COL761

MININGMAVERICKS

Assignment 1

Chinmay Mittal *(2020CS10336)*
Parth Shah *(2020CS10380)*
Shubh Goel *(2020EE10672)*

Date of submission of Report: September 25, 2023

1 Frequent Subgraph Mining

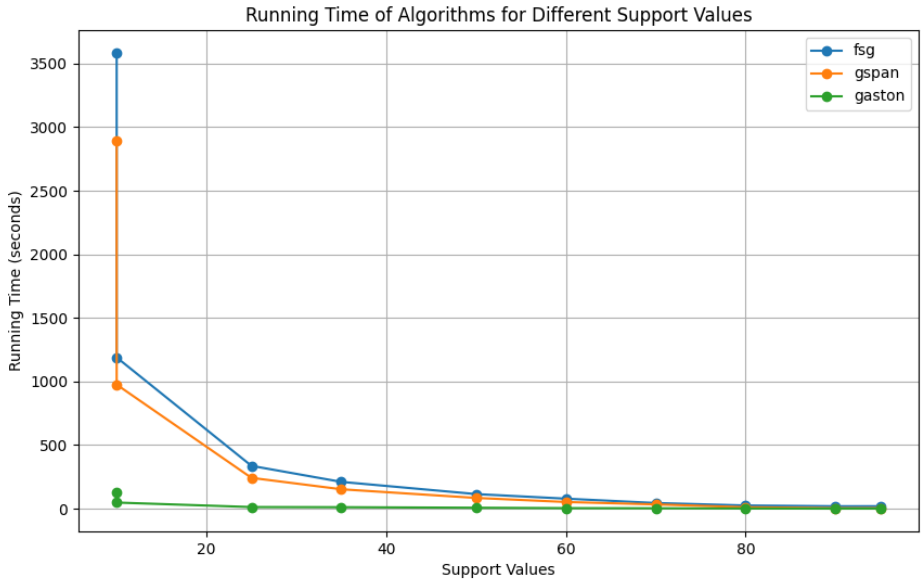


Figure 1: Running Time for Frequent Sub-graph Mining Algorithms

Qualtitaive overview of all the three algorithms:

- **gSpan** relies on a depth-first search approach to mine frequent subgraphs. It uses a canonical representation for graphs (minimum DFS code) and avoids duplicate computations by checking the canonical forms. It effectively computes the frequency of the child, given the frequency of the parent without repeated databases scans and does not generate any false candidates. In addition, gSpan introduces the concept of rightmost extension to guide its DFS exploration, which reduces the search space.

- **FSG** (PAFI) constructs an adjacency matrix based canonical representation for every frequent subgraph. It starts with a single edge and grows the subgraphs by adding edges to the current frequent subgraphs in a breadth-first search manner. The adjacency matrix along with vertex in-variants based ordering ensures that isomorphism checks are efficient. But this algorithm generates a lot of false candidates, which increases running time.

- **Gaston**: Gaston uses a heuristic to quick start the frequent subgraph computation based on statistical properties of common datasets. It utilizes efficient data structures and algorithms for support counting. Gaston's approach to counting support is one of its primary advantages over the other methods. Gaston also prunes the search space effectively and performs incremental candidate generation.

The running time of these algorithms is **exponential** in nature with respect to support values.

The task of frequent subgraph mining is inherently exponential in nature due to the vast number of potential frequent subgraphs that could exist in a given dataset for smaller support values. As the size of support decreases, the number of potential subgraphs grows exponentially.

As you lower the minSup threshold, more subgraphs qualify as "frequent", thus increasing the search space and the number of isomorphism checks the algorithm has to perform. At

extremely high minSup values (e.g., 95%), the number of qualifying frequent subgraphs decreases dramatically, hence running times might decrease.

Explaining the observed **order of performance**:

Gaston is the fastest: Its efficient support counting mechanism and effective pruning strategy make it quicker. Gaston uses a level-wise method to locate all common subgraphs, taking into account first basic routes, then more complicated trees, and lastly the most complicated cyclic graphs. The most common graphs in practice don't seem to be particularly complex structures; Gaston makes good use of this quickstart observation to arrange the search space. Gaston uses an occurrence list-based approach to calculate the frequency of graphs, where all occurrences of a small collection of graphs are recorded in main memory.

gSpan depth-first search and canonical form-based approach helps in reducing the number of isomorphic checks. The rightmost extension principle ensures that it avoids many unnecessary subgraph candidates. However, the method of generating subgraphs and counting support is not as efficient as Gaston's approach which uses heuristics to prune the search space effectively and also maintains efficient data structures.

FSG: generates a larger number of subgraph candidates compared to gSpan and Gaston, which can slow it down, especially for lower support thresholds. Also it has to perform expensive

In conclusion, the efficiency of these algorithms comes down to their strategies for candidate subgraph generation, support counting, and pruning of the search space. Gaston's efficient approach to these aspects makes it faster than gSpan and FSG in many scenarios. However, it's worth noting that the performance can also be influenced by dataset-specific characteristics, so these trends might vary depending on the nature and structure of the input data. For most datasets that are tackled in the real-world like the one provided the running time follows the order

$$\text{Gaston} < \text{gSpan} < \text{FSG}$$

2 K-Means Clustering

One of the challenges in applying k-Means Clustering is determining the appropriate value for the parameter 'k,' which represents the number of clusters that should be formed when analyzing a given dataset. To address this challenge, data analysts often turn to a visualization technique known as the "Elbow Plot."

The Elbow Plot is a graphical representation that assists in choosing the optimal value of 'k' for k-Means Clustering. It is constructed by plotting the average distance of each data point from its assigned cluster center("Inertia") against the number of clusters.

To mitigate the sensitivity of the k-Means Clustering algorithm to the initial choice of cluster centers, we ran the algorithm multiple times(40) on the same dataset, each time with different initializations of cluster centers. By doing so, it's possible to obtain a more stable and robust clustering result. We then calculated the average of "inertia" across all runs. This averaged inertia value was then used to create a smoother and more representative Elbow Plot.

We got the best plot for the dataset with **dimension 4** and found out the optimal value of k to be **6**.

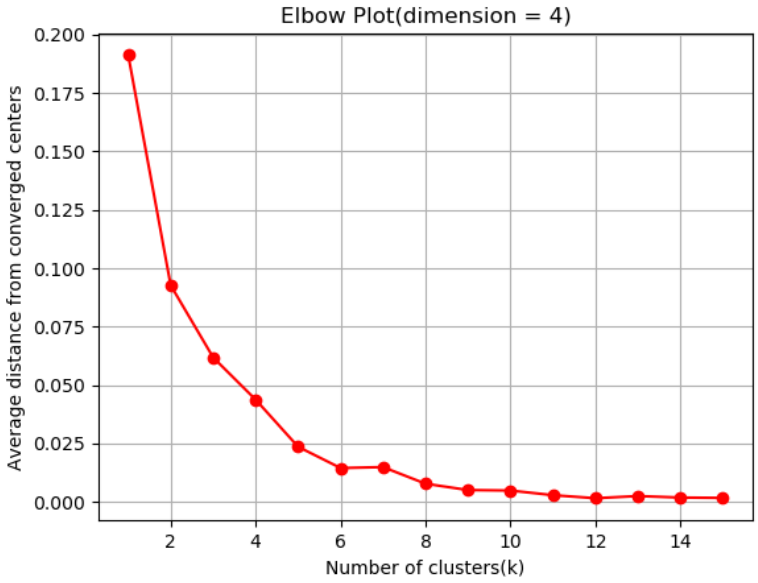


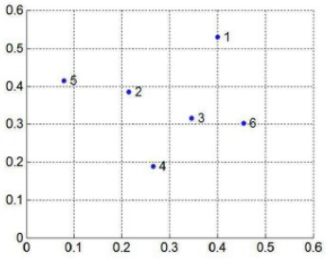
Figure 2: Elbow point at k = 6

3 Single Linkage Clustering

let **D** = *set of data points*
let **C** = *set of initial clusters, initialized to n clusters with each cluster containing one data point.*
Single linkage clustering is performed in the following way:

1. Find two clusters, $\mathcal{X}, \mathcal{Y} \in \mathbf{C}$ (i.e. two clusters which are part of the current cluster set) such that,
$$\mathcal{F}(\mathcal{X}, \mathcal{Y}) = \min_{\mathcal{P}, \mathcal{Q} \in \mathbf{C}} \mathcal{F}(\mathcal{P}, \mathcal{Q})$$
where,
$$\mathcal{F}(\mathcal{X}, \mathcal{Y}) = \min_{u \in \mathcal{X}, v \in \mathcal{Y}} \|u, v\|$$
here $\|u, v\|$ = metric distance between u and v
2. $\mathcal{Z} = Merge(\mathcal{X}, \mathcal{Y})$
3. Delete $\mathcal{X}$,$\mathcal{Y}$ from **C** and insert $\mathcal{Z}$ into **C**
4. Repeat steps $1 - 3$ until only one cluster remains i.e. $||C|| = 1$

Dendogram



Point	x	y
1	0.40	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	0.30

Figure 3: Data Points

	1	2	3	4	5	6
1	0	0.234307	0.21587	0.367696	0.34176	0.235372
2	0.234307	0	0.143178	0.194165	0.143178	0.243516
3	0.21587	0.143178	0	0.158114	0.284605	0.10198
4	0.367696	0.194165	0.158114	0	0.284253	0.219545
5	0.34176	0.143178	0.284605	0.284253	0	0.386005
6	0.235372	0.243516	0.10198	0.219545	0.386005	0

Figure 4: Distance Matrix

1. After 1st iteration:

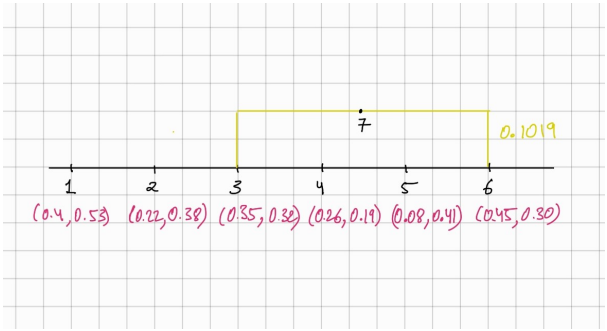


Figure 5: $\mathcal{F}(\{3\},\{6\}) = \|3,6\| = 0.1019$

2. After 2nd iteration:

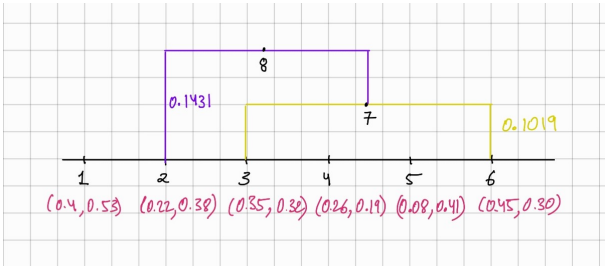


Figure 6: $\mathcal{F}(\{3,6\},\{2\}) = \|2,3\| = 0.1431$

3. After 3rd iteration:

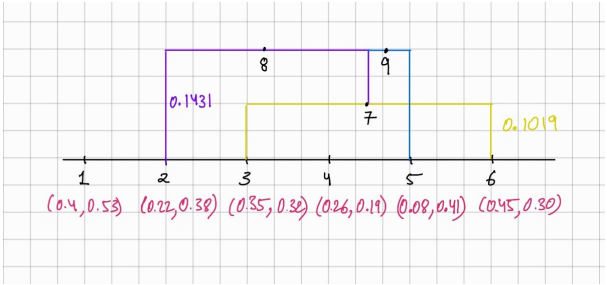


Figure 7: $\mathcal{F}(\{2,3,6\},\{5\}) = \|2,5\| = 0.1431$

4. After 4th iteration:

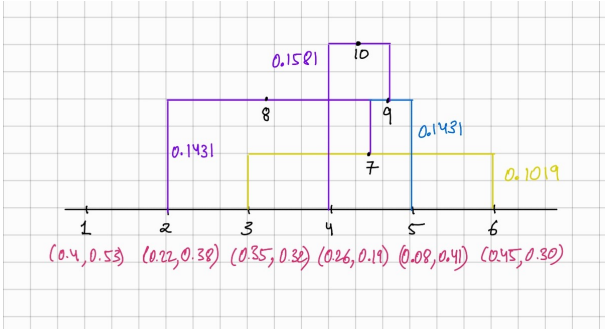


Figure 8: $\mathcal{F}(\{2,3,5,6\},\{4\}) = \|3,4\| = 0.1581$

5. After 5th iteration:

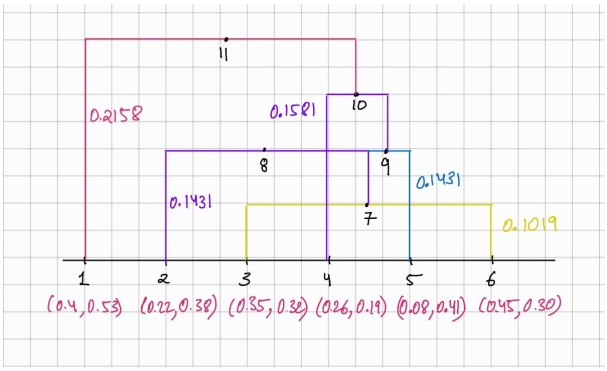


Figure 9: $\mathcal{F}(\{2,3,4,5,6\},\{1\}) = \|1,3\| = 0.2158$

Theorem

If the $Merge(\mathcal{X},\mathcal{Y})$ operation is defined as the addition of an edge between $a(\in \mathcal{X})$ and $b(\in \mathcal{Y})$ having edge weight $\|a,b\|$ such that,

$$\|a,b\| = \min_{u\in\mathcal{X},v\in\mathcal{Y}} \|u,v\|$$

then, the single cluster remaining at the end of the clustering represents the *Minimum Spanning Tree* of the graph $\mathcal{G}(V,E,W)$ with,

$$V = \mathcal{D}$$
$$E = VXV$$
$$W(u,v) = \|u,v\|, \forall (u,v) \in E$$

Proof

Let $\mathcal{T}$ denote the single cluster remaining at the end.
Lemma $\mathcal{T}$ is a tree.

1. At each iteration, an edge is added between points belonging to different clusters. Hence, $\mathcal{T}$ does not contain any cycle.
2. $\forall u,v \in \mathcal{T}$, there exists a unique path between them. Since, there are no cycles in $\mathcal{T}$, there exists at most one path between u and v. Suppose, there is no path between them. This would imply that u and v belong to two different clusters, *contradiction!*

From 1 and 2 we conclude that $\mathcal{T}$ is a tree.

Let $wt(\mathcal{T})$ = weight of $\mathcal{T}$.
Suppose, $\mathcal{T}$ is not an MST. This means there exists an edge, $e \in \mathcal{T}$ and $e' \in E - \mathcal{T}$ such that,

$$wt(\mathcal{T} + e' - e) < wt(\mathcal{T}) \tag{1}$$

Since, $\mathcal{T} + e' - e$ is a tree, e must be a cut edge corresponding to e'. let $e' = (u',v')$ and $e = (u,v)$. let $\mathcal{X}$ and $\mathcal{Y}$ represent connected components of $\mathcal{T}$ such that $u,u' \in \mathcal{X}$ and $v,v' \in \mathcal{Y}$. From (1), we get,

$$\|u',v'\| < \|u,v\| \tag{2}$$

That means,

$$\|u,v\| \neq \min_{a\in\mathcal{X},b\in\mathcal{Y}} \|a,b\| \tag{3}$$

Hence, the edge (u,v) should not have been added while merging $\mathcal{X}$ and $\mathcal{Y}$, which is a *contradiction!*

Algorithm Pseudocode

Algorithm 1 Single Linkage Clustering

1: **Input:** $\mathbf{D}(|\mathcal{D}| = n)$
2: $V \leftarrow \mathbf{D}$
3: $E \leftarrow V \times V$
4: $W \leftarrow \{\|u, v\| | \forall (u, v) \in E\}$
5: $\mathcal{T} \leftarrow prims\_algorithm(V, E, W)$
6: *sort edges in* $\mathcal{T}$ *in the increasing order of edge weight*
7:
8: $cluster\_id \leftarrow \{\}$
9: $cluster\_size \leftarrow \{\}$
10: $cluster\_height \leftarrow \{\}$
11: $point\_id \leftarrow \{\}$
12:
13: $Inititialize : cluster\_id, cluster\_size, cluster\_height, point\_id$
14:
15: $new\_cluster\_id \leftarrow n + 1$
16: $new\_clusters \leftarrow \{\}$
17:
18: **for** $(u, v) \in \mathcal{T}$ **do** ▷ Hierarchical clustering begins
19: $cluster\_height\,[new\_cluster\_id] \leftarrow weight(u, v)$
20: $new\_clusters\,[new\_cluster\_id]\,.left \leftarrow cluster\_id\,[\textbf{\textit{Find(u)}}]$
21: $new\_clusters\,[new\_cluster\_id]\,.right \leftarrow cluster\_id\,[\textbf{\textit{Find(v)}}]$
22: ***Join(u,v)***
23: $cluster\_id\,[\textbf{\textit{Find(u)}}] \leftarrow new\_cluster\_id$
24: $new\_cluster\_id \leftarrow new\_cluster\_id + 1$
25: **end for**

Algorithm 2 Find(u)

1: **while** $u \neq point\_id\,[u]$ **do**
2: $u = point\_id\,[u]$
3: **end while**
4: return u

Algorithm 3 Join(u,v)

1: $u \leftarrow$ ***Find(u)***
2: $v \leftarrow$ ***Find(v)***
3: **if** $cluster\_size\,[u] < cluster\_size\,[v]$ **then**
4: *Swap(u,v*
5: **end if**
6: $cluster\_size\,[u] \leftarrow cluster\_size\,[u] + cluster\_size\,[v]$
7: $point\_id\,[v] \leftarrow u$

Time Complexity Analysis(Algorithm 1)

1. Line 5 is the application of well known *Prim's Algorithm*. It takes $O(|V^2|)$ time to give a MST of graph, $\mathcal{G}(V, E, W)$. Hence, this line takes $O(n^2)$ time.

2. Line 6 takes $O(n \log(n))$ time.

3. Line 13 takes $O(n)$

4. Lines 18-25 is a modification of *Kruskal's Algorithm* which takes $O(E \log(E))$ time where E represents the number edges($n - 1$ in this case). The functions ***Find(u)*** and ***Join(u,v)*** take $O(\log(n))$ time. Hence lines 18-25 take $O(n \log(n))$ time. *Kruksal's Algorithm* can be implemented by the Disjoint-Set Union data structure efficiently using the path-compression and union by rank optimizations.

Overall, the algorithm takes $O(n^2)$ time. Note that this is the best time complexity possible for any single clustering algorithm, since for any dataset any algorithm will at least have to compute the distance between all pair of points which will take $O(n^2)$ time.