Homework 4

CS6720 : Data Mining

CS15M015 : Hasit Bhatt

Indian Institute Of Technology, Madras

gSpan, FSG and Gaston comparision

For the trend analysis for different support values on different algorithms, the given dataset for AIDS was converted into 2 different formats for gaston,FSG(PAFI) and gSpan.

On the 42682 graphs, the running times observed for thresholds of 5%, 10%, 25%, 50% and 95% were found as below. The running times were observed on 2.2 GHz Pentium 4, 6 GB, 64-Bit Ubuntu 15.04.

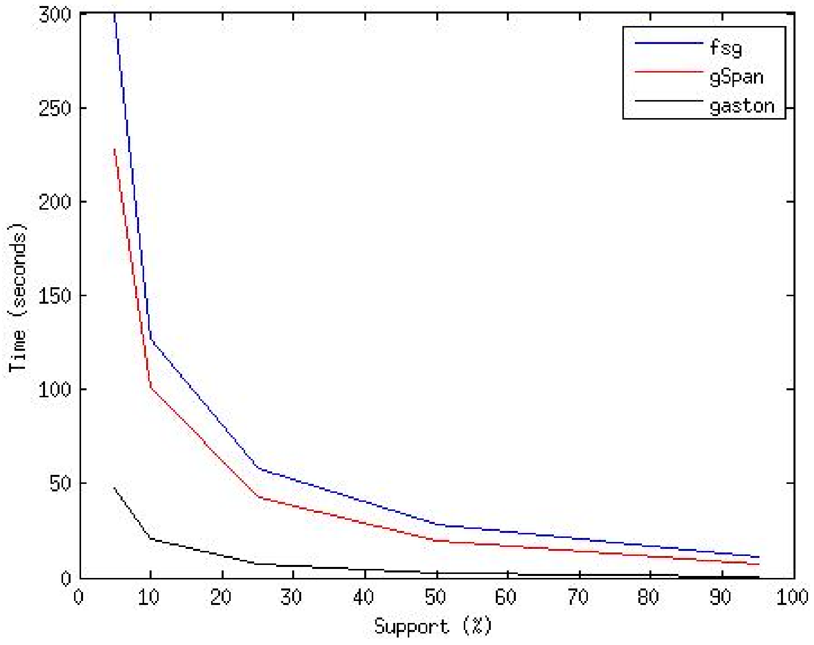

Fig. 1. Running Times for gSpan, FSG and gaston

Observations As can be observed from the plot, the gaston outperforms both algorithms. gSpan performs better than FSG (PAFI) but performs worse than gaston.

To understand the reason behind the behaviour, when we go through the algorithms, FSG is an Apriori based approach unlike Gaston and gSpan. So in informal terminology, it does BFS traversal and tries all possible frequent patterns and finds support for those. This candidate generation is a huge overhead as seen from the plot.

Now, gSpan and GASTON both are pattern-growth techniques (uses DFS order) so they perform better than FSG. Whereas gSpan uses concept of minimum lexicographic order DFS code to prune isomorphic subgraphs if its DFS ordering is not minimum, GASTON uses embedding lists containing the parent-child relations. It is similar to a hashlist(hashtable) for tree representations.

GASTON considers subgraphs in following different types.

- Paths
- Rooted Trees
- Free Trees
- Cyclic Graphs

Using the parent-child relationship, the enumeration for subgraphs of type paths or trees becomes extremely fast, and using this as a quick start, then it is extended for cyclic graphs by adding sets of found frequent subgraphs. Therefore, GASTON performs much faster than gSpan.

Other than the comparision on running time, as we decrease the threshold, the number and size of frequent subgraphs (in terms of nodes and edges) increases. And with decrease in support threshold, these subgraphs increases exponentially and therefore, the time also increases(irrespective of the nature of algorithm). Therefore trend of exponentially decrease can be observed from the graph.

Classification [1]

The Code can be found under Q2 folder, and can be run using
sh classify.sh <trainset filename containing graphs ><active graph IDs filename ><inactive graph IDs filename ><testset filename containing graphs >

[1] The following two problems were done in team with CS15S020: Neha Dubey

Finding Frequent Subgraphs

For the classification of graphs, we have used FSG binary (PAFI) to find minimum subgraphs for given threshold. gSpan is faster but FSG has maximal subgraphs options available therefore it was used for ease. From the dataset, we firstly partition data into two classes as given in input file, active and inactive graphs.

Feature/Subgraph Selection

Using different thresholds, we find different frequent subgraphs for both active and inactive graphs separatly. Now frequent subgraphs from inactive dataset which are neither superset or subset of frequent subgraphs of active datasets are found and stored as features. The similar process is followed for finding maximal subgraphs on both datasets. But now, frequent maximal subgraphs from active datasets which are not related to maximal subgraphs mined from inactive datasets are chosen. These subgraphs are also added as feature with the features we stored previously.

The threshold value is set to 25% empirically.

Feature Population

Using subgraph isomorphism we check whether these features(selected frequent subgraphs) are present in training data using binary. The same is done in the training data.

Classification

Once for each graph vector indicating whether given subgraphs are present or not is calculated, we have binary features and a label to predict, a textbook classification problem. We use Voting Machine (min-rule) with estimating classifiers as BaggingClassifier with DecisionTrees, RandomForest, ExtraTreeClassifier, Weighted Linear Regression for the prediction whether a graph is active or not.

The tests were done using 5-fold cross validation on training before finding frequent subgraphs.

Minimum DFS Code

The Code can be found under Q3 folder, and can be run using
sh generateFromDFS.sh

Algorithm

The algorithm used for finding the Minimum DFS code is very similar to the branch and prune algorithm. The code was written in C++ following C++11 standard. In a generic manner, the algorithm can be explained as followed.

```
Generate edgeList and adjacencyList from graph
Sort edgeList
$E_{min}$ = set of edges with minimum lexicographic representation
while $E_{min}$ not empty do
    Pop the minimum edge e from $E_{min}$
    Explore DFS Codes starting with $<0,1,e_s,e_l,e_d>$
end
```

Algorithm 1: Starting Point

In the algorithm e_s,e_l,e_d represents the source, label of edge and destination respectively.

edgeList Population From the given graph, an adjacency list, and the list of edges were found. The edges being undirected, each edge in graph was stored in edgelist twice (both directions).

Sorting the edges Now these edges were sorted based on their source node, edge label and destination node. The edges are sorted both in edgeList(global) and adjacency list for different nodes.

Exploring the DFS Codes A DFS Code is a collection of edges written in lexicographic order mentioned in gSpan paper. For each edge with minimum lexicographic representation, the DFS Codes are explored. This is because each DFS code will always start with <0,1,label(source),label(edge),label(destination) >.

So the minimum DFS code must contain minimum edge as first DFS edge. Therefore, only all edges with minimum lexicographic representation should be considered as starting point. For each edge of this kind, source is assigned timestamp 0, and destination is assigned timestamp 1. And after putting both in visited list, the DFS code is further explored from the destination node.

Exploring the DFS Codes from a node If the DFS Code for a given node in an exploration is not minimum (we store current minimum DFS code globally) stop exploring the node and return back to source node.

Now, from the destination node, we choose all the edges and divide it into two parts, back edges and forward edges. Back edges are edges having destination nodes which are already visited before the current node. Forward edges are edges to nodes which are yet to be discovered. All the backward edges in order of their discovery and added to current DFS code instance. Now, from forward edges, all with minimum lexicographic order is chosen for exploration. In case of multiple minimum edges, all the edged will be considered. For each edges, we add the d_e in visited list and start exploring the codes further from that node.

Stopping Criteria If there are no forward edges for a node, we pop the node (keeping the current DFS code containing the node removed) and go back to the parent and explore other edges similar to actual Depth-First Algorithm. When we reach the node with timestamp 0 and all the nodes in its adjacency list is visited we stop and update minimum DFS code.

The algorithm is recursive in programming terminology. And, at each node, if the current DFS code is greater than minimum DFS code, the whole branch is pruned.

It is also noticed that unless at least one node has more than one forward edges with minimum lexicographic order only one path will be traversed and it will be the minimum DFS code. But talking about worst case scenario, it can be non polynomial for a binary tree with only one node having different label and all other having same labels or similar case.

To tackle the problem of non-ending loop for large and complex cyclic graphs, hard time limit of 0.2 seconds is put for every graph, so if the algorithm is still exploring other codes (permutations with smallest edges at some node) , it will give current local DFS code instead.