TOPICS IN DATA MANAGEMENT: GRAPH DATABASES

TEAM 2 PROJECT: CPI CONSTRUCTION

HARNISHA GEVARIA (HGG5350) KUSHAL GEVARIA (KGG5247) KU, WEI-YAO (WXK6489)

1. Design and implement query decomposition described in Section 3. (**Submitted CPI_QueryDecomposition.java**)

In query decomposition, we split the graph into three sub-structures namely core, forest, and leaf as mentioned in the paper [1]. The first part it to decompose the whole query graph into the core and temporary forest. The core is defined as a structure where all the nodes are connected, and it forms a proper graph where there are no nodes with degree one edges. To find this we use the core-forest decomposition technique mentioned in [1], where we recursively remove all the nodes with degree one edges until there are no nodes left with one-degree edges. This remaining structure is called the core of the query graph and we store it in a different array list in java. Now to build the rest of the forest structure we follow this:

```
// creating the remaining forest
for (int key : core.keySet()) {
    forest.get(key).removeAll(core.get(key));

    // if no more elements left then that node is not part of the forest
    if (forest.get(key).isEmpty()) {
        forest.remove(key);
    }
}
```

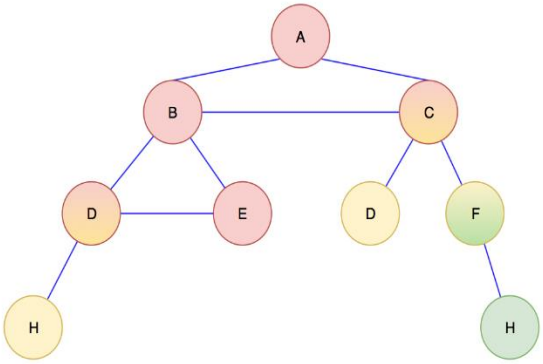
We initialize the forest structure with whole query graph first, then in the above code we iterate through all the core nodes, and for those nodes in the forest, we remove the edges that are present in the core. If after removal of the core edges, there are no more edges left then that node is completely part of the core and hence we remove that node from the forest. We do this for all the node present in the core.

Now we have temporary forest which we need to decompose further into forest and leaf called as forest leaf decomposition. Now we create leaf structure from the temporary forest using the code below:

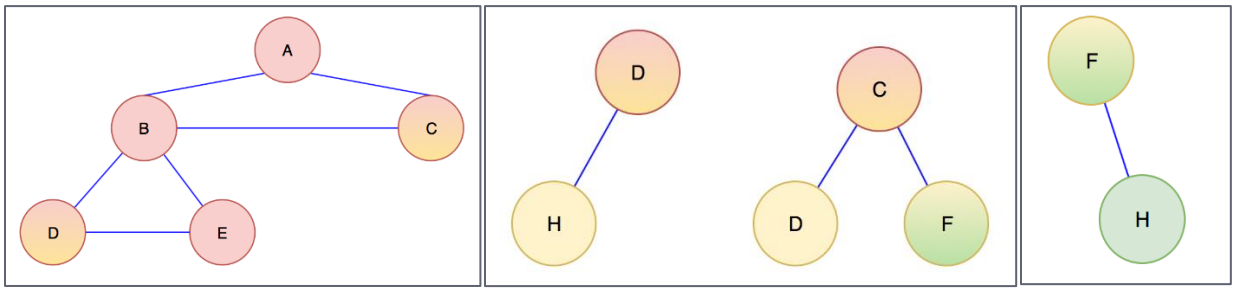
```
private void performForestLeafDecomposition() {
    leaf = new HashMap<>();

    for (int key : forest.keySet()) {
        if (forest.get(key).size() == 1) {
            /*
             * so if a key(node) or it's one child is part of the core then
             * that node is considered as the part of forest and not the
             * leaf
             */
            if (core.containsKey(forest.get(key).get(0)) || core.containsKey(key)) {
                continue;
            }
            int parentKey = forest.get(key).get(0);
            if (forest.get(parentKey).size() <= 2) {
                leaf.put(key, new ArrayList<>(forest.get(key)));
                leaf.put(forest.get(key).get(0), new ArrayList<>(Arrays.asList(key)));
            }
        }
    }
```

In the above code, we traverse through all the nodes in the temporary forest, and for all those node that contains only one edge, and if that node is not the part of the core and if it's parent's degree is less than or equal to 2 then only we add that node to leaf structure, otherwise it remains the part of forest structure. For instance, consider the example mentioned below:



In this structure D and E would be part of the forest and not leaf because though H has the edge to one degree but with our assumption we only add this to leaf if the degree of its parent i.e. D should not be part of the core, and if it is not part of the core, then it's degree should also be less than equal to 2, which is false here since the degree of D is 3 so we don't add it to leaf. But all these conditions satisfy for the node H so we add H to leaf and F-H to leaf structure. Once we have the leaf structure, we remove those from the forest structure. The above query graph is finally decomposed into following core, forest, and leaf.



It is possible to have an empty core, forest or leaf structure, in that case, the above code will give an empty structure which we can check later to see if anything is empty.

2. Design and implement CPI using Neo4j along with root node selection described in Appendix A.6. Report your design decisions. (**Submitted a CPI_Generation.java file which contains the method for root selection.**)

We decided to store all the query structure and the candidate sets in the form of Hash maps where the keys is the query node numbers and the values contains the array list of the candidates, which itself are represented by the neo4j node ids, so that accessing the underlying structure from neo4j in embedded form becomes much faster and easier.

For root node selection, the procedure mentioned in the paper [1] is to select the root node from the core structure. So, for root node selection, we first compute the candidate set for the nodes in the core, using the technique of search space computation of naïve subgraph. We add the node in the candidate set only when it's degree is greater or equal to the degree of query node and its name i.e. label matched with query node. This is how we build the candidate set for all the query nodes present in the core.

We then sort the order of query node based on the size of their candidate set, and for top three candidates with smallest candidate set size, for each candidate in that set, we verify that node using CandVerify i.e. algorithm 6 from the paper [1]. If it does not pass the verification we remove that node from the candidate set.

```
for (Object key : sortedHashMap.keySet()) {
    if (j == 3) {
        // found top three candidates for root node
        break;
    }
    int u = (int) key;
    // System.out.println("Top three nodes for root: " + u);

    ArrayList<Integer> removeNonCnd = new ArrayList<>();
    for (int v : candidateSet.get(u)) {
        if (!CandVerify(u, v)) {
            removeNonCnd.add(v);
        }
    }
    candidateSet.get(u).removeAll(removeNonCnd);
    j++;
    candidateSize = candidateSet.get(u).size();
    queryNodeDegree = qd.queryGraph.get(u).size();
    if (minValue > candidateSize / queryNodeDegree) {
        root = u;
        minValue = candidateSize / queryNodeDegree;
    }
}
// System.out.println("Root Node:" + root);
```

Once we have the reduced candidate set, we perform the division and select the node with the smallest value of $\frac{|C(u)|}{d_q(u)}$. We implemented CandVerify for our structure following the algorithm mentioned in [1].

```
Top three nodes for root: 3
new Candidate Set: 135
Top three nodes for root: 4
new Candidate Set: 280
Top three nodes for root: 5
new Candidate Set: 340
Root Node:3
```

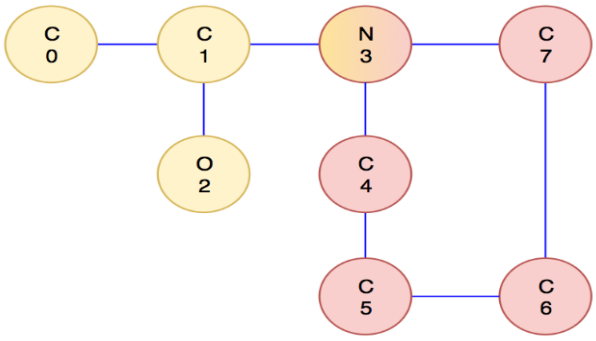


Figure 1: Root node selection

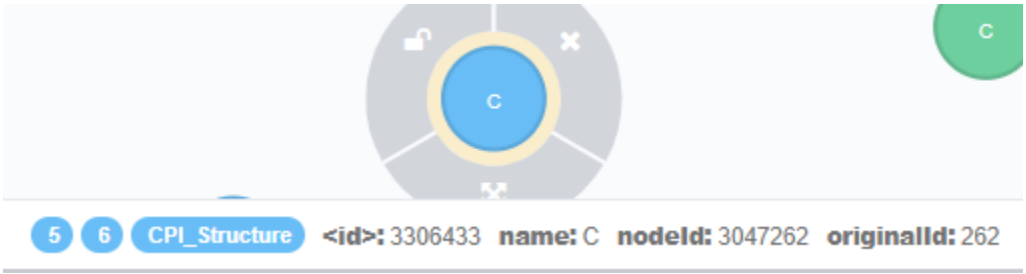
As we can see above the top three nodes we get are 3, 4, and 5, we reduce the candidate set using CandVerify and then select the one with the smallest value of the equation which in the above case is 3.

One of the edge case to handle here is that if the core is Empty. In that case, the root node itself is the core. For this, in our implementation, we pass forest as the query graph and select the root for the forest using the same technique mentioned above and then continue the CPI construction for the forest using that root node. Here since the core is just a single node, there is no point in building the CPI for one node and verifying everything, because that node will be anyways verified with forest. So, in our code, we just skip core matching, if the core is empty and continue normally.

3. Theorem 4.1 states that CPI can be used to compute subgraph matching. Using the ground truth in Proteins, decompose the different query graphs, compute CPIs for the decompositions, and provide a Java program that automatically checks if all the expected solutions are contained in the computed CPIs. (**Submitted a CPI_Generation.java and GenerateProtiensDatabase.java**)

For computing the CPI Structure, we followed the exact algorithm 3 and 4 provided in the paper [1]. We first build the BFS query tree from the query graph starting from the root node, and store the nodes level wise, as the algorithm goes through nodes level wise. Firstly, for all the potential candidates of the root node, we perform candidate filtration on them using and CnVerify method and then add them to the candidate set of the root node. Once we have this we mark root node as visited and continue the algorithm from level 2 to the max level present. For query nodes in each level, we perform forward candidate generation. We store all the necessary information in Hash maps with data for the same node having the same key, this was accessing all the element from the key becomes convenient. The property v.cnt is stored as the attribute of neo4j node which can be accessed easily using the internal id which is stored in the candidate set. We have initialized the cnt for each node in target graph to 0 while loading the protein database in the code **GenerateProtiensDatabase.** Once we follow the whole procedure and get the candidate set for all the nodes at one level, we continue the algorithm to perform backward candidate pruning, in which we use variable v.cnt and CNT to identify the wrong candidate and prune them.

Once we have the candidate set, we create the nodes for each of this candidates in neo4j with attributes as “id” being the internal neo4j id of that node, “originalId” being the actual Id of that node in proteins which can be used to verify with ground truth and “name” as the label of the node like C, N etc and the labels as “CPI_Structure” and the query node of which it is a candidate for instance if v1 is the candidate of u0, then we add label 0 to the new node v1 we created. Once we have these nodes in place we continue with the algorithm 3 and perform adjacency list generation, where instead of storing this structure in Java, we directly build the edges between the new nodes we generated which can be again easily accessed with the help of label “CPI_Structure” and their attributes. While performing this procedure, we keep track whether there exists a path already from node A to node B to avoid building multiple paths, this helps us keep the CPI structure compact and clean, as the extra edges are of no use here. Once we complete this part of the algorithm, we continue to next level and repeat the whole procedure again.



After we have created a CPI_Structure, we follow the algorithm 4 to perform the bottom up refinement and here we directly use the new neo4j structure and the candidate set from the previous algorithm to perform each step and refine our structure furthermore. Example CPI structure is shown below:

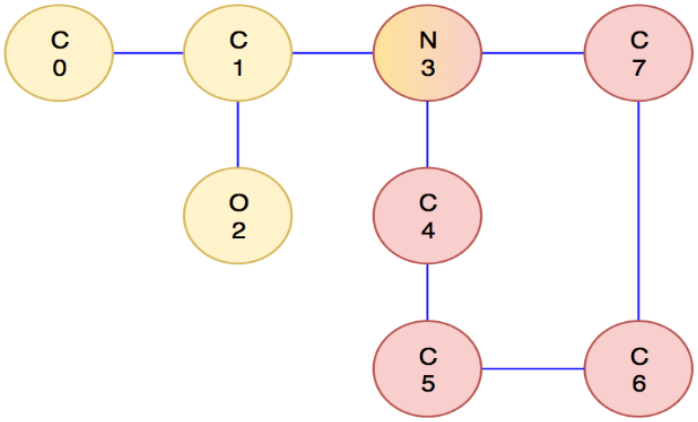


Figure 2: Query Graph

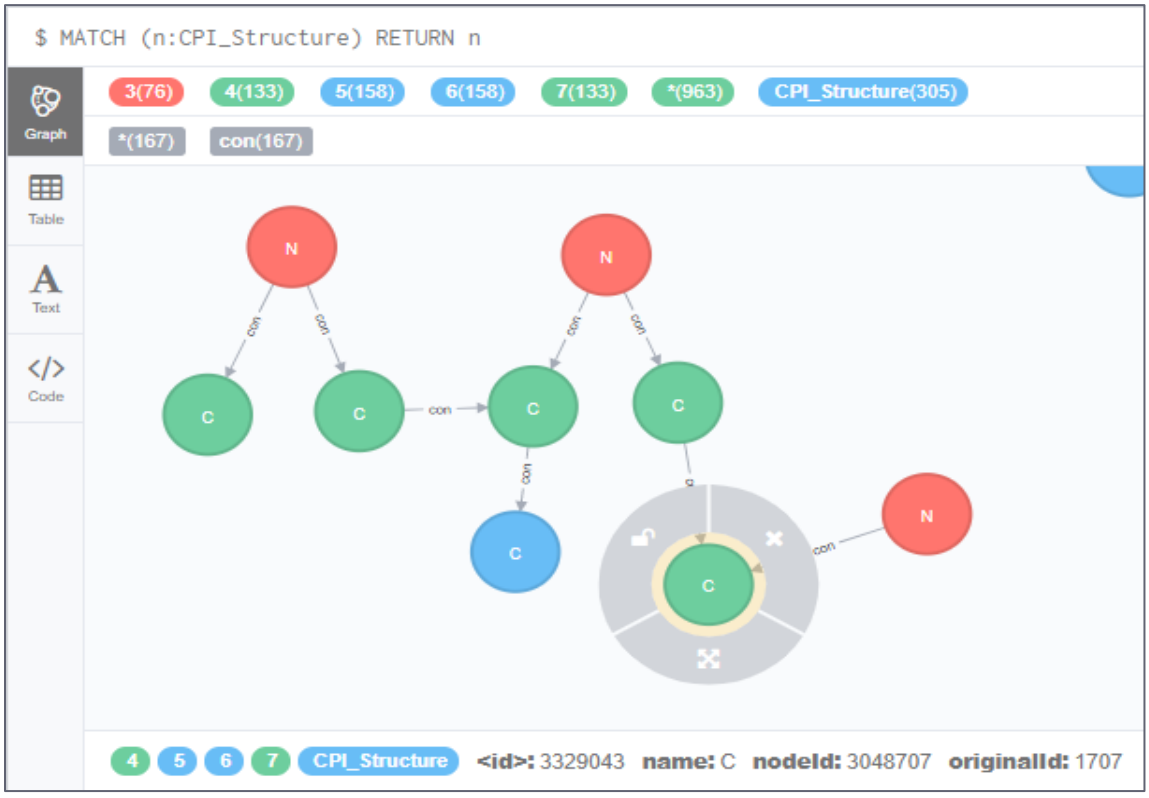


Figure 3: CPI for Core

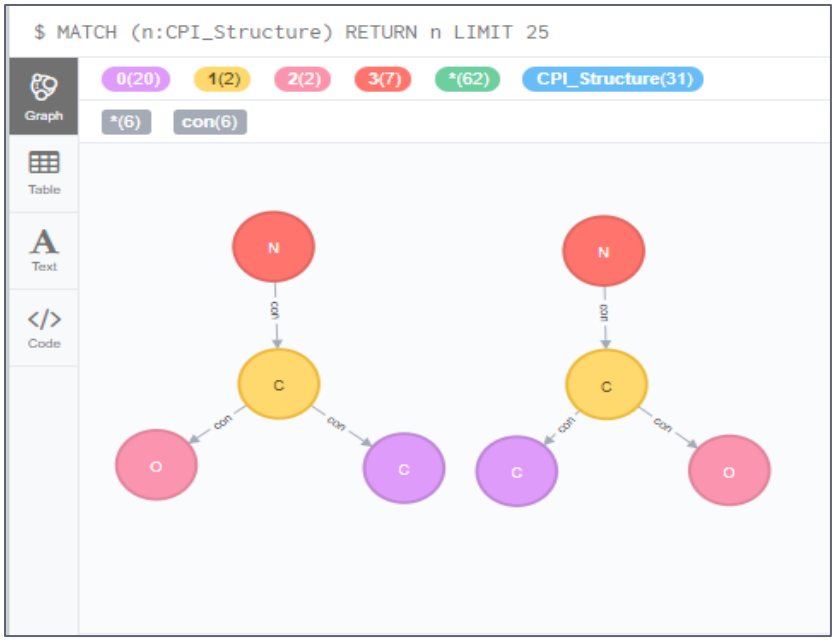


Figure 4: CPI for forest

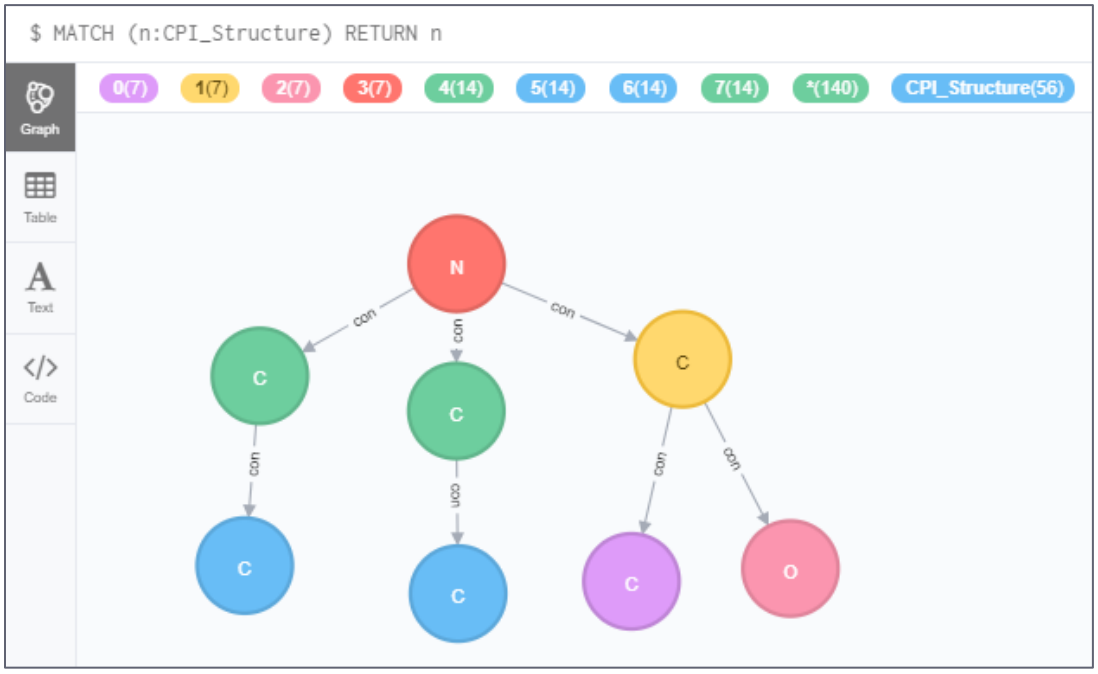


Figure 5: CPI for whole query graph

After compute and store the CPI result. The method “computeGroundTruthSubsetMatching” would read in ground truth result from a file and match it to CPI result in Neo4j. Check if the node ID present in the ground truth is a subset of CPI result and print out the matching result if necessary.

4. Report your performance results. (**Submitted CPI_mainFile.java**)

We test all query graph with different protein size (10 query per size and total 60 query graph) and match with five random picked target files (backbones_140L.grf, backbones_1NOD.grf, backbones_2X3T.grf, human_1B9G, human_2KSA.grf). The experiment will run the process as descript in Question 3 which generate CPI for the whole and decomposed graph and perform the subset matching with the ground truth. The program prints out the query execution time of each query pair, the ground truth math result only If the subset match fails, and the average time after finishing the whole set of the query.

The example output as:

----------Query Set #50 ----------
Target File: human_2KSA.grf
Query File: saccharomyces_cerevisiae_2GB8.32.sub.grf
Time for decomposed query graph: 0.32186666 (Min)
Time for whole query graph: 0.26678333 (Min)

Total Query Count: 50
Total Time for this 50 query graph (decomposed and whole): 16.60745 (Min)

Average Time for Decomposed Match: 0.21619566 (Min)
Average Time for the Whole Match: 0.11594401 (Min)
Average Time per Query: 0.332149 (Min)

Our experiment results indicate that all the ground truth subset match is a success and the performance as shown in following tables and the figure 6:

Decomposed	**Protein 8**	**Protein 16**	**Protein 32**	**Protein 64**	**Protein 128**	**Protein 256**
Max (Min)	0.2547	0.3771	0.6458	0.7971	1.9941	3.7188
Min (Min)	0.1130	0.0198	0.0240	0.0293	0.0301	0.0768

Table 1: The minimal and maximal execute time with different query size in decomposed graph set.

Whole	**Protein 8**	**Protein 16**	**Protein 32**	**Protein 64**	**Protein 128**	**Protein 256**
Max (Min)	0.1384	0.2758	0.39365	0.6909	1.0577	1.7942
Min (Min)	0.0143	0.0170	0.0126	0.0167	0.0107	0.0227

Table 2: The minimal and maximal execute time with different query size in whole graph set.

Average time (Min)	Protein 8	Protein 16	Protein 32	Protein 64	Protein 128	Protein 256
Decomposed	0.0675	0.1269	0.2161	0.3284	0.567	1.1651
Whole	0.0362	0.0924	0.1159	0.2340	0.1178	0.1413

Table 3: Average execute time in different query size.

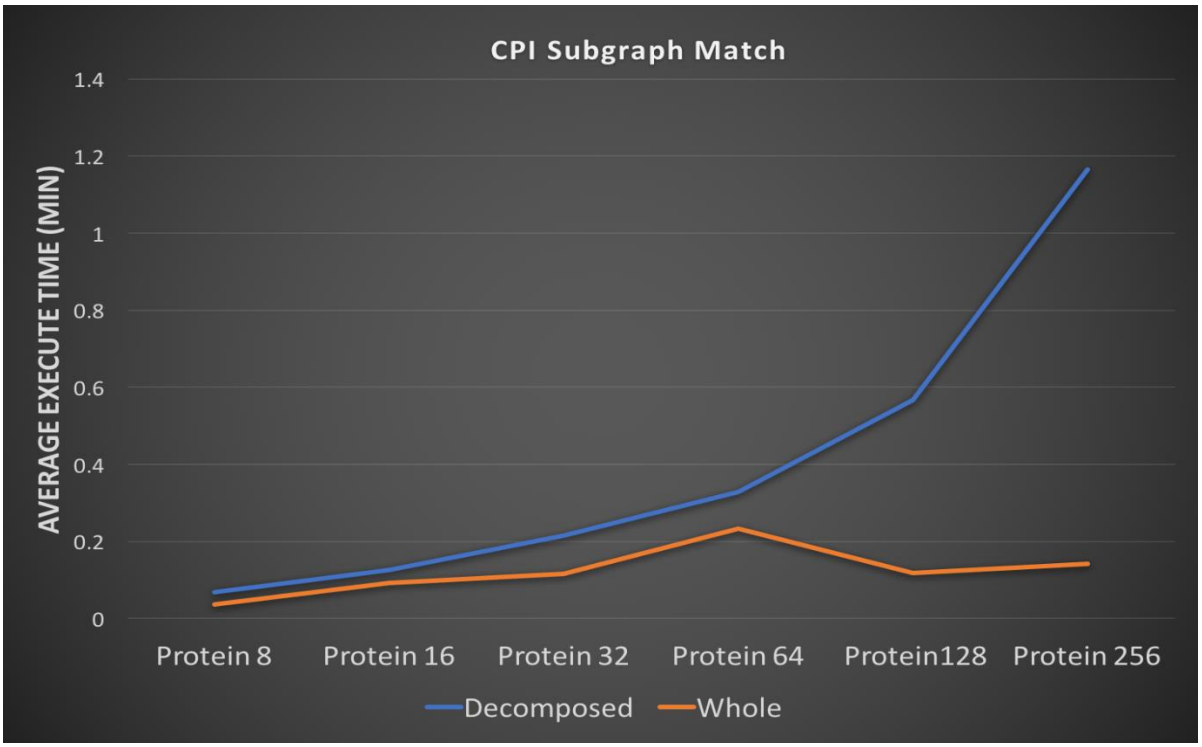


Figure 6: Average time comparison with different protein size.

We can notice that the query time of decomposed set is always longer than the whole graph set that is because the decomposed set with the core, forest and leaf contain many overlapping nodes, so we can expect the longer execute time in CPI construction and subset matching. We are unable to compare CPI performant with the previous algorithm like VF2 plus because we didn't perform the framework matching and the best search ordering in the task.

Reference:

[1] Bi F, Chang L, Lin X, Qin L and Zhang W. *Efficient Subgraph Matching by Postponing Cartesian Products* in International Conference on Management of Data, 1199-1214, July 2016.