17037, 17062, 18229]
Component 928: Density 2.2500, Vertices: [7713, 7735, 7736, 10117]
Average Density Across All Components: 0.1898
```

Code Explanation

First, the data from the cit-HepTh.txt file was to be read into a structure that would be able to represent its graph nature. The most suitable data type for representing a graph is a Struct, so in a separate module called graph_subroutines, a “Graph” struct was defined with three fields: n of usize to represent the number of nodes, outedges of type AdjacencyList, and id_to_node as

Vec<usize>(AdjacencyList type is defined as a vector of vectors storing Vertex type, were vertex type is a usize variable). Accompanying this Graph struct were two simple and useful methods: “add_directed_edges” adds directed edges to the graph by appending target nodes to the adjacency list of source nodes, and “sort_graph_lists” method organizes each node's adjacency list in ascending order for consistent traversal and lookup.

To create a graph instance from the tab separated file, a “read_file” method was implemented for the Graph struct, which takes a file name and returns a Graph. This method relied on functions provided by the io and fs modules of the standard library. First opening the file in reading mode with File::Open then reading each line of the file with BuffReader. The nodes of the graph contain arbitrary non-sequential titles, so two HashMaps were initialized to keep track of each index and its NodeId. node_map maps node ids to indices and id_to_node maps indices back to node ids. A for loop iterated over each line in reader.lines(), and split the content of the line into elements of a vector based on the ‘whitespaces’ between them. The first element of the vector was parsed as usize and assigned to “citing_node”, the second element was assigned to “cited_node”. citing_node is then mapped to an index in citing_index, and cited_node is mapped to a node in cited_index. The mapping utilizes a separate function “map_node” that assigns a unique numeric index to a node if it doesn't already have one, and returns that index. Both citing and cited nodes are then pushed into a vector called “edges”. A new instance of Graph is created, and using add_directed_edges and sort_graph_list its edges field is populated with a sorted adjacency list. Returning to the main function, the “Cit-HepTh.txt" file was read into a mutable struct called citation graph using the read_file function.

With the graph prepared, the next step was to generated a “denser_subgraph” that removed the weakest connected nodes. Returning to the graph_subroutines module, a method “calculate_out_degree” was created that calculated the length of the outedges of each node and returned a vector of tuples where each tuple was a node index and its outward edges count. Then a calculate_density method was created to calculate the average out-degree of nodes. It sums the total number of edges and divides it by the count of nodes s. If there are no such nodes, it returns 0.0. With these methods prepared, returning to the main file, a function named “denser_subgraph” was defined that took a mutable reference to a graph as input and returned a new graph that only retained the 75% of nodes that were most densely connected. The function first applied the “calculate_out_degree” method to the graph and then sorted the result by degree. This result is a vector of tuples, to obtain the indices of the 75% of nodes with the most outward edges, iterate over the tuples in reverse order and use the “take()” function with the number representing 75% of nodes as the parameter. This yields a vector of roughly 20,000 indices. To build a graph with only those 20,000 nodes - it was first initialized with an empty vector of length “num_to_keep” as the “outedges” field, and a clone of the “to_keep” vector as its “id_to_node” field. Then, a HashMap named “node_map” was built from iterating over an enumerated “to_keep”, mapping each node to index. Next, a for loop iterates over each retained node, maps it to its new index in the new graph, and adds edges to the new graph only if both the current node and its neighbor are in the retained set “to_keep”. The new_graph is returned at the end of the function.

To identify the connected components within the denser subgraph, a new method “mark_component_dfs” was defined in the graph_subroutines module. It utilized a depth-first search to label all nodes in a connected component of a graph with a unique component number. It starts from a given vertex, and recursively explores all unvisited neighbors, marking them as part of the same component in the component vector. In the main function a for loop iterates over each index in the denser_subgraph and if it is not in the components vector, it passes the index to “mark_component_dfs” to perform a depth first search and mark its component.

With the connected components identified, the necessary parameters for a function that finds the densest components was ready to be built. This function takes the denser_subgraph and components vector as inputs and return a vector of tuples(usize, f64, Vec<usize>) where the first item is the component number, the second is the calculated density, and the third is the vertices in the component. The first step of the function is to group the vertices by component, by instantiating a HashMap called vertex count, and populating it with the component id as a key and the list of vertices belonging to that component as values(as was done in the calculate_average_density function. Then, the function iterates through each component and its vertices, but only components with fewer than 50 nodes are processed(it is unlikely that a citation ring exceeds that size). Within the loop, new subgraphs are built from the components vertices by initializing an empty graph with the same number of nodes as "vertices", then creating a mapping to reindex the component's nodes for the subgraph, and finally adding the edges to the subgraph if both nodes of an edge are within the component. Once the subgraph is built, its density is calculated with the "calculate_density" method and pushed to the densities vector along with the component id number and a clone of the vertices in the component. Once all the components have been iterated through, the tuples are sorted by density, and the first 10 tuples are retained and returned. These components are printed in the main function as "Component __ : Density __, Vertices: [__]".

To justify these nodes being suspicious, their density must be compared to the average density of all components in the graph. To find such density a new function called "find_average_density" was defined that took the denser_subgraph and components vector as input and returned an f64 representing the average density of all components. It first groups nodes into their respective components using the component vector, storing each component's nodes in a HashMap. For each component, it builds a subgraph by extracting only the nodes and edges within the component, then calculates the density of this subgraph using the calculate_density method. Finally, it computes the average density by dividing the total density of all components by the number of components and returns it.

Examining the final results of the program, the average density of all components in the denser subgraph was 0.1898, while the densities of the 10 most densely connected components ranged from 2.2500 to 5.1333. The most densely connected component(Component 411) is 27 times more densely connected than the average. The significant difference between the density of these subgraphs and the average justifies further investigation on whether or not these suspicious groups are indeed “citation rings”.

Works Cited

Khuller, S., & Saha, B. (20YY). *On Finding Dense Subgraphs*. ACM Journal Name. Presented at the 36th International Colloquium on Automata, Languages and Programming (ICALP 2009).

Chawla, D. S. (2023, May 3). Researchers who agree to manipulate citations are more likely to get their papers published. Nature. https://doi.org/10.1038/d41586-023-01532-w

Leskovec, J., & Krevl, A. (2014). SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data