Chinese Journal of Electronics
Vol.30, No.2, Mar. 2021

Novel Approach to Minimize the Memory Requirements of Frequent Subgraph Mining Techniques

BİLGİN Turgay Tugay[1] and OĞUZ Murat[2]

(1. *Department of Computer Engineering, Bursa Technical University, Bursa, Turkey*)

(2. *Microsoft Corporation Turkey Branch, İstanbul, Turkey*)

Abstract — Frequent subgraph mining (FSM) is a subset of the graph mining domain that is extensively used for graph classification and clustering. Over the past decade, many efficient FSM algorithms have been developed with improvements generally focused on reducing the time complexity by changing the algorithm structure or using parallel programming techniques. FSM algorithms also require high memory consumption, which is another problem that should be solved. In this paper, we propose a new approach called Predictive dynamic sized structure packing (PDSSP) to minimize the memory needs of FSM algorithms. Our approach redesigns the internal data structures of FSM algorithms without making algorithmic modifications. PDSSP offers two contributions. The first is the Dynamic Sized Integer Type, a newly designed unsigned integer data type, and the second is a data structure packing technique to change the behavior of the compiler. We examined the effectiveness and efficiency of the PDSSP approach by experimentally embedding it into two state-of-the-art algorithms, gSpan and Gaston. We compared our implementations to the performance of the originals. Nearly all results show that our proposed implementation consumes less memory at each support level, suggesting that PDSSP extensions could save memory, with peak memory usage decreasing up to 38% depending on the dataset.

Key words — Frequent Subgraphs, Data Mining, Memory, Space Complexity.

I. Introduction

Data mining extracts useful information from large databases. Graph mining is a subdomain of data mining that focuses on information extraction from graph-based structured data and is used in a variety of real-world and scientific applications, such as social networks and bioinformatics. The size of the graphs used in graph mining is growing rapidly, having now increased well into the petabytes[1]. In 2000, 2.1 billion vertices and 15 billion edges existed on the web graph created by search engines[2]. Today, search engines leverage new infrastructures that support even larger web graphs, which can include over one trillion vertices[3].

Genome sequencing is another important research area using graph mining. Recent works have attempted to solve the genome assembly problem by using graphical representations of the genomes. One example is the de Bruijn graph that was generated based on a k-mers calculation in Velvet algorithms. The maximum number of nodes in this graph reached nearly 4 million[4].

The most widely used application domain of graph mining looks at graph pattern mining problems, also referred to as Frequent subgraph mining (FSM). These challenges are wellstudied with numerous applications in areas such as computational chemistry, bioinformatics, and social networks[5]. FSM considers the enumeration of frequent subgraphs, either in a single graph or a graph database. During the enumeration of subgraphs, memory requirements become exponentially larger compared to the size of the input data. For this reason, various methods exist to optimize memory and CPU usage of these FSM algorithms. Improvements have focused on distributing the computational power and memory requirements to different nodes by parallelization techniques, like Map/Reduce or the Message passing interface (MPI). Using compression techniques, changing the data storing structure or format with new technologies, such as Cassandra or Hadoop, offer additional working solutions. However, all these optimization techniques require modifications to the FSM algorithms.

Applying High-performance computing (HPC) for

Manuscript Received May 9, 2017; Accepted Dec. 21, 2019.
 DOI:10.1049/cje.2021.01.003

FSM applications is an active area of research, especially on Symmetric multiprocessing (SMP) systems. In these systems, one shared memory is used by more than one core. Any FSM algorithm paralleled on an SMP system requires more memory than a single-threaded counterpart configured to increase the level of parallelization owing to the data sharing needs between threads. As a result, space complexity becomes a more significant problem for this type of systems[6].

In this study, we develop a new approach to decrease the memory requirements of FSM algorithms when they are run as a single-threaded implementation or an HPC-based structure. Our proposed approach can be applied to various FSM algorithms and does not require these algorithms to be redesigned[7].

The remainder of this paper is organized as follows. We first define several FSM preliminaries and challenges in Section II. In Section III, we discuss our method and describe our experimental study in Section IV. The final section offers conclusions of our research.

II. Definitions

A graph is constructed by pairing a set of vertices V with a set of edges E and is defined by $G = (V, E)$, if $E \subseteq VxV$ and every edge $e \in E$ relates to a pair of vertices $(v1, v2)$.

Two graphs G_1 and G_2 are isomorphic if $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are topologically identical. This relationship means that there is a mapping from G_1 to G_2 such that each edge in E_1 is mapped to a single edge in E_2 and vice versa. If the graph includes labels, then this mapping must also exist between the labels on the vertices and edges.

The graph $G_2 = (V_2, E_2)$ is a subgraph of another graph $G_1 = (V_1, E_1)$ when $V_2 \subseteq V_1, E_2 \subseteq E_1(v1, v2) \in E_2 \rightarrow V_1 \in V$, and $v_2 \in V_2$ can be found, as in Fig.1.

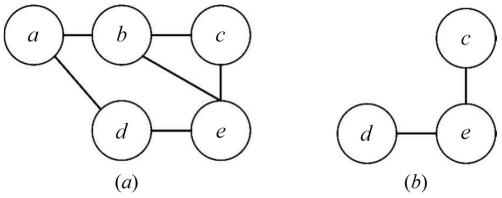

Fig. 1. Representation of graph. (*a*) G_1; (*b*) Subgraph of G_1

The graph $G_1 = (V_1, E_1)$ in Fig.1(a) containing the vertex set $V_1 = \{a, b, c, d, e\}$ and edge set $E_1 = \{ab, ad, bc, be, ce, de\}$ is shown. So, the graph $G_2 = (V_2, E_2)$ of Fig.1(b) with vertex set $V_2 = \{c, d, e\}$ and edge set $E_2 = \{ce, de\}$ is a subgraph of graph G_1.

Subgraph isomorphism occurs between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ when an isomorphism between G_2 and a subgraph of G_1 exists. In other words, identifying this characteristic is important to determine if G_2 is included in G_1.

A frequent subgraph is defined as a graph that frequently occurs within a graph database, which is a special type of database comprising of a single large graph or multiple small graphs. Given a labeled graph dataset $G_D = \{G_1, G_2, \ldots, G_k\}$, the support or frequency of a subgraph g is the percentage (or number) of graphs in G_{D} where g exists as a subgraph[8]. If D is the input database, then the graph support G_D is denoted by Sup (G_D).

Frequent subgraph mining is the process of discovering subgraphs within a given set of graphs[9]. For example, with the presented graphs in Fig.2(a) and 2(b), a frequent subgraph is the graph shown in Fig.2(c).

Many efficient FSM algorithms have been developed, such as gSpan[10], Gaston[11], CloseGraph[12], SPIN[13], Mofa[14], EDC[15], and FSG[16]. In these studies, two basic approaches were applied to the FSM problem. The first approach shares similar characteristics with Apriori-based frequent item set mining algorithms as it begins searching for small-sized subgraphs, and then extends the search by joining subsequently found subgraphs. The well-known Apriori-based frequent subgraph mining algorithms include AGM, FSG, and an edge-disjoint path-join algorithm[17].

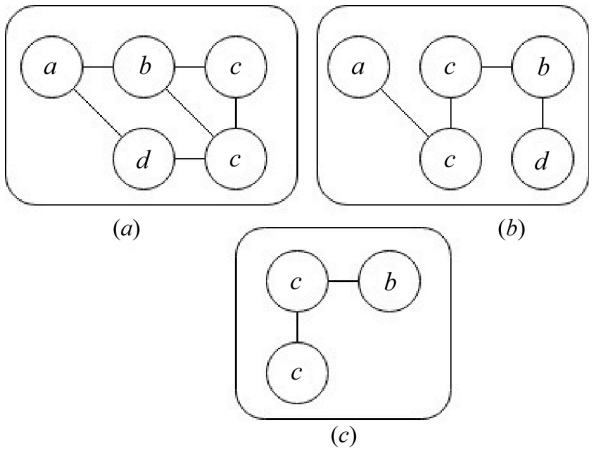

Fig. 2. Representation of graph. (*a*) Input graphs; (*b*) and (*c*) Frequent subgraph

The second approach employs a pattern-growth algorithm that begins at an initial edge and extends the graph by directly adding a new edge in every possible position. Then, the process checks if the generated graph supports a threshold value. Well-known pattern-growth-based graph mining algorithms include gSpan, MoFa, SPIN, and Gaston[18,19].

The gSpan and Gaston algorithms are applied to testing our proposed approach. gSpan (graph-based Substructure pattern) uses DFS-codes for presenting and storing graphs. The test to search and compare frequent

subgraphs for isomorphism is performed via a DFS code tree. So, with this feature, gSpan does not require candidate pattern generation as it generates all exact frequent subgraphs.

gSpan also guarantees the completeness of the mining results with minimum DFS codes by pruning non-minimal children from the solution space. Algorithm 1 describes the pseudo-code of gSpan, which is an integration of the algorithm descriptions presented in Ref.[10].

Algorithm 1 gSpan

Method 1 *GraphSet_projection(GS, FS)*
sort labels of the vertices and edges in GS by frequency;
remove infrequent vertices and edges;
relabel the remaining vertices and edges (descending);
$S1$:= all frequent 1-edge graphs;
sort S^1 in DFS lexicographic order;
$FS := S^1$;
for each edge e in S^1 **do**
init g with e, set $g.DS = \{h|h \in GS, e \in E(h)\}$;
Subgraph_mining(GS, FS, g);
$GS := GS - e$;
if |GS|< $minSup$
break;
Method 2 *Subgraph_mining(GS, FS, g)*
if $g \neq min(g)$
return;
$FS := FS \cup \{g\}$;
enumerate g in each graph in GS and count g's children;
for each c (child of g) **do**
if $support(c) \geq minSup$
Subgraph_mining(GS, FS, c);
Enumeration of g: Finding all the exact positions of g in another graph

All pattern-growth algorithms generate duplicated candidates during the enumeration process. In gSpan, these duplicated candidates represent non-minimal codes. Instead of calculating the minimum DFS code of s from all possible DFS codes and selecting the smallest to compare with s, gSpan defines an efficient function 'isMin(s)' in the 'Subgraph_mining' method. As a heuristic search designed using the DFS lexicographic order, this search concludes when a prefix of a DFS is generated and is less than s, representing that s is not minimal.

For support calculation and candidate enumeration, gSpan uses a Transaction ID (TID) list that includes the ID of each graph in the database containing the corresponding subgraph.

To find the set of frequent graphs, we use the memory-based algorithm Gaston (Graph sequence tree extraction), which is based on the observation that most frequent substructures in practical graph databases are free trees. This algorithm employs a highly effective strategy to enumerate the frequent free trees first. Gaston first stores all embeddings (nodes and edges) to generate only the refinements that appear and achieve fast isomorphism testing. The paths and trees are checked next, and the subgraph isomorphism test is performed last. Gaston only outputs the cycled graphs, so it works faster than gSpan, FFSM, and Mofa.

To summarize the Gaston algorithm, let *P(U)* be a subgraph found in U. The first step finds all frequent edges in the database, F_1. For each frequent edge p, the algorithm generates the descendants G of p with the set of allowable extended edges L (for each block). According to the type of G and the extended edges, the algorithm determines the paths, trees, or cyclic graphs in the database. The pseudocode if-else sections perform these operations, as outlined in Algorithm 2.

Algorithm 2 Gaston

Input: U, one of the units of the database
sup, minimum support.
Output: $P(U)$, the set of frequent subgraphs in U.
F_1 = {frequent edges in U};
for each $p \in F_1$ {
L = {allowable extended edges of p};
for each allowable extended edge $l \in L$ {
G' = Adding l to p;
L' = {allowable extended edges of G'};
if l is a node refinement {
if G' is a path
find paths with G' and L';
else
find trees with G' and L';
}
else
find cyclic graphs with G' and L' ;
}
}

III. Predictive Dynamic Sized Structure Packing (PDSSP)

In this section, we describe our contribution to the standard FSM algorithm, which reads datasets from an external resource, processes the data, and writes back to the disk. A flowchart of the standard FSM algorithm is depicted in Fig.3(a).

Our proposed PDSSP approach redesigns the internal data structures of the FSM algorithm without making algorithmic modifications, thereby acting only as an extension. After applying the PDSSP to a standard FSM implementation (FSM_PDSSP), the modifications are depicted in Fig.3(b).

All FSM implementations use a fixed-type integer for all variables. The idea of the PDSSP is that if the maximum value to be stored in the integer-based variables can be estimated, then varying-length integer data types can also be employed. In this way, the memory requirements of FSM implementations can be reduced.

We analyzed FSM implementations and observed that only unsigned integers are utilized due to the structure of the input data. Table 1 lists the standard

unsigned integer types with corresponding storage sizes and value ranges[20]. We experienced high memory usage when choosing improper integer data types. Consequently, we focused on exploring a solution to create varying-length unsigned integer data types.

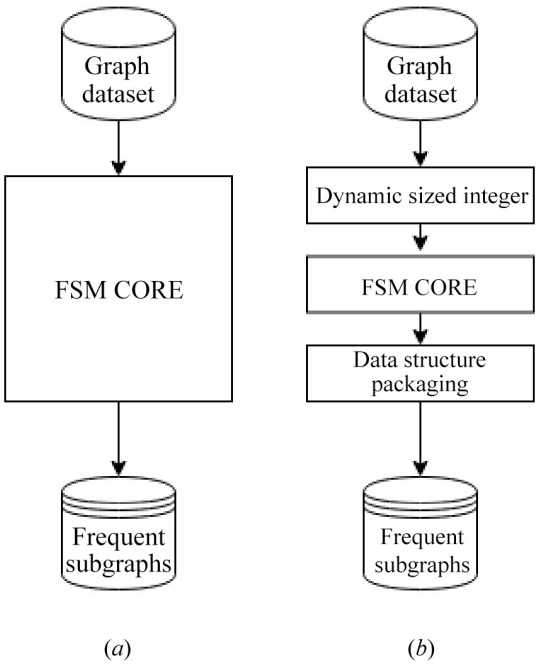

Fig. 3. Flowcharts of standard FSM and PDSSP_FSM

Table 1. Standard unsigned integer types and their storage sizes and ranges

Type	Storage size	Value range
Unsigned char	1 byte	0 to 255 (2^8)
Unsigned short int	2 bytes	0 to 65,535 (2^{16})
Unsigned int	4 bytes	0 to 4,294,967,295 (2^{32})
Unsigned long int	8 bytes	0 to 18,446,744,073,709,551,616(2^{64})

FSM_PDSSP offers two contributions. First, the Dynamic sized integer type (ds_Int) is a newly designed unsigned integer data type that features a varying capacity range from 0 to 2^{64}, which can be changed on demand. Second, a "data structure packaging" component uses a data structure packing technique that adjusts the behavior of the compiler. Details about these contributions are discussed in the following sections.

FSM implementations store the maximum possible values of graph features in memory as an integer data type regardless of the number of samples, edges, and vertices in the dataset. PDSSP is designed to convert this static memory usage into a dynamic state. Hence, our proposed model includes two stages with a flowchart of its implementation presented in Fig.4.

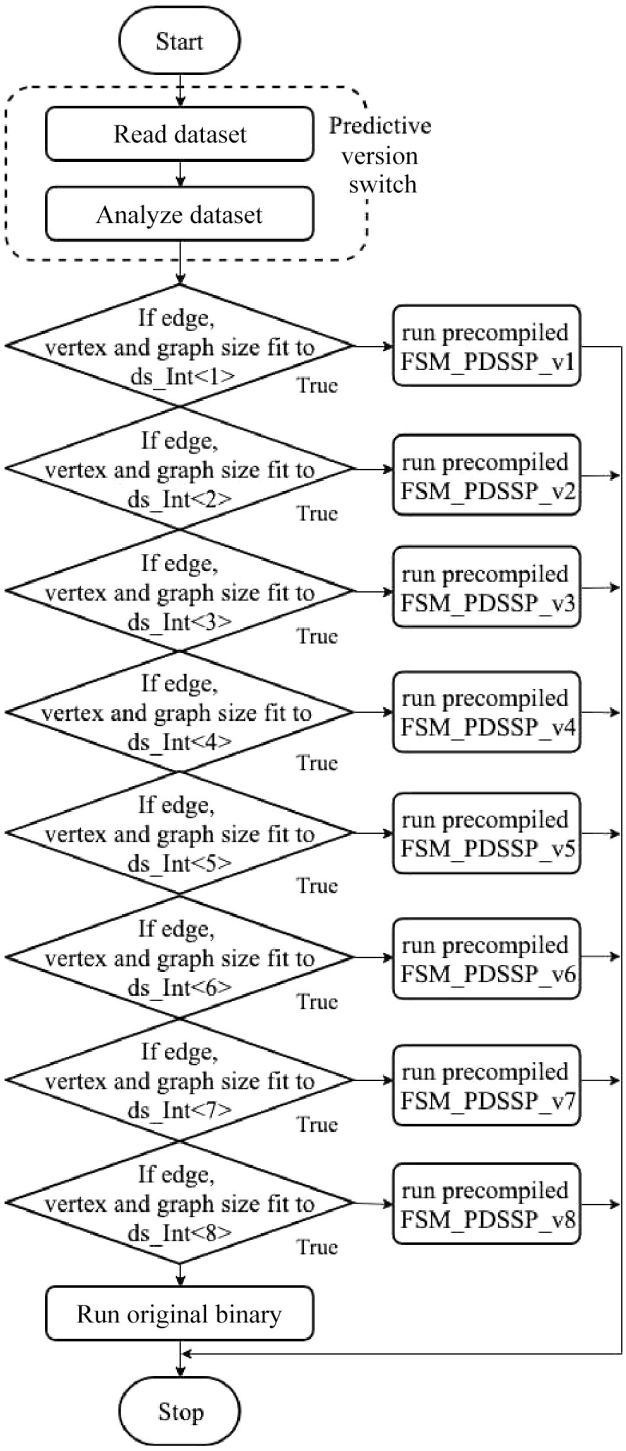

Fig. 4. Our proposed flowchart to use FSM with PDSSP

The first stage is the Predictive version switch (PVS) that scans the input dataset file containing the stored graphs digitized in the DIMACS[21] format, as shown in Table 2. On the "analyze dataset" module in Fig.4, PVS determines the proper integer value range and selects the appropriate pre-compiled PDSSP version to execute. FSM_PDSSP is an FSM version for which the primitive integer data type is replaced by the dynamic length ds_Int data type. An appropriate size of ds_Int is

determined by the range of values given in Table 3.

Table 2. Input dataset file format

t # <graph_id>
v <vertex_id><vertex_label>
e <edge_from><edge_to><edge_label>
<next_graph_or_end_of_file>

Table 3. Comparison of ds_Int and standard integer types

Value range	Standard data type / size (byte)	ds_Int type / size (byte)	Saving (byte)
0 to 2^8	Unsigned char / 1	ds_Int<1>/ 1	0
0 to 2^{16}	Unsigned short int / 2	ds_Int<2>/ 2	0
0 to 2^{24}	Unsigned int / 4	ds_Int<3>/ 3	1
0 to 2^{32}	Unsigned int / 4	ds_Int<4>/ 4	0
0 to 2^{40}	Unsigned long int / 8	ds_Int<5>/ 5	3
0 to 2^{48}	Unsigned long int / 8	ds_Int<6>/ 6	2
0 to 2^{56}	Unsigned long int / 8	ds_Int<7>/ 7	1
0 to 2^{64}	Unsigned long int / 8	ds_Int<8>/ 8	0

The C/C++ languages enable changing primitive integer data types only at compile time and not during runtime. Therefore, the FSM_PDSSP code is pre-compiled before the operation for each type of ds_Int. As shown in Fig.4, a determiner module chooses the optimum pre-compiled binary FSM_PDSSP based on the number of edges, vertices, and graph size. If the determiner module cannot identify a version to execute, then the original binary is executed. Finally, the detected frequent sub-graphs are written to the disk.

1. Predictive version switch (PVS)

The PVS is a preprocessing module developed as an independent software that is modified to meet the requirements of gSpan and Gaston. PVS takes as input parameters the input dataset file, the name of the algorithm, minimum support level, and output file, and its pseudo-code is given in Algorithm 3.

After scanning the input dataset, PVS finds the maximum integer values used to store the transaction count, vertex number, edge number, and label numbers. Then, PVS runs the appropriate PDSSP version.

2. Dynamic sized integer type (ds_Int)

A new unsigned integer variable type, ds_Int, is designed as part of the solution. As seen in its pseudo-code implementation in Algorithm 4, the idea of ds_Int is to store unsigned integer values into an unsigned array using bitshifting operations. Leveraging this method, ds_Int can also support larger numbers than 2^{64} with minimal algorithmic modification. For this study, we limited this scale to 2^{64} to compare with the unsigned long integer type.

The difference from the standard unsigned integer type is that ds_Int can be declared to store 1 to 7 bytes and from 0 to 2^{64} value correspondingly. In the C/C++ languages, primitive unsigned integers typically require 1, 2, 4, or 8 bytes but not 3, 5, 6, or 7 bytes. This dynamically sizable ds_Int enables the programmer to define 3, 5, 6, or 7 byte integers; hence, a significant amount of memory is saved, depending on the dataset.

Algorithm 3 Predictive Version Switch (PVS)

PVS (A dataset file ds_file, an algorithm selection alg_select, a minimum support level min_sup, an output file out_file)
Set TransactionCount, maxVertex, maxEdge, maxLabel to zero;
Read graphs from ds_file into Dataset
for each *row* in *Dataset* **do**
  if type of *row* is transaction **then**
    Increase *TransactionCount* by 1
    else if type of *row* is label **and** label_ID of row is greater than *maxLabel* **then**
      Set *maxLabel* to label_ID
    else if type of row is vertex **and** vertex_ID of row is greater than *maxVertex* **then**
      Set *maxVertex* to vertex_ID
    else if type of *row* is edge **and** edge_ID of *row* is greater than *maxEdge* **then**
    Set maxEdge to edge_ID
  end if
end for
if alg_select is gSpan **then**
  Run "gSpan_PDSSP" with $ds\_file, min\_sup, out\_file$
  else if alg_select is Gaston **then**
    Run "Gaston_PDSSP" with $ds\_file, min\_sup, out\_file$
end if

Algorithm 4 ds_Int

  struct ds_Int (An unsigned integer data InputData, size of ds_Int value byte_size)
    Set StoredData with an empty unsigned char array in byte_size size
    function *get* () returns integer
      Set *OutputData* to zero
      for i from 0 to byte_size **do**
        Set *OutputDatawithOutputData* & ($i \times 8$ byte left shifted *StoredData*[i])
      end for
      return OutputData
    end function
    function *set* (*InputData*) returns nothing
      for j from 0 to byte_size **do**
        Set *StoredData*[j] to *StoredData* & ($j \times 8$ byte right shifted *InputData*)
      end for
    end function

Comparisons of ds_Int and standard unsigned integer data types with corresponding storage savings are shown in Table 3. When the upper limit of the value range is higher, especially greater than 2^{32}, the memory space savings increases. To demonstrate the benefit of using ds_Int, we tested the following example. Assume 2^{22} integer items exist in an array and the maximum value to be stored is 2^{35}. If standard integer types are used to store the array, then an unsigned long integer type must be preferred owing to its supported

size limit. The memory space required for this operation is calculated to be $\left(\sum_{n=1}^{2^{22}} 8\right)/1024 = 32,768$KB. When a ds_Int<5> integer type is used for the same operation, the memory space requirement reduces to $\left(\sum_{n=1}^{2^{22}} 5\right)/1024 = 20,480$KB, which is a reduction in the memory requirement of $\left(1 - \frac{20480}{32768}\right) = 37\%$

3. Data structure packing

Data structure packing is the final and fundamental component of the PDSSP approach. First, we review how data is stored and accessed in memory.

In computer systems, stored data in memory have two properties: value and storage location (address in memory). Data alignment means that the address of the data should be evenly divisible by any power of 2 because the CPU does not read one byte at a time. By default, the value of the word size depends on the architecture of a system. Word sizes tend to be 4, in most cases, and if the size of the data is smaller than a word size, empty spaces are added to the end of the data for data alignment. This technique is called "padding".

Compiler padding is illustrated in the following example. Here, an int is assumed to be 4 bytes, and char is a single byte. struct mydata {char C; int L; char B; int J; };

Fig.5 illustrates how 'struct mydata' would be padded to align with 4 byte boundaries. As the alignment of an 'int' on this platform is 4 bytes, 3 bytes are added after 'char C', and 3 bytes are added at the end of 'char' B. Because of this padding, the addresses of the data in this structure are evenly divisible by 4, which is a process referred to as structure member alignment. As a consequence, the size of the structure in memory increases.

In this case, the CPU must perform additional operations to access the data, such as loading two chunks of data, shifting out unwanted bytes, and then combining them. These extra operations can slow the performance of the CPU[22].

As described above, our new data type ds_Int is fully adjustable from 1 byte to 8 byte storage sizes and avoids the misaligned data access caused by compiler alignment options. In our C/C++ implementation, we use the "#pragma" preprocessor directive to configure the compiler to work with the specified structure packing size when activated[23].

The PVS scans the input dataset file and finds the integer value range during the "analyze dataset" step in Fig.4 before determining the appropriate ds_Int type. As a result, if 3 bytes fit for the integer data type, our proposed "data structure packaging system" selects the ds_Int<3> data type.

Fig.6 illustrates the new memory alignment when then integer data type reduces to 3 bytes. As a result, 6 bytes of waste have been saved with our proposed method.

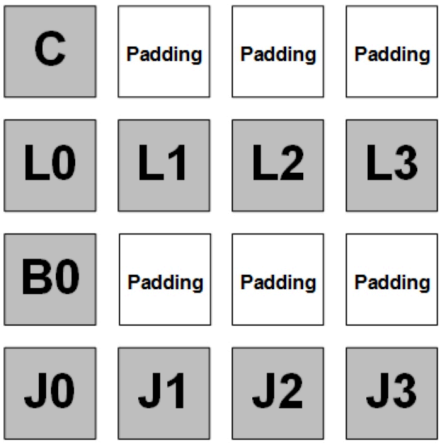

Fig. 5. Memory alignment and padding of struct mydata

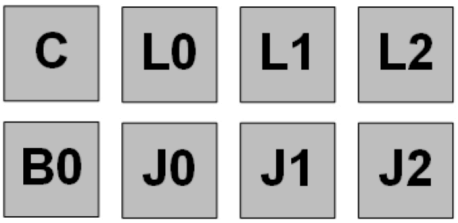

Fig. 6. Memory alignment when ds_Int<3> used

4. Embedding PDSSP into FSM algorithms

As mentioned in the previous section, PDSSP is designed as an extension to FSM implementations. To embed PDSSP into an FSM implementation, memory profiling will determine the most memory-demanding data structures. If any greedy data structures are found, these may be replaced by our ds_Int types.

We used the Valgrind and Massif Visualizer tools[24] to profile the memory consumption of the gSpan and Gaston implementations. The most memory-demanding data structures were determined and replaced with the ds_Int type. The original data structures and replacements are shown in Table 4.

IV. Experiments

We examined the effectiveness and efficiency of the PDSSP approach by embedding it into two state-of-the-art algorithms called gSpan and Gaston. We then compared our proposed FSM_PDSSP implementation with the classic implementation.

1. Datasets

To evaluate the performance of our PDSSP approach, we conducted experiments with three real-world datasets, Anticancer screen datasets (NCI)[25], Dobson and Doig (DD) molecule dataset[26], and the AIDS antiviral screen dataset (AIDS)[27]. In addition to these real datasets, we also generated three synthetic datasets, named

T10KV5KE14K, T58KV100E100, and T114KV200E200. The metadata for each benchmark dataset is listed in Table 5.

Table 4. The comparison of original data structures and PDSSP structures

Original data structure	
gSpan-1	struct Edge{ int from; int to; int elabel; unsigned int id; //other codes };
gSpan-2	struct PDFS { unsigned int id; Edge * edge; PDFS * prev; //other codes };
Gaston-1	struct Legoccurrence{ Tid tid; OccurrenceID ccurrenceid; NodeId tonodeid, fromnodeid; //other codes };
PDSSP data structure	
gSpan-1	*#pragma pack(n)* struct Edge{ ds_Int<v_max>from; ds_Int<v_max>to; ds_Int<elabel_max>elabel; ds_Int<e_max>id; //other codes }; *#pragma pack()*
gSpan-2	*#pragma pack(n)* struct PDFS { ds_Int<tid_max>id; Edge * edge; PDFS * prev; //other codes }; *#pragma pack()*
Gaston-1	*#pragma pack(n)* struct LegOccurrence{ ds_Int<tid_max>tid; ds_Int<tid_max+1>occurrenceid; ds_Int<v_max>tonodeid, fromnodeid; //other codes }; *#pragma pack()*

2. Test environment

For this study, we downloaded the gSpanCORK implementation of the gSpan algorithm, which was developed by Thoma, Marisa, *et al.*[28]. Gaston was downloaded from its official web site[29]. Because both implementations are open source and coded in C/C++, we easily embedded our proposed PDSSP implementation. All source codes were compiled for x64 architecture in CentOS Linux release 7.1, with GCC 4.8.3. The C++ compiler version that supports C11 standards was used to compile the PDSSP binaries. The test hardware included a 2-core Intel Xeon CPU E5-2670 2.60 GHz processor and 4 GB RAM. These implementations were developed to run in a singlethreaded mode.

Table 5. Comparison of gSpan and gSpan+PDSSP on benchmark datasets

Dataset $\mathbf{G}$	$\lvert\mathbf{G}\rvert$	$\lvert V_{MAX}(\mathbf{G})\rvert$	$\lvert E_{MAX}(\mathbf{G})\rvert$	$\lvert L_{MAX\text{-}V}\rvert$	$\lvert L_{MAX\text{-}E}\rvert$
NCI	20586	112	119	64	3
DD	1178	5747	14267	88	0
AIDS	56213	221	247	61	3
T10KV5 KE14K	10317	5747	14267	88	3
T58KV1 00E100	58242	112	119	63	3
T114KV 200E200	114455	221	247	63	3

Note: $\lvert\mathbf{G}\rvert$: the total number of graphs in the dataset;
$\lvert V_{MAX}(\mathbf{G})\rvert$: maximum number of vertices in any graph;
$\lvert E_{MAX}(\mathbf{G})\rvert$: maximum number of edges in any graph;
$\lvert L_{MAX\text{-}V}\rvert$: maximum number of vertex labels;
$\lvert L_{MAX\text{-}E}\rvert$: maximum number of edge labels.

3. Experimental results

We executed our implementation and the original on the benchmark datasets given in Table 5. The original implementations of gSpanCORK and Gaston were compared to the corresponding FSM_PDSSP implementations that we label as gSpan+PDSSP and Gaston+PDSSP. The running times and maximum memory consumptions (*e.g.*, peak memory) were collected for various support levels. All tests were performed three times to ensure consistency.

The results are shown in Tables 6 and 7 with the memory usage listed in megabytes and run times in seconds for each benchmark. The "gSpan Orig" column corresponds to the original implementation, and "gSpan+PDSSP" corresponds to the PDSSP embedded version. The bold values in these tables indicate better results.

The tests were repeated for various support levels ranging from 5% to 30%. At the lower support levels, run times were longer, as expected, which is why it requires more time to find the frequent subgraphs at a low support level while also consuming more memory. Due to the subgraph isomorphism tests performed during the frequent subgraph mining, the run time and allocated memory increase exponentially with the size of the dataset.

Tables 8 and 9 show the overall memory and run time improvements achieved through our embedded PDSSP approach. As shown in Table 8, the memory usage is significantly reduced in all the cases, with savings

ranging from 27.44% to 19.88% on all six benchmark datasets, and an average improvement of 25.66%. The table also indicates that improvement is accomplished by our PDSSP approach. The run time is also slightly better than the original implementation, as suggested from the average run time of the six benchmarks that our PDSSP embedded gSpan algorithm requires 1.15% less time to run.

Table 6. Comparison of gSpan and gSpan+PDSSP on benchmark datasets

Dataset	Supp.(%)	Mem. usage (MB's)		Run time (Sec)	
		Gaston Orig.	Gaston + PDSSP	Gaston Orig.	Gaston + PDSSP
NCI	5	142,57	**107,29**	37,97	**37,84**
	10	113,4	**85,26**	11,15	**11,03**
	15	94,33	**71,18**	6,66	**6,44**
	20	86,01	**62,83**	4,68	**4,58**
	25	74,36	**54,38**	3,71	**3,69**
	30	75,8	**54,75**	**3,3**	3,32
DD	5	84,68	**59,72**	718,24	**717,87**
	10	82,46	**57,17**	171,21	**168,14**
	15	80,95	**55,95**	**77,03**	77,89
	20	81,14	**55,38**	47,72	**47,37**
	25	79,54	**54,51**	32,56	**31,72**
	30	78,79	**54,39**	23,47	**23,28**
AIDS	5	147,82	**110,12**	10,93	**10,7**
	10	96,98	**71,83**	5,19	**5,11**
	15	92,86	**68,18**	4,1	**4,02**
	20	86,27	**65,86**	3,32	**3,29**
	25	78,4	**58,27**	2,68	**2,61**
	30	75,86	**57,26**	2,47	**2,39**
T58KV100E100	5	2434	**1827**	684,71	**671,71**
	10	1699	**1304**	179,46	**172,87**
	15	1418	**1050**	102,59	**99,27**
	20	1188	**906,99**	71,42	**69,81**
	25	1001	**765,91**	58,29	**57,62**
	30	877,57	**648,09**	47,87	**47,7**
T114KV200E20	5	1927	**1594**	236,45	**227,01**
	10	1412	**1086**	103,9	**103,18**
	15	1191	**956,61**	73,16	**72,45**
	20	1042	**841,67**	59,03	**57**
	25	927,74	**746,53**	51,2	**50,99**
	30	919,89	**731,68**	46,27	**45,88**
T10KV5KE14K	5	190,18	**138,58**	**25,80**	25,99
	10	153,78	**110,98**	12,54	**12,38**
	15	145,49	**104,14**	**10,04**	10,18
	20	135,00	**99,48**	**9,13**	9,39
	25	128,77	**92,43**	**8,65**	8,69
	30	125,51	**91,95**	**8,08**	8,18

The results of our second experiment are shown in Table 9. As with gSpan embedded with PDSSP, the memory usage is also significantly reduced for Gaston with PDSSP. The reduction ranges from 38.42% to 14.03% across the six benchmark datasets with an average improvement of 26.65%. However, the run times for all experiments are worse than the original implementation, with an average run time 15.16% longer than the original. To consider this observation, gSpan uses an adjacency list for graph representation and Gaston uses a hash table, which makes Gaston the fastest out of the four algorithms, MoFa, gSpan, FFSM, and Gaston[30]. We selected gSpan as a "fairly optimized" representative, whereas Gaston was chosen as a "well-optimized" representative to demonstrate our approach. So, our embedded PDSSP may cause some delay in Gaston, which represents a worst

Table 7. Comparison of Gaston and Gaston+PDSSP on benchmark datasets

Dataset	Supp.(%)	Mem. usage (MB's)		Run time (Sec)	
		Gaston Orig.	Gaston + PDSSP	Gaston Orig.	Gaston + PDSSP
NCI	5	77,01	**48,48**	**4,54**	5,8
	10	55,88	**34,81**	**1,45**	1,93
	15	45,33	**28,62**	**0,87**	1,13
	20	36,41	**24,13**	**0,6**	0,76
	25	30,49	**20,13**	**0,46**	0,59
	30	28,46	**13,89**	0,39	**0,27**
DD	5	40,27	**34,15**	**142,86**	164,15
	10	37,68	**31,55**	**40,32**	50,37
	15	35,2	**30,43**	**21,12**	23,32
	20	33,88	**29,04**	**12,61**	15,42
	25	33,33	**28,74**	**8,42**	9,83
	30	32,47	**27,8**	**6,18**	7,06
AIDS	5	53,84	**34,23**	**1,36**	1,88
	10	48,36	**29,86**	**0,7**	0,93
	15	43,55	**27,47**	**0,51**	0,66
	20	41,06	**26,25**	**0,4**	0,52
	25	35,64	**23,84**	**0,32**	0,38
	30	33,14	**21,71**	0,25	**0,05**
T58KV100E100	5	1224	**772,82**	**97,68**	112,91
	10	836,11	**513,39**	**29,58**	34,6
	15	636,45	**397,22**	**18,15**	20,86
	20	527,4	**331,93**	**12,37**	14,85
	25	427,06	**279,71**	**9,82**	12,13
	30	382,12	**246,1**	**7,95**	9,08
T114KV200E20	5	964,08	**747,84**	**38,57**	44,56
	10	627,82	**490,56**	**17,79**	21,15
	15	489,02	**390,57**	**11,21**	12,73
	20	427,09	**351,73**	**8,22**	10,06
	25	395,7	**318,78**	**6,61**	8,02
	30	387,75	**309,25**	**5,48**	6,01
T10KV5KE14K	5	107,21	**90,07**	**6,51**	8,71
	10	71,99	**59,63**	**1,47**	1,83
	15	45,63	**38,70**	**0,75**	0,83
	20	41,35	**35,52**	**0,65**	0,73
	25	36,81	**33,23**	**0,60**	0,66
	30	37,25	**32,77**	0,53	**0,41**

case for our approach of up to a 15% delay. While an actual delay is expected to be much smaller, 15% remains acceptable as the Gaston algorithm can operate up to 50% faster compared to gSpan. The corresponding rows from Tables 6 and 7 may be compared to the run times of gSpan and Gaston on the same dataset with the same support level.

The memory requirements of the FSM algorithms

are inversely proportional to the level of support. Nearly all results suggest that our proposed PDSSP implementation consumes less memory for each support level. The experimental results show that FSM_PDSSP saves memory with peak memory usage decreasing up to 38%, depending on the dataset.

Table 8. Memory and run time improvement for the PDSSP employed gSpan algorithm

Dataset	Memory usage gSpan with PDSSP(%)	Run time gSpan with PDSSP(%)
NCI	–25,90	–1,13
DD	–30,87	–0,81
AIDS	–25,31	–2,06
T58KV100E100	–24,57	–2,09
T114KV200E200	–19,88	–1,72
T10KV5KE14K	–27,44	0,90
AVERAGE	–25,66	–1,15

Table 9. Memory and run time improvement for the PDSSP employed Gaston algorithm

Dataset	Memory usage gaston with PDSSP(%)	Run time gaston with PDSSP(%)
NCI	–38,42	19,15
DD	–14,58	17,25
AIDS	–35,87	11,54
T58KV100E100	–36,70	17,55
T114KV200E200	–20,29	16,89
T10KV5KE14K	–14,03	11,44
AVERAGE	**–26,65**	**15,64**

V. Results and Discussion

Frequent subgraph mining is one of the most challenging problems in the graph-mining domain. This research provides the novel approach of PDSSP to minimize memory consumption of FSM algorithms.

To demonstrate the efficiency of PDSSP, experiments were performed on real and large synthetic datasets. The total run times and maximum memory consumption were compared with the original implementations. Our results suggest that PDSSP significantly decreases memory usage, although some delay may be introduced in the run time of welloptimized FSM implementations, such as Gaston.

Our PDSSP approach offers two contributions of a new Dynamic Sized Integer Type as a designed unsigned integer data type and a "Data structure packaging" component. For future work, we plan to use MapReduce and the MPI to improve the performance of our PDSSP embedded FSM algorithms.

References

[1] A. Thusoo, Z. Shao, S. Anthony, *et al.*, "Data warehousing and analytics infrastructure at Facebook", *ACM SIGMOD/PODS 2010 Conference*, Indianapolis, Indiana, USA, pp.1013–1020, 2010.

[2] L. Wang, Y. Xiao, B. Shao, *et al.*, "How to partition a billion-node graph", *2014 IEEE International Conference on Data Engineering*, Chicago, IL, USA, pp.568–579, 2014

[3] A. S. Muttipati and P. Padmaja, "Analysis of large graph partitioning and frequent subgraph mining on graph data", *International Journal of Advanced Research in Computer Science*, Vol.6, No.7, pp.29–40, 2015.

[4] G. V. Carlos and M. Esteban, "Comparative analysis of de Bruijn graph parallel genome assemblers", *2018 IEEE International Work Conference on Bioinspired Intelligence* (*IWOBI*), IEEE, pp.1–8, 2018.

[5] N. Talukder and M. J. Zaki, "A distributed approach for graph mining in massive networks", *Data Mining and Knowledge Discovery*, Vol.30, No.5, pp.1024–1052, 2016.

[6] G. Di Fatta and M. R. Berthold, "High performance subgraph mining in molecular compounds", *International Conference on High Performance Computing and Communications*, Springer, Berlin, Heidelberg, pp.866–877, 2005.

[7] A. Stratikopoulos, G. Chrysos, I. Papaefstathiou, *et al.*, "HPC-gSpan: An FPGA-based parallel system for frequent subgraph mining", *IEEE 2014 24th International Conference on Field Programmable Logic and Applications* (*FPL*), Munich, Germany, pp.1–4, 2014.

[8] C. C. Aggarwal, M. A. Bhuiyan and M. Al Hasan, "Frequent pattern mining algorithms: A survey", in: C. C. Aggarwal, J. Han (editors), *Frequent Pattern Mining*, Switzerland: Springer International Publishing, pp.19–64, 2014.

[9] D. C. Anastasiu, J. Iverson, S. Smith, *et al.*, "Big data frequent pattern mining", in: C. C. Aggarwal, J. Han (editors), *Frequent Pattern Mining*, Springer International Publishing Switzerland, pp.225–259, 2014.

[10] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining", in: *IEEE 2003 IEEE International Conference on Data Mining*, Melbourne, Florida, USA, pp.721–724, 2003.

[11] S. Nijssen and J. N. Kok, "The Gaston tool for frequent subgraph mining", *Electronic Notes in Theoretical Computer Science*, Vol.127, No.1, pp.77–87, 2005.

[12] X. Yan and J. Han, "CloseGraph: Mining closed frequent graph patterns", in: *ACM SIGKDD 2003 International Conference on Knowledge Discovery and Data Mining*, Washington, DC, USA, pp.286–295, 2003.

[13] K. Lakshmi and D. T. Meyyappan, "A comparative study of frequent subgraph mining algorithms", *International Journal of Information Technology Convergence and Services*, Vol.2, No.2, pp.23–39, 2012.

[14] C. Borgelt and M. R. Berthold, "Mining molecular fragments: Finding relevant substructures of molecules", *IEEE 2003 International Conference on Data Mining*, Melbourne, Florida, USA, pp.51–58, 2003.

[15] B. Guan, X. Z. Zan, B. Y. Xiao, *et al.*, "Detecting dense subgraphs in complex networks based on edge density coefficient", *Chinese Journal of Electronics*, Vol.22, No.3, pp.517–520, 2013.

[16] M. Kuramochi and G. Karypis, "Frequent subgraph discovery", *IEEE 2001 International Conference on Data Mining*, San Jose, California, USA, pp.313–320, 2001.

[17] N. Vanetik, E. Gudes and S. E. Shimony, "Computing

frequent graph patterns from semistructured data", *IEEE 2002 International Conference on Data Mining*, Maebashi City, Japan, pp.458–465, 2002.

[18] S. U. Rehman, S. Asghar and S. J. Fong, "Optimized and frequent subgraphs: How are they related?", *IEEE Access*, DOI:10.1109/ACCESS.2018.2846604, pp.37237–37249, 2018.

[19] S. Yang, R. Guo, R. Liu, *et al.*, "cmFSM: A scalable CPU-MIC coordinated drug-finding tool by frequent subgraph mining", *BMC Bioinformatics*, Vol.19, Article No.98, 2018.

[20] I. Horton, "Working with fundamental data types", in: S. Anglin (lead editor), *Beginning C++*, NY, USA, pp.55–77, 2014.

[21] D. A. Bader, H. Meyerhenke, P. Sanders, *et al.*, "Benchmarking for graph clustering and partitioning", *Encyclopedia of Social Network Analysis and Mining*, pp.73–84, 2012.

[22] Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A programmer's Perspective*, 3rd edition, Pearson, 2016.

[23] Microsoft, "Working with packing structures", *https://docs.microsoft. com/en-us/previousversions/ms253935*, 2017-12-8.

[24] Valgrind and Massif Visualizer tools, *http://valgrind.org/info/tools.htm*, 2017-12-8.

[25] Nikil Wale, Ian A. Watson and George Karypis, "Comparison of descriptor spaces for chemical compound retrieval and classification", *Knowledge and Information Systems*, Vol.14, pp.347–375, 2008.

[26] P. D. Dobson and A. J. Doig, "Distinguishing enzyme structures from non-enzymes without alignments", *Journal of Molecular Biology*, Vol.330, No.4, pp.771–783, 2003.

[27] D. Wajdi, A. Sabeur and N. E. Mephu, "MR-SimLab: Scalable subgraph selection with label similarity for big data", *Information Systems*, Elsevier, Vol.69, pp.155–163, 2017.

[28] Thoma M, Cheng H, Gretton A, *et al.*, "Discriminative frequent subgraph mining with optimality guarantees", *Statistical Analysis and Data Mining*, The ASA Data Science Journal, Vol.3, pp.302–18, 2010.

[29] Gaston official web site, *http://liacs.leidenuniv.nl/∼nijssensgr/gaston/index.html*, 2017-9-8.

[30] Wörlein M., Meinl T., Fischer I., *et al.*, "A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston", *European Conference on Principles of Data Mining and Knowledge Discovery*, Springer, Berlin, Heidelberg, pp.392–403, 2005.

BİLGİN Turgay Tugay (corresponding author) was born in Kirklareli, Turkey, in 1977. He is now an Associate Professor in Bursa Technical University. His current research interests include data mining, big data analysis, high performance computing.
(Email: turgay.bilgin@btu.edu.tr)

OĞUZ Murat was born in Sinop, Turkey, in 1977. He is now an Engineer in Microsoft Turkey Branch. His current research interests include data mining, big data analysis, high performance computing. (Email: murat.oguz@microsoft.com)