Assignment 2 report

Shreyans Nagori 2018CS10390
Rajat Singh 2020CSZ8507
Chhavi Agarwal 2020CSY7654

October 2021

1 Comparison of FSG, Gaston and GSpan for frequent subgraph mining :

Firstly lets see how each of the algorithm solves the frequent subgraph mining problem:

1. FSG: FSG is the oldest algorithm as compared with gSpan and Gaston. It uses the sparse graph representation to minimise processing and storage cost. FSG is similar to Apriori algorithm, from a set of given frequent k-edge subgraphs it generates a list of candidate (k+1) edge subgraphs by combining each k edge subgraph and generating all possible subgraphs. It uses heuristics like canonical labelling to avoid generating subgraphs already generated.

2. GSpan: GSpan intends to avoid the issues related to the apriori approach used by FSG, i.e. the candidate generation cost and the false positive detection during the isomorphism testing. GSpan adopts a DFS based approach to solving the frequent subgraph mining problem. It sorts edges in lexicographic order and then at each level adds the smallest lexicographical edge not added to the current subgraph and checks if the generated DFS code is the minimal DFS code of some frequent subgraph. If not it backtracks accordingly, and explores each subgraph once only.

3. Gaston: Gaston is based on gSpan. . It uses a hash table representation which justifies its performance over the other algorithms. It follows a level-wise approach in which first simple paths are considered, then more complex trees and finally the most complex cyclic graphs. It appears that in practice most frequent graphs are not actually very complex structures; Gaston uses this quick start observation to organize the search space efficiently. To determine the frequency of graphs, Gaston employs an occurrence list based approach in which all occurrences of a small set of graphs are stored in main memory.

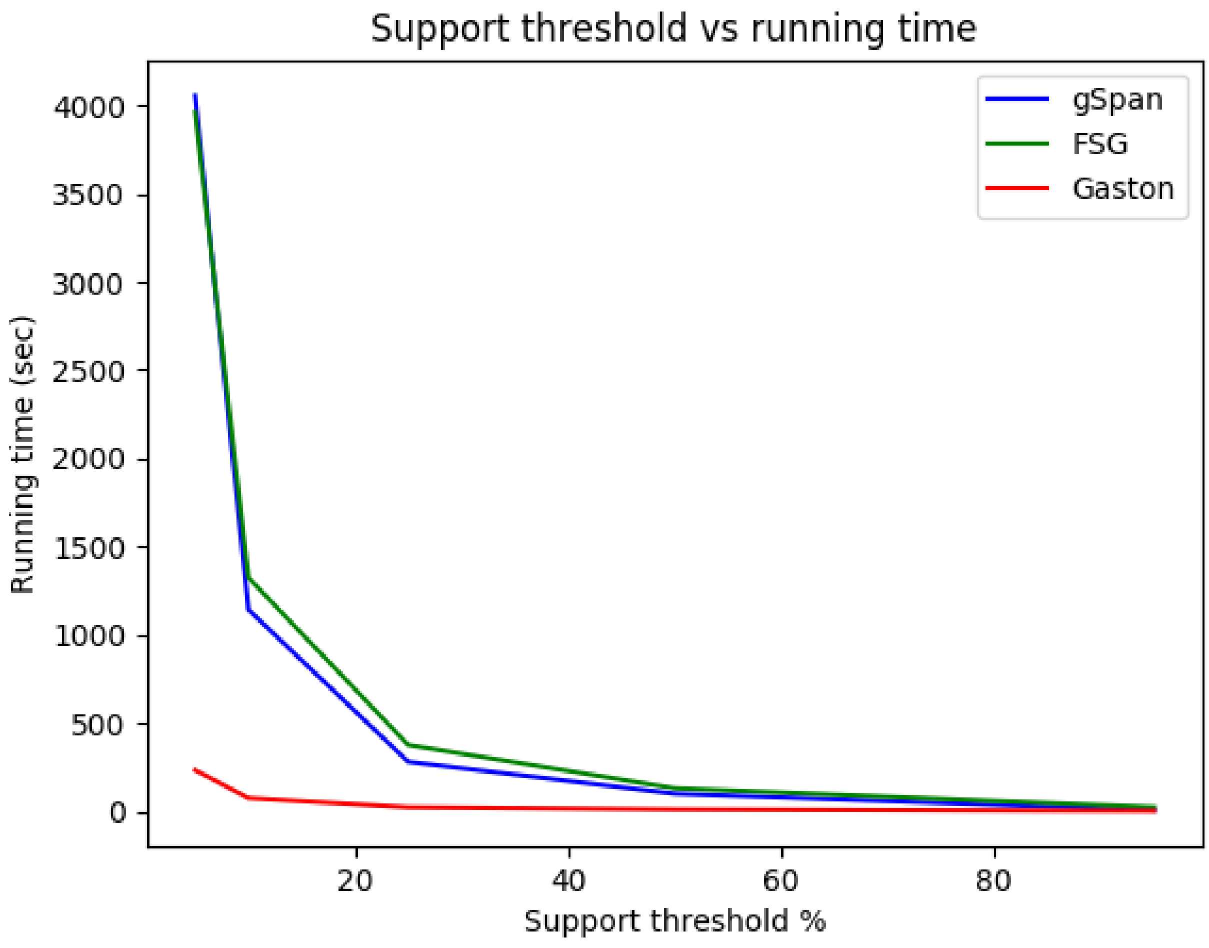

Figure 1: Plot of time vs Support threshold

In Part 1 of the assignment we ran gSpan, FSG, and Gaston algorithm for different support thresholds and checked the time taken by each algorithm to find the number of frequent subgraphs. Figure 1 Shows the Running time vs Support threshold plot for gSpan, FSG, and Gaston algorithm. From the plots we can see the following observations :

1. For all the three algorithms (gSpan, FSG, and Gaston) time taken to find the number of frequent subgraphs is inversely proportional to the support threshold. Thus as the support threshold is decreasing, the time taken by each of the algorithm is increasing with respective rate.
2. From Figure 1 we can observe that for gSpan and FSG time is increasing exponentially as the support threshold decreasing, but on the other hand for Gaston it is increasing linearly.
3. From Figure 1 we can observe that Gaston is the fastest algorithm as compared with gSpan and FSG.

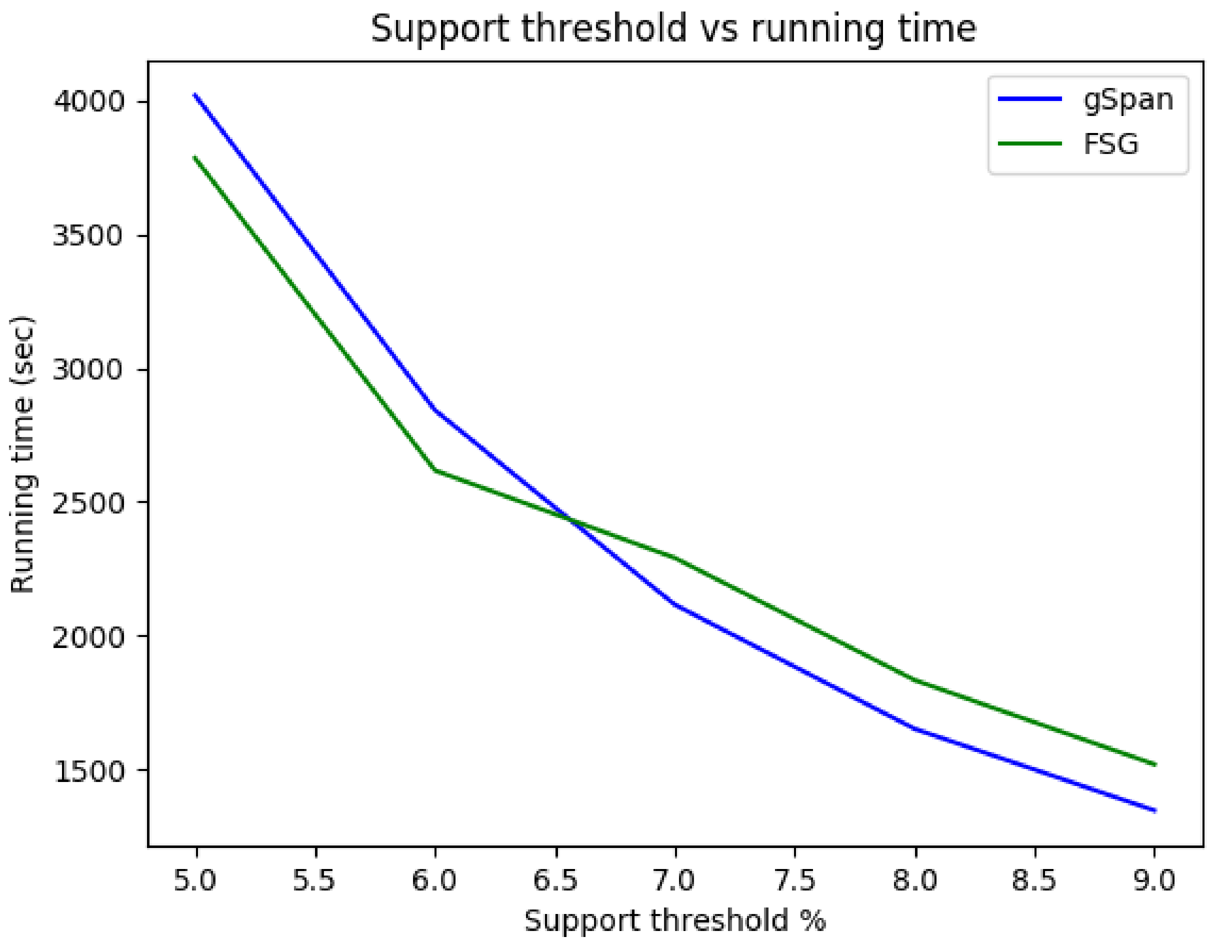

Figure 2: Plot of time vs Support threshold

4. From figure 1 and Figure 2 we can observe that gSpan is faster than FSG after a certain support threshold. For very small support threshold, FSG is faster than gSpan.
5. In Figure 3 we can see that at higher support threshold Gaston is better than gSpan and gSpan is better than FSG.

Below are some reasons of the above observations :

1. For lower support threshold, the number of frequent subgraphs will be very large (grows exponentially) and vice versa for the higher support threshed.
2. As we know that FSG use BFS strategy which includes storing intermediate candidates generation and subgraphs that are being discovered during the entire process and pruning the infrequent subgraphs. Since all these are expensive process which makes FSG as the slowest algorithm.

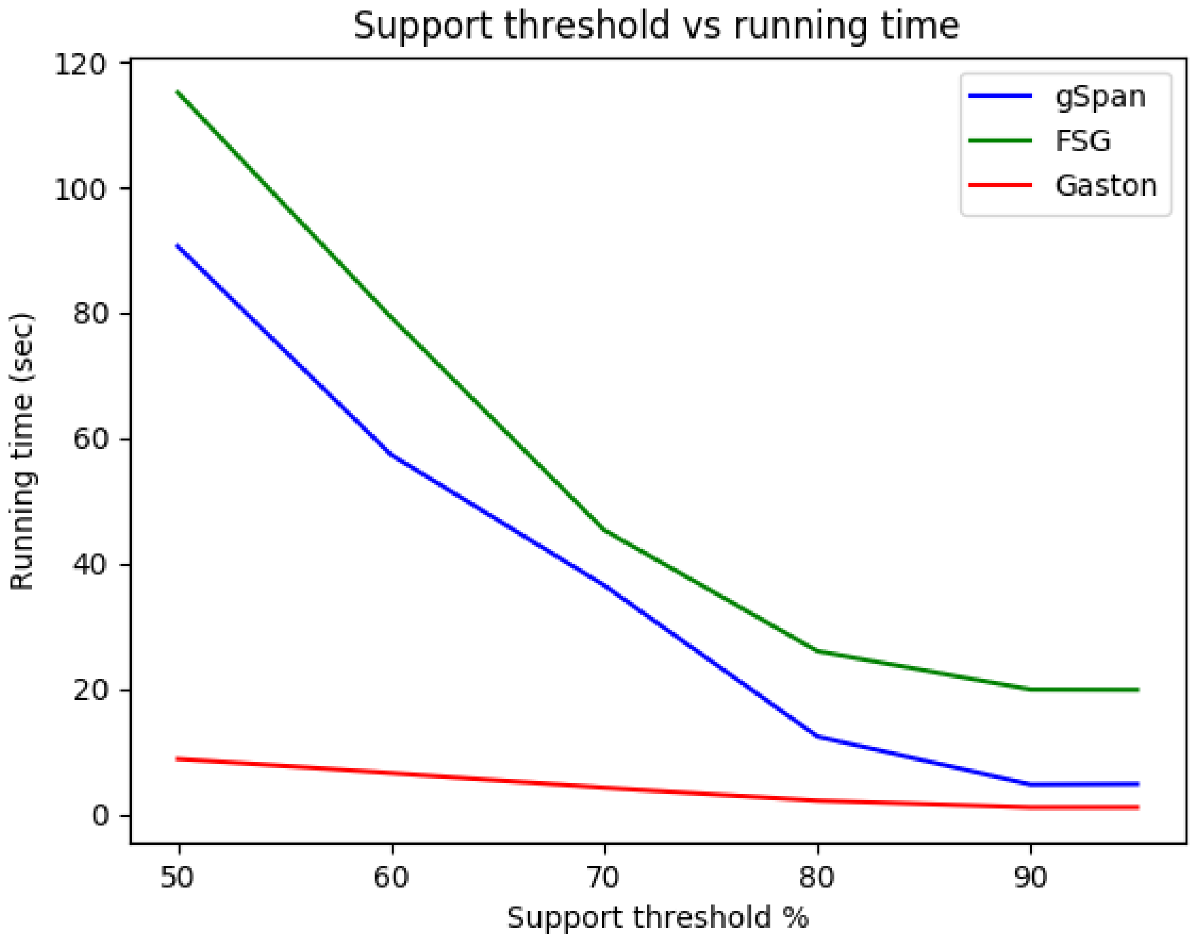

Figure 3: Plot of time vs Support threshold

3. As we know that gSpan use DFS approach and thus it does not generate candidate graph set. Since gSpan does not generate the candidate graph set, it avoids the cost intensive problem like redundant candidate generation and Isomorphism testing. Thus this makes this algorithm better than FSG.

4. For lower support threshold FSG is better than gSpan because at lower support threshold the number of frequent subgraphs are very high and random thus other costs dominate over the main algorithm.

2 Comparison of Brute Force Vs KD Tree Vs M Tree For KNN:

Lets consider how these indexing structures index our data:

1. KD Tree: The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis.

2. M-Tree: As in any Tree-based data structure, the M-Tree is composed of Nodes and Leaves. In each node there is a data object that identifies it uniquely and a pointer to a sub-tree where its children reside. Every leaf has several data objects. For each node there is a radius r that defines a Ball in the desired metric space. Thus, every node n and leaf l residing in a particular node N is at most distance r from N, and every node n and leaf l with node parent N keep the distance from it.

Pseudocode to find kNN in KD Tree:

Lets first see how KNN is performed on KD Trees:
1. Algorithm first searches for point as it does in Knn search.
2. Then it recurses from deepest node found to root, searching for best node on the path from found node to root, and updating accordingly.
3. It then creates a hypersphere of radius of current best distance around query point and searches in the other half of KD tree which we haven't traversed yet.
4. Eliminate those branches which are guaranteed to fall outside the sphere, and return best answer.

In k nearest neighbours we maintain k current bests instead of 1, and accordingly prune branches if and only iff the branch is guaranteed to have all points outside the k nearest neighbours of the query point.

Pseudocode to find kNN in M Tree:

PR is a priority queue of pointers to active sub-trees, i.e. subtrees where it is possible to get candidate k nearest neighbours, and NN maintains the result of k best neighbours found till now. A lower bound, $d(T(O_r))$, on the distance of any object in $T(O_r)$ with value $= \max(0, d(O_r,q) - r(O_r))$. Search radius is distance between k nearest neighbour and query point:
1. Choose the node with the minimum value of d from the priority queue.
2. For each subtree by triangle inequality we know an upper bound on the distance of the query point from any point in that subtree.
3. On an internal node, it first determines active sub-trees and inserts them into the Priority queue.

4.Then, if needed, it calls the N-Update function to perform an ordered insertion in the N array and receives back a (possibly new) value of covering radius.
5.This is then used to remove from PR all sub-trees for which the d$_{min}$ lower bound exceeds covering radius.

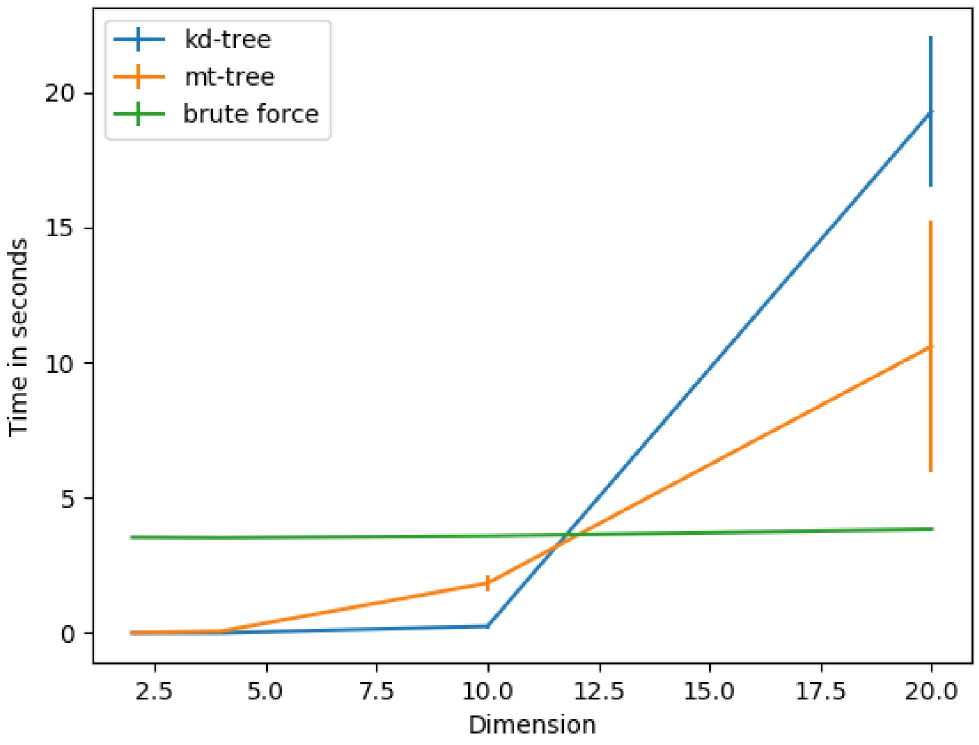

Figure 4: Plot of time vs dimension

In figure we can see how for lower dimensions sicpy's implementation of KD-Tree and our library's M Tree perform better than brute force, but as dimensionality increases the time taken by brute force stays nearly constant and the time taken by KD trees and M Trees grows due to curse of dimensionality. The speed of growth for KD Tree is in particular more than that of M-Tree, because unlike M Tree depth of KD-Tree increases linearly with that of dimension.