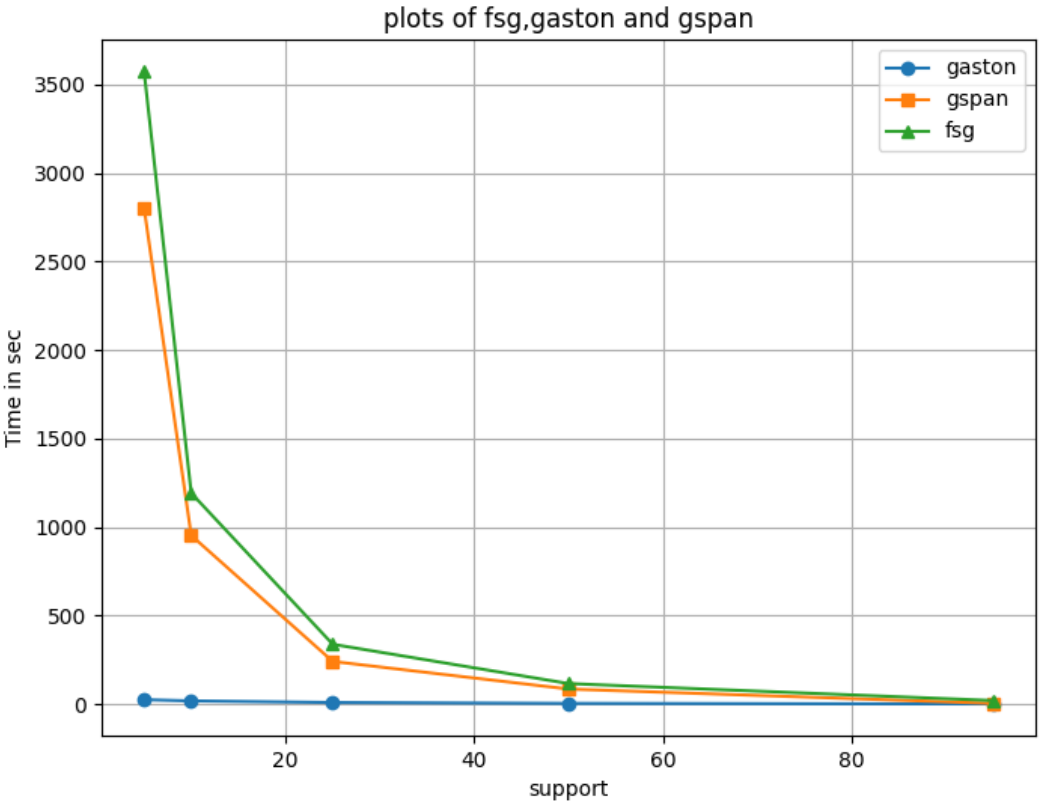
Q1)



Minimum Support	FSG (sec)	Gspan (sec)	Gaston (sec)
95	20	4	1
50	116	85	4
25	338	241	9
10	1192	953	18
5	3576	2804	25

- As evident from the table above, the approximate runtime trend of FSG, Gspan and Gaston is FSG > Gspan > Gaston. As the minimum support threshold decreases the runtime increases exponentially due to increase in number of fragments. FSG uses joint based approach while Gspan and Gaston uses pattern growth approach.

- The likely reason of why Gaston is outperforming or why Gspan and FSG are nearly equal in performance does depend on several factors such as number of graphs in dataset, average size of graphs, average size of frequent subgraph, number of different labels and edges etc.

- Comparing the performance of FSG on this dataset against others, we can see that it requires more time for this yeast dataset, this is because in the yeast dataset, edge and vertex labels may have non-uniform distribution. As we decrease the minimum support, larger frequent

subgraphs start to appear which generally contain only carbon and hydrogen and a single bonding type.

- FSG does not perform good when there is a smaller number of different vertexes and edges for same size of graphs and potential subgraphs in dataset. This is because as the number of edge and vertex labels increases there are fewer automorphisms and subgraph isomorphisms required, which leads to fast candidate generation and frequency counting. Also, by having more edge and vertex labels, we can effectively prune the search space of isomorphism.

- FSG works well when the graph is sparce and number of different labels are more but in our dataset number of different labels is around 11. Same goes for Gspan as well, as it also works well with diverse graphs who has tree like structure that is consistent with DFS structure of Gspan. Therefore, both FSG and Gspan have to do larger number of subgraph isomorphism tests. Although Gspan should outperform FSG by high margin, authors in $\text{paper}^{[1]}$ discussed about the performance comparison between FSG and Gspan and for synthetic dataset FSG turns out to be better than Gspan. Thus, we can say that the datasets used by us might have different distribution than the dataset used by the Gspan authors where they found Gspan to have an edge over other algorithms.

- On the other hand, Gaston uses embedding list structures that helps in speeding up subgraph isomorphism test especially for large fragments. As Gaston uses cluster type structure the time taken in checking isomorphism is lower compared to other algorithms. At lower minimum support the number of frequent subgraphs will be more so isomorphism checking needs to be done many times which gives edge to Gaston over others.

REFERENCES

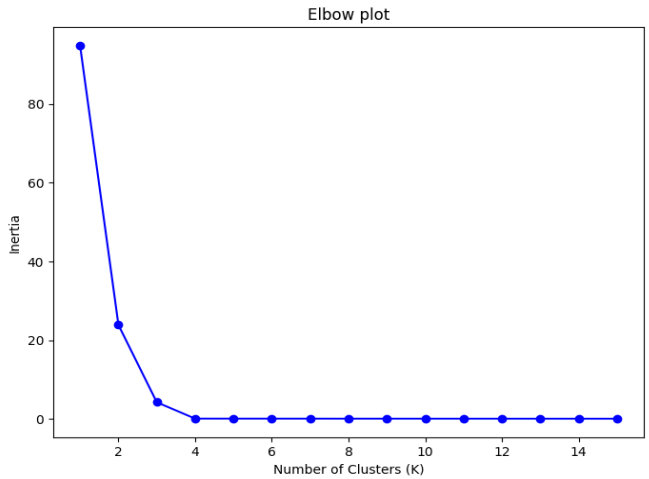
[1] Kuramochi, Michihiro, and George Karypis. "An efficient algorithm for discovering frequent subgraphs." *IEEE transactions on Knowledge and Data Enginessering* 16.9 (2004): 1038-1051.

Q2) For K-means clustering, K-means library which resides in scikit-learn library is used. We are using 4-dimensional dataset namely 'AIB232079_generated_dataset_4D'. K-means uses NumPy and SciPy library in backend computation which are using optimized c and Fortran libraries. K-means uses init initialization which has several initialization methods such as kmeans++, random, array like etc. out of which kmeans++ is preferable as it follows these steps.

- select first cluster centroid randomly. Calculate the squared Euclidean distance between each data point and the nearest centroid that has already been chosen.
- For each data point, compute the probability of selecting it as the next centroid. The probability is proportional to the squared distance from the data point to its nearest existing centroid. In other words, data points that are farther away from existing centroids are more likely to be chosen as new centroids.
- Randomly select the next centroid based on these probabilities.
- Repeat the process of calculating distances and selecting centroids until you have chosen k centroids.

After that fit method is called onto the algorithm in which It iteratively refines the cluster assignments of data points and updates the cluster centroids until convergence. The goal is to minimize the within-cluster variance (inertia) or another specified criterion. The algorithm stops when one of the stopping criteria is met, such as a maximum number of iterations or when the centroids no longer change significantly. After fitting the model, each data point in the input dataset is assigned to one of the clusters based on its proximity to the cluster centroid.

For plotting elbow plot, we have run kmeans algorithm for different number of clusters ranging from 1 to 15. For every number of cluster, we checks Within Cluster Sum of Squares (WCSS) method to quantify the quality of clusters. Smaller WCSS represents dense and compact clusters. Result will be plotted on graph by keeping varying number of clusters on X-axis and WCSS values on Y-axis. The point at which WCSS values stops changing significantly is called elbow point and corresponding cluster number is the optimal number of clusters that should be used in kmeans algorithm. In our experimental dataset, the observed value of k comes out to be 4. The elbow point represents the balance between model complexity and model performance.



Q3): Given data/points

Point	X	y
1	0.4	0.53
2	0.22	0.38
3	0.35	0.32
4	0.26	0.19
5	0.08	0.41
6	0.45	0.3

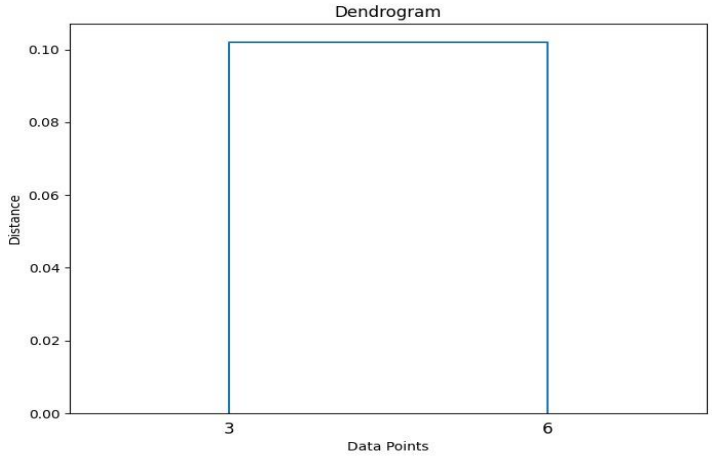
The distance between 2 points (x_1,y_1) and (x_2,y_2) is given by $\sqrt{((x_2-x_1)^2 + (y_2-y_1)^2)}$

Step1:

The distance matrix for the given points is:

Point	1	2	3	4	5	6
1	0	0.2343	0.2159	0.3677	0.3418	0.2354
2	0.2343	0	0.1432	0.1942	0.1432	0.2435
3	0.2159	0.1432	0	0.1581	0.2846	0.102
4	0.3677	0.1942	0.1581	0	0.2843	0.2195
5	0.3418	0.1432	0.2846	0.2843	0	0.386
6	0.2354	0.2435	0.102	0.2195	0.386	0

From the distance matrix it is evident that the smallest distance is between points 3 and 6 which is 0.102. So, we will group points 3 and 6 together into a single point. The dendogram for that will be as follows:

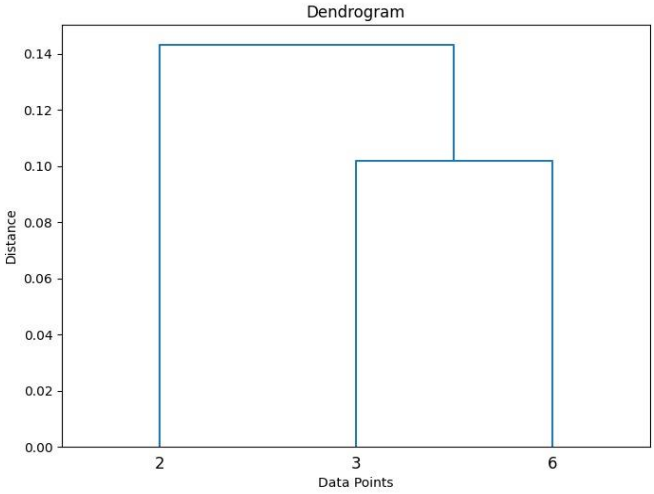


Step2:

The updated distance matrix will be:

Point	1	2	3,6	4	5
1	0	0.2343	0.2159	0.3677	0.3418
2	0.2343	0	0.1432	0.1942	0.1432
3,6	0.2159	0.1432	0	0.1581	0.2846
4	0.3677	0.1942	0.1581	0	0.2843
5	0.3418	0.1432	0.2846	0.2843	0

From the distance matrix it is evident that the smallest distance is between point 2 and group of points (3,6) which is 0.1432. (Points 5 and 2 distance is also same but we consider it later as 3 and 6 were grouped together already). So, we will combine point 2 into the group of (3,6) points into a single point. The dendogram for that will be as follows:

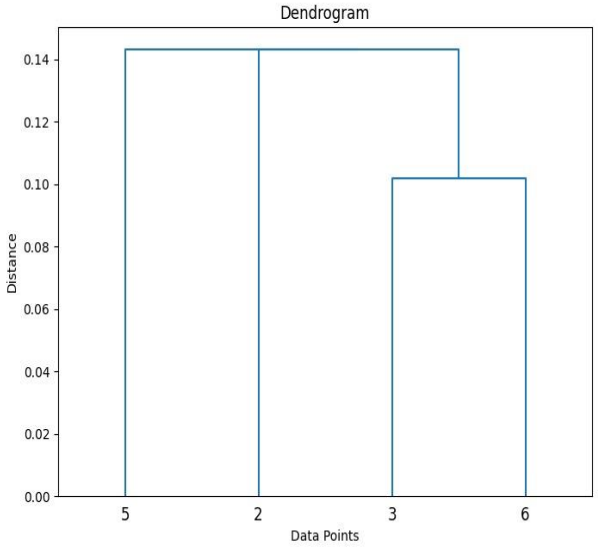


Step3:

The updated distance matrix will be:

Point	1	2,3,6	4	5
1	0	0.2159	0.3677	0.3418
2,3,6	0.2159	0	0.1581	0.1432
4	0.3677	0.1581	0	0.2843
5	0.3418	0.1432	0.2843	0

From the distance matrix it is evident that the smallest distance is between point 5 and group of points (2,3,6) which is 0.1432. So, we will combine point 5 into the group of (2,3,6) points into a single point. The dendogram for that will be as follows:

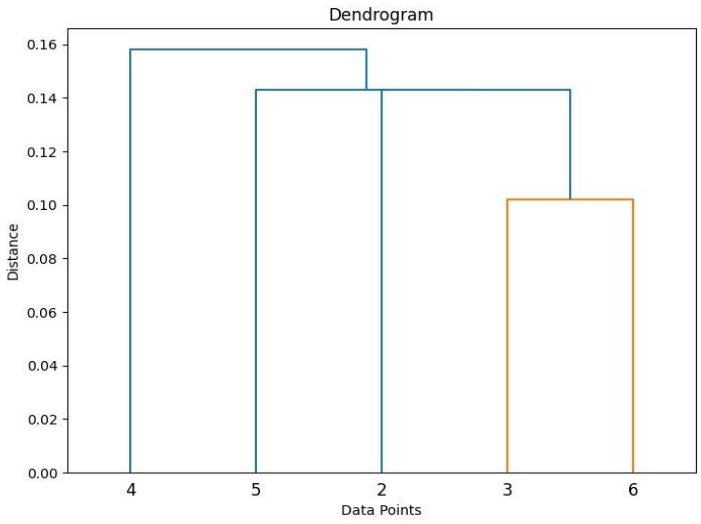


Step4:

The updated distance matrix will be:

Point	1	2,3,5,6	4
1	0	0.2159	0.3677
2,3,5,6	0.2159	0	0.1581
4	0.3677	0.1581	0

From the distance matrix it is evident that the smallest distance is between point 4 and group of points (2,3,4,6) which is 0.1581. So, we will combine point 4 into the group of (2,3,5,6) points into a single point. The dendogram for that will be as follows:



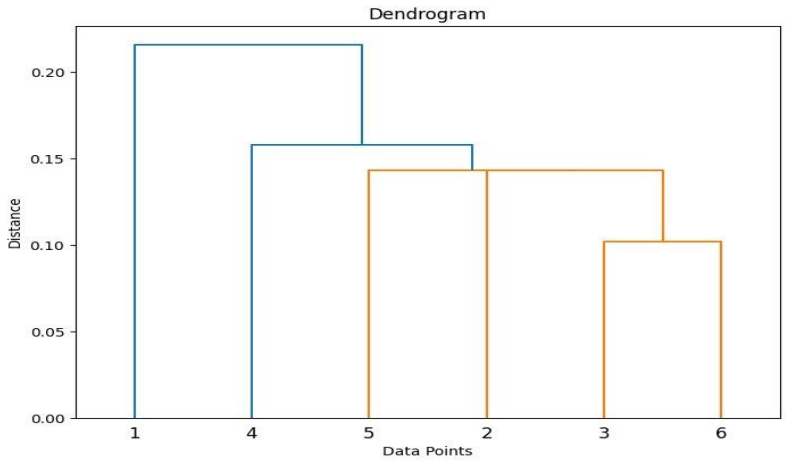
Step5:

The updated distance matrix will be:

Point	1	2,3,4,5,6
1	0	0.2159
2,3,4,5,6	0.2159	0

From the distance matrix the only distance left is between point 1 and the group of points 2,3,4,5,6. So, we will combine point 1 into the group of (2,3,4,5,6) points into a single point.

The final dendogram after adding and including all the points will be as follows:



Complexity of the fastest possible algorithm : $O(n^2)$

Pseudocode

```
class SingleLinkage
{
   vector<int> parent,size
public:
   SingleLinkage (int n)
         size.initialize(n+1,1)
         for(p=0:p<=n;p++)
                  parent[p]=p


   int find_parent(int point)
         if(point==parent[point])
                  return point
         return parent[point]=find_parent[parent[point]]


   void union_by_size(int p1,int p2)
         int ultp_p1=find_parent(p1)
```

```
        int ultp_p2=find_parent(p2)
        if(size[ultp_p1]<size[ultp_p2])
                parent[ultp_p1]=ultp_p2
                size[ultp_p2]+=size[ultp_p1]
        else
                parent[ultp_p2]=ultp_p1
                size[ultp_p1]+=size[ultp_p2]
}

void find_min_distance(points,map)
        map m
        n=points.size();
         for(int i=0;i<n;i++)
                set s<-empty
                distance=INT_MAX
                for(int j=0;j<n;j++)
                        if(i==j) continue;
                        if(euclid_distance(points[i],points[j])<distance)
                                s<-empty
                                distance=euclid_distance(points[i],points[j])
                                s<-(i+1,j+1)
                m<-{s,distance}

main()
{
   points<-data_base
   map m
   find_min_distance(points,m)
   min_heap=heap_sort_based_on_m.second(m)
   SingleLinkage sl(points.size())
   while(!min_heap.empty())
   {
      sl.union_by_size(min_heap.top().first[0],min_heap.top().first[1])
      minheap.pop()
   }
}
```

Complexity Analysis:

1) Variables used:

   Points: all data points
   m:map stores data of nearest points and respective distance ((p1,p2),distance)
   parent: parent of each data point in a cluster
   size: size of the cluster
   ultp_p1: ultimate parent of p1 data point
   ultp_p2: ultimate parent of p2 data point
   sl: object of SingleLinkage class

data_base contains all the data points of D dimension. store that data in points variable, initialize map m to store the minimum distance points with respect to their distance.

find_min_distance:

This function finds the nearest point to each point in points

Initialize empty set and distance=INT_MAX for each point i and evaluate the Euclidean distance to each and every point j (i notequalto j). Update the set and distance whenever euclid_distance(points[i],points[j])<distance),Insert the set (i+1,j+1) and distance into map m

(i) This takes ***O(n^2)*** Time complexity as each point is checked with each and every other point in the data.

Finally map m contain the data of nearest points and distance

heap_sort_based_on_m.second(m):

It generates the minheap based on the distance (I.e. m.second)

(ii) Time complexity ***O(n logn)***

Create an object sl of *SingleLinkage* class, the constructor creates the array of ***parent*** and ***size***. parent variable stores the pointer to the parent. Initializing parent of each point to itself. size variable stores the size of each cluster to which that data point belong, initializing size array to 1 as each point is considered as one cluster initially.

We traverse the minheap until it is empty and take the top element from min_heap and pop it.

Call function *union_by_size* by pair of point numbers .

union_by_size:

ultp_p1 and ***ultp_p2*** stores the ultimate parent of each point. we are pointing the smaller size parent cluster to the larger size parent cluster by comparing the variable ***size*** of parents and then the small sized cluster will be merged with large sized cluster. It decreases the height of the tree . Also, add the size of smaller cluster to the larger cluster to track the size of cluster.

find_parent

It is a recursive function required for path compression between child and ultimate parent .it decreases the path between the each newly added point and root parent by directly pointing it to ultimate parent point, this is done when two clusters are merged.

(iii) Time complexity : ***O(n*4) =O(n)*** which is deduced from inverse Ackermann function[2]

Where n is for each pair in min heap and 4 is for union and path compression which is constant time[1][2]

From (i), (ii) & (iii)

Total time Complexity = O(n^2) + O(n logn) + O(4*n) = O(n^2)

References:

[1]. https://en.wikipedia.org/wiki/Disjoint-set_data_structure#Time_complexity

[2]. https://en.wikipedia.org/wiki/Iterated_logarithm