YRF Work Report

- Shaun Mathew

I. Reading Research Papers

1. Unravelling the Influence of Osmolytes on Water Hydrogen-Bond Network: From Local Structure to Graph Theory Analysis

- DETAILED SUMMARY: [LINK]
- Introduction to the basic methods of graph analysis and formation of graphs in computational biology.
- Learnings:
  - How Graphs are Created?
  - What interactions are mapped?
  - Which measures and properties of Graphs are used in the analysis of the molecular network?
  - How are simulations carried out?
  - What are the different settings that can be used for simulating?

2. Dynamics of Hydrogen Bonds between Water and Intrinsically Disordered and Structured Regions of Proteins

- DETAILED SUMMARY: [LINK]
- Introduces the concept of Intrinsically Disordered Proteins and methods of analyzing them using graphs and networks.
- Learnings:
  - How are ionic interactions taken into consideration?
  - What is the behaviour of the water molecules around the IDR and IDPs?
  - Another setting used for biological simulation.
  - Introduced the concept of Relative Surface Area and its impact on interactions.

3. Spectral Graph Analyses of Water Hydrogen-Bonding Network and Osmolyte Aggregate Structures in Osmolyte–Water Solutions

- DETAILED SUMMARY: [LINK]
- Introduces the eigenanalysis of graph networks in the context of Biological Networks.
- Learnings:
    - What are the methods used for eigenanalysis of a Graph network?
    - What is the relevance of these methods in the context of Biological Networks?
    - How do we interpret the derived results from the analysis?
    - Hyperparameters (cutoff distance, bonding angle, etc.) determination.
    - Calculating the radial distribution functions for different sets of molecular interactions.

4. Revised Centrality Measures Tell a Robust Story of Ion Conduction in Solids

- DETAILED SUMMARY: [LINK]
- Introduces about the conduction in ionic solids and the role of the movement of the ions in the conduction
- Learning:
    - How do we model the ionic movements and calculate the conduction through graph analysis?
    - Earlier Simulation Methods in the domain: KMC (Kinetic Monte Carlo) trajectories. More about this: [LINK].
    - Use of centrality measures to analyse and conclude results.

5. Per|Mut: Spatially Resolved Hydration Entropies from Atomistic Simulations

- DETAILED SUMMARY: [LINK]
- Introduces a new method of calculating spatially resolved hydration entropies with the help of techniques like Permutation Reduction and Mutual Information Expansion. (more about the methods : [LINK])
- Learning:
    - Graph Theoretical, Spatial Distribution Movement and MD Simulation methods of mixture analysis
    - Nearest Neighbour Search using the KNN estimator and the KD Tree method.
    - The essence and importance of entropy in biophysical systems.
    - Estimation methods for Entropy Analysis and Calculations in biophysical systems.

II. Understanding Coding Approaches

- Basic Starter Code provided to carry out further optimizations: [LINK]

Learnings:

- Libraries that are commonly used in graph network formation and analysis.
- Explored various functionalities of NetworkX, MDTraj, and other libraries.
- Understanding the application of Graph analysis methods in the given context (Shortest Paths, Centrality Measurements, Clustering Analysis, etc.)
- Understanding the input and output format obtained from the .xtc and .gro files (obtained from carrying out the simulation of the given mixture under appropriate settings with the help of GROMACS).
- Reading and feeding the input file into the specified library functions.

Improvements:

- Further analysis techniques.
- Optimizations for making the code run faster.

III. Learning Improvement Techniques

Improvement in the speed of the code

- Multi-Threading: Concurrent execution of multiple threads within a process, enabling better parallelization and responsiveness.
- Multi-Processing: Multiprocessing involves the simultaneous execution of multiple processes, each with its own memory space, allowing for parallel computation.
- DataStructures for better Clustering Analysis (KDTree): [LINK]
- Faster Shortest Path Calculation Techniques: Explored various techniques like Multiple_BFS (Unweighted graphs), Floyd Warshall method, Dijkstra's, etc. Checked the best and the fastest method and reported the results.
- Approximated Centrality Calculations: Heuristic measures to calculate approximated centrality values in a much quicker manner.

Learning Further Analysis Techniques

- Learned about eigen and spectral analysis techniques of graph networks and their application in the domain of computational biology.
- Learnt about community and motif analysis of graphs that helps in finding clusters and common patterns in the graphs. More about this: [LINK]

IV. Implementing the Ideas

```
import concurrent.futures  # For implementing threading and multiprocessing
from scipy.spatial import cKDTree  # Using kdtree data structure for
optimization
import networkx as nx  # Using the networkx library for graph-based
calculations
```

- Used these libraries to implement the ideas, and the final working code can be found here: [LINK]
- GitHub Repository with all the codes, results and extra details: [LINK]

V. The Code Blocks

A. Calculation of different types of centralities with the help of the NetworkX library

```
def calculate_closeness_centrality(graph):
    return nx.closeness_centrality(graph, distance="weight")

def calculate_betweenness_centrality(graph):
    return nx.betweenness_centrality(graph, weight="weight")

def calculate_degree_centrality(graph):
    return nx.degree_centrality(graph)

def calculate_eigenvector_centrality(graph):
    return nx.eigenvector_centrality(graph, weight="weight", max_iter=1000)

def calculate_katz_centrality(graph):
    return nx.katz_centrality(graph, weight="weight")

def calculate_pagerank_centrality(graph):
    return nx.pagerank(graph, weight="weight")
```

Closeness Centrality

Closeness centrality measures how quickly a node can reach other nodes in the network, taking into account the weighted distances. It is calculated as the reciprocal of the sum of the shortest path distances from a node to all other nodes.

Betweenness Centrality

Betweenness centrality quantifies the influence of a node in controlling the flow of information in a network. It is computed as the fraction of shortest paths passing through a node, considering the weights of the edges.

Degree Centrality

Degree centrality is a simple measure based on the number of edges connected to a node. It is calculated as the ratio of the number of edges incident to a node to the total number of nodes in the network.

Eigenvector Centrality

Eigenvector centrality assesses a node's importance by considering both the node's direct connections and the connections of its neighbors. It is calculated using the dominant eigenvector of the adjacency matrix.

Katz Centrality

Katz centrality measures a node's centrality based on the sum of influence from its neighbors, with attenuation by a constant factor. It considers paths of all lengths and is useful for capturing global influence.

PageRank Centrality

PageRank centrality evaluates the importance of a node based on the structure of the entire network. It assigns higher importance to nodes with more incoming links, and the weights of the edges are considered in the calculation.

B. Loop Optimisation using KDTree

```
kdtree = cKDTree(coord[frame])
potential_acceptors_indices = [acc_index for acc_index in acceptors if
distance_sq(frame, index1, acc_index) <= kd_tree_threshold]
```

A KD-tree (K-dimensional tree) is a data structure used for efficient multidimensional search operations, particularly in spatial search applications. It divides space into regions to organize and optimize the search for nearest neighbors or points within a certain distance threshold.

KD-trees are efficient for applications like spatial databases, machine learning, and computational geometry. They reduce the search space, optimizing nearest neighbor searches in multidimensional datasets.

More about KDTree: [LINK]

C. Different Shortest Path Methods

```
 if shortest_path_mode == "floyd-warshall":
        path = dict(nx.floyd_warshall(hbnet))   ## Applying Floyd Warshall
on the graph
 elif shortest_path_mode == "all-pair-dijkstras":
        path = dict(nx.all_pairs_dijkstra_path_length(hbnet))
 elif shortest_path_mode == "unweighted-all-pair-shortest-paths":
        path = dict(nx.all_pairs_shortest_path_length(hbnet))
```

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm finds the shortest paths between all pairs of nodes in a weighted graph. It computes the shortest path distances by iteratively considering all possible intermediate nodes.

All-Pairs Dijkstra's Algorithm

The All-Pairs Dijkstra's algorithm computes the shortest path lengths between all pairs of nodes in a weighted graph. It is an extension of Dijkstra's algorithm, which efficiently finds the shortest paths from a single source to all other nodes.

Unweighted All-Pair Shortest Paths

The unweighted all-pair shortest paths algorithm computes the shortest path lengths between all pairs of nodes in an unweighted graph. It is based on breadth-first search and is suitable for graphs where edges do not have weights.

D. Threading and MultiProcessing

```
with concurrent.futures.ThreadPoolExecutor() as executor:
            clcen_future = executor.submit(calculate_closeness_centrality, 
hbnet)
            btcen_future = 
executor.submit(calculate_betweenness_centrality, hbnet)

            clcen1 = clcen_future.result()
            btcen1 = btcen_future.result()

with concurrent.futures.ProcessPoolExecutor() as executor:
            clcen_future = executor.submit(calculate_closeness_centrality, 
hbnet)
            btcen_future = 
executor.submit(calculate_betweenness_centrality, hbnet)

            clcen1 = clcen_future.result()
            btcen1 = btcen_future.result()
```

Threading

Threading enables concurrent execution within the same memory space, suitable for I/O-bound tasks, but faces limitations in parallelizing CPU-bound tasks due to the Global Interpreter Lock (GIL).

MultiProcessing

Multiprocessing executes multiple processes with independent memory spaces, allowing concurrent and parallel execution of CPU-bound tasks, overcoming the GIL limitations in languages like Python.

E. Community and Motif Analysis

```
 motif_counts = nx.algorithms.tree.motifs.tree_motif_counts(hbnet, 3)
        print("Motif Counts:")
        for motif, count in motif_counts.items():
            print(f"Motif {motif}: {count}")

        # Community Detection
        communities =
nx.algorithms.community.greedy_modularity_communities(hbnet)

        # Create a mapping of node to community index
        community_mapping = {node: i for i, community in
enumerate(communities) for node in community}

        # Add community information to nodes
        nx.set_node_attributes(hbnet, community_mapping, "community")

        # Draw the graph with nodes colored by community
        pos = nx.spring_layout(hbnet)  # You can use a different layout
algorithm if needed
        node_colors = [hbnet.nodes[node]["community"] for node in hbnet]
        nx.draw(hbnet, pos, node_color=node_colors,
cmap=plt.cm.get_cmap("viridis"), with_labels=True)
        plt.show()
```

Community Analysis

Community analysis in graph theory involves identifying groups of nodes (or vertices) within a network that are more densely connected to each other than to nodes outside the group. These groups are known as "communities."

Motif Analysis

Motif analysis involves identifying recurring, small, and often subgraph patterns within a larger graph that occur more frequently than expected in a random network.

More about this: [LINK]

--- Thank You ---