Technical Report

An updated dashboard of Complete Search FSM Implementations in Centralized Graph Transaction Databases

Rihab AYED, Mohand-Saïd HACID, Rafiqul HAQUE and Abderrazek JEMAI

Author's Address

Université Claude Bernard, Lyon 1 (UCBL)
Laboratoire d'InfoRmatique en Image et Systèmes d'information (LIRIS)
43, boulevard du 11 Novembre 1918
69622 Villeurbanne cedex
France
Email: rihab.ayed@liris.cnrs.fr

Abstract

In this report, we present and analyze the results of an experimental study regarding the Frequent Subgraph Mining algorithms implementations. We evaluated the 6 most known and available algorithms. We used datasets from the state of the art and metrics. The idea of this evaluation campaign came when we started looking for a Frequent Subgraph Mining algorithm for indexing large graphs databases for aggregated search.[1]

Keywords

Graph Mining, Graph Transaction Databases, Centralized Environment, Frequent Subgraph Mining(FSM), FSM Algorithms, Complete Search

[1] https://www.irit.fr/CAIR/fr

Table of Contents

1 Introduction

Graph mining is one of the most important approaches in data mining that transforms graph data into knowledge [3]. Frequent Subgraph Mining (FSM) is a subcategory of Graph Mining [28]. The objective of traditional FSM [36] is to extract subgraphs, in a given dataset, whose occurrence counts (*aka* frequency) are above a specified threshold (Minimum Support Threshold). The extracted subgraphs, called Frequent Subgraphs, are (directly) useful for analysis in areas like, biology, co-citations, chemistry, semantic web, social science and finance trade networks [41,73]. They could also be used for other relevant purposes such as classification, clustering, graph indexing and similarity search [44,73].

In the scope of the CAIR[1] project, we are investigating efficient and effective approaches for aggregated search [51] over distributed repositories. One of the building blocks of our approach is graph indexing. The index data can be provided by resorting to an FSM algorithm [90].

Given that there are many algorithms proposed for FSM in the literature, the first task was to categorize the algorithms according to some relevant and desired features. The existing studies [23, 47, 66, 72, 73, 83] devoted to benchmarking FSM algorithms are not satisfactory for our requirements for the following reasons: *(i)* conclusions about algorithms found in these studies do not explicitly consider the effect of the variability of inputs on the performance results. The variability includes the following: the tested real datasets characteristics (size, density) and the minimum support threshold interval (low values, high values) ; *(ii)* two different implementations of a given algorithm - by original authors and the third party implementers - reported different performance results ; *(iii)* the most recent experimental comparisons (2014) [8,19,72,73] is concerned only with four (4) algorithms at most. These algorithms are relatively old (2001 - 2007). About thirteen new algorithms of the same category have been proposed since 2007 ; *(iv)* there is no comparison of the currently available FSM algorithms in the literature ; *(v)* implementations of some algorithms are refined without any experimental study on their performances (*e.g.,* gSpan (2002) [88] release v.6 2009).

2 Contributions

From the set of all complete search FSM algorithms for centralized graph transaction databases proposed so far, we selected a subset. The selection obeys a well-founded approach.

We conducted an experimental study of the selected algorithms using datasets from the state of the art. Only algorithms provided with usable implementations are considered and different implementations of one algorithm are included in the study. We analyzed the behavior of the algorithms according to the following parameters: *(i)* execution time, *(ii)* memory consumption and *(iii)* the number of frequent subgraphs. We analyze the behavior of the algorithms by varying two input parameters: datasets and minimum support threshold. We categorize datasets according to their size: *(i)* small, *(ii)* medium and *(iii)* large. We categorize the minimum support threshold into : low or high. We used only real datasets. We left out synthetic datasets due to the fact that synthetic datasets are generated randomly and they could display features that are not easy to compare. According to [83] "by considering the average of the results of these kind of datasets, no valid conclusion can be made". Also, we study the use of some environment variations (used OS, labeling strategy and format of datasets) to test their effect on the experimental results. We compared our found results with the literature.

This report is organized as follows: Section 3 present some notions of the FSM field. Section 4, contains our theoretical study of FSM algorithms. Section 6 describes the evaluation of the selected FSM algorithms and discusses the results. We conclude in Section 7 and present some perspectives of further FSM studies.

3 Preliminaries

FSM algorithms can be classified according to three different aspects [45,47]: (i) database setting which depends on the applications (*e.g.,* Chemical graphs, Social Networks), (ii) nature of input and output graphs and (iii) the strategy of subgraph search and matching. In what follows, we briefly explain these aspects.

3.1 Database Setting

There are two distinct problem formulations for frequent subgraph mining in graph datasets: (i) graph-transaction setting and (ii) single-graph setting.

3.1.1 Graph-Transaction Setting In this case, the input is a collection of moderate sized graphs (transactions), and a subgraph is considered frequent if it appears in a large number of graphs. A subgraph occurrence is counted only once per transaction, independently of the possible multiple occurrences in the same transaction [36]. Graph Transaction mining is applied in biochemical structure analysis, program control flow analysis, XML structure analysis, image processing and analysis [3,41].

3.1.2 Single-Graph Setting This setting involves mining frequent subgraphs in different regions of one large sized graph. The frequency of a subgraph is based on the number of its occurrences (*i.e.,* embeddings) in the large graph. Special support metrics are used, by considering, for example, the overlapping of two subgraphs [50]. Single Graph mining is dedicated to applications such as social networks, citation graphs, or protein-protein interactions in bioinformatics [20].

In this report, we are interested in algorithms that mine a collection of graphs instead of a single large garph.

3.2 Nature of Input and Output Graphs

In centralized graph transaction mining, the input graphs which are used in most of the FSM algorithms are assumed to be *labeled (vertices and edges) simple*[2] *connected undirected graphs* and the output are *connected subgraphs*. There are algorithms developed for specific graphs (*e.g.,* complex graphs [57], unconnected subgraphs [74], vertex labeled graphs [94], see Table 1) rather than the general ones.

3.3 Subgraph Search and Matching Strategy

FSM algorithms can be classified according to search strategy : *complete* and *incomplete* (or heuristic) search. Also, they can be classified according to the type of isomorphism test (matching) performed between the mined subgraphs : *exact* and *inexact* matching. We explain these categories in what follows.

[2] A simple graph is "an un-weighted and un-directed graph with no loops and no multiple links between any two distinct nodes" [26]

Table 1: FSM Algorithms with specific graphs

Input Cases	**Algorithms**
Complex graphs	MgVEAM [57]
Directed graphs	mSpan [53]
Directed Acyclic graphs	DIGDAG [76]
Unlabeled graphs	The smoothing-clustering framework [13]
Vertex-labeled graphs	Cocain [94], TSMiner [42]
Relational graphs	CODENSE [32], CLOSECUT & SPLAT [91], Fp-GraphMiner [79]
Geometric graphs	gFSG [50], MaxGeo [10], FREQGEO [68]
Uncertain graphs	Monkey [96], RAM [95], MUSE [98]
Output Cases	**Algorithms**
Cliques and quasi-cliques from dense graphs	CLAN [81], Cocain [94]
Unconnected subgraphs	UGM [74]

3.3.1 Complete Search The complete search[3] algorithms perform a complete mining *i.e.,* it guarantees to find all frequent subgraphs from the input data, above a minimum frequency threshold [37,48].

a) Exact Matching: It consists in finding all possible frequent subgraphs as they appear in the input data [37, 48]. The complete search must return a frequent subgraph (*e.g.,* subgraph (1) shown in Figure 1) and all of its possible subgraphs that are necessarily frequent as well (*e.g.,* subgraphs (2), (4), (5), (6) and (7) shown in Figure 1).

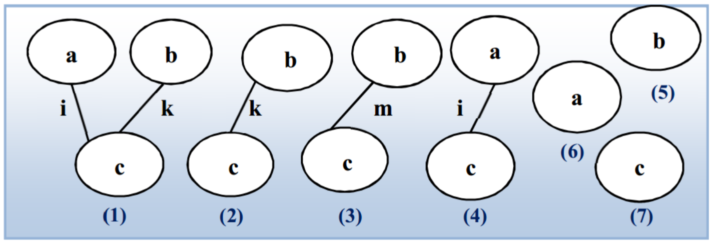


Fig. 1: Example of all Frequent Subgraphs (Exact Matching)

b) Approximate Matching: It consists in finding all frequent subgraphs, with an assumption that subgraphs having the same structure and different labels, will all be returned as the same subgraph [53]. This is considered as a complete search because all possible frequent subgraphs could be verified in the output set with the abstraction of labels (edges or vertices). Figure 2 illustrates the approximate matching where graphs with different edge labels are considered the same. For example, the subgraphs (2) and (3) in Figure 1 could be represented with the approximate matching by one subgraph (2') in Figure 2.

[3] Complete search is also called ”exact search” [41,73], we will use, in this report, only the designation ”complete search”

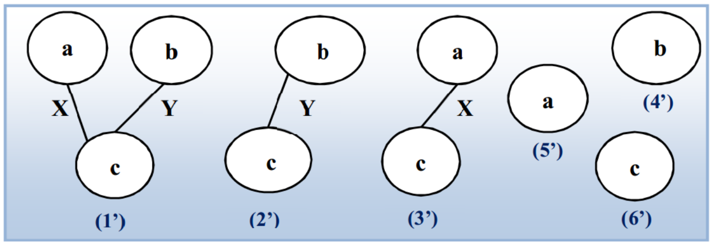


Fig. 2: Example of all Frequent Patterns (Approximate Matching)

3.3.2 Incomplete or Heuristic Search The incomplete and heuristic search algorithms discover a set of frequent subgraphs whose cardinality is greater or lower than the one returned by the complete search. This category of FSM search is used to : (i) reduce the set of frequent subgraphs (use of exact [89] or approximate matching [15]), or (ii) add more frequent subgraphs than the complete search in order to consider the innacuracy or uncertainty of the input data (use of approximate matching) [99].

c) Exact Matching: It consists in returning a subset of frequent subgraphs [82] by setting a supplementary calculable parameter (*e.g.,* maximum size of frequent subgraphs, closed subgraphs, maximal subgraphs, maximum support threshold) [6, 34, 89], besides the minimum support threshold. Figure 3 shows an example returning a subset of frequent subgraphs (see all frequent subgraphs, Figure 1) where the set parameter is the maximum size of frequent subgraphs (set to 2 edges).

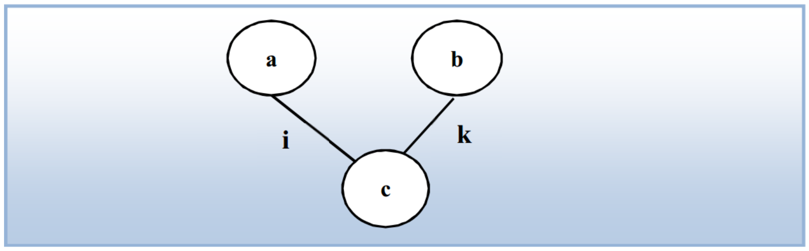


Fig. 3: Example of a subset of Frequent Subgraphs (Exact Matching)

d) Approximate Matching: It consists in either (i) reducing the output by returning a set of representative frequent patterns or (ii) enriching the frequent subgraphs output by considering the inaccuracy or uncertainty of data [40, 99]. For the first case, a representative frequent pattern is a frequent subgraph similar to a set of other frequent subgraphs. In other words, frequent subgraphs that have some differences regarding edges, vertices and labels are represented by one pattern in the output [4]. For the second case, it consists in adding infrequent subgraphs that are similar to frequent subgraphs with respect to the structure or labels [2].

For the four subcategories a, b, c and d, there are 31, 1, 22 and 15 algorithms respectively.[4] to have the list of all FSM algorithms in centralized graph transaction databases.

[4] Please refer to https://liris.cnrs.fr/rihab.ayed/ACFSM.pdf

In our work, we are interested in algorithms that perform a complete search (subcategories a, b). Our main objective is to identify an efficient FSM algorithm for generating subgraphs which will be used to index large repositories. The incomplete search algorithms (subcategories c, d) that are available and usable for general purposes[5] propose to return : (i) closed subgraphs ([75, 89]), (ii) maximal subgraphs [6, 34], (iii) significant subgraphs [86], (iv) sample of fixed size subgraphs [73] or (v) approximate subgraphs [40]. The closed and maximal subgraphs could not be used for the purpose of indexing [90]. The sampling and approximation of subgraphs can be used for indexing. However, we did not select probabilistic or approximation algorithms to avoid the impact of their output set (*i.e.,* frequent subgraphs) on our indexing approach.

To the best of our knowledge, no exhaustive list of Complete search FSM algorithms has been provided so far. Also, there is no study that cites all the currently available FSM implementations. In the next section, we provide a list of all algorithms and highlight their availability and usefulness. We will finally select a few of them.

4 Selection of candidate FSM algorithms

In this section, we describe our approach and criteria for selecting candidate algorithms. To establish our selection process, we defined a set of criteria which includes: *performance reported in literature*, *availability of implementation*, and *miscellaneous weaknesses*. We also point out the ambiguities, found in state of the art regarding the most efficient algorithm to use. We provide details of the experiments settings reported in literature, in order to make our further experimental choices understandable.

4.1 List of FSM Algorithms

We identified thirty-two algorithms (in the literature) designed to extract all possible frequent subgraphs above a minimum support threshold (see Table 2).

Before studying the performance and availability of these algorithms, we investigated their usage. We define the usage of an algorithm in accordance with three facets: (i) *the number of experiments*[6] *performed with the algorithm for centralized graph transaction datasets*, (ii) *the number of real datasets* used for testing, and (iii) *the most recent experiment (i.e., paper*[7]*) with the algorithm.* In Table 3, E, D and R denote these facets, respectively. We found that eleven out of the thirty two algorithms are relatively more popular. Table 3 shows that the most tested algorithms in the literature are: gSpan [88], Gaston [64], FSG [48] and FFSM [33].

Additionally, Table 3 illustrates that the recent FSM algorithms (*e.g.,* LC-Mine [19]) are compared with the least recent algorithms (*e.g.,* gSpan [88], FSG [48]), instead of the most recent ones. Questions are raised in the FSM field about the availability and performances of each algorithm among the 32 ones proposed.

In the following sections, we discuss the outcome of our investigations in terms of performance, availability and miscellaneous weaknesses.

[5] https://liris.cnrs.fr/rihab.ayed/ACFSM.pdf

[6] We counted the number of distinct authors experiments. Authors that experimented the algorithm in many papers are counted once

[7] original paper of the algorithm is not considered

Table 2: An exhaustive list of FSM Centralized Algorithms (Complete Search)

Algorithm	Author	Algorithm	Author
WARMR	[17]	ADI-Mine & GraphMiner	[82, 86]
AGM	[36]	TSMiner	[42]
FARMER	[63]	FSP	[29]
MOLFEA	[46]	DMTL	[5]
AcGM	[39]	gRed	[23]
B-AGM	[37, 38]	FSMA	[84]
FSG	[48]	mSpan	[53]
FREQGEO	[68]	SyGMA	[18]
MoFa/MoSS	[12]	CGM & UGM	[74]
DPMine	[27]	gdFil	[22]
gSpan	[87, 88]	grCAM	[24]
Topology	[31]	ADI-Minebio	[16]
FFSM	[33]	Fp-GraphMiner	[79]
DSPM	[14]	FSMA	[25]
AGM-H	[59]	LC-Mine framework: FG-MAC & AC-miner	[19]
GASTON	[64, 65]	IDFP-tree	[58]

Table 3: The usage of Centralized FSM algorithms (Complete Search)

Algorithm (year)	E	D	R (year)
gSpan (2002) [88]	25	25	[58] (2015)
Gaston (2004) [64]	11	14	[73] (2014)
FSG (2001) [48]	9	11	[19] (2014)
FFSM (2003) [33]	5	10	[72] (2014)
AcGM (2002) [39]	4	3	[73] (2014)
MoFa (2002) [12]	3	6	[74] (2009)
FSP (2007) [29]	2	3	[72] (2014)
ADI-Mine (2004) [80]	2	3	[81] (2006)
FSMA (2008) [84]	2	0	[79] (2011)
MOLFEA (2001) [46]	2	2	[38] (2005)
WARMR (1998) [17]	2	1	[64] (2004)
LC-Mine (2014) [19]	1	10	-
The remaining 20 algorithms	1	<5	-

4.2 Performance of FSM Algorithms

Studies in the literature reported that the performance of four algorithms, namely WARMR [17], FARMER[8] [63], UGM & CGM[9] [74] and MOLFEA [46], is commonly poor. Also, we found that FSMA algorithm [25] was experimented moderately and was not compared with any FSM algorithm.

[8] WARMR and FARMER were both used for itemsets and complex relations

[9] MoFa is competitive with UGM&CGM. MoFa has a poor performance compared to Gaston, gSpan, FFSM [62, 83]

Therefore, we removed these five algorithms from the list of potential candidates. It is worth noting that

Table 4: Contextual performance of FSM algorithms

a) Is Gaston or gSpan a more efficient algorithm?	
Gaston is the fastest graph mining algorithm compared to gSpan and FSG [64]	For the large dataset NCI and for low support threshold, Gaston was slower than gSpan [83]
Gaston RE is the best memory consumer over Gaston, FFSM and gSpan [66]	gSpan is the best memory consumer comparing to Gaston and FFSM [83]
b) Is FFSM more efficient than gSpan ?	
FFSM outperformed gSpan [33] FFSM achieves a considerable performance gain over gSpan [69]	gSpan was slightly faster than FFSM. GSpan was the best algorithm regarding its memory requirements compared to FFSM, MoFA, Gaston [83] gSpan is almost as competitive as Gaston and FFSM, at least with not too big fragments [19]
c) Is FSG an efficient algorithm to use?	
gSpan outperforms FSG by an order of magnitude in terms of runtime [87]. AcGM is faster than FSG [39]	gSpan and FSG are placed among the most efficient graph miners in their respective categories [19]

we found ambiguities in several experiments of well-known FSM algorithms. This led to a confusion in terms of choosing the best candidates. Table 4 shows some examples of ambiguities, including: (i) no general conclusion determines which of the two algorithms FFSM and gSpan is the most efficient one (see *case b* in Table 4); (ii) the performance comparison of Gaston and gSpan depends on the dataset (*e.g.,* Large NCI dataset [83]) and the used implementation (Gaston or Gaston RE) (*case a* shown in Table 4). The contexts of the experiments (*e.g.* FSM implementation, the support threshold interval, datasets characteristics) were not defined adequately in order to have a complete view of the algorithms' performance. In Section 6, we conduct such an analysis, and provide a choice of an algorithm with the specification of its performance cases.

4.3 Weaknesses

We removed four algorithms for limitations regarding input graphs (see Table 5). In fact, we intend to compare algorithms that propose generic usage. The analysis of the algorithms weaknesses allowed to keep a list of twenty-three algorithms.

Table 5: FSM Algorithms with specific uses

Algorithm	**Input Graphs Case**
FREQGEO [68]	Geometric Graphs (2D or 3D)
TSMiner [42]	Graphs with unlabeled edges
SyGMA [18]	The number of labels has to be small
ADI-MineBio [16]	- The input data is relational tables - Dedicated for specific biomedical data

4.4 Availability of Software

We tried to collect the implementations of the twenty-three algorithms. However, only one-third implementations (7 out of 23) are publicly available. According to our study, the reasons of unavailability are (see Table 6): (i) legal constraint (intellectual property right), (ii) codes are lost and (iii) no response from the authors following our requests.[10]

There are different implementations of the seven algorithms. Table 8 shows the variants. AcGM and four implementations of gSpan, FFSM and Gaston were removed from the list due to technical shortcomings (see Table 7 for the details). We could have tried to debug the algorithms but our main objective is to use and compare existing implementations as such without making any changes.

Table 6: Unavailable FSM algorithms

Algorithms	**Unavailability**
AGM [36], Topology [31], AGM-H [59], B-AGM [37], ADI-Mine [80], FSP [29], FSMA [84], mSpan [53], LC-Mine framework [19], IDFP-tree [58]	No answer from authors
gRed [23], gdFil [22], grCAM [24]	Under intellectual propoerties
DPMine [27], DSPM [14], Fp-GraphMiner [79]	The code is lost

The final list of candidate algorithms contains six algorithms with thirteen implementations. We performed an experimental study with these implementations.

Table 7: FSM Implementations with Technical Drawbacks (Complete Search)

Implementation	**Technical Drawbacks**
gSpan ParSeMis	- Quality of Frequent Subgraphs (redundancy) - Error during the execution
gSpan (Kudo, 2004)	- Requiring an additional software (MATLAB)
FFSM Original	- Error with Input Files (No answer from authors about this error)
AcGM Original	- No information about Memory Consumption or Runtime (binary code and no response from authors) - The output is only the DFS code of frequent subgraphs
Gaston ParSeMis	- Error during the execution

4.5 Experimental Setting in Literature

We found different experimental settings in literature used for testing FSM algorithms. In this section, we briefly describe these settings.

[10] 1 request and 2 reminders have been sent to authors
[11] ParMol framework could be provided by authors [56, 83]
[12] No theoretical work is related to this implementation

4.5.1 Datasets The largest datasets - for experimenting FSM implementations in centralized environment - found in the literature does not exceed 274 860 graphs with an average graph size (|T|) that contains a maximum of 50 edges and a maximum of 90 labels (|N|) for vertices and 4 for edges. For the most dense datasets, the average graph size does not exceed 3636 vertices and 206 747 edges where the number of graphs (|D|) is 11. The largest dense datasets contain a maximum of 1178 graphs with an average graph size not exceeding 360 vertices and 910 edges. Table 9 shows the largest, most dense and largest dense datasets characteristics. Synthetic datasets do not exceed 100 000 graphs. A dense synthetic dataset contains (generally) a maximum of 400 vertices and 1000 edges.

For evaluationg FSM implementations in a distributed environment, we found real datasets that can contain 46 703 496 graphs [54] and synthetic datasets that can contain 100 000 000 graphs [8].

4.5.2 Memory Resources The maximum size of main memory used in most of the experiments found in the literature does not exceed 4 GB except for (i) gSpan, Gaston, FFSM, FSG and AcGM in [66] with 10 GB, (ii) gSpan and Takigawa Algorithm [75] with 48 GB and (iii) gSpan and Gaston [73] with 128 GB.

4.5.3 Evaluation Metrics Typically, three common metrics have been used to compare implementations: (i) execution time, (ii) memory consumption and (iii) number of extracted frequent subgraphs. More detailed metrics about subtasks efficiency and the quality of subgraphs (*e.g.*, the execution time of the subtasks [66], the sub-optimality [83], the number of duplicate candidates [24]), were used as well.

Table 8: Available Implementations of FSM algorithms (Complete Search)

Algorithm	**Available versions**	**Last Release**
FSG [48]	FSG Original v1.37 (PAFI v1.0.1) [43]	2003
gSpan [88]	gSpan Original v.6 [85]	2009
	gSpan Original 64-bit v.6 [85]	2009
	gSpan ParSeMis [30, 70]	2011
	gSpan (Kudo) [67]	2004
	gSpan ParMol [11]	2013
	gSpan (Zhou)[12] [97]	2015
MoFa/MoSS [12]	MoFa ParMol [56, 83]	2013
	MoSS ParMol [56, 83]	2013
	MoFa/Moss Original (Miner v6.13) [11]	2015
AcGM [39]	AcGM Original [35]	-
FFSM [33]	FFSM Original v3.0 [21]	2010
	FFSM ParMol [56, 83]	2013
Gaston [64]	Gaston Original v1.1 [61]	2005
	Gaston Original RE v1.1 [61]	2005
	Gaston ParMol [56, 83]	2013
	Gaston ParSeMis [30, 70]	2011
DMTL [5]	DMTL Original v1.0 (g++ 4.8 compiler) [92]	2006

Table 9: Characteristics of Tested Centralized Graph Transaction Datasets in the Literature

Dataset Type (Name)	**|D|**	**|T|**	**|N|**
Largest dataset (DS3) [8]	274860	40-50 (e)	-
Most Dense dataset (US Stock Market) [81, 93, 94]	11	3636 (v) 206747 (e)	-
Largest Dense dataset (DD) [19]	1178	284 (v) 716 (e)	82 (v) 1 (e)

In our work, we do not consider potential differences induced by optimization techniques that were used in different implementations. We only focus on three variables: *the number of discovered/extracted frequent subgraphs*, *the execution time* and *the memory consumption*.

5 A brief description of candidate FSM algorithms

In this section, we briefly explain the algorithms that we retained following our analysis in the previous section. An FSM algorithm can be considered as efficient according to its strategies [41]: (i) graph representation structure (*e.g.,* adjacency list, adjacency matrix, hash table, tries), (ii) subgraph candidate generation (*i.e.,* extending, joining or combinational) by using a search approach (*i.e,* apriori with breadth-first search or pattern-growth with depth-first search), (iii) canonical graph representation for filtering duplicates (*i.e.,* the two main representations are CAM : Canonical Adjacency Matrix or M-DFSC : Minimum DFS Code), (iv) subgraph isomorphism detection strategy to compute the support (*i.e.,* keeping embedding of patterns or explicit subgraph isomorphism). We define the selected 6 algorithms in our study mainly according to these features.

5.1 FSG

FSG (Frequent Subgraph Discovery) [48] uses adjacency lists for storing graphs [47]. It uses an apriori approach. It requires a large amount of memory because it employs BFS and generates a large volume of candidate patterns. Consequently, it scans many times the database and examines a large number of candidates [58]. It uses the CAM canonical representation [83]. It generates candidates using the level-wise join technique. It uses transaction list for support counting. It has a bad performance on graphs with many vertices and edges of identical labels and could be inefficient for mining large-sized subgraph patterns.[13]

5.2 gSpan

GSpan (Graph-based Substructure Pattern Mining) [88] uses adjancency matrix. It uses M-DFSC as a canonical representation. It uses a DFS lexicographic ordering to construct a tree-like lattice over all possible patterns, resulting in a hierarchical search space called a DFS code tree [19]. It performs a rightmost path expansion as subgraph extension [83] which means that the k subgraphs are generated by one edge expansion from the k-th level of the DFS tree. Unlike embedding list saving algorithms, gSpan saves transaction list for each discovered pattern which saves on memory usage. GSpan, with some minor changes, can accommodate to directed graphs [41].

[13] http://web.ecs.baylor.edu/faculty/cho/4352/

5.3 MoFa/MoSS

MoFa (Molecular Frequent Miner) or MoSS (Molecular Substructure miner) [12] is a specialized miner for molecular data. It enables to find frequent molecular substructures and discriminative fragments. However, it can also work on arbitrary graphs. The algorithm is inspired by the Eclat algorithm[14] for frequent item set mining. MoFa stores graphs in adjacency matrices. It follows the pattern growth approach. It uses a rightmost path extension. New subgraphs are built by extending old subgraphs with an edge (and a node if necessary). It uses embedding lists to remove duplicates [56]. It is able to mine directed graphs [41].

5.4 FFSM

FFSM (Fast Frequent Subgraph Mining) [33] is based on gSpan. It uses adjacency matrix for graphs. It follows pattern-growth approach. FFSM uses the CAM representation for canonical graph representation [83]. The CAM tree of the database is built dynamically using two matrix operations of join and extension [22]. FFSM completely avoids subgraph isomorphism testing by maintaining an embedding set for each frequent subgraph [33]. The embedding lists allow to avoid excessive subgraph isomorphism tests and therefore avoid exponential runtime. However, as a trade-off, FFSM faces exponential memory consumption instead [19]. FFSM cannot be used in the context of directed graphs due to its use of triangle matrices [41, 83].

5.5 Gaston

Gaston (GrAph/Sequence/Tree extractiON) [64] is based on gSpan. It uses a hash table representation which justifies its performance over the other algorithms [47]. It follows the pattern-growth approach. Also, Gaston is the fastest among other algorithms [64] due to the fact that it performs subgraph extension using a quick-start principle where paths and trees are considered at first, and general graphs with cycles are enumerated at the end [47]. To detect the duplicate subgraphs, a well-known algorithm, namely Nauty [55] is used to handle the NP-complete subgraph isomorphism problem [29]. Gaston scans the database only once because it uses embedding lists stored in main memory [52]. Gaston cannot be used in the context of directed graphs unless considering major changes [41, 83].

5.6 DMTL

DMTL (Data Mining Template Library) [5] is a library for frequent pattern mining. It offers implementations to mine four type of patterns - itemsets, sequences, trees and graphs - in a unified platform. It performs the joining of two patterns to generate one or more new candidates. It counts support by using a vertical representation of patterns named Vertical Attribute Table (VAT) (*i.e.,* a list of transactions in which the pattern occurs). This vertical representation is typically faster than the horizontal representation of the database due to I/O cost reduction. The joining of patterns is associated with a back end operation : the intersection of two VAT tables of patterns.

[14] Eclat webpage: http://www.borgelt.net/eclat.html

6 The Experimental Study with FSM Implementations

In this section, we present the results of our experiments. We provide the description of our experimental setting. We split our study into : (i) intra-algorithm study where various implementations of a given algorithm are compared, (ii) a comparison of results (for each algorithm) with those of state of the art and (iii) inter-algorithms study where implementations of several algorithms are compared. We conclude this section by a final selection of the most efficient algorithms and some learned lessons regarding the performance of FSM algorithms.

6.1 Experimental Setup

Our experimental settings include: (i) the inputs of implementations (*i.e.,* datasets and minimum support threshold), (ii) the used resources, (iii) information about implementations settings, and (iv) the metrics used to evaluate the efficiency of the implementations.

6.1.1 Input of Implementations There are two inputs: the tested datasets and the minimum support threshold.

Datasets To the best of our knowledge, about thirty-one real datasets with four different formats (TXT, SDF, SMILES, XML) were tested with FSM implementations. However, the available FSM implementations accept only the TXT format, except for ParMol and MoSS Original that accept other formats. For instance, ParMol is able to parse TXT and SDF. MoSS Original parses only chemical formats of data (*e.g.,* SDF, SMILES). Selecting the datasets which were used the most for the experiments reported in literature is an important issue ; because it would enable us to compare the results with existing studies.

We conducted our experiments with twelve available datasets of the two most used formats (TXT, SDF). For all implementations, the default choice was TXT format except for MoFa Original implementation. SDF datasets were converted to TXT format using ParMol parsers [1]. Table 10 displays the datasets we collected for experiments. In the table, */E/* denotes the number of FSM experiments (*i.e.,* papers) in the literature performed on the dataset, *F* is the original format of the dataset, *S* is the dataset size on disk in KB, */D/* is the number of graphs in the dataset, */T/* is the average size of a graph by vertex(v)/edge(e) count, */N/* is the number of labels (vertex/edge) in the dataset and */M/* is the maximum size of a graph by vertex/edge count.

We use the term *"Large datasets"* (*Small*, *Dense*, *Medium*) to refer to the largest datasets in the FSM literature. However, typically the term *"Large"* refers to a huge volume of data which is not the case in this context. We selected the PTE dataset used in 22 FSM experiments, the HIV/AIDS dataset (all releases) used in twenty experiments. We found two available HIV releases: AID2DA99 (October 1999) and AIDS (unknown release). The dataset HIV-CA (all releases) was used in eleven experiments. We found an available HIV-CA release (March 2002) that was used in six experiments.

The rest of the datasets (shown in Table 10) were selected due to their: (i) availability, (ii) format (*i.e.,* TXT or SDF) compatible with the FSM implementations, and (iii) characteristics (*e.g.,* dense, large, medium). In some cases, we carried out the following tasks on some datasets: (i) correcting the parsing errors (NCI250 dataset) with potential graph elimination (AID2DA99, CAN2DA99 datasets), (ii) conversion from SDF to TXT format (*e.g.,* AID2DA99), (iii) grouping a set of files into one dataset file (AIDS, NCI145, NCI330 datasets), (iv) converting string vertex labels to integer ones (DS3 dataset).

Table 10: Available Datasets used in the Literature

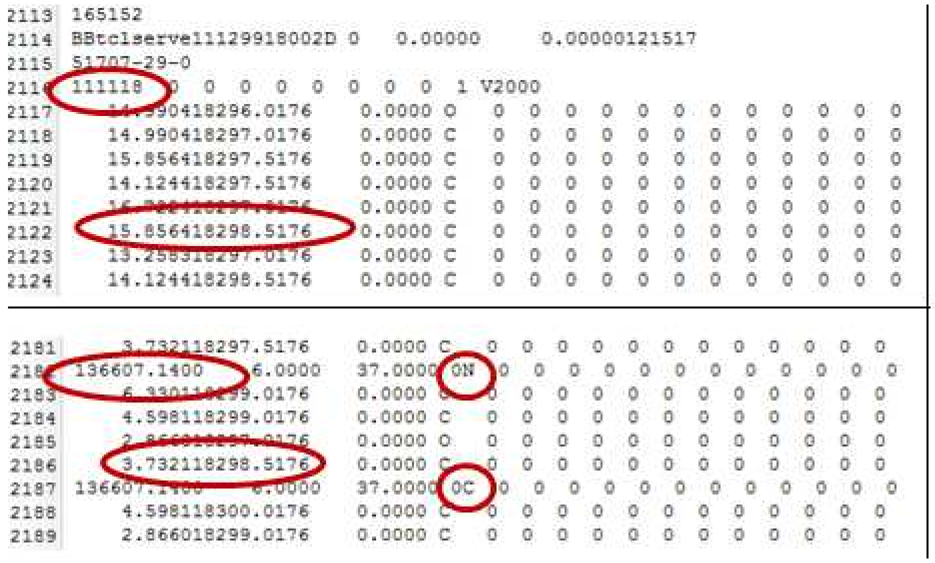
Dataset	|E|	F	S	|D|	|T|	|N|	|M|
			Small datasets				
PTE[15] [61,85]	22	TXT	169.7	340	27(v)/ 27(e)	66(v)/ 4(e)	214(v)/ 214(e)
HIV-CA [85]	6	TXT	285.2	422	40(v)/ 42(e)	21(v)/ 4(e)	189(v)/ 196(e)
			Medium Datasets				
NCI145 [77]	1	TXT	9 900	19 553	30(v)/ 32(e)	53(v)/ 3(e)	110(v)/ 116(e)
NCI330 [77]	1	TXT	9 700	23 050	25(v)/ 27(e)	57(v)/ 3(e)	120(v)/ 132(e)
CAN2DA99 [60]	4	SDF	266000(SDF)/ 14500(TXT)	32 553	26(v)/ 28(e)	66(v)/ 3(e)	229(v)/ 236(e)
AIDS [77]	1	TXT	26 200	56 213	28(v)/ 30(e)	62(v)/ 4(e)	222(v)/ 247(e)
AID2DA99 [60]	4	SDF	111000(SDF)/ 18500(TXT)	42 682	26(v)/ 28(e)	62(v)/ 3(e)	222(v)/ 247(e)
			Large Datasets				
NCI250 [60]	1	SDF	960000(SDF)/ 89600(TXT)	250 251	21(v)/ 23(e)	82(v)/ 3(e)	252(v)/ 276(e)
DS3 [16]	1	TXT	101 700	273 324	22(v)/ 24(e)	83(v)/ 3(e)	99(v)/ 99(e)
			Dense Datasets				
PS[17]	1	TXT	2 975	90	67(v)/ 256(e)	21(v)/ 3(e)	76(v)/ 320(e)
DD [77]	1	TXT	13 100	1 178	284(v)/ 716(e)	82(v)/ 1(e)	5748(v)/ 14267(e)
			Very Dense Datasets				
PI[18]	1	TXT	3 500	3	81607 (e)	2154 (v)/ 1 (e)	136264 (e)

We used the available codes of ParMol software [1] with small modifications to perform these tasks. We eliminated graphs from two datasets CAN2DA99 (4 deleted graphs) and AID2DA99 (7 deleted graphs) due to file[19] errors (for examples, see Figure 4).

For the DS3 dataset, the labels of vertices are either strings (*e.g.,* 1u, 33e) or integers. For ParMol implementations, we modified this dataset because its TXT parser considers integer labels only. We named this modified dataset *DS3M*. Additionally, gSpan Original and Gaston Original implementations parse the DS3 by taking the integer part of labels (*e.g.,* 1e -> 1). This is because gSpan Original and Gaston Original work also with integer labeled datasets. FSG Original is the only implementation able to parse string and integer labeled TXT datasets.

Minimum Support Threshold (MST) Different implementations of FSM algorithms use different internal minimum frequency (integer) because of the conversion of the minimum support threshold (float). The conversion is done by carrying out one of these options: (i) *Truncation* (denoted by **L**), (ii) *Truncation+1* (denoted by **H**), and (iii) *Rounding* (denoted by **L/H**). Table 11 shows the conversion strategy (denoted by C) of FSM solutions and the input type of minimum threshold values: a relative value (*i.e.,* support) (S), an absolute value (*i.e.,* frequency) (F) or both (*e.g.,* ParMol implementations allow both input types, see Table 11). Later in this report, we compared the implementations of the same strategy (L or H).

[19] Please refer to: http://c4.cabrillo.edu/404/ctfile.pdf for a basic SDF file format



```
165152
BBtclserve11129918002D 0   0.00000     0.00000121517
51707-29-0
111118  0   0   0   0   0   0   0   0   1 V2000
   14.990418296.0176    0.0000 O   0   0   0   0   0   0   0   0   0   0   0   0
   14.990418297.0176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
   15.856418297.5176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
   14.124418297.5176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
   16.722418297.0176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
   15.856418298.5176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
   13.258318297.0176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
   14.124418298.5176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0

     3.732118297.5176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
 136607.1400      6.0000    37.0000 ON  0   0   0   0   0   0   0   0   0   0   0   0
     6.330118299.0176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
     4.598118299.0176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
     2.866018297.0176    0.0000 O   0   0   0   0   0   0   0   0   0   0   0   0
     3.732118298.5176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
 136607.1400      6.0000    37.0000 OC  0   0   0   0   0   0   0   0   0   0   0   0
     4.598118300.0176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
     2.866018299.0176    0.0000 C   0   0   0   0   0   0   0   0   0   0   0   0
```

Fig. 4: Examples of SDF file errors - AID2DA99 dataset

6.1.2 Used Resources All of our experiments were performed using a machine with 4 GB of memory and a Quad core processor except for the experiment with a very dense dataset PI (see Table 12). For experimenting the PI dataset we used a different machine with 7 GB of memory and a Quad core processor. We used Linux OS for deploying all FSM solutions. The Windows OS was used only to estimate the effect of varying the OS on the performance results (see Section 6.5.1).

[16] the packages of gSpan and Gaston contain the PTE dataset
[17] DS3 is provided by authors [8]
[18] PS is provided by authors [73]
[19] PI is provided by authors [7]

Table 11: Algorithms' strategy of Minimum Support/Frequency Input

Algorithm Implementation	**S**	**F**	**C**
gSpan Original v.6 2009	x		Truncation
gSpan-64bit Original v.6 2009	x		(L)
gSpan (Keren Zhou, 2015)	x		
ParMol (Gaston, gSpan, FFSM, MoFa, MoSS)	x	x	Truncation+1 (H)
MoFa Original v6.13	x	x	
Gaston Original v1.1		x	
Gaston Original RE v1.1		x	-
DMTL Original		x	
FSG Original (PAFI v1.0.1)	x		Rounding (L/H)

Table 12: Machine Characteristics

Cases	**Default**	**Very dense datasets case**
Processor	Intel Core i3 Quad Core	
	2.40GHz	3.2GHz
RAM	4 GB	7 GB
Hard Disk	192.8 GB	226 GB
OS	*Default* : Ubuntu (14.04) : All Software	
	Windows 7 : ParMol and MoFa Original	

6.1.3 Implementations settings Theoretically, complete search FSM algorithms return all frequent subgraphs that are above a specified minimum support threshold. However, in practice, the available FSM solutions of complete search algorithms produce a lower number of graphs compared to the complete set. According to the contacted authors of ParMol and gSpan Original, this happens because of other internal thresholds and rounding effects defined in the implementation.

Table 13 shows the type of implementation of FSM algorithms, and programming language used for implementing these solutions. For Java based solutions, we used the default Java Heap Space (1 GB) and increased it when required (up to 3.8 GB). We ran each solution three times for each support value. The results that are reported in this report are the mean of the three executions. Some of the solutions (gSpan Original, gSpan (Keren Zhou), ParMol) propose optional multi-threading execution. We used single thread in our experiments.

Table 13: Frameworks characteristics

Programming Language (Open Source)	Java	ParMol, MoFa Original
	C++	gSpan (Zhou, 2015), Gaston Original v1.1, Gaston Original RE v1.1, DMTL Original
Binary Code	-	gSpan Original v.6, gSpan Original 64-bit v.6, FSG Original
Java configurations	Java Heap Space(JHS)	1GB (default)
	Increased JHS	3.8GB
	JVM	1.8.0_65-b17
	IDE	Eclipse Mars 4.5.1 (64bits)
Number of Runs	3	

The execution time is composed of parsing time and the time to extract frequent subgraphs. It is worth noting that FSG Original is the only implementation which does not provide information about parsing time. Thus, in this case, we estimated the parsing time by using an external time calculation function (see Table 14).

For some solutions, we found only binary codes (see Table 13); they do not return information about the memory consumption. In these cases, we tried to deduce the limit of memory consumption by testing the lowest support threshold values that could be reached by the solution. We verified that the failure at low support values is due to a lack of memory (by resorting to a machine with 128 GB of memory).

ParMol algorithms and MoFa Original are set by default to return only closed frequent subgraphs. We set off this option. Additionally, by default, ParMol is not set to parse TXT format. However, there is a TXT parser (LineGraphParser) in the ParMol package [1]. We used it for our TXT datasets. For ParMol implementations, we set the following arguments : graphFile (*i.e.,* graph input), outputFile (frequent subgraphs output), minimumFrequencies (minimum support), memoryStatistics set to true (memory consumption) and debug set to 1 (subtasks runtime). It is worth noting that adding some arguments (*i.e.,* memoryStatistics argument) can change the performance of the implementations (see Section).

MoFa Original v6.13 parses chemical datasets (*e.g.,* SDF, SLN). Therefore, we tested it only with 3 SDF format datasets. Also, the software proceeds to a special modification of edge labels (conversion of found Kekule representations[20] into aromatic bonds [21] [12]). We ran two versions of the software : *(a)* with Kekule Representation conversion step and *(b)* without conversion.

Abbreviations of implementations will be used further in experimentation results. They are as follows, **SO** : gSpan Original v.6 (2009), **SO64** : gSpan-64bit Original v.6 (2009) **SP** : gSpan ParMol, **SK** : gSpan (Keren Zhou, 2015), **GO** : Gaston Original v1.1, **GR** : Gaston Original RE v1.1, **GP**: Gaston ParMol, **D**: DMTL Original, **F** : FSG Original, **FF** : FFSM ParMol, **MFP** : MoFa ParMol, **MSP** : MoSS ParMol, **MOa** : MoFa Original with Kekule Representation Conversion, **MOb** : MoFa Original without Kekule Representation Conversion and **P** : all ParMol implementations.

Table 14: Our Estimated Parsing Time of the FSG Algorithm

Dataset	**Parsing Time (sec)**
HIV-CA	0.5
PTE	0.3
AID2DA99	11
CAN2DA99	8
AIDS	15
NCI145	5
NCI330	5
NCI250	52
DS3	57
DD	7
PS	13

6.1.4 Evaluation Metrics

We use the three common metrics as in the literature (see Section 4.5.3): (i) the execution time, (ii) the memory consumption, and (iii) the number of returned frequent subgraphs. The solutions will be compared between each other considering one of the three metrics.

[20] Alternating between labels 1 and 2 in a chemical ring
[21] relabeling edges by label 4

In the following, **Comp** will denote the qualitative comparison between two implementations, **Diff** will denote a quantitative interval corresponding to the difference (e.g., runtime) between two implementations at the lowest and the highest support value. The symbol $\approx$ will indicate that the versions have approximately equal values. The symbol **(F)** will indicate that the versions have fluctuation in performance (*i.e.,* one version can be better than another in a run and be worse in another run).

6.2 An Intra-Algorithm Performance Study

In this section, we compare different implementations of one algorithm in order to use the best implementations in a further comparison with the other algorithms. There are three algorithms with more than one implementation, namely gSpan, Gaston and MoFa/MoSS (see Table 8). Only gSpan and Gaston implementations are evaluated in this Section. MoFa/ MoSS implementations will be evaluated with all the other algorithms (see Section 6.4).

6.2.1 gSpan Implementations We tested four implementations of gSpan: Two original implementations provided by authors of gSpan [88] (gSpan Original v.6 2009, gSpan 64-bit Original 2009) and an third-party implementations (gSpan ParMol, gSpan (Keren Zhou, 2015)).

Table 15: Number of FS by gSpan (L strategy) - HIV-CA

Min Sup	SP (L)	SO/SO64	SK
4%	-	6825311	-
5%	905299	905298	723603
6%	293406	293404	250518
7%	65260	65259	60183
8%	28559	28558	26304
9%	17512	17511	15945
10%	15973	15972	14486
15%	4476	4476	4152
20%	937	936	915
25%	248	248	239
30%	124	124	120
40%	60	60	56
45%	39	39	35
50%	32	32	29
60%	19	19	16

Number of Frequent Subgraphs GSpan (Keren Zhou, 2015) is able to run with only small datasets (*e.g.,* HIV-CA or PTE). It was not able to run with larger datasets (*e.g.,* AID2DA99, CAN2DA99) or dense datasets (*e.g.,* DD). In addition, gSpan (Keren Zhou, 2015) generates significantly fewer frequent subgraphs than, the two other solutions (ParMol, Original) (see Table 15). The two versions of gSpan Original (v.6 and 64bit v.6) generate the same number of frequent subgraphs except for the NCI330

Table 16: gSpan Original vs gSpan Original 64bit : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**
NCI330		
5%	SO = SO64	-
6%, 8%	SO > SO64	15, 4
9% - 90%	SO = SO64	-
The rest of datasets		
SO = SO64		

dataset (for 6% and 8% minimum support threshold, there is a difference respectively of 15 and 4 graphs, see Table 32).

Typically, gSpan ParMol (L) and gSpan Original (v.6, 64-bit v.6) generate the same number of frequent subgraphs. Sometimes, it can produce one or two graphs in more or less than gSpan Original (v.6, 64-bit v.6) (*e.g.,* HIV-CA dataset, see Table 15). Additionally, in some exceptional cases, such as for PTE dataset, with low support threshold 1.5% and 2%, gSpan Original generates 53 and 49 (respectively) more graphs than the gSpan ParMol version (see Table 17).

It is worth noting also that gSpan algorithm implemented by Original authors and in ParMol, can compute the frequent subgraphs differently. For example, for NCI330 dataset with 6% MST, the two implementations of gSpan generate 4 subgraphs with different frequency[22] values. However, the frequency values are close (*e.g.,* the frequency values for a selected frequent subgraph out of the 4 are 4990 and 5107 for SO and SP, respectively).

Memory Consumption In our experiment, we found that gSpan (Keren Zhou, 2015) required considerably more memory than gSpan ParMol (see Table 18). Also, it was not able to reach the same low support thresholds as gSpan Original v.6, due to high memory consumption (see Table 19).

For low support threshold, gSpan-64bit Original v.6 required more memory than gSpan (Keren Zhou, 2015), and significantly more than gSpan Original v.6 and gSpan ParMol. For example, gSpan Original-64bit could run with a threshold greater or equal to 8% for the HIV-CA dataset, while gSpan Original could run with 4% successfully (see Table 19). The lowest MST we used for this experiment is 1.5%.

GSpan-64bit Original and gSpan (Keren Zhou, 2015) could not run with low support threshold values (see Table 19), unlike gSpan ParMol and gSpan v.6 Original. GSpan Original v.6 is the only implementation that could reach the lowest minimum support threshold (*e.g.,* 4% for the HIV-CA dataset and 1.5% for the DD dataset, see Table 19).

Runtime Our experiments show that gSpan (Keren Zhou, 2015) is the fastest algorithm for high support threshold values (see Figures 5 & 6). However, this version could not be used in the context of dense datasets (*e.g.,* dataset DD) or datasets that are not small in size (number of graphs) (*e.g.,*

[22] The frequency is the number of occurrences of the subgraph, it is the absolute value, while the support is the relative value.

Table 17: gSpan Original vs gSpan ParMol : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Large Datasets**		
HIV-CA			**AID2DA99, CAN2DA99**			**NCI250**		
5% - 20%	SO > SP	1 - 1	SO = SP			2% - 90% \{3%}	SO = SP	-
30% - 90%	SO = SP	-	**AIDS**			3%	SO > SP	931
PTE			1.5% - 2%	SO < SP	2 - 1	**DS3M**		
1.5% - 3%	SO < SP	53 - 1	3% - 90%	SO = SP	-	5%, 20%	SO < SP	1, 1
4% - 90%	SO = SP	-	**NCI330**			3% - 90% \{5%, 20%}	SO = SP	-
Dense Datasets			4%, 6%	SO > SP	1142 , 15	**Dense Datasets**		
DD			5% - 90% \{4%,6%}	SO = SP	-	**PS**		
4% - 90%	SO = SP	-	**NCI145**			80% - 90%	SO = SP	-
			2% - 90% \{3%}	SO = SP	-			
			3%	SO < SP	1			

AID2DA99, CAN2DA99). In addition, this version generated, significantly fewer frequent subgraphs than the other versions (see Table 15).

Table 18: Examples of Memory Consumption of two gSpan versions (L strategy)

Implementation	**Memory (GB)**	**Number of FS**
1.5% (PTE)		
SP (L)	1.1	721 249
SK	48	698 934
5% (HIV-CA)		
SP (L)	2.01	905 299
SK	87	723 603

Table 19: Minimal Support threshold value reached by gSpan versions (L strategy)

	Small		**Dense**	**Medium**		**Large**
Dataset	*PTE*	*HIV-CA*	*DD*	*CAN2DA99*	*NCI330*	*NCI250*
Version	**Min Support Threshold Reached**					
SP	1.5%	5%	4%	2%	4%	2%
SO	1.5%	4%	1.5%	1.5%	3.5%	2%
SK	2.5%	7%	-	-	-	-
SO64	3%	8%	20%	3%	5%	4%

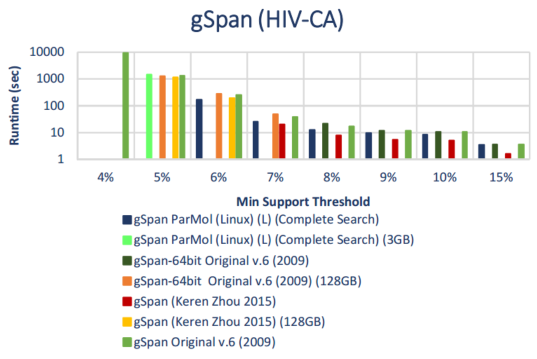


Fig. 5: gSpan Runtime (Low Support Threshold) - HIV-CA

Tables 20, 21 and 22 show the runtime comparison between the rest of the gSpan implementations for datasets and support threshold intervals. The difference between execution times (*Diff*) is mentioned in seconds by an interval corresponding to the runtime difference of the lowest and the highest support value. For example, with the minimum support value 2%, gSpan Original v.6 consumes about 241 seconds more than gSpan Original v.6 64-bit for the AID2DA99 dataset and 5 seconds more for 90% (see Table 20). Our experiments also show that gSpan-64bit Original v.6 is faster than gSpan

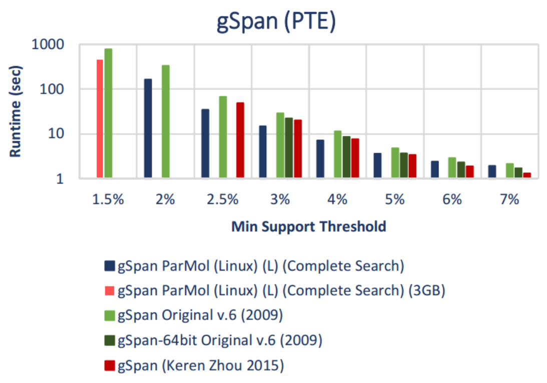


Fig. 6: gSpan Runtime (Low Support Threshold) - PTE

Original v.6 for all the tested datasets. However, for low support threshold values (*e.g.,* 8% - 15% for HIV-CA, see Table 20), it required much more memory compared to gSpan Original. For further lower support values (*e.g.,* $<$ 8% for HIV-CA), it was unable to run due to high memory consumption.

Furthermore, our experiments reveal that gSpan ParMol (L) is faster than gSpan Original v.6 for small and medium datasets (*e.g.,* AID2DA99, see Table 21). It can be slower if the support threshold is very low (*e.g.,* 2% for NCI145). For low and medium support values, gSpan ParMol (L) is slower than gSpan Original v.6 for large (*e.g.,* NCI250) and dense (*e.g.,* DD) datasets (see Table 21). GSpan-64bit Original v.6 was faster than gSpan ParMol for dense and large datasets for all reached low and

Table 20: gSpan Implementations Runtime Comparison (gSpan Original versions)

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**			**Large Datasets**		
HIV-CA			**AID2DA99**			**DS3**		
8%	SO $<$ SO64	4	2% - 90%	SO $>$ SO64	241 - 5	5% - 90%	SO $>$ SO64	123 - 27
9% - 15%	SO $\approx SO64$	-	**AIDS**			**NCI250**		
20% - 90%	SO $>$ SO64	0.1 - 0.083	2% - 90%	SO $>$ SO64	1008 - 6.7	4% - 90%	SO $>$ SO64	200 - 24
PTE			**CAN2DA99**			**Dense Datasets**		
2.94%	SO $<$ SO64	4	3% - 80%	SO $>$ SO64	187 - 4.1	**DD**		
3% - 90%	SO $>$ SO64	7 - 0.004	**NCI145**			20% - 30%	$SO \approx SO64$	1
			5% - 90%	SO $>$ SO64	251 - 2.7	40% - 90%	SO $>$ SO64	2
			NCI330			**PS**		
			5% - 90%	SO $>$ SO64	45 - 2.5	80% - 90%	SO $>$ SO64	3 - 0.04

medium support threshold values (see Table 22). It was slower than gSpan ParMol for medium and small datasets and low support threshold values except for the NCI330 dataset. It had a competitive performance compared to gSpan ParMol for high support threshold values.

Summary

– Of all gSpan solutions, gSpan Original v.6 is the most efficient one in terms of memory consumption for very low support threshold values. GSpan ParMol failed to achieve the search for some low

Table 21: gSpan Implementations Runtime Comparison (gSpan ParMol vs. gSpan Original)

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5%	SO $<$ SP	109	1.5% - 90%	SO $>$ SP	850 - 3.4
6% - 10%	SO $>$ SP	91 - 2.4	**AIDS**		
15%	$SO \approx SP$	-	1.5% - 90%	SO $>$ SP	4636 - 4
20% - 80%	SO $<$ SP	0.4 - 0.27	**CAN2DA99**		
PTE			2% - 80%	SO $>$ SP	837 - 2.54
1.5% - 7%	SO $>$ SP	321 - 0.08	**NCI145**		
8% - 90%	SO $<$ SP	0.23 - 0.34	2%	SO $<$ SP	169
Large Datasets			3% - 90%	SO $>$ SP	1558 - 1.8
NCI250			**NCI330**		
2% - 70%	SO $<$ SP	2089 - 1.5	4% - 5%	SO $<$ SP	292 - 16
80% - 90%	SO $>$ SP	3 - 5	6% - 90%	SO $>$ SP	6.6 - 1.5
Dense Datasets			**Dense Datasets**		
DD			**PS**		
4% - 90%	SO $<$ SP	5882 - 11	80%	SO $>$ SP	13
			90%	SO $<$ SP	0.7

Table 22: gSpan Implementations Runtime Comparison (gSpan ParMol vs. gSpan Original 64bit)

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
8% - 10%	SO64 > SP	8 - 2	2%	SO64 > SP	90	5% - 9%	SO64 > SP	125 - 13
15%	$SO64 \approx SP$	-	3% - 4%	SO64 (F) SP	16/38 - 1	10% - 50%	SO64 (F) SP	9 - 0.4
20% - 90%	SO64 < SP	0.5 - 0.35	5% - 9%	SO64 < SP	25 - 4	60% - 90%	SO64 < SP	0.03 - 0.9
PTE			10% - 90%	SO64 (F) SP	4 - 1.5	**NCI330**		
3% - 4%	SO64 > SP	6 - 1.45	**AIDS**			5% - 8%	SO64 < SP	61 - 5.5
5% - 6%	$SO64 \approx SP$	-	2% - 60%	SO64 > SP	712 - 0.1	9% - 60%	SO64 > SP	1.25 - 0.3
7% - 90%	SO64 < SP	0.3 - 0.34	70%	$SO64 \approx SP$	-	70% - 90%	SO64 < SP	0.4 - 1
Dense Datasets			80% - 90%	SO64 < SP	3 - 2.1	**Large Datasets**		
DD			**CAN2DA99**			**NCI250**		
20% - 90%	SO64 < SP	64 - 14	3% - 5%	SO64 > SP	43 - 6	4% - 90%	SO64 < SP	1020 - 18
PS			6% - 40%	SO64 (F) SP	6 - 0.7			
80%	SO64 > SP	10	60% - 80%	SO64 < SP	0.34 - 1.5			
90%	SO64 < SP	0.8						

threshold values (*e.g.,* HIV-CA 4%) and gSpan-64bit Original failed earlier (*e.g.,* HIV-CA 8%). The failures are mainly due to memory consumption. However, gSpan Original v.6 was able to complete the execution successfully (*e.g.,* HIV-CA 4%).

- GSpan-64bit Original v.6 can be used in a context where support threshold values are not low and the required execution time is critical.
- Instead of gSpan Original v.6, the open source implementation gSpan ParMol can be used for better runtime performance if the dataset is small or medium, not dense, and the support values are not too low.

6.2.2 Gaston Implementations There are three implementations of Gaston: two (Gaston Original v1.1, RE v1.1) are from original authors [64] and the other one (Gaston ParMol) is from a third-party implementer [83]. The implementations are written in two different programming languages (see Table 13). It is worth noting that according to [64,66], Gaston Original v1.1 is much faster and requires much more main memory than Gaston Original RE v1.1 due to the use of different structures for counting the occurrences of a graph.

Number of Frequent Subgraphs In general, Gaston Original versions (v1.1, RE v1.1) generate the same number of frequent subgraphs. However, there could be some exceptions such as the ones we found for the DD (under 20%) and PS (under 80%) datasets. For example, for the 2% MST of DD dataset, Gaston Original v1.1 produced 1359 frequent subgraphs more than Gaston v1.1 RE (see Table 23).

Gaston Original (v1.1, RE v1.1) (H) and Gaston ParMol produced a different number of frequent subgraphs. This is shown in Table 27 where *Diff* denotes the difference, in terms of frequent subgraphs, produced by the two implementations for the lowest and the highest support values.

Table 23: Gaston Original vs Gaston Original RE : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**
Dense Datasets		
DD		
2% - 20%	GO $>$ GR	1359 - 4
30% - 90%	GO $=$ GR	-
PS		
60% - 80%	GO $>$ GR	17013 - 45
90%	GO $=$ GR	-
Rest of Datasets		
GO $=$ GR		

Unlike Gaston ParMol, Gaston Original versions (v1.1, RE v1.1) do not include frequent subgraphs with single vertex (0 edges). With PTE dataset, for 1.5% MST, Gaston Original versions (v1.1, RE v1.1) (H) generated, 57 946 frequent cyclic graphs, 282 724 frequent trees and 2268 frequent paths. Gaston ParMol generated 57 951 frequent cyclic graphs, 284 294 frequent trees and 2234 frequent paths. The difference in the number of frequent subgraphs between Gaston ParMol and Gaston Original (v1.1, RE v1.1) needs to be explained by the authors.

Memory Consumption Typically, Gaston ParMol consumed more memory for all datasets and produced different numbers of frequent subgraphs compared to Gaston Original versions (see Table 25, Figure 8). However, for relatively high values of support threshold and small datasets (*e.g.,* above 6% MST for PTE, and 20% MST for HIV-CA), Gaston ParMol required fewer memory than Gaston Original versions (see Figure 7).

Gaston Original RE v1.1 was proposed by Nijssen et *al.* in order to reduce the memory consumption of Gaston Original v1.1. We found that Gaston Original RE, when it is able to run, had in fact a linear

Table 24: Gaston Implementations: Number of Frequent Subgraphs Comparison (Gaston ParMol vs Gaston Original) - (L/H strategy)

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99/CAN2DA99**[23]			**NCI330**		
6% - 7%	GP $<$ GO	32 - 34	2% - 90%	GP $>$ GO	10 - 3	4% - 90%	GP $>$ GO	90 - 1
8% - 90%	GP $>$ GO	8 - 3	**AIDS**			**Large Datasets**		
PTE			2%	GP $<$ GO	5	**NCI250**		
1.47% - 2.94%	GP $<$ GO	4324 - 10	3% - 90%	GP $>$ GO	8 - 2	60% - 90%	GP $>$ GO	3 - 1
3%	GP $=$ GO	-	**NCI145**			**Dense Datasets**		
4% - 90%	GP $>$ GO	16 - 1	2% - 90%	GP $>$ GO	1443 - 1	**DD**		
			6% - 90%	GP $>$ GO	4 - 18	4% - 5%	GP $<$ GO	49 - 21
						PS		
						80% - 90%	GP $>$ GO	7399 - 8

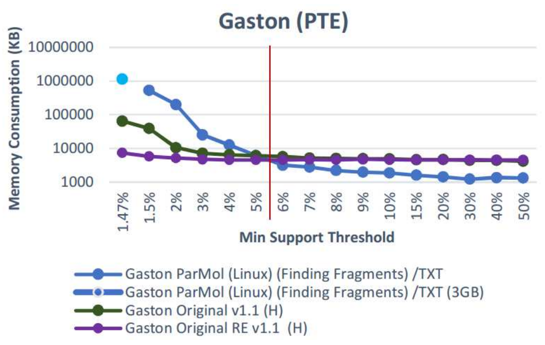


Fig. 7: Memory Consumption of Gaston - PTE

memory consumption lower than Gaston Original (see Figures 7 and 8) for all the tested datasets except for the dense DD and PS datasets.

However, for very low support threshold values (*e.g.,* 3% for NCI330, 6% for HIV-CA) or for relatively large datasets (*e.g.,* DS3, NCI250), Gaston Original RE produced an exception and hence the operation was terminated. For the same cases, Gaston Original completed successfully (see Table 26).

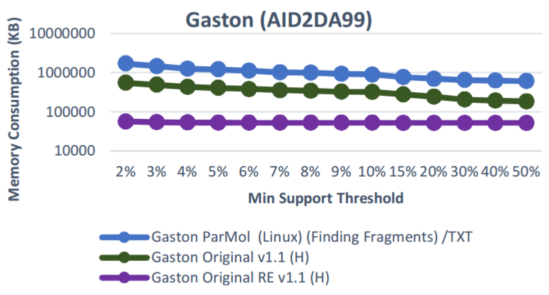


Fig. 8: Memory Consumption of Gaston Original versions - AID2DA99

Runtime The results show that for all used datasets, the runtime performance of Gaston Original v1.1 was the best among all Gaston versions. Gaston Original RE v1.1 required less memory than Gaston Original v1.1, as a trade-off, it was slower (see Figure 9). For the dense DD and PI datasets, Gaston Original RE required more time and memory than Gaston Original with a different number of frequent subgraphs.

Summary

– Gaston ParMol consumed the highest amount of memory amongst all Gaston versions (except for small datasets and high support values), yet it is the slowest solution (*e.g.,* for AID2DA99 dataset,

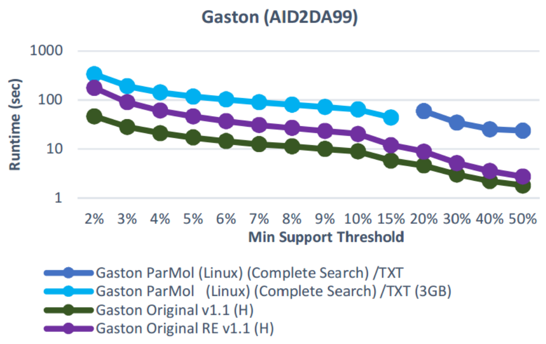


Fig. 9: Gaston Runtime - AID2DA99

see Figure 9) and it produced a number of frequent subgraphs different from what Gaston Original versions produced.

- Gaston Original RE v1.1 can be used to save the memory (despite the consumed time) for the following cases: (a) support threshold values not too low (*e.g.*, above 6% MST for HIV-CA) and (b) datasets that are not large (*e.g.*, smaller than DS3, NCI250) and not dense (*e.g.*, less than DD). If neither (a) nor (b) are verified, then (c) the provided RAM memory should be large enough to handle the mining task. If none of the cases (a) and (b), or (c) are true, then Gaston Original v1.1 has to be used.
- Gaston Original v1.1 should be used for applications where runtime is critical.

6.3 Comparison with the State of the Art

In this section, we compare our results regarding the six algorithms with the results we found in state of the art. The comparison shows the similarities and differences between the results. According

Table 25: Examples of Memory Consumption and Number of Frequent Subgraphs of Gaston versions

Implementation	**Memory (MB)**	**Number of FS**
3% (PTE)		
GO (H)	7	18 121
GR (H)	4	18 121
GP	25	18 121
2% (AID2DA99)		
GO (H)	554.916	25 197
GR (H)	56.300	25 197
GP	1729.857	25 206
5% (DD)		
GO (H)	251812	795623
GR (H)	50964	795717
GP	876978	795696

to our understanding, differences might occured due to different machine characteristics and different implementation releases (*e.g.*, gSpan v.5, gSpan v.6). It is worth noting that we eliminated some graphs from the datasets AID2DA99 and CAN2DA99. Therefore, our versions of these datasets contain slightly fewer number of graphs (7 and 4 graphs, respectively) than the ones tested in state of the art; we believe this could have an impact on the outcome and so on results.

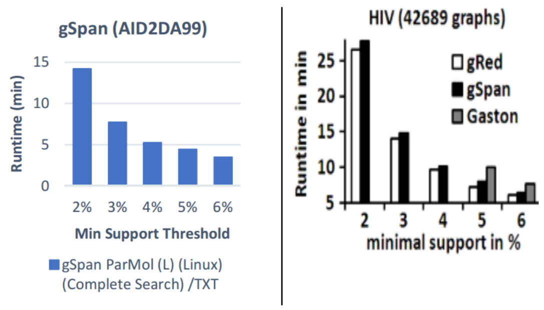


Fig. 10: Comparison of our Runtime (Left) with the Literature [22–24] (Right) - gSpan ParMol (AID2DA99)

6.3.1 GSpan Comparison Our experiment with gSpan Original v.6 (2009) generated a number of frequent subgraphs that is different (superior) from the result found in [47,64,66] for the datasets PTE and HIV-CA (see Table 27). However, the same result was produced by gSpan Original reported in [8] for the DS3 dataset and the result found in [89] is approximately[24] the same for the dataset HIV-CA.

[24] The results are given in a graphical form, we could not deduce a more precise conclusion

Table 26: Limits of Memory Consumption (KB) of Gaston versions for low support threshold

Version/ Dataset	**GO (L)**	**GR (L)**		**GP**
Min Sup	**4%**	**6%**	**7%**	**6%**
HIV-CA	15456	*Segmentation Fault*	3956	183400
Min Sup	**1%**	**3%**	**3.5%**	**4%**
NCI330	238512	*Segmentation Fault*	46072	676437
Min Sup	**2%**	**90%**		**60%**
NCI250	2759732	*Segmentation Fault*		2557084
Min Sup	**2%**	**90%**		**50%**
DS3M	3067400	*Segmentation Fault*		*Out Of Memory*
Min Sup	**1%**	**1.5%**	**2%**	**3.5%**
DD	66944	*Killed*	2744180	2114479

Table 27: Number of Frequent Subgraphs in our Experiment and Nijssen et *al.* Experiment [64, 66] - gSpan - PTE

Our experiments					**Nijssen et *al.***
Min Sup	**SP (L)**	**SP (H)**	**SO**	**SK**	**SO (L/H)**
2% (6.7)	344513	**136981**	344464	338284	**136949**
3% (10.2)	**22786**	18146	**22785**	**22200**	**22758**
4% (13.59)	8776	**5955**	8776	8706	**5935**
5% (17.0)	**3627**	**3627**	**3627**	**3607**	**3608**
6% (20.4)	**2343**	2138	**2343**	**2326**	**2326**
7% (23.8)	1861	**1786**	1861	1845	**1770**
8% (27.19)	**1339**	1240	**1339**	**1323**	**1323**
9% (30.6)	1065	**993**	1065	1049	**977**
10% (34.0)	**860**	**860**	**860**	**844**	**844**
20% (68.0)	**199**	**199**	**199**	**190**	**190**
30% (102.00001)	**75**	75	**75**	**68**	**68**

Considering the runtime performance, gSpan Original v.6 (2009) was slightly slower than gSpan Original - reported in [8, 47, 64, 66] for PTE, HIV-CA and DS3 datasets. The result reported in [89] for HIV-CA dataset was approximately similar to ours. According to our understanding, the difference regarding runtime could be due to different machine characteristics and to the number of generated frequent subgraphs.

Regarding the impact of the differences between the gSpan versions (2002-2009), Xifeng Yan - the author of gSpan Original explained the following: *"They are the same, except the new one supports more labels and it is running on a 64 bit system"... "The new version supports multi-threads, and more labels. Therefore, it consumes more memory (50%-100%)..."*

Our experiment with gSpan ParMol for PTE dataset produced the same number of frequent subgraphs as in [22]. For AID2DA99 dataset, our result was considerably faster (see Figure 10), consumed slightly more memory (see Figure 11) and produced approximately the same number of duplicates (see

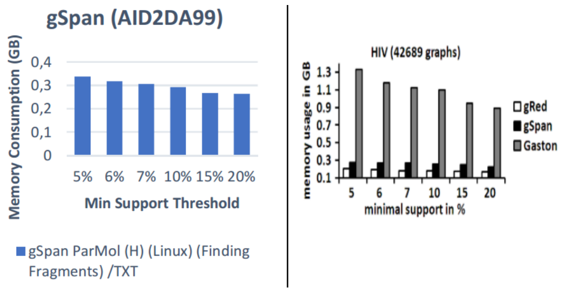


Fig. 11: Comparison of our Memory Consumption (Left) with the Literature [23] (Right) - gSpan ParMol (AID2DA99)

Figure 12) compared to gSpan ParMol result found in [22–24]. The difference in runtime could not be completely understood. Although, initially we assumed that the runtime performance is different due to different machine specification. However, we found that even using a machine with the same resource specification did not alleviate this difference.

Concerning the different versions of ParMol, Thorsten Meinl, one of the authors of ParMol, stated the following: *"We released several versions ... the changes had only minor effects on runtime and memory consumptions since those are mostly determined by the algorithm and not by the implementation"*.

6.3.2 Gaston Comparison *Gaston Original:* In our experiment, Gaston Original (v1.1, RE v1.1) generated the same number of frequent subgraphs as in [47, 64] (for PTE and HIV-CA respectively). It generated a fewer (one less) number of frequent subgraphs than the result in [8] (for 30% MST, DS3 dataset).

Table 28: Gaston Memory Consumption (MB): Comparison with the Literature [64, 66] - PTE

Our experiments					**Nijssen et al.**	
Min Sup	**GO (L)**	**GO (H)**	**GR (L)**	**GR (H)**	**GO (L/H)**	**GR (L/H)**
2%	38.786	**10.421**	5.669	**5.096**	**9.1**	**1.5**
3%	**7.158**	7.062	**4.688**	4.618	**4.4**	**1.3**
4%	6.588	**6.354**	4.518	**4.518**	**3.4**	**1.3**
5%	**5.946**	**5.946**	**4.518**	**4.518**	**3.0**	**1.3**
6%	**5.688**	**5.598**	**4.558**	**4.520**	**2.7**	**1.3**
7%	5.237	**5.088**	4.516	**4.576**	**2.1**	**1.3**
8%	**5.042**	4.945	**4.552**	4.510	**1.9**	**1.3**

Gaston Original (v1.1, RE v1.1) consumed more memory than the version found in [64, 66] (PTE dataset, see Table 28). Its runtime performance was better than the result in [64, 66] (PTE dataset,

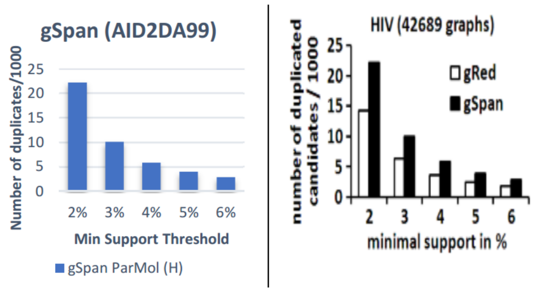


Fig. 12: Comparison of our Number of Duplicates (Left) with the Literature [22–24] (Right) - gSpan ParMol (AID2DA99)

see Table 29) and in [47] (HIV-CA dataset). However, Gaston Original v1.1 has competitive runtime as in [8] (for the DS3 dataset).

It is worth noting that we used different resource specification including a more powerful processor[25] than the one used in [64, 66] and different from the ones used in [8, 47].

Table 29: Gaston Runtime: Comparison with the Literature [64, 66] - PTE

Our experiments					**Nijssen et *al.***	
Min Sup	**GO (L)**	**GO (H)**	**GR (L)**	**GR (H)**	**GO (L/H)**	**GR (L/H)**
2%	6.6275	**2.2836**	24.0183	**9.9545**	**7.9**	**39.6**
3%	**0.4545**	0.3635	**2.0841**	1.6018	**1.7**	**8.5**
4%	0.1932	**0.1509**	0.8405	**0.6583**	**0.6**	**2.7**
5%	**0.0959**	**0.0959**	**0.3836**	**0.3836**	**0.4**	**1.6**
6%	**0.0684**	**0.0599**	**0.2463**	**0.2258**	**0.3**	**1.0**
7%	0.0529	**0.0501**	0.1915	**0.1797**	**0.3**	**0.8**
8%	**0.0426**	0.0415	**0.1432**	0.1213	**0.2**	**0.6**

Gaston ParMol: Gaston ParMol generated a number of frequent subgraphs which is different from the number reported in [22] (for AID2DA99, see Table 30).

Table 30: Number of Frequent Subgraphs - Gaston : Comparison with the Literature - AID2DA99

Our experiments		**Gago-Alonso et *al.* [22]**
Min Sup	**GP**	**GP**
3%	18121	18146
4%	5951	5955
5%	3625	3627
30%	75	75
40%	62	62
50%	37	37

According to our results, Gaston ParMol was faster than the one tested in [22–24] (AID2DA99 and PTE datasets, see Figure 14). However, it consumed slightly lesser memory than the one reported in [23] (AID2DA99 dataset, see Figure 14).

6.3.3 FSG Comparison In our experiment, FSG Original *v1.37* generated different number of frequent subgraphs compared to the version tested by Kuramochi et *al.* [49] for some threshold values (*e.g.* 2%, 7.5% for PTE, see Table 34). However, the experiments reported in [8, 47, 62] produced the same number of subgraphs as in our experiments (for PTE, HIV-CA and DS3 datasets, respectively). The

[25] Nijssen et al. used a single processor of a 2GHz Pentium, see Table 12 for our processor characteristics

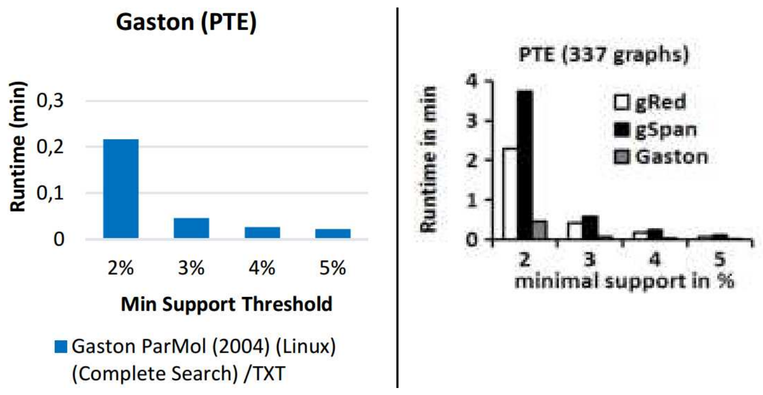


Fig. 13: Gaston Runtime: Comparison with the Literature - PTE

runtime performance in our experiments with FSG Original was close to the performance[26] reported in [8, 47] (for HIV-CA and DS3 datasets, respectively).

However, it showed a better performance (*i.e.,* two or three times) than the FSG evaluated[27] in [62] (for PTE dataset, see Table 32).

FSG had considerably better runtime (up to 50 times less) than the one reported in [38, 89] (for PTE and HIV-CA datasets, respectively). It is worth noting that we used a more powerful processor than the ones used in [38, 89]. However, the difference in FSG results cannot be only related to the processor. In fact, gSpan in our experiments did not have such a huge difference, with the the same literature results (less than 2 times slower [38, 89]). We relate the difference of the FSG results to FSG version evolution. Since our experiments rely on binary release of FSG, we could not compare memory consumption with state of the art.

[26] It is worth noting that the processor of [47] and [8] are different from our
[27] It is worth noting that our processor was more powerful than in [62]

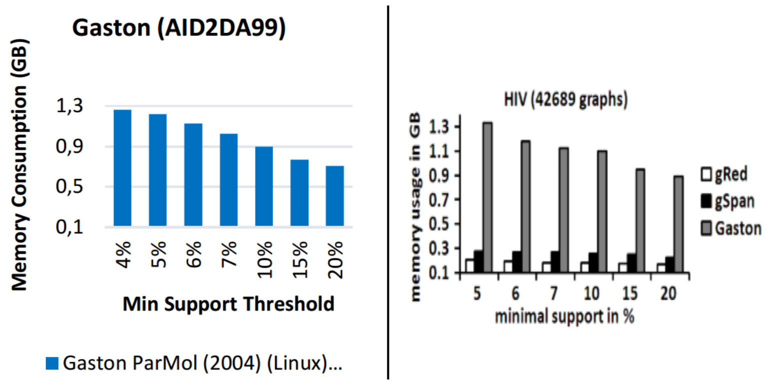


Fig. 14: Gaston Memory Consumption (GB): Comparison with the Literature - AID2DA99

Table 31: Number of Frequent Subgraphs - FSG : Comparison with the Literature - PTE

Min Sup	**Our experiments**	**Kuramochi et *al.* [49]**
	F	**F**
2%	*136949*	*136927*
3%	22758	22758
4%	5935	5935
5%	3608	3608
6%	2326	2326
7%	1770	1770
7.5%	*1459*	*1590*
8%	1323	1323
9%	977	977
10%	844	844

6.3.4 DMTL Comparison We found only one *available real* dataset tested with DMTL in the literature (the dense dataset PI [7]). In [7], the basic version of DMTL crashes within few minutes with a 2 GB of RAM. We left DMTL running for days, it did not complete. We then aborted the execution.

6.3.5 FFSM Comparison The comparison of our results with the ones found in the literature [22] shows that the number of duplicates generated by FFSM ParMol is the same for the PTE dataset and a slightly more for the AID2DA99 dataset. This raises a question - if we have *less* graphs and labels in our modified[28] AID2DA99 dataset than the one in [22], what makes the number of found duplicates in our result *more* than the one in [22].

[28] This is due to eliminated graphs with parsing errors, see Section 6.1.1

Table 32: FSG Runtime: Comparison with the Literature [62] - PTE

Min Sup	**Our experiments**	**Nijssen et *al.***
	F(L/H)	**F(L/H)**
2%	128.5333	307.4
3%	18	43.9
4%	4.4	11.0
5%	2.5	6.3
6%	1.6	4.0
7%	1.2	2.9
8%	0.9	2.4
9%	0.7	1.8
10%	0.6	1.6
20%	0.2	0.6
30%	0.1	0.3

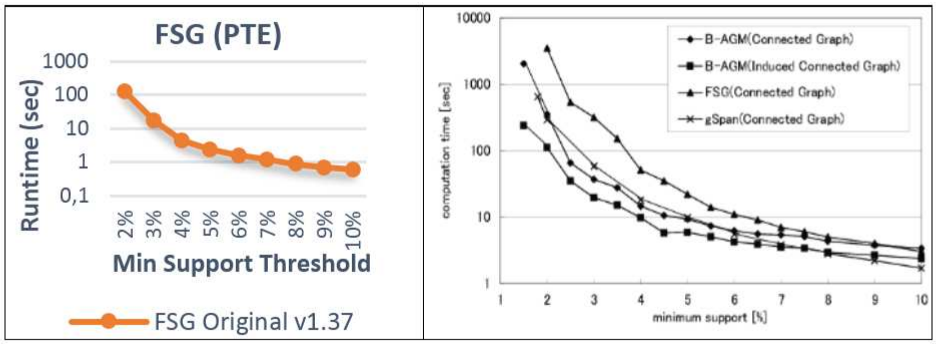


Fig. 15: Comparison of our Runtime (Left) with the Literature [38] (Right) - FSG Original - PTE

The number of frequent subgraphs is reported only in a graphical form in the literature [74]. Therefore, it was not possible to make a precise conclusion.

Our experiment result show that FFSM ParMol was considerably faster[29] than the result reported in [22] for the PTE dataset with the same number of duplicates. We found competitive runtime with the result reported in [74] with approximately the same number of frequent subgraphs (graphical estimation). However, since no information about the system specification was provided in [74], this makes the conclusion of runtime closeness useless.

No information was reported about FFSM ParMol memory consumption in the literature.

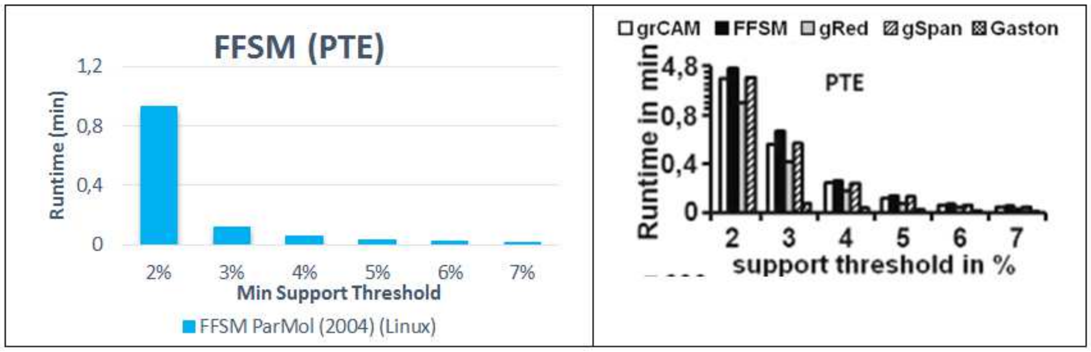


Fig. 16: Comparison of our Runtime (Left) with the Literature [22] (Right) - FFSM ParMol - PTE

6.3.6 MoFa Comparison We were unable to compare our experiment results of MoFa ParMol with the ones found in the literature. The reasons are: (i) unavailability of the dataset reported in [83], (ii) lack of sufficient details about the experiment (no information about machine characteristics was provided in [74]), and (iii) the ambiguity about the used implementation (MoFa or MoSS) [22].

[29] The processing power of our resource is better than the one in [22] (Intel Core 2 Duo 2.2 GHz processor)

Also, we were not able to compare our results regarding MoFa Original with state of the art mainly, because of the unavailability of HIV-CM dataset and the lack of efficiency results[30] in state of the art [12].

6.4 An Inter-Algorithms Performance Study

In this section, we compare the performance between different algorithms implementations. It is important to note that some algorithms were tested with the (H) strategy and some with the (L) strategy (see Table 11). We considered the used strategy in our comparative study. In this report, we report part of the result (*e.g.,* for one dataset) found through our experiments. Additionnally, we present a summarized comparison between some competitive implementations for all datasets and support threshold values. It is worth noting that all conclusions in this report are based on all experimental results[31] and not only about the some results shown in this report.

6.4.1 Number of Frequent Subgraphs The Gaston ParMol generated a number of frequent subgraphs which is different from the other implementations, for the low support threshold values (*e.g.,* PTE dataset, see Table 33 and for all datasets, see Table 34). GSpan (Keren Zhou, 2015) produced a number of frequent subgraphs considerably different from gSpan versions (as mentioned in section 6.2.1). It was also different from Gaston Original for low and medium support threshold values (see Table 35).

Table 33: Number of Frequent Subgraphs by FSM algorithms - (H strategy) - PTE

Min Sup	**SP (H)**	**GP (H)**	**GO (H)**	**F**	**FF (H)**	**D (H)**	**MFP (H)**	**MSP (H)**
2%	136981	136513	136949	136949	136981	136949	136981	-
4%	5955	5951	5935	5935	5955	5935	5955	-
5%	3627	3625	3608	3608	3627	3608	3627	-
7%	1786	1786	1770	1770	1786	1770	1786	654
9%	993	993	977	977	993	977	993	464
10%	860	860	844	844	860	844	860	390
20%	199	199	190	190	199	190	199	120
25%	126	126	117	117	126	117	126	76
40%	62	62	58	58	62	58	62	36
50%	37	37	34	58	37	34	37	26

Three ParMol implementations (gSpan, FFSM, MoFa) produced the same number of frequent subgraphs for all tested datasets (*e.g.,* PTE, see Table 33). Since the number of subgraphs produced by gSpan ParMol and gSpan Original v.6 were almost the same (1 or 2 more or less, see Table 36), we conclude that ParMol (gSpan, FFSM, MoFa) and gSpan Original v.6 produce almost the same number of frequent subgraphs (with a few exceptions).

Typically, the Gaston Original versions and FSG produced the same number of frequent subgraphs, with some exceptions. For example, for DD dataset under 20% MST there is a difference between

[30] Experiments were mainly focusing on the quality of results [12]
[31] For all results, see https://liris.cnrs.fr/rihab.ayed/DFSM.pdf

Table 34: Gaston ParMol vs gSpan ParMol: Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
5% - 7%	GP $<$ SP	151 - 42	2%	GP $>$ SP	1	2% - 7% \{3%}	GP $>$ SP	1435 - 1
8% - 90%	GP $=$ SP	-	3% - 90%	GP $=$ SP	-	3%	GP $<$ SP	365
PTE			**CAN2DA99**			8% - 90%	GP $=$ SP	-
1.5% - 5%	GP $<$ SP	1575 - 2	2% - 3%	GP $>$ SP	2 - 1	**Large Datasets**		
6% - 90%	GP $=$ SP	-	4% - 90%	GP $=$ SP	-	**NCI250**		
Dense Datasets			**AIDS**			50% - 90%	GP $=$ SP	-
DD			2%	GP $<$ SP	15	**DS3M**		
4% - 90%	GP $=$ SP	-	3% - 90%	GP $=$ SP	-	50% - 90%	GP $=$ SP	-
PS			**NCI330**					
80% - 90%	GP $=$ SP	-	4% - 7%	GP $>$ SP	84 - 1			
			8% - 90%	GP $=$ SP	-			

Gaston Original v1.1, v1.1 RE and FSG Original. Also, for AIDS dataset with 2% MST and NCI330 dataset with 7% MST, we observed a difference between Gaston Original v1.1 and FSG (see Table 37). Noticeably, Gaston Original and FSG Original compute differently the frequency of subgraphs. For example, for AIDS with 2% MST, the two implementations generated 27 frequent subgraphs - out of 17 694 - that are the same but with different frequency values (*e.g.,* one frequent subgraph out of 27 has a frequency by GO equal to 13558 and by F equal to 13553).

DMTL produced significantly a fewer number of frequent subgraphs than the others for the NCI330 and NCI145 datasets. For the other datasets, it produced the same number as Gaston Original versions (see Table 38).

The number of subgraphs produced by Gaston Original group is generally different from gSpan Original, MoFa Original (b) and the ParMol (gSpan, FFSM, MoFa). Table 39 shows this difference.

This difference could be partially justified for some support threshold values due to the fact that Gaston Original does not include frequent subgraphs with one vertex, unlike gSpan Original (*e.g.,* [3%, 50%] for PTE). However, other differences (*e.g.,* 1.5% for PTE) cannot be rationalized by the same fact. In fact, for some cases, Gaston Original produced more frequent subgraphs than gSpan

Table 35: gSpan (Keren Zhou, 2015) vs Gaston Original : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**
Small Datasets		
HIV-CA		
5% - 20%	SK $<$ GO	181687 - 17
30% - 60%	SK $=$ GO	-
PTE		
1.5% - 5%	SK $<$ GO	22279 - 1
6% - 50%	SK $=$ GO	-

Table 36: Number of Frequent Subgraphs by FSM algorithms - (L strategy) - PTE

Min Sup	**SP (L)**	**SO**	**SK**	**GO (L)**	**F**	**D (L)**
1.5%	721249	721196	698934	721213	721213	-
3%	22786	22785	22200	22758	22758	22758
5%	3627	3627	3607	3608	3608	3608
6%	2343	2343	2326	2326	2326	2326
8%	1339	1339	1323	1323	1323	1323
10%	860	860	844	844	844	844
15%	437	437	424	424	424	424
20%	199	199	190	190	190	190
25%	126	126	117	117	117	117
30%	75	75	68	68	68	68
40%	62	62	58	58	58	58
50%	37	37	34	34	58	34

Table 37: FSG Original vs Gaston Original/DMTL : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Dense Datasets**			**Large Datasets**		
HIV-CA, PTE			**DD**			**NCI250**		
F = GO			7% - 9%	F < GO	24 - 8	2%	F < GO	1
Medium Datasets			30% - 90%	F = GO	-	3% - 90%	F = GO	-
AID2DA99, CAN2DA99, NCI330, NCI145			**PS**			**DS3M**		
F = GO			80% - 90%	F < GO	1733033 - 7415	1% - 2%	F < GO	16 - 1
AIDS						4% - 90%	F = GO	-
1.5% - 2%	F > GO	1 - 1						
3% - 90%	F = GO	-						

Table 38: DMTL Original vs Gaston Original/FSG Original: Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
5% - 80%	D = GO	-	40% - 90%	D = GO	-	2% - 80%	D < GO	449691 - 4
PTE			**CAN2DA99**			90%	D = GO	-
1.5% - 90%	D = GO	-	20% - 90%	D = GO	-	**NCI330**		
Dense Datasets			**AIDS**			4% - 60%	D < GO	49081 - 1
DD			80% - 90%	D = GO	-	70% - 90%	D = GO	-
3% - 30%	D < GO	274 - 1	**Dense Datasets**					
40% - 90%	D= GO	-	**PS**					
			70% - 90%	D = F	-			

Original. It is also worth noting that the two implementations can compute differently the frequency of subgraphs. For example, for AIDS with 10% MST, they produced 16 frequent subgraphs - out of 510 - that have different frequency values (*e.g.*, 21 759 and 21 761 are the frequency for one graph out of the 16 by SO and GO, respectively).

Table 39: Gaston Original vs gSpan Original: Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Dense Datasets**		
HIV-CA			**DD**		
4% - 80%	SO > GO	8 - 3	2% - 5%	SO < GO	1434 - 21
PTE			6% - 90%	SO > GO	4 - 18
1.5% - 2%	SO < GO	17 - 15	2%	SO < GR	75
3% - 90%	SO > GO	27 - 1	3% - 90%	SO > GR	133 - 18
Medium Datasets			**PS**		
AID2DA99/CAN2DA99/AIDS/NCI145[32]			60%	SO < GO	25814977
2% - 90%	SO > GO	9 - 3	70%	SO > GO	176572
NCI330			80%	SO < GO	7399
3% - 90%	SO > GO	8 - 1	90%	SO > GO	8
Large Datasets			60%	SO < GR	25797964
NCI250/DS3/DS3M			70%	SO > GR	176572
2% - 90%	SO > GO	8 - 1	80%	SO < GR	7354
			90%	SO > GR	8

The MoSS ParMol produced a number of frequent subgraphs which is considerably different from all implementations for PTE (see Table 36), PS and HIV-CA datasets (see Table 40).

Table 40: MoSS ParMol vs MoFa ParMol : Number of Frequent Subgraphs Comparison

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI330**		
8% - 90%	MSP < MFP	23525 - 1	20% - 90%	MSP = MFP	-	20% - 90%	MSP = MFP	-
PTE			**CAN2DA99**			**Large Datasets**		
7% - 50%	MSP < MFP	1132 - 11	20% - 80%	MSP = MFP	-	**NCI250**		
70% - 90%	MSP = MFP	-	**AIDS**			70% - 90%	MSP = MFP	-
Dense Datasets			10% - 90%	MSP = MFP	-	**DS3M**		
DD			**NCI145**			80% - 90%	MSP = MFP	-
10% - 90%	MSP = MFP	-	20% - 90%	MSP = MFP	-			
PS								
90%	MSP < MFP	8						

MoFa Original with case b (MOb) produced the same number of frequent subgraphs as MoFa ParMol for the 3 SDF used datasets (*e.g.,* AID2DA99, see Table 41). However, MoFa Original with case a (MOa) produced significantly different number of frequent subgraphs. This is due to the edge relabeling strategy of chemical aromatic bonds [12]. According to our results, the 13 FSM implementations can

Table 41: Number of Frequent Subgraphs by MoFa Implementations - (H strategy) - AID2DA99

Min Sup	**MFP**	**MSP**	**MOa**	**MOb**
2%	25205	-	9741	25205
3%	11531	-	4395	11531
4%	6670	-	2566	6670
5%	4442	-	1763	4442
6%	3162	-	1224	3162
8%	1869	-	695	1869
9%	1484	-	590	1484
10%	1185	-	484	1185
20%	326	326	146	326
30%	133	133	75	133
40%	71	71	37	71
50%	45	45	33	45
70%	19	19	11	19
90%	3	3	2	3

be classified according to their similarity in the number of frequent subgraphs (see Figure 17). This classification is critical for unbiased comparison analysis of runtime and memory consumption.

We found in state of the art that gSpan, Gaston, FFSM and FSG original implementations produced the same number of frequent subgraphs (reported in [47,64,66]). Additionally, ParMol implementations (gSpan, Gaston, FFSM, MoFa) produced the same number of frequent subgraphs (reported in [22–24]). However, in [8], Gaston, FSG and gSpan original versions generate different[33] number of frequent subgraphs. In our experiments, we found different number of subgraphs (see Figure 17).

6.4.2 Runtime Our experiment shows that DMTL Original was significantly slower than all the other implementation for the same number of frequent subgraphs. Among all, for all the used datasets (*e.g.,* PTE, see Figure 18), Gaston Original v1.1 performed the best regarding runtime and Gaston Original RE the second.

GSpan Original may require a significant runtime for parsing a dataset (*e.g.,* for DS3 dataset, it consumed 28 seconds), while Gaston is faster in parsing (*e.g.,* for DS3 dataset, less than 0.2 seconds). Furthermore, gSpan is slower than Gaston for extracting frequent subgraphs.

We observed a competitive performance between gSpan versions, FFSM ParMol and FSG Original in terms of runtime (see Figures 19, 20). Thus, these three FSM algorithms are investigated futher. We conducted the following comparison: (i) FSG Original with gSpan versions and (ii) FFSM ParMol with gSpan versions.

[33] Authors tried to explain this difference for their tests [9] (French paper)

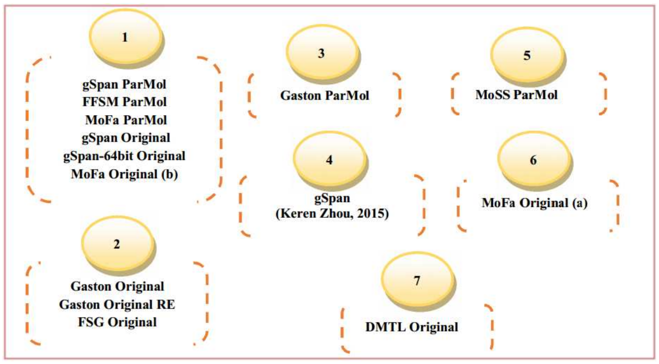


Fig. 17: Classification of FSM Implementation by the Number of Frequent Subgraphs

We compared FSG Original with the fastest versions of gSpan (gSpan ParMol or gSpan Original). Table 42 displays the results for all the datasets.

The FSG Original is faster than gSpan versions for low support threshold and medium or large datasets. For small datasets, it is slower than gSpan versions for low support threshold and slightly faster or close to gSpan versions for high support threshold. For dense datasets, it is slower than gSpan versions. It is worth noting that the number of subgraphs produced by FSG was less than the result produced by gSpan versions (see Table 43).

Table 44 shows a runtime comparison between FFSM ParMol and gSpan ParMol. In Table 44, 'F' stands for Fluctuation. FFSM ParMol was slower than gSpan ParMol for medium (*e.g.*, 10%) and high support threshold values (*e.g.*, 50%) for medium sized datasets. For low support threshold, it could be slower or faster depending on the dataset (*e.g.*, AID2DA99, AIDS).

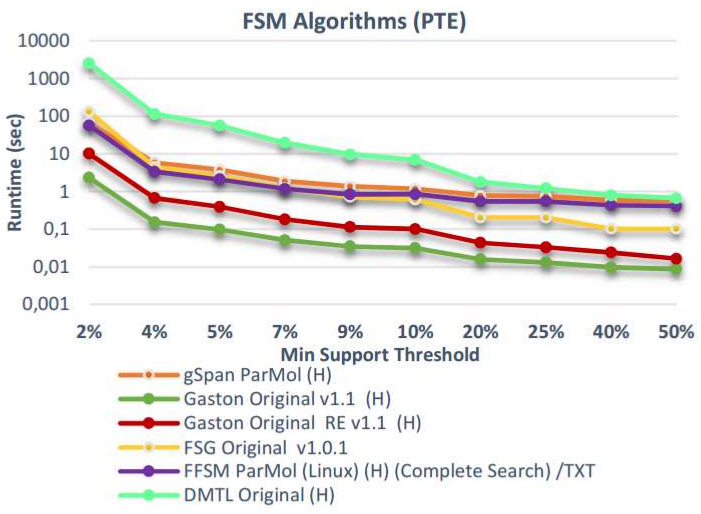


Fig. 18: FSM Algorithms Runtime (PTE) - (H strategy)

Table 42: FSG Original vs. gSpan versions (L/H strategy): Runtime Comparison

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5% - 15%	F > SP	256 - 0.4	1.5% - 5%	F < SP	510 - 24
20% - 80%	$F \approx SP$	0.1	6% - 90%	F > SP	14 - 7
PTE			**CAN2DA99**		
1.5% - 3%	F > SP	526 - 3.3	2% - 6%	F < SP	398 - 16
4% - 60%	F < SP	1 - 0.1	7% - 80%	F > SP	15 - 4.5
Large Datasets			**AIDS**		
NCI250			1.5% - 4%	F < SP	3239 - 12.5
2% - 20%	F < SO	4371 - 1	5% - 90%	F > SP	2.6 - 10
30% - 90%	F > SO	1.4 - 18	**NCI145**		
Dense Datasets			2% - 6%	F < SP	5263 - 8
DD			7% - 90%	F > SP	22 - 4
7% - 90%	F > SO	14234 - 6	**NCI330**		
PS			4% - 90%	F > SP	85 - 3
80% - 90%	F > SO	0.5 - 12.9			

Table 43: FSG Original vs. gSpan versions (L strategy): Number of Frequent Subgraphs

Support Interval	**Comp**	**Diff**	**Support Interval**	**Comp**	**Diff**
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5% - 90%	F < SP	9 - 2	1.5% - 90%	F < SP	8 - 1
PTE			**CAN2DA99**		
1.5% - 90%	F < SP	36 - 1	2% - 80%	F < SP	8 - 3
Large Datasets			**AIDS**		
NCI250			1.5% - 90%	F < SP	9 - 2
2% - 90%	F < SO	8 - 1	**NCI330**		
Dense Datasets			4% - 90%	F < SP	6 - 1
DD			**NCI145**		
7% - 90%	F < SO	20 - 18	2% - 90%	F < SP	8 - 1
PS					
80% - 90%	F < SO	16 - 13			

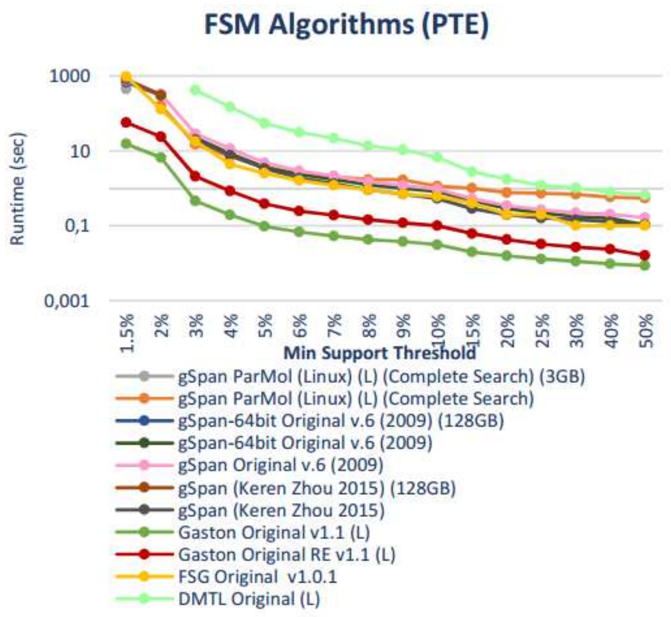


Fig. 19: FSM Algorithms Runtime (PTE) - (L strategy)

FFSM ParMol was faster than gSpan ParMol for dense datasets (see Table 44). However, for large datasets FFSM ParMol was slower. For small datasets, it was slightly faster or almost equal to gSpan ParMol except for very low support threshold values where it could be slower (*e.g.,* 5% for HIV-CA, Table 44).

Our results show that MoFa ParMol was the slowest among gSpan, FFSM and Gaston ParMol, for all the tested datasets (see Figure 20). Also, we found that MoSS ParMol was the slowest among ParMol implementations (see Figure 20). The number of subgraphs produced by MoSS ParMol can be considerably fewer than all the other implementations (*e.g.,* about the half, see Table 39). MoFa Original (b) was the fastest implementation amongst all ParMol implementations for the two medium datasets AID2DA99 and CAN2DA99. Also, we observed that it had close runtime to the one reported for Gaston Original v1.1 with low support threshold values (see Figure 20). However, for the large dataset NCI250, it was slower than ParMol implementations (gSpan, Gaston, MoFa) with the high reached support values (90%).

6.4.3 Memory Consumption For low support threshold values, gSpan (Zhou, 2015) consumed the highest amount of memory among all implementations (see Figure 21). For any support threshold value, the largest memory was consumed by DMTL implementation (see Figures 21 and 22).

For further comparison, we considered the implementations that were found to be competitive (see Figure 21). This is the case for gSpan ParMol, Gaston Original versions and FFSM ParMol. We ignored Gaston ParMol because of substantial difference that was observed regarding frequent subgraphs (see Figure 17).

Table 44: gSpan versions vs. FFSM ParMol (L strategy): Runtime Comparison

Support Interval	**Comp**	**Diff (sec)**	**Support Interval**	**Comp**	**Diff (sec)**
Small Datasets			**Medium Datasets**		
HIV-CA			**AID2DA99**		
5%	FF > SP	756	2%	FF (F) SP	-
6% - 50%	FF < SP	13.7 - 0.06	3% - 8%	FF < SP	23 - 3
60% - 80%	FF > SP	0.05 - 0.02	9% - 90%	FF > SP	33 - 2
PTE			**CAN2DA99**		
2% - 40%	FF < SP	10 - 0.14	2% - 80%	FF > SP	166 - 3.5
50%	$FF \approx SP$	-	**AIDS**		
60% - 90%	FF < SP	0.06 - 0.1	2% - 70%	FF > SP	1238 - 5
Large Datasets			80% - 90%	FF < SP	0.1 - 0.05
NCI250			**NCI145**		
30% - 90%	FF > SP	1882 - 87	2% - 9%	FF < SP	2357 - 11
DS3M			10% - 90%	FF > SP	2 - 0.5
40% - 90%	FF > SP	2463 - 214	**NCI330**		
Dense Datasets			4% - 6%	FF < SP	403 - 5
DD			7% - 90%	FF > SP	3.9 - 0.02
4% - 90%	FF < SP	4204 - 12			
PS					
80% - 90%	FF < SP	3 - 0.4			

Table 45 shows the comparison between gSpan ParMol and Gaston Original versions with respect to their memory consumption. For small datasets, gSpan ParMol consumed more memory for low support threshold and lesser for high support threshold than the two Gaston Original versions.

For medium size datasets, gSpan ParMol consumed more memory than Gaston Original RE. However, it required lesser memory than Gaston Original with low and medium support threshold. For some cases with low support threshold, it consumed more than Gaston Original (*e.g.,* 3% for NCI145 dataset, see Table 45).

Futhermore, for large datasets, gSpan ParMol consumed lesser memory than Gaston Original for low support threshold. However, it consumed more memory for high support threshold.

For dense datasets, gSpan ParMol consumed more memory than Gaston Original. Additionally, it consumed more memory than Gaston Original RE for low support threshold and lesser memory for high support threshold.

We compared memory consumption of FFSM ParMol and gSpan ParMol (see Table 46). In Table 46, 'F' stands for fluctuation[34] of the performance.

For small datasets, FFSM ParMol consumed lesser memory than gSpan ParMol for low support threshold values. However, its consumption was close to gSpan ParMol for high support threshold values.

For medium and dense datasets, FFSM ParMol consumed more memory than gSpan ParMol. Additionally, for large datasets, it consumed significantly more memory than gSpan ParMol.

The FSG Original and gSpan Original are provided as binary codes with no information about memory consumption. Therefore, we tried to deduce their respective limits regarding memory con-

[34] None of solutions performed consistently for more than two successive support threshold values

Table 45: gSpan ParMol vs. Gaston Original versions (L strategy): Memory consumption comparison

Support Interval	**Comp**	**Diff (MB)**	**Support Interval**	**Comp**	**Diff (MB)**	**Support Interval**	**Comp**	**Diff (MB)**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
5% - 10%	SP > GO	2031 - 17	1.5% - 90%	SP > GR	468 - 84	2% - 90%	SP > GR	1702 - 53
20% - 80%	SP < GO	3.2 - 2.8	1.5% - 15%	SP < GO	66 - 10	2% - 3%	SP > GO	1324 - 76
7% - 20%	SP > GR	105 - 0.7	20% - 90%	SP > GO	19 - 39	4% - 50%	SP < GO	84 - 10
25% - 80%	SP < GR	1.8 - 1.7	**CAN2DA99**			60% - 90%	SP > GO	15 - 6
PTE			2% - 80%	SP > GR	369 - 100	**NCI330**		
1.5% - 5%	SP > GO	1066 - 1.2	2% - 20%	SP < GO	121 - 1.9	4% - 90%	SP > GR	392 - 24
6% - 90%	SP < GO	2.5 - 1.7	40% - 80%	SP > GO	36 - 40	4% - 5%	SP > GO	212 - 15
1.5% - 5%	SP > GR	1114 - 2.8	**AIDS**			6% - 20%	SP < GO	0.7 - 7.7
6% - 90%	SP < GR	1 - 3.1	1.5% - 90%	SP > GR	480 - 116	30% - 90%	SP > GO	7.5 - 6.7
Dense Datasets			1.5% - 70%	SP < GO	429 - 68	**Large Datasets**		
DD			80% - 90%	SP > GO	51 - 82	**DS3M**		
4% - 10%	SP > GR	1020 - 78	**Dense Datasets**			2% - 10%	SP < GO	908 - 14
20% - 90%	SP < GR	24 - 55	**PS**			20% - 90%	SP > GO	196 - 419
4% - 90%	SP > GO	1348 - 38	80%	SP > GO	26	**NCI250**		
			90%	SP < GO	1.8	2% - 10%	SP < GO	885 - 44
			80%	SP > GR	25	20% - 90%	SP > GO	178 - 386
			90%	SP < GR	2.9			

Table 46: gSpan ParMol vs. FFSM ParMol (L/H strategy): Memory consumption comparison

Support Interval	**Comp**	**Diff (MB)**	**Support Interval**	**Comp**	**Diff (MB)**	**Support Interval**	**Comp**	**Diff (MB)**
Small Datasets			**Medium Datasets**			**Medium Datasets**		
HIV-CA			**AID2DA99**			**NCI145**		
5% - 10%	FF < SP	502 - 7	2% - 90%	FF > SP	512 - 44	2% - 90%	FF > SP	211 - 40
15% - 80%	FF (F) SP	1.9 - 0.4	**CAN2DA99**			**NCI330**		
PTE			2% - 80%	FF > SP	589 - 84	4% - 90%	FF > SP	123 - 18
2% - 3%	FF < SP	93 - 5	**AIDS**			**Dense Datasets**		
4% - 90%	FF (F) SP	1.8 - 0.1	2% - 90%	FF > SP	1101 - 141	**DD**		
Large Datasets			**Large Datasets**			4% - 80%	FF > SP	568 - 15
NCI250			**DS3M**			90%	FF < SP	6
30% - 90%	FF > SP	1675 - 1377	40% - 90%	FF > SP	1592 - 1674	**PS**		
						80%	FF < SP	12
						90%	$FF \approx SP$	-

sumption by testing the lowest support threshold values. FSG Original was not able to run with low support threshold for some datasets (*e.g.,* DD dataset, see Table 47). We conclude that FSG used more memory than gSpan Original, for low support threshold values.

Table 47: Minimal Support threshold value reached by FSM Algorithms- (L strategy)

Algorithm	**HIV-CA**	**DD**
SP	5%	4%
SO	4%	1%
F	5%	7%
GO	4%	1%

We could not conclude about the memory consumption limit of gSpan Original compared to Gaston Original. However, it is worth noting that for some datasets (*e.g.,* NCI330) and with low support thresholds, gSpan Original took a huge time without completing the mining (*e.g.,* 6 days for NCI330 with 2% MST). Gaston Original completed it in a more reasonable time and with a lower support (*e.g.,* 9 hours for NCI330 with 1% MST).

Summary

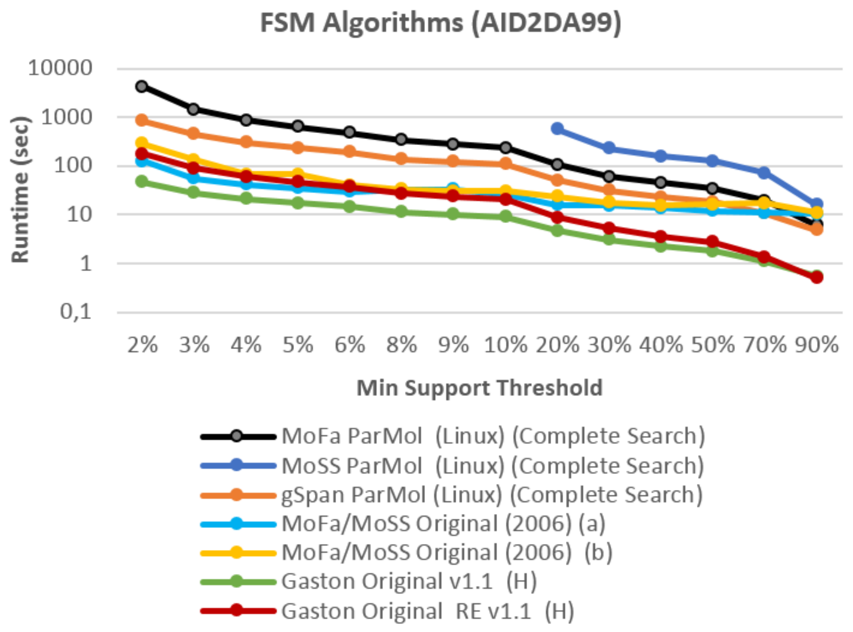


Fig. 20: FSM Algorithms Runtime (AID2DA99) - (H strategy)

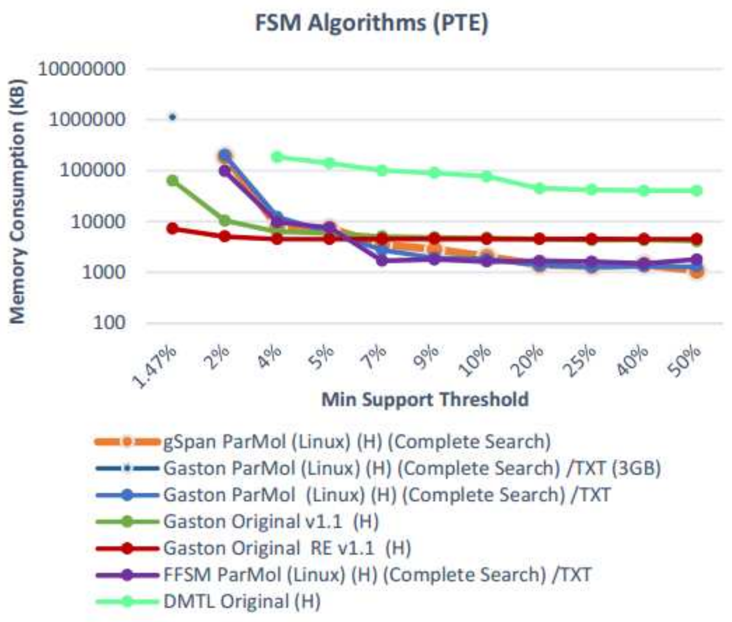


Fig. 21: FSM Algorithms Memory Consumption (PTE) - (H strategy)

– According to our analysis, the gSpan ParMol is more suitable than Gaston Original versions, for memory bound systems, in the following cases: (i) for large datasets and low support threshold, (ii) for small datasets and high support threshold values.

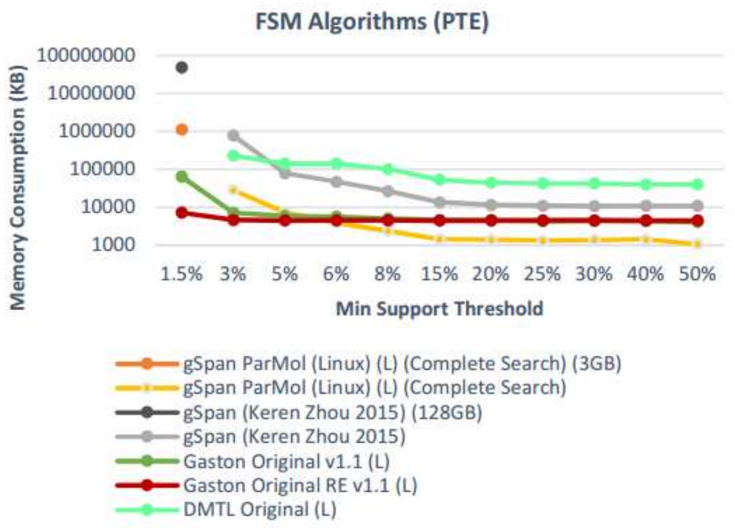


Fig. 22: FSM Algorithms Memory Consumption (PTE) - (L strategy

- Based on our study, we conclude that for memory bound systems, FFSM ParMol can be used instead of gSpan ParMol if the dataset is small and the support threshold values are low. However, it is better to use Gaston Original versions for this case.
- According to our analysis, for memory bound systems, gSpan Original is more suitable to use than FSG Original for low support threshold values.

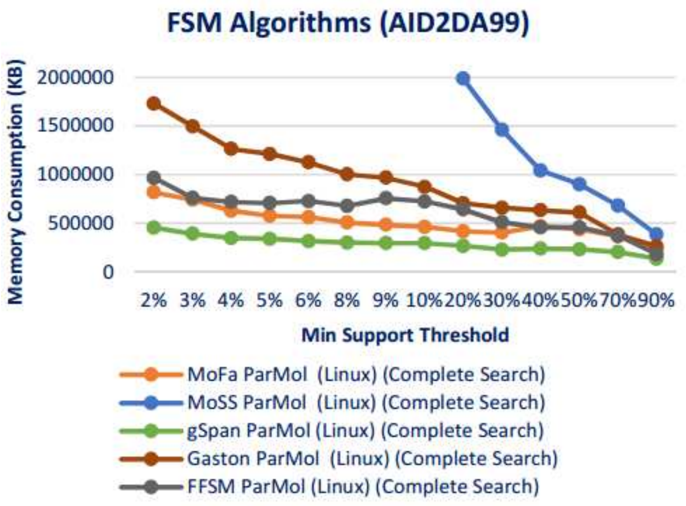


Fig. 23: FSM Algorithms Memory Consumption (AID2DA99) - (H strategy)

According to our results for MoFa/MoSS solutions, we observed that MoFa ParMol and FFSM ParMol consumed the same amount of memory (see Figure 23). MoSS ParMol consumed more memory with a number of subgraphs potentially lesser than all the other implementations (see Figure 23). Additionally, MoFa Original (b) consumed about twice (or one half) the memory[35] of MoFa ParMol for the medium (AID2DA99, CAN2DA99) and large (NCI250) datasets (see Table 48).

To sum up, MoFa ParMol was the slowest among Gaston, FFSM and gSpan ParMol implementations and it consumed an amount of memory close to FFSM ParMol.

6.4.4 Bottleneck experiments Table 50 shows the limits of the Gaston Original implementation which we experimented using our machine characteristics; with eleven datasets and very low support threshold values.

We emphasized on Gaston Original because it is the most efficient solution. The notations we used in the table are, S : reached support threshold, N : number of frequent subgraphs, M : Max size of frequent subgraphs (vertices), R: consumed RAM memory (MB), O : output file size (MB), U : non-reached support threshold, C : cause of failure. For all tested datasets, Gaston Original was able to run with 1% MST, except for HIV-CA and PS datasets. In fact, the implementation was not able to run under 4% MST for HIV-CA due to lack of disk space. For PS with 50% MST, it spent 8

[35] We estimated the memory consumption of MoFa Original by the JVM it required

Table 48: Memory Consumption (MB) of two MoFa implementations (AID2DA99) - (H strategy)

Min Sup	**MFP**	**MOb**
AID2DA99		
2%	817	]1400 - 1500]
5%	576	]1100 - 1200]
10%	462	]1000 - 1100]
50%	437	]700 - 800]
90%	230	]500 - 600]
NCI250		
80%	1862	> 3500
90%	1611	]2400 - 2500]

Table 49: Minimal Support threshold value reached by FSM Algorithms - (L) strategy - DS3 vs. DS3M

Implementation	**DS3**	**DS3M**
SO	1%	1%
SO64	5%	6%
F	1%	1%
GO	1%	1%

days running without completing the experiment, yet an output file of 3 GB was created. Similarly, in another experiment, it spent 4 days running without completing for the same support and with a maximum size of frequent subgraphs (28 vertices).

Table 50: Bottleneck Experiment of Complete Search FSM Algorithms (Gaston)

Dataset	**Success of Mining Limit**				
	S	N	M	R	O
PTE	1%	48732156	42	637.5	21900
HIV-CA	4%	6825303	48	15.4	3400
AID2DA99	1%	107693	20	623.7	19
CAN2DA99	1%	176292	21	586.2	33
AIDS	1%	335483	27	1038.9	9
NCI145	1%	235740772	44	470.05	103000
NCI330	1%	268761360	42	238.5	192000
DD	1%	159820929	15	66.9	20100
PS	60%	63641199	28	5.9	13500
NCI250	1%	70405	21	3033.4	0.00003
DS3	1%	83310	21	3429.5	15.7

Table 51: Execution of Implementations with Very Dense Datasets - PI

Algorithms versions	**P**	**SO**	**SO64**	**GO**	**GR**	**F**	**D**
Dataset Processing	-	-	-	-	+	-	+

Table 51 displays the scalability of FSM implementations with a very dense dataset (PI). Only Gaston Original RE and DMTL were able to process[36] the PI dataset without generating an error. We did not experiment DMTL any further for mining frequent subgraphs since it performs only a complete search, rather we experimented Gaston RE which is able to reduce the mining set.

Table 52 shows the limits of Gaston Original RE in mining PI dataset, where *MSF*, *Min Sup*, *R*, *RM*, *DM*, *NF* denote the maximum size of frequent subgraphs (vertices), the minimum support threshold, runtime, the used RAM memory, the used disk memory, and the number of frequent subgraphs, respectively.

Gaston RE was able to find frequent subgraphs of maximum size 5 with 100% MST. However, it was not able to complete the mining with the same support for maximum size 10. The same applies for the cases of 70% MST and max size 3 or 50% MST and max size 2. These findings approve the results of [7, 73] about the limits of complete search algorithms with dense datasets.

Table 52: Mining Performance of Gaston RE with Very Dense Datasets - PI (Incomplete Search)

MSF	**Min Sup**	**R (sec)**	**RM (GB)**	**DM (MB)**	**NF**
2	70% - 100%	1.017	2.21	0.0079	256
	50%	-	*Segmentation fault*	0.0041	-
3	100%	1.578	2.21	0.0852	1928
	70%	-	*Aborted*	0.0011	-
5	100%	873.745	2.84	12	1578086
10	100%	-	*Killed*	3300	-

6.5 Discussion

According to our observations, the sources of results ambiguities in state of the art (see Section 4.2) are as follows: different style of implementating the FSM algorithm (*e.g.,* ParMol or Original), the used dataset (small, large, dense), and the support threshold values (*e.g.,* [2%, 50%], [10%, 90%]). For example, Gaston ParMol was the highest memory consumer among gSpan ParMol and FFSM ParMol. However, Gaston Original consumed lesser memory than gSpan ParMol and FFSM ParMol.

The experimental study we conducted allowed to alleviate some of the ambiguities and specify some cases of FSM implementations performance. According to our results, eight implementations

[36] No abortion in the beginning of the execution

Table 53: FSM Algorithms with performance drawbacks

Algorithm version	**Performance Characteristics**
DMTL	(-) Huge time and memory consumption (+) Able to process very dense datasets
MoFa ParMol	(-) Huge time and memory consumption (+) Extra mining options for biochemical data
MoSS ParMol	(-) Very small Number of FS compared to others
	(-) Huge consumption of Memory
MoFa Original	(+) Comparable runtime with Gaston Original v1.1 at low support values and medium datasets
	(-) Bad memory consumption
	(+) Extra mining options for biochemical data
Gaston ParMol	(-) Number of FS different
	(-) Important consumption of memory
gSpan-64bit Original v.6 (2009)	(-) Slower than Gaston Original for high support values
	(-) For Low support values, it consumes a lot of memory compared to gSpan Original, gSpan ParMol and Gaston Original
gSpan (Zhou, 2015)	(-) Number of FS different
	(-) Huge consumption of memory
	(-) Dedicated for small datasets
FFSM ParMol	(-) More memory consumption than gSpan ParMol and Gaston Original.
	(-) It is always slower than Gaston Original versions.

among thirteen (see Table 53) are not adequately efficient due to : (i) their high memory and/or time consumption, (ii) a number of frequent subgraphs different from the other implementations, (iii) their inability to handle relatively large datasets or run for low support thresholds.

We selected five implementations out of thirteen as efficient, including Gaston Original, gSpan ParMol, gSpan Original, FSG Original and Gaston Original RE. The first four implementations (see Table 54) were selected based on the following criteria : (i) they consumed the least amount of memory, among all the thirteen FSM implementations, (ii) they are relatively fast (Gaston Original is the fastest), (iii) they were able to complete the mining with relatively large datasets or for low support threshold values. Gaston Original RE was chosen due its ability to process very dense datasets unlike the four others and its good performance with medium datasets.

Then we filter the four selected implementations (Gaston Original, gSpan ParMol, gSpan Original and FSG Original) to two usable implementations (Gaston Original and gSpan Original) for two general cases : (i) applications that need to save memory, and (ii) applications where runtime is critical. Both Gaston Original and gSpan Original are suitable for the former and Gaston Original is suitable for the latter.

During our experiments, we realized that the size of the dataset and the minimum support influenced the performance of the tested FSM solutions. Therefore, we changed some other variables of experiment environment in order to observe their impact on the performance. In the next section, we discuss our results.

Table 54: FSM Algorithms with performance advantages

Algorithm version	**Performance Characteristics**
Gaston Original	(+) Second/Third in memory consumption
	(+) The fastest
	(+) Able to run with relatively large datasets or very low support values
gSpan ParMol	(+) Third/Fourth best memory consumption for medium, large datasets or for low support values
	(+) Third fastest for small or medium datasets and not low support threshold values
	(-) Unable to run for very low support values reached by gSpan Original
gSpan Original	(+) Able to run for some very low support threshold values or for relatively large datasets
	(+) First/Second best memory consumption for low support threshold
	(+) Third fastest for dense datasets or for high support values and large datasets
	(-) Unable to finish in a reasonable time for some very low support threshold values unlike Gaston Original
FSG Original	(+) Able to run for low support threshold or relatively large datasets
	(-) Requiring more memory than gSpan Original and gSpan ParMol for low support values
	(+) Third fastest for medium, large datasets and for low support values
Gaston Original RE	(+) Second in Runtime
	(+) First/Second in memory consumption with medium datasets
	(-) Not to be used with Large datasets or very low support threshold values
	(+) Able to process very dense datasets (*e.g.,* PI)

6.5.1 Impacts of the Environment variations on the results

The variation impact of the experimental environment on the performance of FSM implementations are discussed in the following. The environment variables include the dataset size, the operating system, the used IDE, the implementations arguments, the input data format and the labelling strategy of data.

Dataset variation We studied the impact of tested datasets[37] on the *runtime*, *memory* and the *number of frequent subgraphs.* We considered two variables for datasets : *size* and *density.* These parameters are the same as defined in Section 6.1.1. We discuss our results of experiments with Gaston Original, gSpan ParMol and gSpan Original. *Runtime:* According to our observation, experiments with small datasets required typically the lowest runtime among all datasets (*e.g.,* PTE, see Figure 24). However, the experiments with low support threshold over the small datasets (*e.g.,* HIV-CA) required more or equal time than the other datasets (*e.g.,* DS3) to complete the execution.

Medium datasets had similar runtime performance (*e.g.,* NCI330 and CAN2DA99, see Figure 24). However, for experiments with low support threshold, two medium datasets (NCI330 and NCI145) were considerably slower than the other medium datasets (*e.g.,* AID2DA99).

Typically, large datasets (*e.g.,* DS3) required more time than small and medium datasets (*e.g.,* CAN2DA99), except with very low support threshold values.

[37] The used datasets were defined in Section 6.1.1

Dense datasets (*e.g.,* DD) required similar amount of time compared to medium datasets (*e.g.,* CAN2DA99) with high support thresholds (*e.g.,* 30%, see Figure 24). However, with low support threshold (*e.g.,* 3%), they required more time than large datasets (*e.g.* DS3).

Memory Consumption: Our experiments with Gaston Original and *small datasets* consumed the lowest amount of memory (*e.g.,* PTE, see Figure 25). They are followed by dense (*e.g.,* DD), medium (*e.g.* NCI330) and large (*e.g.* DS3) datasets experiments, respectively (see Figure 25).

Our experiments with gSpan ParMol and *dense datasets* consumed lesser memory than medium datasets with high support values. However, with low support values, they consumed more memory than medium datasets.

The consumption of memory for all datasets was typically linear (see Figure 25). However, we observed some exceptions for small datasets and low support threshold values where there was an important increase of memory (*e.g.,* PTE with Gaston or gSpan ParMol, HIV-CA with gSpan ParMol). The memory consumption for this case (*e.g.,* 1% PTE, see Figure 25) was similar or more than the memory consumed by medium and large datasets.

Number of Frequent Subgraphs: Different sizes of datasets produced typically close number of frequent subgraphs (*e.g.,* DS3 and CAN2DA99, see Figure 26). However, our experiment with the *dense* dataset DD produced a considerably larger number than the other datasets (*e.g.,* DS3). Additionally, with low support values (lower than 6% MST), experiments with small (HIV-CA, PTE) and some medium datasets (NCI330, NCI145) produced signifcantly a larger number than the one produced by all the other datasets (see Figure 26).

OS and IDE variation We used the ParMol framework to test this effect. We conducted experiments using two IDEs : Eclipse with two versions (Mars 4.5.1, Neon 4.6) and Netbeans 8.2. We also experimented ParMol on a terminal. We used the JDK version 1.8_77. The same machine was used as previous experiments (see Table 12, Section 6.1.2). In figures 27 and 28, *Eclipse N*, *Eclipse M*, *Netbeans* and *Terminal +* denote the use of IDE Eclipse Neon, Eclipse Mars, Netbeans and the Terminal, respectively. The results show that using the same OS (Windows or Linux) and different IDEs (*e.g.,* Eclipse or Netbeans) did not affect the runtime (see Figure 27) or memory consumption performance (see Figure 28). However, changing the OS (Linux to Windows) did have an impact on the runtime performance. This is due to the use of the argument 'memoryStatistics' in ParMol that calculates the

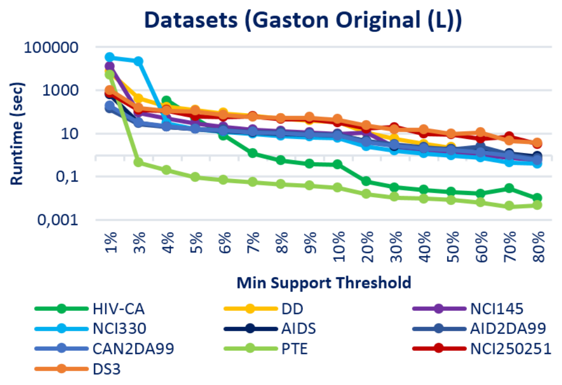


Fig. 24: Dataset Variation Effect on the Runtime - Gaston Original (L)

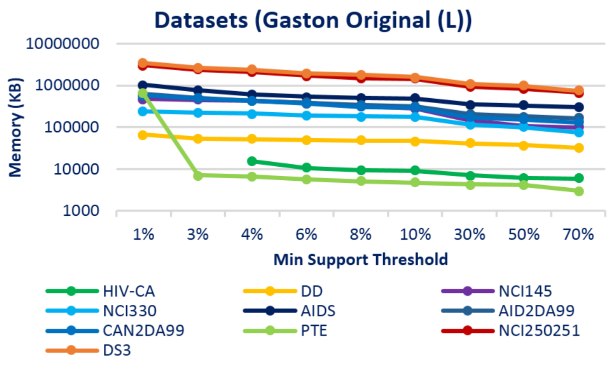


Fig. 25: Dataset Variation Effect on the Memory - Gaston Original (L)

memory consumption. With this argument, the Windows OS had worse runtime performance than Linux (see Figure 27) and the same memory consumption (see Figure 28). In case this argument (*memoryStatistics*) is set to false, we found no impact of OS variation on the performance (see Figure 29, the Windows and Linux *Terminal* - results).

Argument variation We performed experiments with ParMol to study the impact of changing its arguments on performance. Among ParMol arguments, 'memoryStatistics' is the one that had an impact on runtime performance (see Figure 29). In Figure 29, *Terminal* + denotes for experiments performed on a terminal with the argument 'memoryStatistics' set to true and *Terminal* - denotes for experiments with the argument 'memoryStatistics' set to false.

File Format variation We performed experiments over datasets serialized in different formats including TXT and SDF. No noticeable change in runtime performance or memory consumption was observed.

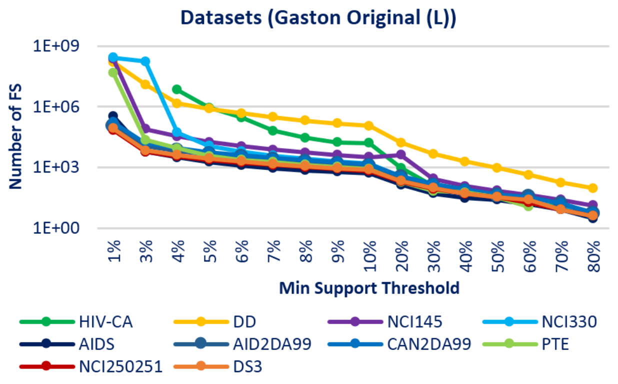


Fig. 26: Dataset Variation Effect on the Number of FS - Gaston Original (L)

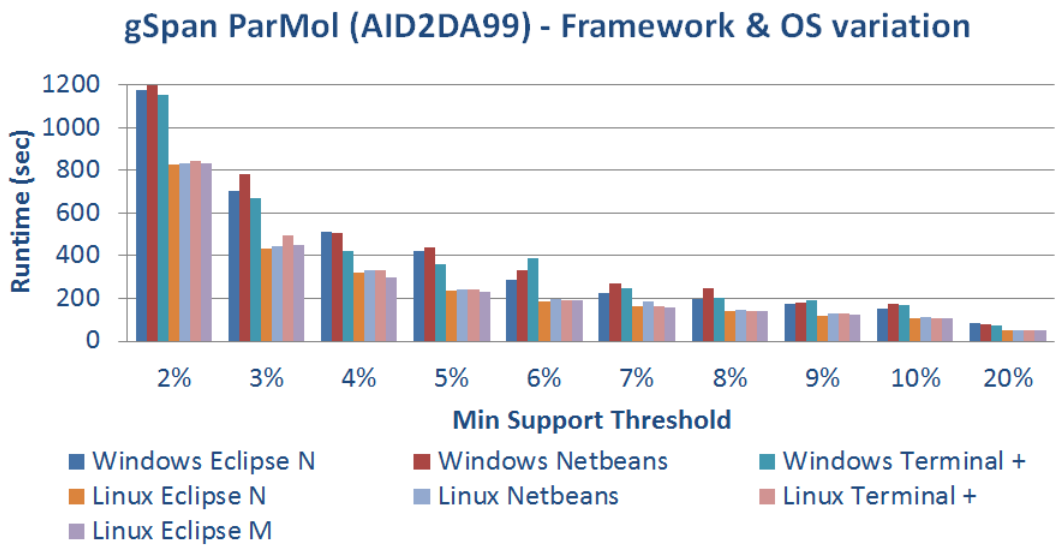


Fig. 27: gSpan ParMol Runtime performance by OS and IDE - AID2DA99 dataset

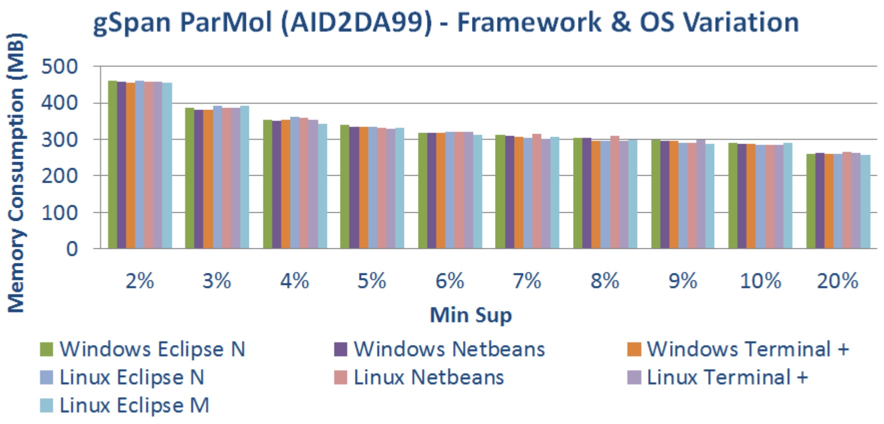


Fig. 28: gSpan ParMol Memory Consumption performance by OS and IDE - AID2DA99 dataset

Labeling strategy variation In our experiments, we modified the DS3 dataset[38] that contains vertices labeled with integer and string (*e.g.,* '1', '1u', '2f', '36'). The modification resulted in a dataset labeled with integer only which we named DS3M. Only FSG Original is able to parse integer and string labeled TXT datasets. Hence, this experiment was performed with FSG Original.

The labeling strategy did not affect the performance of FSG Original regarding the number of frequent subgraphs (see Table 55) and the runtime (see Table 56).

[38] Please refer to Section 6.1.1 for DS3 characteristics

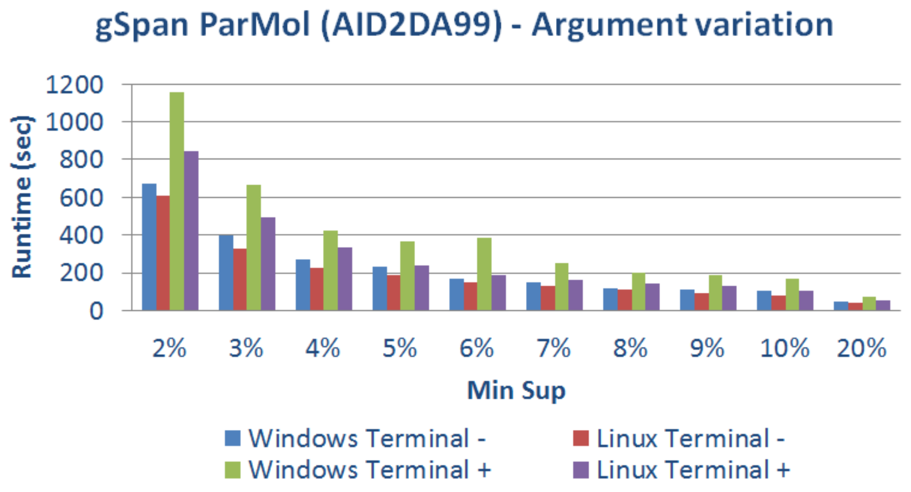


Fig. 29: gSpan ParMol Runtime performance by argument variation and OS - AID2DA99 dataset

Table 55: Number of Frequent Subgraphs for FSG Original - DS3 vs. DS3M

Min	**F**	
Sup	**DS3**	**DS3M**
1%	80722	80722
3%	6534	6534
5%	2651	2651
7%	1414	1414
10%	725	725
30%	93	93
50%	35	35
80%	4	4

6.5.2 Reducing the set of Frequent Subgraphs and other options of FSM Implementations It is possible to perform incomplete search (see Section 3.3.2) using the complete search FSM available implementations (see Table 57). This optional setting is important because the search space of complete FSM mining is rich but it is exponential [3, 71]. There is a need to reduce the set by eliminating the redundancy of subgraph isomorphism [78]. The proposed settings include the following : (i) specifying the maximum and minimum size of frequent subgraphs to return (gSpan ParMol, Gaston Original), (ii) specifying the minimum and maximum support threshold (gSpan ParMol), (iii) returning only supergraphs (*i.e.,* closed or maximal Subgraphs) (gSpan ParMol, FSG Original) (see Table 57).

Also, other options are available such as using multi-threads to perform the mining faster (see Table 57).

Table 56: SG Original Runtime - DS3 vs. DS3M

Min Sup	**F**	
	DS3	**DS3M**
1%	13657.7	13666.2
3%	1463.1	1459.4
5%	891.8	891.8
7%	678	678.6
10%	505.7	505.7
30%	201.633	201.6
50%	130.266	130.2
80%	70.8	70.4

7 Conclusion

This report presented a comprehensive study of complete search FSM implementations in centralized graph transaction databases. We studied all the algorithms found in literature and outlined their merits and demerits. Additionally, we presented the results of an experimental study with the selected and available FSM implementations. We investigated the difference between the algorithms in a quantita-

Table 57: Optional settings for FSM Implementations

FSM solution	**P**	**SO, SO64**	**GO**	**GR**	**F**	**D**
Incomplete Search Options						
Min and Max Support threshold	x					
Min of Frequent Subgraphs Size	x	x			x	
Max of Frequent Subgraphs Size	x		x	x	x	
Closed Frequent Subgraphs	x					
Maximal Frequent Subgraphs					x	
Trees	x		x	x		
Paths	x		x	x		
Maximum number of subgraph isomorphisms		x				
Input Options						
String labeled datasets	x					x
Output Options						
Dataset statistics	x				x	
TXT format	x	x	x	x	x	
DFS code Format		x		x		x
TID List	x	x		x	x	
PC List					x	
Other Options						
Multi-threading	x	x				
Significant/Discriminative Patterns		x				
Weighted graphs		x				

tive manner (*e.g.,* gSpan takes X time more than FFSM for a support threshold interval Y), instead of an abstract way (*e.g.,* gSpan is slower than FFSM in general). Our study unearthed the differences and similarities between different implementations of one single algorithm and between different implementations of algorithms. Also, we experimented the FSM solutions regarding different datasets and different thresholds. Such a comparison will provide the readers a detailed and comprehensive view about these implementations. Hence, it could assist them to make decision regarding the selection of an implementation for a specific context. This work could be seen as an updated observation of existing FSM implementations.

Several future work are lined up including analysis and explanations that should be linked to this work about the difference between results (number of frequent subgraphs, runtime and memory). We plan to conduct a rigorous experiment with FSM algorithms over large and/or dense datasets. Also, this study has been performed only on the literature datasets, we will conduct another study with generic datasets.

8 Acknowledgments

This work has been elaborated as a part of the CAIR[39] project. Special thanks are addressed to FSM authors especially to Xifeng Yan, Thorsten Meinl, Andrés Gago-Alonso, Christian Borgelt, Mohammad Al Hasan and Sabeur Aridhi for sending us software, datasets, also for providing clarifications and for their availability.

References

1. Parmol. `http://en.verysource.com/parmol_1346_2006-08--100787.html`. [Online; accessed 2016-05-30].
2. N. Acosta-Mendoza, A. Gago-Alonso, and J. E. Medina-Pagola. Frequent approximate subgraphs as features for graph-based image classification. *Knowledge-Based Systems*, 27:381–392, Mar. 2012.
3. C. C. Aggarwal, H. Wang, and others. *Managing and mining graph data*, volume 40. Springer, 2010.
4. M. Al Hasan, V. Chaoji, S. Salem, J. Besson, and M. J. Zaki. ORIGAMI: Mining Representative Orthogonal Graph Patterns. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 153–162, Oct. 2007.
5. M. Al Hasan, V. Chaoji, S. Salem, N. Parimi, and M. J. Zaki. DMTL: A Generic Data Mining Template Library.
6. M. Al Hasan and M. Zaki. Musk: Uniform Sampling of k Maximal Patterns. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, Proceedings, pages 650–661. Society for Industrial and Applied Mathematics, 2009.
7. M. Al Hasan and M. J. Zaki. Output Space Sampling for Graph Patterns. *Proc. VLDB Endow.*, 2(1):730–741, Aug. 2009.
8. S. Aridhi, L. d'Orazio, M. Maddouri, and E. Mephu Nguifo. Density-based data partitioning strategy to approximate large-scale subgraph mining. *Information Systems*, 48:213–223, Mar. 2015.
9. S. Aridhi, L. d'Orazio, M. Maddouri, and E. M. Nguifo. Un partitionnement basé sur la densité de graphe pour approcher la fouille distribuée de sous-graphes fréquents. *Technique et Science Informatiques*, 33(9-10):711–737, 2014.
10. H. Arimura, T. Uno, and S. Shimozono. Time and Space Efficient Discovery of Maximal Geometric Graphs. In V. Corruble, M. Takeda, and E. Suzuki, editors, *Discovery Science*, number 4755 in Lecture Notes in Computer Science, pages 42–55. Springer Berlin Heidelberg, Oct. 2007. DOI: 10.1007/978-3-540-75488-6_6.

[39] https://www.irit.fr/CAIR/fr/

11. C. Borgelt. Moss - molecular substructure miner. `http://www.borgelt.net/moss.html`. [Online; accessed 2016-05-30].
12. C. Borgelt and M. R. Berthold. Mining molecular fragments: finding relevant substructures of molecules. In *2002 IEEE International Conference on Data Mining, 2002. ICDM 2003. Proceedings*, pages 51–58, 2002.
13. C. Chen, C. X. Lin, X. Yan, and J. Han. On Effective Presentation of Graph Patterns: A Structural Representative Approach. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 299–308, New York, NY, USA, 2008. ACM.
14. M. Cohen and E. Gudes. Diagonally Subgraphs Pattern Mining. In *Proceedings of the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '04, pages 51–58, New York, NY, USA, 2004. ACM.
15. D. J. Cook and L. B. Holder. Substructure Discovery Using Minimum Description Length and Background Knowledge. *J. Artif. Int. Res.*, 1(1):231–255, 1994.
16. R. de Sousa Gomide, C. D. de Aguiar Ciferri, R. R. Ciferri, and M. T. P. Vieira. ADI-Minebio: A Graph Mining Algorithm for Biomedical Data. *Journal of Information and Data Management*, 2(3):433, Sept. 2011.
17. L. Dehaspe, H. Toivonen, and R. D. King. Finding Frequent Substructures in Chemical Compounds. pages 30–36. AAAI Press, 1998.
18. C. Desrosiers, P. Galinier, P. Hansen, and A. Hertz. Sygma: Reducing symmetry in graph mining. Technical report, Les Cahiers du GERAD, 2007.
19. B. Douar, M. Liquiere, C. Latiri, and Y. Slimani. LC-mine: a framework for frequent subgraph mining with local consistency techniques. *Knowledge and Information Systems*, 44(1):1–25, July 2014.
20. M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proc. VLDB Endow.*, 7(7):517–528, Mar. 2014.
21. H. Fei. Fast frequent subgraph mining (ffsm). `https://sourceforge.net/projects/ffsm/`. [Online; accessed 2016-05-30].
22. A. Gago-Alonso and J. A. Carrasco-Ochoa. Full Duplicate Candidate Pruning for Frequent Connected Subgraph Mining. *Integrated Computer-Aided Engineering*, 17(3):211–225, 2010.
23. A. Gago-Alonso, J. E. M. Pagola, J. A. Carrasco-Ochoa, and J. F. Martínez-Trinidad. Mining Frequent Connected Subgraphs Reducing the Number of Candidates. In W. Daelemans, B. Goethals, and K. Morik, editors, *Machine Learning and Knowledge Discovery in Databases*, number 5211 in Lecture Notes in Computer Science, pages 365–376. Springer Berlin Heidelberg, Sept. 2008. DOI: 10.1007/978-3-540-87479-9_42.
24. A. Gago-Alonso, A. Puentes-Luberta, J. A. Carrasco-Ochoa, J. E. Medina-Pagola, and J. F. Martínez-Trinidad. A New Algorithm for Mining Frequent Connected Subgraphs based on Adjacency Matrices. *Intelligent Data Analysis*, 14(3):385–403, Aug. 2010.
25. Z. Gao, L. Shang, and Y. Jian. Frequent subgraph mining based on the automorphism mapping. In *2012 2nd International Conference on Computer Science and Network Technology (ICCSNT)*, pages 1518–1522, 2012.
26. A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, June 1985.
27. E. Gudes, S. E. Shimony, and N. Vanetik. Discovering Frequent Graph Patterns Using Disjoint Paths. *IEEE Transactions on Knowledge and Data Engineering*, 18(11):1441–1456, Nov. 2006.
28. J. Han, J. Pei, and M. Kamber. *Data Mining: Concepts and Techniques.* Elsevier, June 2011.
29. S. Han, W. K. Ng, and Y. Yu. FSP: Frequent Substructure Pattern mining. In *2007 6th International Conference on Information, Communications Signal Processing*, pages 1–5, 2007.
30. T. Henderson. Parsemis. `https://github.com/timtadh/parsemis`. [Online; accessed 2016-05-30].
31. M. Hong, H. Zhou, W. Wang, and B. Shi. An Efficient Algorithm of Frequent Connected Subgraph Extraction. In K.-Y. Whang, J. Jeon, K. Shim, and J. Srivastava, editors, *Advances in Knowledge Discovery and Data Mining*, number 2637 in Lecture Notes in Computer Science, pages 40–51. Springer Berlin Heidelberg, Apr. 2003. DOI: 10.1007/3-540-36175-8_5.
32. H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(suppl 1):i213–i221, June 2005.

33. J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 549–552. IEEE, 2003.
34. J. Huan, W. Wang, J. Prins, and J. Yang. SPIN: Mining Maximal Frequent Subgraphs from Graph Databases. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 581–586, New York, NY, USA, 2004. ACM.
35. A. Inokuchi. Acgm. kwansei gakuin university. `http://ist.ksc.kwansei.ac.jp/~inokuchi/acgm.zip`. [Online; accessed 2016-05-30].
36. A. Inokuchi, T. Washio, and H. Motoda. An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. pages 13–23, 2000.
37. A. Inokuchi, T. Washio, and H. Motoda. Complete Mining of Frequent Patterns from Graphs: Mining Graph Data. *Machine Learning*, 50(3):321–354, Mar. 2003.
38. A. Inokuchi, T. Washio, and H. Motoda. A general framework for mining frequent subgraphs from labeled graphs. *Fundamenta Informaticae*, 66(1-2):53–82, 2005.
39. A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. Technical report, IBM, 2002.
40. Y. Jia, J. Zhang, and J. Huan. An efficient graph-mining method for complicated and noisy data with real-world applications. *Knowledge and Information Systems*, 28(2):423–447, Feb. 2011.
41. C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(01):75–105, Mar. 2013.
42. R. Jin, C. Wang, D. Polshakov, S. Parthasarathy, and G. Agrawal. Discovering Frequent Topological Structures from Graph Datasets. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 606–611, New York, NY, USA, 2005. ACM.
43. G. Karypis. Pafi software package for finding frequent patterns in diverse datasets. Karypis Lab. `http://glaros.dtc.umn.edu/gkhome/project/dm/software?q=pafi/overview`. [Online; accessed 2016-05-30].
44. Y. Ke and J. Cheng. Efficient Correlation Search from Graph Databases. *IEEE Trans. Knowl. Data Eng.*, 20(12):1601–1615, 2008.
45. M. R. Keyvanpour and F. Azizani. Classification and Analysis of Frequent Subgraphs Mining Algorithms. *ResearchGate*, 7(1):220–227, Jan. 2012.
46. S. Kramer, L. De Raedt, and C. Helma. Molecular Feature Mining in HIV Data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 136–143, New York, NY, USA, 2001. ACM.
47. V. Krishna, N. R. Suri, and G. Athithan. A comparative survey of algorithms for frequent subgraph discovery. *CURRENT SCIENCE*, 100(2):190, 2011.
48. M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, ICDM '01, pages 313–320, Washington, DC, USA, 2001. IEEE Computer Society.
49. M. Kuramochi and G. Karypis. Discovering frequent geometric subgraphs. In *2002 IEEE International Conference on Data Mining, 2002. ICDM 2003. Proceedings*, pages 258–265, 2002.
50. M. Kuramochi and G. Karypis. Finding Frequent Patterns in a Large Sparse Graph. *Data Min. Knowl. Discov.*, 11(3):243–271, Nov. 2005.
51. M. Lalmas. Aggregated Search. In M. Melucci and R. Baeza-Yates, editors, *Advanced Topics in Information Retrieval*, number 33 in The Information Retrieval Series, pages 109–123. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-20946-8_5.
52. G. Lee and U. Yun. *An Efficient Approach for Mining Frequent Sub-graphs with Support Affinities*, pages 525–532. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
53. Y. Li, Q. Lin, G. Zhong, D. Duan, Y. Jin, and W. Bi. A Directed Labeled Graph Frequent Pattern Mining Algorithm Based on Minimum Code. In *Third International Conference on Multimedia and Ubiquitous Engineering, 2009. MUE '09*, pages 353–359, June 2009.
54. W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in mapreduce. In *2014 IEEE 30th International Conference on Data Engineering*, pages 844–855. IEEE, 2014.

55. B. D. McKay et al. *Practical graph isomorphism.* Department of Computer Science, Vanderbilt University Tennessee, US, 1981.
56. T. Meinl, M. WÃ¶rlein, O. Urzova, I. Fischer, and M. Philippsen. The ParMol Package for Frequent Subgraph Mining. *Electronic Communications of the EASST*, 1(0), July 2007.
57. N. A. Mendoza, J. A. Carrasco-Ochoa, A. Gago-Alonso, J. F. Martinez-Trinidad, and J. E. Medina-Pagola. Representative Frequent Approximate Subgraph Mining in Multi-Graph Collections. 2015.
58. M. H. Nadimi-Shahraki, M. Taki, and M. Naderi. IDFP-TREE: An Efficient Tree for interactive mining of frequent subgraph patterns. *Journal of Theoretical and Applied Information Technology*, 74(3), 2015.
59. P. C. Nguyen, T. Washio, K. Ohara, and H. Motoda. Using a Hash-Based Method for Apriori-Based Graph Mining. In J.-F. Boulicaut, F. Esposito, F. Giannotti, and D. Pedreschi, editors, *Knowledge Discovery in Databases: PKDD 2004*, number 3202 in Lecture Notes in Computer Science, pages 349–361. Springer Berlin Heidelberg, Sept. 2004. DOI: 10.1007/978-3-540-30116-5_33.
60. M. C. Nicklaus. Downloadable structure files of nci open database compounds. national cancer institute. `https://cactus.nci.nih.gov/download/nci/index.html`. [Online; accessed 2016-05-30].
61. S. Nijssen. Gaston - download. `http://liacs.leidenuniv.nl/~nijssensgr/gaston/download.html`. [Online; accessed 2016-05-30].
62. S. Nijssen. Performance comparison of graph mining algorithms on pte. `http://liacs.leidenuniv.nl/~nijssensgr/farmer/results.html`, 2003. [Online; accessed 2016-05-30].
63. S. Nijssen and J. Kok. Faster association rules for multiple relations. In *In International Joint Conference on Artificial Intelligence*, pages 891–896. Morgan Kaufmann, 2001.
64. S. Nijssen and J. N. Kok. A Quickstart in Frequent Structure Mining Can Make a Difference. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 647–652, New York, NY, USA, 2004. ACM.
65. S. Nijssen and J. N. Kok. The Gaston Tool for Frequent Subgraph Mining. *Electronic Notes in Theoretical Computer Science*, 127(1):77–87, Mar. 2005.
66. S. Nijssen and J. N. Kok. Frequent subgraph miners: Runtime dont say everything. In *Proceedings of the International Workshop on Mining and Learning with Graphs (MLG 2006*, pages 173–180, 2006.
67. S. Nowozin. gboost graph boosting toolbox for matlab. `http://www.nowozin.net/sebastian/gboost/#intro`. [Online; accessed 2016-05-30].
68. S. Nowozin and K. Tsuda. Frequent Subgraph Retrieval in Geometric Graph Databases. In *2008 Eighth IEEE International Conference on Data Mining*, pages 953–958, Dec. 2008.
69. H. J. Patel, R. Prajapati, M. Panchal, and M. Patel. A Survey of Graph Pattern Mining Algorithm and Techniques. *International Journal of Application or Innovation in Engineering & Management*, 2(1):125–129, 2013.
70. M. Philippsen. Parsemis - the parallel and sequential mining suite. `https://www2.cs.fau.de/EN/research/zold/ParSeMiS/index.html`. [Online; accessed 2016-05-30].
71. S. Ranu and A. K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *2009 IEEE 25th International Conference on Data Engineering*, pages 844–855, Mar. 2009.
72. S. U. Rehman, S. Asghar, Y. Zhuang, and S. Fong. Performance Evaluation of Frequent Subgraph Discovery Techniques. *Mathematical Problems in Engineering, Mathematical Problems in Engineering*, 2014, 2014:e869198, Aug. 2014.
73. T. K. Saha and M. A. Hasan. FS^3: A Sampling based method for top-k Frequent Subgraph Mining. *arXiv:1409.1152 [cs]*, Sept. 2014. arXiv: 1409.1152.
74. Ł. Skonieczny. Mining for Unconnected Frequent Graphs with Direct Subgraph Isomorphism Tests. In K. A. Cyran, S. Kozielski, J. F. Peters, U. Stańczyk, and A. Wakulicz-Deja, editors, *Man-Machine Interactions*, number 59 in Advances in Intelligent and Soft Computing, pages 523–531. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-00563-3_55.
75. I. Takigawa and H. Mamitsuka. Efficiently mining tolerance closed frequent subgraphs. *Machine Learning*, 82(2):95–121, Sept. 2010.
76. A. Termier, Y. Tamada, K. Numata, S. Imoto, T. Washio, and T. Higuchi. Digdag, a first algorithm to mine closed frequent embedded sub-dags. In *Mining and Learning with Graphs, MLG 2007, Firence, Italy, August 1-3, 2007, Proceedings*, 2007.

77. M. Thoma, H. Cheng, A. Gretton, J. Han, H.-P. Kriegel, A. Smola, L. Song, P. S. Yu, X. Yan, and K. M. Borgwardt. Discriminative frequent subgraph mining with optimality guarantees. `http://www.dbs.ifi.lmu.de/cms/Publications/Discriminative_Frequent_Subgraph_Mining_with_Optimality_Guarantees`. [Online; accessed 2016-05-30].
78. J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.
79. N. Vijayalakshmi. FP-GraphMiner - A Fast Frequent Pattern Mining Algorithm for Network Graphs. *Journal of Graph Algorithms and Applications*, 15(6):753–776, 2011.
80. C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable Mining of Large Disk-based Graph Databases. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 316–325, New York, NY, USA, 2004. ACM.
81. J. Wang, Z. Zeng, and L. Zhou. CLAN: An Algorithm for Mining Closed Cliques from Large Dense Graph Databases. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 73–73, Apr. 2006.
82. W. Wang, C. Wang, Y. Zhu, B. Shi, J. Pei, X. Yan, and J. Han. GraphMiner: A Structural Pattern-mining System for Large Disk-based Graph Databases and Its Applications. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 879–881, New York, NY, USA, 2005. ACM.
83. M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston. In A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, editors, *Knowledge Discovery in Databases: PKDD 2005*, number 3721 in Lecture Notes in Computer Science, pages 392–403. Springer Berlin Heidelberg, Oct. 2005. DOI: 10.1007/11564126_39.
84. J. Wu and L. Chen. A Fast Frequent Subgraph Mining Algorithm. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 82–87, Nov. 2008.
85. X. Yan. Software - gspan: Frequent graph mining package. `http://www.cs.ucsb.edu/~xyan/software/gSpan.htm`. [Online; accessed 2016-05-30].
86. X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining Significant Graph Patterns by Leap Search. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 433–444, New York, NY, USA, 2008. ACM.
87. X. Yan and J. Han. gSpan : Graph-Based Substructure Pattern Mining. Technical report, UIUC, UIUCDCS-R-2002-2296, 2002.
88. X. Yan and J. Han. gSpan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. ICDM 2003. Proceedings*, pages 721–724, 2002.
89. X. Yan and J. Han. CloseGraph: Mining Closed Frequent Graph Patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 286–295, New York, NY, USA, 2003. ACM.
90. X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 335–346, New York, NY, USA, 2004. ACM.
91. X. Yan, X. J. Zhou, and J. Han. Mining Closed Relational Graphs with Connectivity Constraints. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 324–333, New York, NY, USA, 2005. ACM.
92. M. J. Zaki. Data mining template library (dmtl). `http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software/Software`. [Online; accessed 2016-05-30].
93. Z. Zeng, J. Wang, J. Zhang, and L. Zhou. FOGGER: An Algorithm for Graph Generator Discovery. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 517–528, New York, NY, USA, 2009. ACM.
94. Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent Closed Quasi-clique Discovery from Large Dense Graph Databases. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 797–802, New York, NY, USA, 2006. ACM.
95. S. Zhang and J. Yang. RAM: Randomized Approximate Graph Mining. In B. Luduscher and N. Mamoulis, editors, *Scientific and Statistical Database Management*, number 5069 in Lecture Notes in Computer Science, pages 187–203. Springer Berlin Heidelberg, July 2008. DOI: 10.1007/978-3-540-69497-7_14.

96. S. Zhang, J. Yang, and V. Cheedella. Monkey: Approximate Graph Mining Based on Spanning Trees. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1247–1249, Apr. 2007.
97. K. Zhou. gspan algorithm in data mining. `https://github.com/Jokeren/DataMining-gSpan`. [Online; accessed 2016-05-30].
98. Z. Zou, J. Li, H. Gao, and S. Zhang. Frequent Subgraph Pattern Mining on Uncertain Graph Data. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 583–592, New York, NY, USA, 2009. ACM.
99. Z. Zou, J. Li, H. Gao, and S. Zhang. Mining Frequent Subgraph Patterns from Uncertain Graph Data. *IEEE Transactions on Knowledge and Data Engineering*, 22(9):1203–1218, Sept. 2010.