I. The Code Blocks

I. The Code Blocks

- A. Calculation of different types of centralities with the help of the NetworkX library
  - 1. Plotting the Eigen Centrality Values:
  - 2. Observations
- B. Loop Optimisation using KDTree
  - 1. Observations:
- C. Different Shortest Path Methods
  - 1. Observations
- D. Threading and MultiProcessing
- E. Community Analysis
  - 1. Plotting the graph:
- F. Subgraph Centrality - Threading
  - 1. Observations
- G. Hydrogen Bond Average Frame Life
  - 1. Observation

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

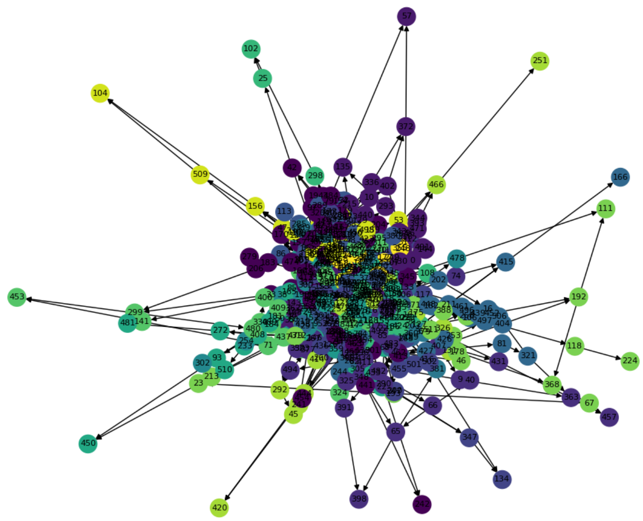
```
def calculate_centrality(G, mode = "closeness", weight_mode = weighted,
approximate_status = approximate, factor = approximation_factor):
    num_nodes = G.number_of_nodes()
    if not approximate_status:
        factor = 1
    k_factor = int(factor * num_nodes)
    if mode == "closeness":
        centrality_value = nx.closeness_centrality(G, distance=weight_mode)
    elif mode == "betweeness":
        centrality_value = nx.betweenness_centrality(G, weight=weight_mode, k =
k_factor)
    elif mode == "degree":
        centrality_value = nx.degree_centrality(G)
    elif mode == "eigen":
        centrality_value = nx.eigenvector_centrality_numpy(G, weight=weight_mode,
max_iter=1000)
    elif mode == "katz":
        centrality_value = nx.katz_centrality(G, weight=weight_mode)
    elif mode == "pagerank":
        centrality_value = nx.pagerank(G, weight=weight_mode)
    return centrality_value
```

- Common Centrality Calculation function for all kinds of centralities required in the graph analysis
- Can set properties like weighted nature of the graph and the approximation factor (for heuristic algorithms) through the functions for faster execution and lesser computational load.

- Eigenvector Centrality:
    - Measures the influence of a node in a network based on the concept that a node's importance is proportional to the importance of its neighbours.
    - Nodes with high eigenvector centrality are connected to other well-connected nodes, indicating their significance in information flow.
- Closeness Centrality:
    - Quantifies how quickly a node can reach other nodes in the network, considering the shortest paths between them.
    - Nodes with high closeness centrality are central in terms of accessibility, playing a key role in efficient information or resource dissemination.
- Betweenness Centrality:
    - Evaluate the importance of a node by measuring the number of shortest paths passing through it.
    - Nodes with high betweenness centrality act as critical bridges or connectors, controlling information flow between different parts of the network.

- Approximation Factor implies that only a set of nodes (approximation_factor*nodes) will be considered for the entire centrality calculation.
- This speeds up the calculation process but gives an approximation error.

1. Plotting the Eigen Centrality Values:



```
nx.set_node_attributes(hbnet, egcen1, 'eigenvector_centrality')

# Visualize the graph with node colors based on eigenvector centrality
node_colors = [egcen1[node] for node in hbnet.nodes()]
pos = nx.spring_layout(hbnet)
nx.draw(hbnet, pos, node_color=node_colors, cmap=plt.cm.RdYlBu, with_labels=True,
font_size=4)

plt.title("Graph with Eigenvector Centrality")
plt.savefig(f"{head}_network_plot_frame_eigen_{frame}.png")
```

- Due to the large number of nodes, it is difficult to visualise and conclude from the graph, We can specifically filter out the plotting for the important nodes based on a specific criterion.

- Eigenvalue centrality is calculated based on the centrality value of its neighbours, too. Here, we get a larger picture as the neighbours of a more influential node become influential themselves, and the information spreads throughout the network.

2. Observations:

1.

Centrality - unweighted	Centrality - weighted
0.338889	1.18837
0.359662	1.246605
0.348153	1.203118
0.355257	1.176847
0.382943	1.251486

- Centrality calculations are fastened by the absence of weights by a factor of 4.

2.

Centrality - multiprocessing	Centrality - threading	Centrality - normal
0.413773	0.451764	0.338889
0.397238	0.527083	0.359662
0.393094	0.698632	0.348153
0.377695	0.636263	0.355257
0.391227	0.530162	0.382943

- Threading and Multiprocessing cause overhead and might decrease the performance of the overall functions.
- Threading Issue:
  - Python Global Locks cause an issue and perform poorly when tasks given are CPU-bound. (not Input-Output based)
  - More on this issue: [LINK]
- Multiprocessing Issue:
  - Overhead of process switching is high and might reduce the overall efficiency rather than increasing it.
  - More on this issue: [LINK]

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]
```

- This is meant to reduce the searching from O(n2) to O(nlogn).
- Though this reduces the time complexity in mathematical terms, there is an additional cost of building the KD-Tree here, which takes up some more computational time.
- When the number of nodes is low OR when a large number of nodes are clustered in a small space, the time taken by this optimisation might actually be larger than the O(n2) brute force approach.

1. Observations:

- Addition of the KDTree doesn't necessarily improve the performance by a great extent for a smaller number of nodes.

Graph Construction without KDTree	Graph Construction with KDTree
2.158541	2.361321
2.196804	2.20435
2.159834	2.440186
2.163356	2.474563
2.229542	2.450693

- Performance with KDTree might be slower at times due to the additional cost of constructing the tree and then also repeatedly searching and reiterating over the potential acceptor set.

```
hbnet.clear_edges()
```

- Clearing just the edges, instead of the whole graph.
- This avoids re-creation of the graph from scratch when the number of nodes is large.
- Saves time on large graphs.

C. Different Shortest Path Methods

```
def calculate_shortest_path(G, mode = default_shortest_path_mode):
    if mode == "floyd-warshall":
        path = matrix_to_dict(nx.floyd_warshall_numpy(G))
    elif mode == "all-pair-dijkstras":
        path = dict(nx.all_pairs_dijkstra_path_length(G))
    elif mode == "unweighted-all-pair-shortest-paths":
        path = dict(nx.all_pairs_shortest_path_length(G))
    return path
```

- Different modes of shortest-path calculations are used here
    - Floyd-Warshall: Most effective when we have a weighted dense graph.
    - All-Pair-Dijkstra: Most effective when we have a weighted sparse graph.
    - Unweighted-All-Pair: Most effective when we have an unweighted graph.
- Using unweighted algorithms fastens the time taken by a factor of 4 - 10, depending on the algorithms.

1. Observations

Shortest Path - weighted	Shortest Path - unweighted
0.436986	0.058031
0.534913	0.06586
0.494195	0.064521
0.450115	0.061458
0.599815	0.073681

- The unweighted shortest path performs much better than the weighted Dijkstra for the sparse graph representing the hydrogen bond network.
- The speed improves by a factor of 9-10.

D. Threading and MultiProcessing

```
centrality_types = ["closeness", "betweeness", "eigen"]
if multiprocessing:
        with ProcessPoolExecutor() as executor:
            results = list(executor.map(calculate_centrality, [hbnet]*3,
centrality_types))
        clcen1, btcen1, egcen1 = results

if threading:
        with ThreadPoolExecutor() as executor:
            results = list(executor.map(calculate_centrality, [hbnet]*3,
centrality_types))
        clcen1, btcen1, egcen1 = results
```

- Code for threading and multiprocessing.
- Doesn't improve performance by a lot as compared to when threading and multiprocessing are not used.
- Can cause a dip in performance due to the reasons stated [ABOVE]

E. Community Analysis

```
communities = nx.algorithms.community.greedy_modularity_communities(hbnet,
cutoff=1)
```

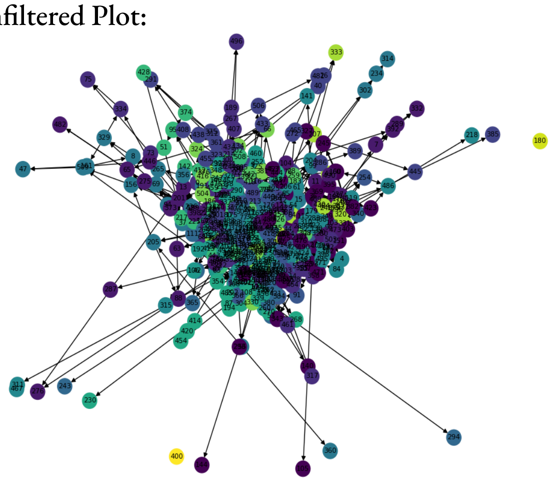
1. Plotting the graph:

```
plt.figure(figsize=(10, 8))
pos = nx.spring_layout(filtered_hbnet)
node_colors = [filtered_hbnet.nodes[node]["community"] for node in
filtered_hbnet]
nx.draw(filtered_hbnet, pos, node_color=node_colors, cmap=colormaps["viridis"],
with_labels=True, font_size=4, node_size = 200)
```

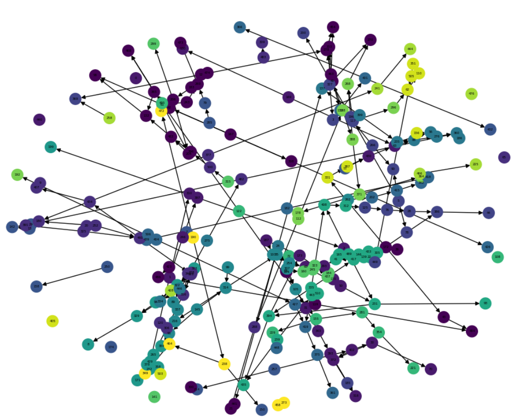
Due to too many number of nodes, we can't understand or conclude from the graph properly, so we can filter out the important nodes based on some criterion.

```
average_degree = 0
max_degree = 0
for node, degree in hbnet.degree():
     average_degree += degree
     max_degree = max(degree, max_degree)
average_degree /= hbnet.number_of_nodes()
degree_threshold = min(average_degree * 1.1, max_degree*0.8)
filtered_nodes = [node for node, degree in hbnet.degree() if degree >=
degree_threshold]
filtered_hbnet = hbnet.subgraph(filtered_nodes)
```

Unfiltered Plot:



Filtered Plot:



F. Subgraph Centrality - Threading

```
def calculate_centrality_for_subgraphs(hbnet, centrality_type):
    connected_components = list(nx.weakly_connected_components(hbnet))
    subgraphs = [hbnet.subgraph(component).copy() for component in
connected_components]

    with ProcessPoolExecutor() as executor:
        centrality_list = list(executor.map(calculate_centrality, subgraphs,
[centrality_type]*len(subgraphs)))

    centrality_combined = {}
    for node_centrality_detail in centrality_list:
        centrality_combined.update(node_centrality_detail)

    centrality_combined = dict(sorted(centrality_combined.items()))
    return centrality_combined
```

- Here we split the subgraph into disconnected components and carry out centrality analysis for each component separately using a thread.
- Since the overall network is well connected, we don't observe any gain in the overall performance using this method on a pure water sample.

1. Observations

Centrality - normal	Centrality - subgraph_threading
0.338889	0.504788
0.359662	0.477514
0.348153	0.54854
0.355257	0.608815
0.382943	0.646934

- Here the threading slows down the performance as multiple threads are created for singular nodes which are not performance costly, but give a big overhead due to thread management.

G. Hydrogen Bond Average Frame Life

```
old_hbond_list = []
latest_hbond_list = []
average_hbond_life_mapping = {}

def update_mapping(frame_no, max_frames=2):
    for key in set(latest_hbond_list) - set(old_hbond_list):
        average_hbond_life_mapping.setdefault(key, {'Start': frame_no, 'End': frame_no,
'Time_List': []})

    for key in set(old_hbond_list) - set(latest_hbond_list):
        average_hbond_life_mapping[key]['End'] = frame_no
        average_hbond_life_mapping[key]['Time_List'].append(
            average_hbond_life_mapping[key]['End'] -
average_hbond_life_mapping[key]['Start']
        )

    if frame_no + 1 == max_frames:
        for key in set(latest_hbond_list):
            average_hbond_life_mapping[key]['End'] = frame_no + 1
            average_hbond_life_mapping[key]['Time_List'].append(
                average_hbond_life_mapping[key]['End'] -
average_hbond_life_mapping[key]['Start']
            )
```

1. Observation

Strength	Number of Bonds
1/5	1440
2/5	579
3/5	291
4/5	105
5/5	48

- The average number of frames a hydrogen bond lasts continuously is defined as the strength of the bond.
- Strength = $\dfrac{\sum_{\text{live ranges}} len(\text{live range})}{count(\text{live range})}$

Credibility:

shaun_outputs > Centrality_histogram_.dat

#Frame	WatClCen	WatBtCen	
0.005000	96	0.002500	298
0.015000	0	0.007500	365
0.025000	0	0.012500	442
0.035000	0	0.017500	452
0.045000	1	0.022500	352
0.055000	10	0.027500	268
0.065000	88	0.032500	184
0.075000	398	0.037500	108
0.085000	922	0.0[illegible]	47
0.095000	878	0.047500	19
0.105000	164	0.052500	13
0.115000	3	0.057500	8
0.125000	0	0.062500	3
0.135000	0	0.067500	0
0.145000	0	0.072500	0
0.155000	0	0.077500	0
0.165000	0	0.082500	1
0.175000	0	0.087500	0
0.185000	0	0.092500	0
0.195000	0	0.097500	0
0.205000	0	0.102500	0
0.215000	0	0.107500	0
0.225000	0	0.112500	0
0.235000	0	0.117500	0
0.245000	0	0.122500	0

hamsa_maam_outputs > Centrality_histogram_.dat

#Frame	WatClCen	WatBtCen	
0.005000	96	0.002500	298
0.015000	0	0.007500	365
0.025000	0	0.012500	442
0.035000	0	0.017500	452
0.045000	1	0.022500	352
0.055000	10	0.027500	268
0.065000	88	0.032500	184
0.075000	398	0.037500	108
0.085000	922	0.042500	47
0.095000	878	0.047500	19
0.105000	164	0.052500	13
0.115000	3	0.057500	8
0.125000	0	0.062500	3
0.135000	0	0.067500	0
0.145000	0	0.072500	0
0.155000	0	0.077500	0
0.165000	0	0.082500	1
0.175000	0	0.087500	0
0.185000	0	0.092500	0
0.195000	0	0.097500	0
0.205000	0	0.102500	0
0.215000	0	0.107500	0
0.225000	0	0.112500	0
0.235000	0	0.117500	0
0.245000	0	0.122500	0

- Similar centrality outputs when run for 5 frames

shaun_outputs > hbweight_histogram_.dat

#weight	wwhist
0.025000	0
0.075000	0
0.125000	0
0.175000	0
0.225000	0
0.275000	0
0.325000	0
0.375000	0
0.425000	0
0.475000	0
0.525000	0
0.575000	0
0.625000	0
0.675000	0
0.725000	0
0.775000	0
0.825000	0
0.875000	0
0.925000	0
0.975000	0
1.025000	4284
1.075000	0

hamsa_maam_outputs > hbweight_histogram_.dat

#weight	wwhist
0.025000	0
0.075000	0
0.125000	0
0.175000	0
0.225000	0
0.275000	0
0.325000	0
0.375000	0
0.425000	0
0.475000	0
0.525000	0
0.575000	0
0.625000	0
0.675000	0
0.725000	0
0.775000	0
0.825000	0
0.875000	0
0.925000	0
0.975000	0
1.025000	4284
1.075000	0

- Similar weight outputs when run for 5 frames

Network_Stats_.dat

shaun_outputs > Network_Stats_.dat

#Frame	#Hbonds	#HBNetNodes	#HBNetE
0	851	512	851
1	862	512	862
2	856	512	856
3	870	512	870
4	845	512	845

Network_Stats_.dat

hamsa_maam_outputs > Network_Stats_.dat

#Frame	#Hbonds	#HBNetNodes	#HBNetEd
0	851	512	851
1	862	512	862
2	856	512	856
3	870	512	870
4	845	512	845

- Similar network stats when run for 5 frames