DataMining
Homework4 Report

Sahiti,CS13B043

Q1

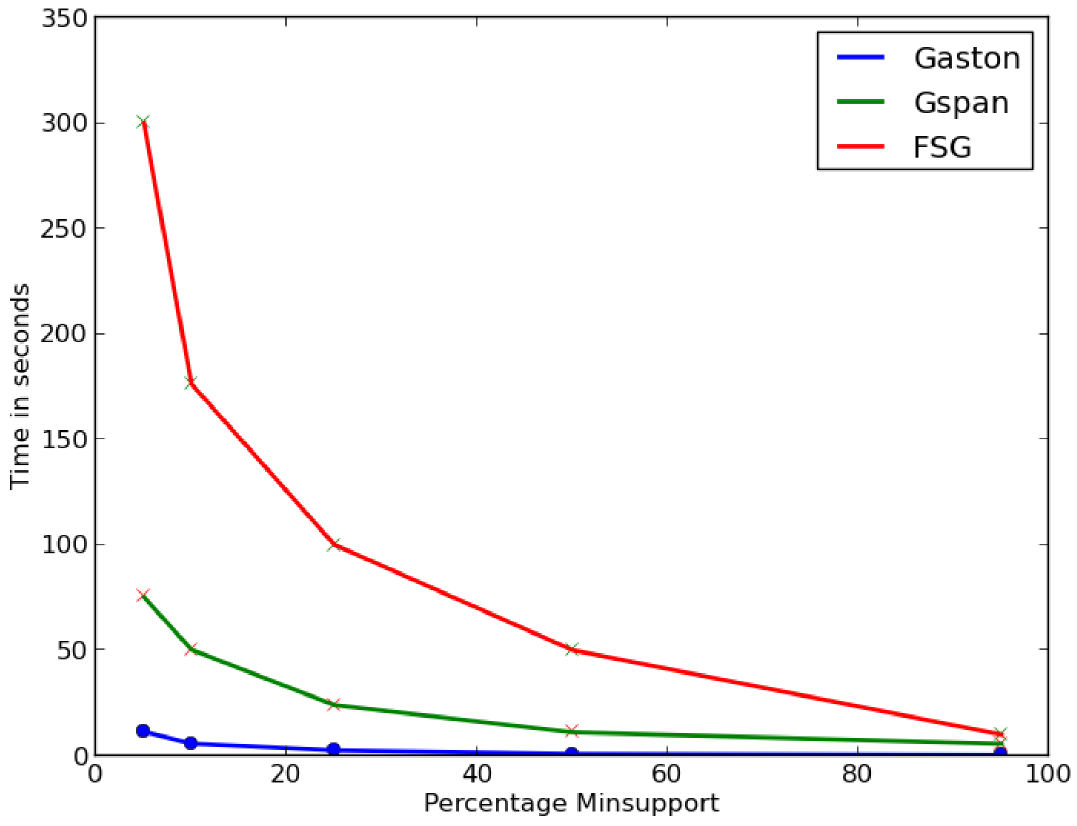

Figure 1: Plot for Runtime against Minsup for FSG,Gspan,Gaston

FSG

The computation time of FSG increasesagments exponentially. FSG computes the candidates with k edges at every iteration and then makes k+1 edge candidates.The noof candidates increase a lot at every iteration.At every iteration the noof candidates generated multiplies.So,if the support is decreased ,the initial noof candidates generated will be more and this increase get multiplied at every iteration,so the overall increase is in exponential.Simply to say,for a n-edge candidate to be selected as frequent,in fsg all its subgraphs are first checked ,and the noof subgraphs are proportional to $2^n - 1$.So when minsup increase the candidates increase exponentially ,so the time inceases exponentially

GSpan

GSpan uses pattern growth approach.It increases the DFSCode incrementally for getting next candidates.It generates the next candidates using 2 pruning ways:1.dfscode can be extended at nodes that lie on the rightmost path of the dfs-tree,2:Only frequent patterns are considered for extension.Additional to this it also prunes away the paths that are already covered (this is checked seeing that if dfs-code obtained is minimal or not.)In this way GSpan is saving lot of time by not generating all candidates.That is why the candidates don't grow exponentially with min support.

Gaston

Gaston does optimization for fast isomorphism testing.It save time by not doing all subgraph isomorphisms.This makes it faster than gspan and fsg .The main insight is that there are efficient ways for to enumarate the paths and trees. By considering the candidates that are paths or trees first, and by only proceeding to general graphs with cycles at the end, a large fraction of the work can be done efficiently.By doing this only in the end it has to do the NP-completeness algorithm of subgraph isomorphism.Gaston defines a global order on cycle-closing edges and only generates those cycles that are "larger" than the last one.Therefore,Gaston takes very less time among all.

2. Graph Classification: Active v Inactive compounds

Here we are given a training set which has active and inactive compounds, we train our classifier on it, and predict the class labels for the given test dataset.

Model :

First of all, we have to deal with the class imbalance in the given dataset.This is done by getting the frequent subgraphs from each classes, using fsg-pafi. Another level of pruning is then done based on the confidences of the features in the two classes, retaining only those which have considerable difference. We then use SVM classifier to learn the model. To run the SVM classifier, we first convert the graphs into vectors of dimension = sum of number of frequent confident subgraphs obtained from the active and inactive classes. Each value is boolean (0 or 1). We then let SVM learn the model, convert the test graphs also into the required format, and predict its labels with the model build.

3. GSpan Minimum DFS

We are given a connected graph , and we have to find its canonical string representation (minimal dfs code)

Algorithm :

We anyway have to consider all possible dfs paths of the given graph to find its minimal one. Only certain heuristics can be used to reduce our search space and make our search faster.

The strategy adopted a breadth first strategy among all possible/ potential DFS codes seen so far, to eliminate most of them at the earliest. Multiple recursive functions are used to optimise thie :
We first start with a single edge at each node which can be the root of a minimal dfs code,ie, at each node which has the least label. Then we iteratively extend the DFS paths starting from these different nodes until we reach an unexplored vertex. We save the list of potential minimal dfs codes seen so far in ArrayList of ArrayList of ArrayList of DFSstruct, named tripledfs. This ensures that the computations done upto stage i is not repeated. Note that there could be multiple potential minimal DFS codes starting from a given potential root node and hence this three-level list is unavoidable. Then at each iteration, we eliminate the non-minimal dfs codes by comparing the codes obtained so far with each other, both at same root/ parent level and different root levels.

When we try to extend a given partly developed potential DFS code, we know the final vertex from where we have to continue the code, and also all the vertices that are covered so far and all the edges that are covered so far in this particular DFS code. To add the next minimal edge, we first see if this node has any uncovered back edges, in which case, the edge to the earliest discovered node is added to the DFS code. Otherwise (if no back edges are present), we have to add a new hitherto unexplored vertex to the dfs code. Clearly, the number we assign to this new vertex is going to be same for all potential unexplored vertices. Hence only the neighbours of this vertex that has got the minimal edge weight from this vertex need to be considered, and all such potential dfs codes are returned back, for comparing and eliminating them at the calling function. For ease of all these comparisons, we have a priority queue for each vertex, where all its incident edges are ordered by the above mentioned rules, and the next best edge is simply the initial equivalent edges in the priority queue of the last vertex.

Even when we can't break ties among all the same partially developed dfs codes at some instant of time, continuing a particular DFS code from where we left is not an issue because the dfs codes have the vertex ids for identifying the actual path corresponding to the seemingly same dfs codes. We keep passing only the list of dfs codes corresponding to each root/node alone when we call it for extension, and get the one step extended dfs code and compare them among all. This comparison is done at 2 stages : for a given root itself, only its minimal dfs codes are returned, and finally the minimal dfs codes from all potential roots are compared in each iteration. The second comparison helps in removing some nodes from our list of potential roots itself, even if we couldn't break ties within the DFS codes of that node/root, thereby pruning a large chunk of potential dfs codes itself.