POLİTEKNİK DERGİSİ

JOURNAL of POLYTECHNIC

ISSN: 1302-0900 (PRINT), ISSN: 2147-9429 (ONLINE)
URL: http://dergipark.org.tr/politeknik

Sık alt çizge madenciliği algoritmalarının bellek gereksinimlerini en aza indirmek için yeni bir yaklaşım

A new approach to minimize memory requirements of frequent subgraph mining algorithms

Yazar(lar) (Author(s)): *Turgay Tugay BİLGİN[1], Murat OĞUZ[2]*

ORCID[1]: 0000-0002-9245-5728

ORCID[2]: 0000-0002-2757-5504

Bu makaleye şu şekilde atıfta bulunabilirsiniz(To cite to this article): Bilgin T. T. ve Oğuz M., "A new approach to minimize memory requirements of frequent subgraph mining algorithms", ***Politeknik Dergisi***, 24(1): 237-246, (2021).

Erişim linki (To link to this article): http://dergipark.org.tr/politeknik/archive

DOI: **10.2339/politeknik.678921**

A New Approach to Minimize Memory Requirements of Frequent Subgraph Mining Algorithms

Highlights

- *Frequent subgraphs*
- *Graph mining*
- *Space complexity*
- *Structure Packaging*
- *Graph Data Structures*

Graphical Abstract

In this study, a new approach called Predictive Dynamic Sized Structure Packing (PDSSP) have been proposed to minimize the memory requirement of FSM algorithms. Proposed approach redesigns the internal data structures of FSM algorithms without any algorithmic modifications.

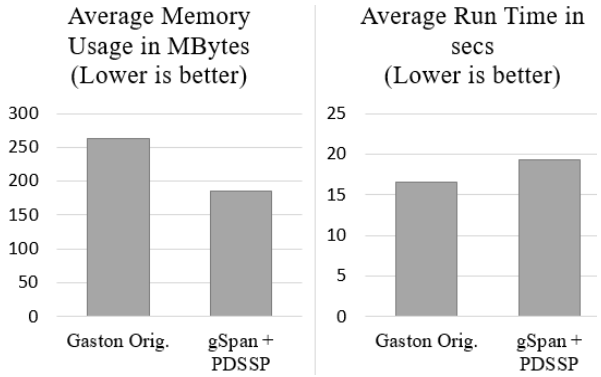

Figure. Avg. Memory usage (MBytes) and avg. Run Time durations (secs) comparison of Gaston to Gaston + PDSSP.

Aim

In this study, a new approach called Predictive Dynamic Sized Structure Packing (PDSSP) have been proposed to minimize the memory requirement of FSM algorithms.

Design & Methodology

Proposed approach redesigns the internal data structures of FSM algorithms without any algorithmic modifications.

Originality

PDSSP has two contributions. The first one is the Dynamic Sized Integer Type (ds_Int) which is a newly designed unsigned integer data type. The second contribution is "Data Structure packaging" component that uses a data structure packing technique which changes the behaviour of the compiler.

Findings

A number of experiments have been conducted to examine the effectiveness and efficiency of the PDSSP approach by embedding it into two state-of-art algorithms called gSpan and Gaston. Proposed implementation have been compared to the official one. Almost all results show that the proposed implementation consumes less memory on each support level.

Conclusion

Predictive Dynamic Sized Structure Packing (PDSSP) extensions can save memory and the peak memory usage may decrease up to 38% depending on the dataset.

Declaration of Ethical Standards

The author(s) of this article declare that the materials and methods used in this study do not require ethical committee permission and/or legal-special permission.

Sık Alt Çizge Madenciliği Algoritmalarının Bellek Gereksinimlerini En Aza İndirmek İçin Yeni Bir Yaklaşım

Araştırma Makalesi / Research Article

Turgay Tugay BİLGİN[1*], Murat OĞUZ[2]

[1]Mühendislik ve Doğa Bilimler Fakültesi, Bursa Teknik Üniversitesi, Bursa, TÜRKİYE
[2]Microsoft Türkiye, İstanbul, TÜRKİYE

(Geliş/Received : 24.01.2020 ; Kabul/Accepted : 13.03.2020)

ÖZ

Sık alt çizge madenciliği (SAÇM), çizge sınıflandırma ve çizge kümeleme için yaygın olarak kullanılan bir çizge madenciliği alt türüdür. Son on yılda, birçok verimli SAÇM algoritması geliştirilmiştir. Geliştirmeler genellikle algoritma yapısını değiştirerek veya paralel programlama teknikleri kullanarak zaman karmaşıklığını azaltmaya odaklanmıştır. SAÇM algoritmalarının çözülmesi gereken en önemli problemlerinden biri yüksek bellek tüketimidir. Bu çalışmada, SAÇM algoritmalarının bellek gereksinimini en aza indirmek için Öngörücü Dinamik Boyutlu Yapı Paketleme (ÖDBYP) adı verilen yeni bir yaklaşım önerilmiştir. Önerilen yaklaşım SAÇM algoritmalarının iç veri yapılarında herhangi bir algoritmik değişiklik yapmadan yeniden tasarlamaya olanak sağlamaktadır. Bu çalışma kapsamında geliştirilen ÖDBYP ile very madenciliği alanına iki önemli katkı sağlanmaktadır. Birincisi, yeni tasarlanmış işaretsiz bir tamsayı veri türü olan Dinamik Boyutlu Tamsayı Türüdür (ds_Int). İkinci katkı, derleyicinin davranışını değiştiren bir veri yapısı paketleme tekniği kullanan "Veri Yapısı paketleme" bileşenidir. ÖDBYP yaklaşımının etkinliğini ve verimliliğini, gSpan ve Gaston adlı güncel algoritmalara gömerek çeşitli deneyler gerçekleştirilmiştir. Çalışma kapsamında geliştirilen yöntem ile algoritmaların original halleri ile kıyaslanmıştır. Neredeyse tüm sonuçlar, önerilen uygulamanın her destek düzeyinde daha az bellek harcadığını göstermektedir. Sonuç olarak, ÖDBYP uzantıları bellek tasarrufu sağlayabilir ve veri kümesine bağlı olarak maksimum bellek kullanımı % 38 kadar düşürülebilmektedir.

Anahtar Kelimeler: Sık alt çizgeler, veri madenciliği, alan karmaşıklığı.

A New Approach to Minimize Memory Requirements of Frequent Subgraph Mining Algorithms

ABSTRACT

Frequent subgraph mining (FSM) is a subsection of graph mining domain which is extensively used for graph classification and graph clustering purposes. Over the past decade, many efficient FSM algorithms have been developed. The improvements generally focus on reducing time complexity by changing the algorithm structure or using parallel programming techniques. FSM algorithms have another problem to solve, which is the high memory consumption. In this study, a new approach called Predictive Dynamic Sized Structure Packing (PDSSP) have been proposed to minimize the memory requirement of FSM algorithms. Proposed approach redesigns the internal data structures of FSM algorithms without any algorithmic modifications. PDSSP has two contributions. The first one is the Dynamic Sized Integer Type (ds_Int) which is a newly designed unsigned integer data type. The second contribution is "Data Structure packaging" component that uses a data structure packing technique which changes the behaviour of the compiler. A number of experiments have been conducted to examine the effectiveness and efficiency of the PDSSP approach by embedding it into two state-of-art algorithms called gSpan and Gaston. Proposed implementation have been compared to the official one. Almost all results show that the proposed implementation consumes less memory on each support level. As a result, PDSSP extensions can save memory and the peak memory usage may decrease up to 38% depending on the dataset.

Keywords: Frequent subgraphs, data mining, space complexity.

1. INTRODUCTION

The size of the graphs used in the graph mining area is growing rapidly up to trillion edges [1]. In the year 2015, one of the largest reported experiments with a real-world graph involved over 1.5 trillion edges [2]. Today, the search engines have new infrastructures to support the bigger web graphs, which has over a trillion vertices [3]. Another important area for graph mining is genome sequencing. Recent works have attempted to solve the genome assembly problem by using graphical representations for genomes. An example of a big genome graph is the de Bruijn graph which was generated based on k-mers calculation in Velvet algorithms. The maximum number of nodes in that graph was about 4 million [4]. The most widely used application domain of graph mining is the graph pattern mining problems, which is also called frequent subgraph mining (FSM). It is a well-studied problem with numerous applications in areas such as computational

*Sorumlu Yazar (Corresponding Author)
e-posta : turgay.bilgin@btu.edu.tr

chemistry, bioinformatics, and social networks [5]. FSM focuses on the enumeration of the frequent subgraphs, either in a single graph or in a graph database. During enumeration of the subgraphs, memory requirements get exponentially larger compared to the input size of the input data. For this reason, various methods have been used to optimise memory and CPU usage of the FSM algorithms. These improvements have focused on distributing the computational power and memory requirements to different nodes by parallelization techniques like Map/Reduce, Message Passing Interface (MPI). Using compression techniques, changing the data-storing structure or format with new technologies like Cassandra and Hadoop are other solutions that work. All these techniques are used to modify and optimize the FSM algorithms.

Using high-performance computing (HPC) for FSM is a timely subject, especially on symmetric multiprocessing (SMP) systems. In SMP systems, there is one shared memory used by more than one core. Any FSM algorithm paralleled on SMP systems needs more memory than a single-threaded counterpart for increasing the level of parallelization due to data sharing needs between threads. As a result, the space complexity is getting a bigger problem for these type of systems [6].

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

The remaining part of this paper is organized as follows: we define some FSM preliminaries and the problem in Section 2. In section 3, we discuss our method, in section 4 we have performed an experimental study. The last section draws the conclusions.

2. DEFINITIONS

A graph is constructed by pairing a set of vertices V and a set of edges E. The graph 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 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 has labels, this mapping must also be between the labels on the vertices and edges.

Subgraph $G_2 = (V_2, E_2)$ of another graph $G_1 = (V_1, E_1)$ is that $V_2 \subseteq V_1$ and $E_2 \subseteq E_1\hat{}(v1, v2) \in E_2 \rightarrow V_1 \in V_2$ and $v_2 \in V_2$ can be found, as in Fig. 1.

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

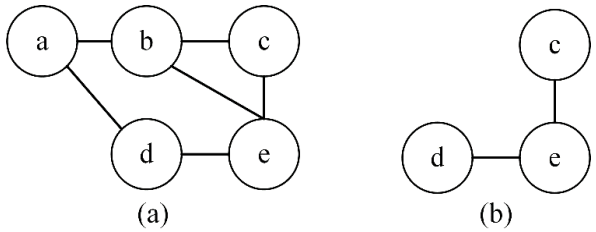

Figure 1. (a) represents a graph, (b) represents a subgraph of (a).

Subgraph isomorphism occur between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is when you find an isomorphism between G_2 and a subgraph of G_1, that is, to determine whether or not G_2 is included in G_1.

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

Frequent subgraph mining is the discovery of subgraphs of the given set of graphs [9]. Let Fig. 2 (a) and Fig. 2 (b) be the given graphs. An example of frequent subgraph would be the graph shown in Fig. 2 (c).

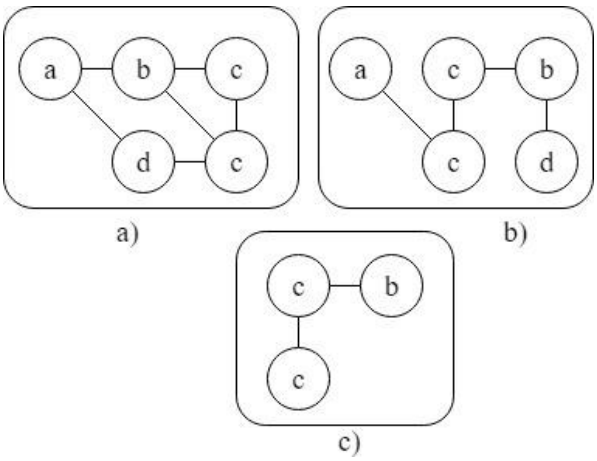

Figure 2. (a), (b) are input graphs, (c) is a frequent subgraph.

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

The second approach employs pattern-growth algorithms that start from an initial edge and extend the graph by directly adding a new edge in every possible position, then checking whether or not this graph supports the threshold. Well-known pattern-growth based graph mining algorithms are gSpan, MoFa, SPIN, and Gaston [18][19].

In this work, two algorithms gSpan and Gaston are used to test our approach. **gSpan** (graph-based Substructure pattern) uses DFS-codes for presenting and storing the graphs. Searching and comparing frequent subgraphs for isomorphism check test is done via DFS code tree. With this feature, gSpan does not require any candidate pattern generation. It generates all exact frequent subgraphs.

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

All pattern growth algorithms generate duplicated candidates during the enumeration process. In gSpan, the duplicated candidates are non-minimal codes. Instead of calculating the minimum DFS code of *s* from all possible DFS codes, picking up the smallest one and comparing it against *s*, gSpan defines a more efficient function `isMin(s)` in `Subgraph_mining` method. A heuristic search was designed using the DFS lexicographic order. Whenever some prefix of a DFS is generated and it is less than *s*, then *s* is not minimal and the search concludes.

Table 1. Pseudo-code of the gSpan algorithm

Algorithm gSpan

Metod 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);
S^1:= 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∈GS, e∈E(h)}*;
Subgraph_mining(*GS, FS, g*);
GS := *GS* - *e*;
if |*GS*| < *minSup*
break;

Metod 2: Subgraph_mining(GS, FS, g)
if $g \neq min(g)$
return;
FS := FS ∪{g};
enumerate *g* in each graph in *GS* and count *g*'s children;
for each *c* (child of *g*) **do**
if *support(c)* ≥ *minSup*
Subgraph_mining(GS, FS, c);

Enumeration of g: Finding all the exact positions of *g* in another graph

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

Table 2. Pseudo-code of Gaston algorithm

Algorithm 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`* ;
}
}

Many memory-based algorithms have been proposed to discover the frequent graphs. In this work, we use the Gaston algorithm to find the set of frequent graphs. The Gaston (Graph sequence tree extraction) algorithm is based on the observation that most frequent substructures in practical graph databases are actually free trees and employs a highly effective strategy to enumerate the frequent free trees first. Gaston stores all embeddings (both nodes and edges), to generate only refinements that actually appear and to achieve fast isomorphism testing. It firstly checks paths and trees, subgraph isomorphism test is done as the last job. Gaston only outputs the cycled graphs. So that, Gaston works faster than both gSpan, FFSM or Mofa.

Table 2 gives an outline of the Gaston algorithm. Let *P(U)* be a subgraph found in the *U*. The first line finds all the 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 types of *G`* and the extended edges, the algorithm will decide to find paths, trees or cyclic graphs in the database. If-else sections perform these operations in pseudo-code given in Table 2.

3. PREDICTIVE DYNAMIC SIZED STRUCTURE PACKING (PDSSP)

In this section, we have described our contribution to the standard FSM algorithm. FSM algorithm read the datasets from an external resource, process the data and

write them into the disk. Basic flowchart of standard FSM algorithm is depicted in Fig. 3 (a).
Our proposed PDSSP approach redesigns the internal data structures of the FSM algorithm without any algorithmic modifications, therefore it acts as an extension to the FSM algorithm. After applying the PDSSP to standard FSM implementation (FSM_PDSSP), modifications are depicted in Fig. 3 (b).

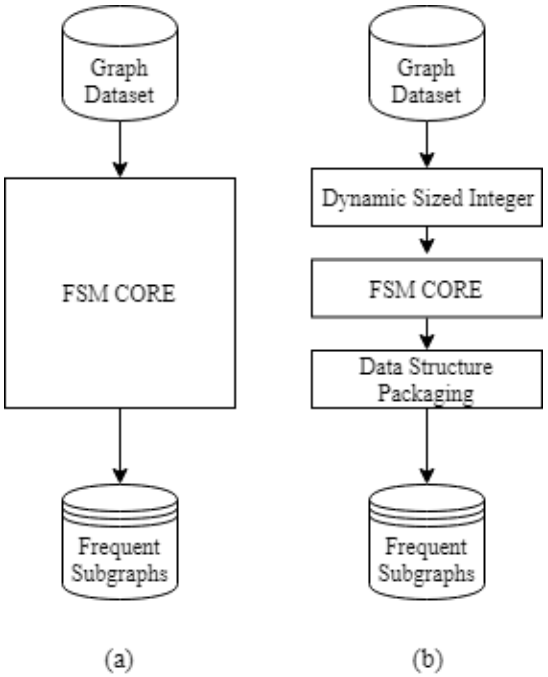

Figure 3. Flowcharts of standard FSM and PDSSP_FSM.

Generally, all FSM implementations use a fixed-type integer for all variables. The main idea behind PDSSP is that, if the maximum value to be stored in the integer-based variables could be estimated, then varying-length integer data types could be employed. In this way, memory requirements of FSM implementations may be reduced.
We have analyzed the FSM implementations and we noticed that, due to the structure of the input data, only unsigned integers are employed. Table 3 shows the standard unsigned integer types with their storage sizes and value ranges [20]. We have observed that choosing improper integer data types causes high memory usage. Consequently, we have focused on finding a solution to create varying-length unsigned integer data type.
FSM_PDSSP has two contributions. The first one is the Dynamic Sized Integer Type (ds_Int) which is a newly designed unsigned integer data type that has a varying capacity range from 0 to 2^{64} . The capacity of ds_Int could be changed on demand. The second contribution is "Data Structure packaging" component that uses a data structure packing technique which changes the behaviour of the compiler. The details of the contributions are given in the following sections.

Table 3. 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})

Table 4. 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>
```

FSM implementations work in such a way that they store maximum possible values of graph features in memory as integer data type regardless of the number of samples, the number of edges and vertices in the data set. PDSSP is designed to convert this static memory usage into a dynamic state. Our proposed model has 2 stages. The flowchart of our implementation has been given in Fig 4. The first stage is Predictive Version Switch (PVS) that scans the input dataset file which stores the graphs digitized in DIMACS [21] format as shown in Table 4. On the "analyze dataset" module in Fig 4, PVS determines the proper integer value range and then chooses the appropriate precompiled PDSSP version to execute. FSM_PDSSP is an FSM version of which primitive integer data types replaced by dynamic length ds_Int data type. An appropriate size of ds_Int is determined by the range of values given in Table 7.
C/C ++ languages allow changing primitive integer data types only at compile time, 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, according to the number of edges, vertices and graph size.
If the determiner module cannot determine which version to execute, then the original binary executed. In the end, the detected frequent sub-graphs are written to the disk.

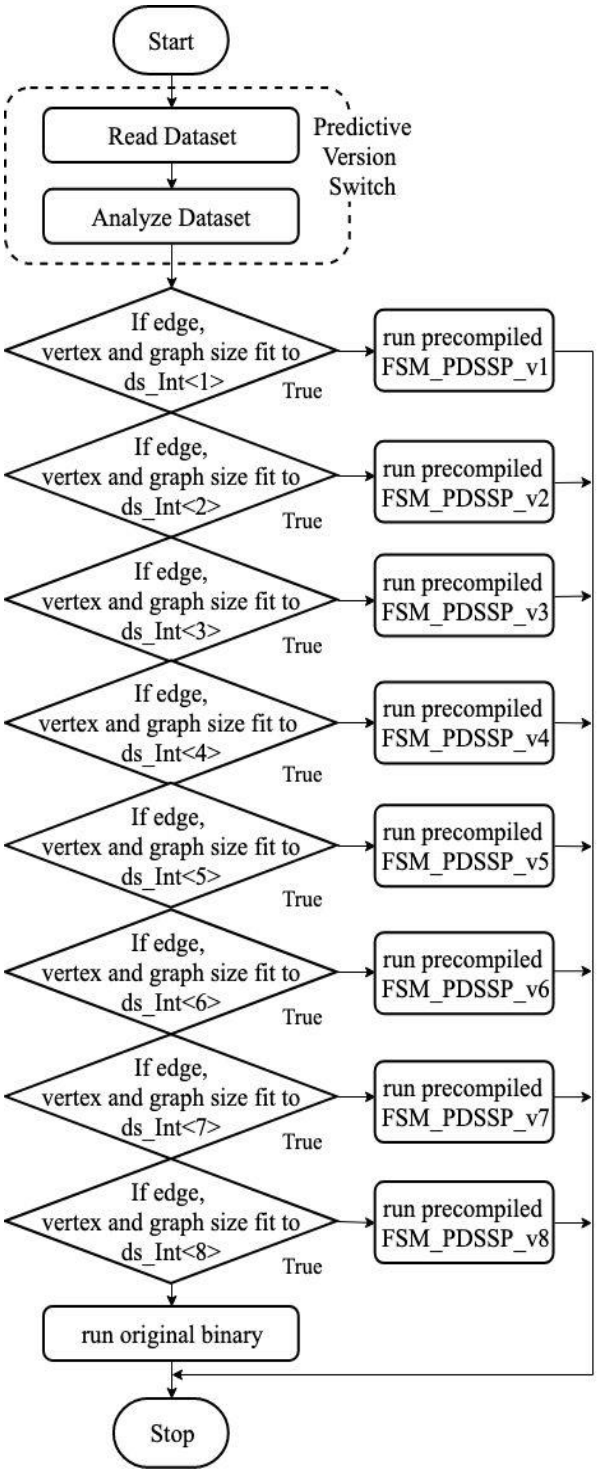

Figure 4. Our proposed flowchart to use FSM with PDSSP.

3.1. Predictive Version Switch (PVS)

PVS is a preprocessing module that has been developed as independent software. In this study, it has been modified to meet the requirements of gSpan and Gaston. PVS takes the following input parameters: input dataset file, the name of the algorithm, minimum support level and output file. The pseudo-code of PVS is given in Table 5.

Table 5. Pseudo-code of Predictive Version Switch (PVS)

Algorithm 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

PVS scans the input dataset and finds the maximum integer values that will be used to store transaction count, vertex number, edge number and label numbers. After the decision process, PVS runs the appropriate PDSSP version.

3.2. Dynamic Sized Integer Type (ds_Int)

A newly designed unsigned integer variable type, ds_Int, has been developed as the part of the solution. As can be seen in the pseudo-code of ds_Int implementation given in Table 6, the main idea is to store unsigned integer values in an unsigned array by using bit-shifting operations. With this method, ds_Int can also support bigger numbers than 2^{64} with minimal algorithmic modification. For this work, we have limited it to 2^{64} in order to compare with the unsigned long integer type.

Table 6. Pseudo-code of ds_Int algorithm

Algorithm 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 *OutputData* with *OutputData* & (*i* × 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* × 8-byte right shifted *InputData)*
end for
end function

The difference between ds_Int and 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 C/C++ languages, the primitive unsigned integers typically require 1, 2, 4 or 8-bytes, but not 3, 5, 6 or 7 bytes. Dynamic sized ds_Int enables the programmer to define 3,5,6 or 7 bytes integers, therefore a significant amount of memory depending on the dataset may be saved.

Table 7. 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

A comparison of ds_Int and standard unsigned integer data types and storage savings are shown in Table 7. As shown in the table, when the value range upper-limit gets higher, especially when it is greater than 2^{32}, memory space savings increases. In order to demonstrate the strength of ds_Int employment, we may give an example. Assume that there are 2^{22} integer items in an array and the maximum value which will be stored is 2^{35}. If standard integer types are used to store the array, an unsigned long integer type has to be preferred due to its supported size limit. If we calculate the memory space requirement for this operation, the amount will be $\left(\sum_{n=1}^{2^{22}} 8\right)/\ 1024\ = 32{,}768\ KB$. Whereas, when ds_Int<5> integer type is used for the same operation, the memory space requirement will be $\left(\sum_{n=1}^{2^{22}} 5\right)$ $/1024 = 20{,}480\ KB$. It can be clearly stated that, using ds_Int results in a reduction in memory requirement up to $\left(1 - \frac{20480}{32768}\right) = 37\%$.

3.3. Data Structure Packing

Data structure packing is the last and fundamental part of the PDSSP approach. Before explaining, it is necessary to understand how data is stored and accessed in the memory.

In computer systems, stored data in memory has two properties. The first one is its value and the second is its 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. Generally, word size is 4 in most cases. If the size of data is smaller than a word size, some extra empty spaces are added to the end of the data for data alignment. This phenomenon is called "padding".

The compiler padding is illustrated in the following example. Here, an `int` is assumed to be 4 bytes and a `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 the padding, the addresses of the data in this structure are evenly divisible by 4. This is called structure member alignment. Obviously, the size of the structure in memory grows as a consequence.

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

In this study, we have created a new data type called dynamic sized integer (ds_Int). It is fully adjustable from 1-byte to 8-byte storage sizes and it avoids misaligned data access by means of compiler alignment options. In our C/C++ implementation, we use "#pragma" preprocessor directive. The pragma directive makes the compiler work with the specified structure packing size when it is activated [23].

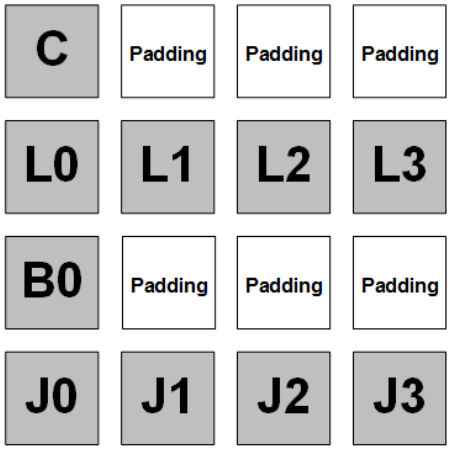

Figure 5. Memory alignment and padding of struct `mydata`.

Predictive Version Switch (PVS) scans the input dataset file. On the "analyze dataset" step in Fig 4, PVS finds the integer value range and then determines the appropriate ds_Int type. As a result, If 3-byte fits for the integer data type, then our proposed "data structure packaging system" will select the ds_Int <3> data type.

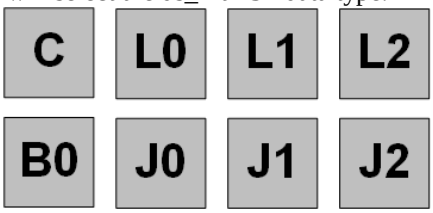

Figure 6. Memory alignment when ds_Int<3> used.

Fig 6 illustrates the new memory alignment when integer data type shrinks to 3 bytes. As a result, totally 6 bytes of waste has been saved by the help of our proposed method.

3.4. Embedding PDSSP into FSM Algorithms

As we mentioned in the previous section, PDSSP is an extension to FSM implementations. In order to embed PDSSP into an FSM implementation, one should perform memory profiling to determine the most memory demanded data structures. If any greedy data structures found, our proposed ds_Int types may replace these.
In this study, we have used Valgrind and Massif Visualizer tools [24] to profile the memory consumption of gSpan and Gaston implementations. By means of these tools, the most memory demanding data structures have been determined. The greedy data structures are replaced with ds_Int type. The original data structures and replaced ones are shown in Table 8.

Table 8. 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,frmnodeid;`
	`  //other codes`
	`};`
	`#pragma pack()`

4. EXPERIMENTS

We conducted experiments to examine the effectiveness and efficiency of the PDSSP approach by embedding it into two state-of-art algorithms called gSpan and Gaston. We compared our proposed FSM_PDSSP implementations to the official implementations.

4.1. Data Sets

To evaluate the performance of the PDSSP approach, we have conducted experiments with three real-world data sets: Anti-cancer screen datasets (NCI) [25], Dobson and Doig (DD) molecule data set [26] and AIDS antiviral screen data set (AIDS) [27]. In addition to the real datasets, we have also generated three synthetic datasets named T10KV5KE14K, T58KV100E100 and T114KV200E200. The metadata of the real and synthetic databases are given in Table 9.

Table 9. Benchmark datasets and their characteristics.

Dataset $G^{'}$	**$\|G^{'}\|$**	**$\|V_{MAX}(G^{'})\|$**	**$\|E_{MAX}(G^{'})\|$**	**$\|L_{MAX-V}\|$**	**$\|L_{MAX-E}\|$**
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

$|G^{'}|$: the total number of graphs in the dataset
$|V_{MAX}(G^{'})|$: maximum number of vertices in any graph
$|E_{MAX}(G^{'})|$: maximum number of edges in any graph
$|L_{MAX-V}|$: maximum number of vertex labels

$|L_{MAX-E}|$: maximum number of edge labels

4.2. Test Environment

In this study, gSpanCORK an implementation of the gSpan algorithm which is developed by Thoma, Marisa, et al. has been employed. It has been downloaded from the web page provided in their article entitled "Discriminative frequent subgraph mining with optimality guarantees" [28]. Gaston was downloaded from the Gaston official web site [29]. Since both implementations are open source and coded in C/C++ programming language, we have easily embedded our proposed PDSSP implementation into them. All source codes are 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 PDSSP binaries. The test hardware had 2-core Intel Xeon CPU E5-2670 2.60GHz processors and 4 GB RAM memory. The implementations were developed to run in single threaded mode.

4.3. Experimental Results

We have executed our implementation and the original one on the benchmark datasets given in Table 9. Original implementations of gSpanCORK and Gaston are compared to the corresponding FSM_PDSSP implementations whose are called gSpan+PDSSP and Gaston+PDSSP. Total running times and the maximum memory consumptions (peak memory) are collected for various support levels. All tests are carried out three times to make sure that they are consistent.

Table 10. Comparison of gSpan and gSpan+PDSSP on benchmark datasets.

Dataset	Supp. (%)	Mem. Usage (MB's)		Run Time (Sec)	
		gSpan Orig.	gSpan + PDSSP	gSpan Orig.	gSpan + 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**
T58KV100E10	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**
T114KV200E200	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 experiments are shown in Table 10 and Table 11. For each benchmark, memory usage in megabytes and run time in seconds are given. "gSpan Orig." column corresponds to the original implementation and "gSpan+PDSSP" column correspond to the PDSSP employed version. In Table 10 and Table 11, bold values indicate better results.

Tests are repeated for various support levels ranging from 5% to 30%. At lower support levels, the run times are longer as expected. That's why it requires more time to find frequent subgraphs at a low support level and consumes more memory. Owing to the subgraph isomorphism tests performed during the frequent subgraph mining, run time and the allocated memory increases exponentially with the size of the dataset.

Table 11. 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
T1	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**

Table 12 and Table 13 show the overall memory and run time improvements achieved through our PDSSP approach. As shown in Table 12, the memory usage is significantly reduced in all the cases. The memory savings range from 27.44% to 19.88% on the 6 of the 6 benchmark datasets. The average improvement is 25.66%. The average memory usage and run time durations for all cases are given in Fig.7.

Table 12 and the charts in Fig 7. indicates that improvement is accomplished by our PDSSP approach. On the other hand, the run time is slightly better than the original implementation. The average run time of 6 benchmarks denotes that our PDSSP employed gSpan algorithm requires 1.15% less time to run.

Table 12. 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**

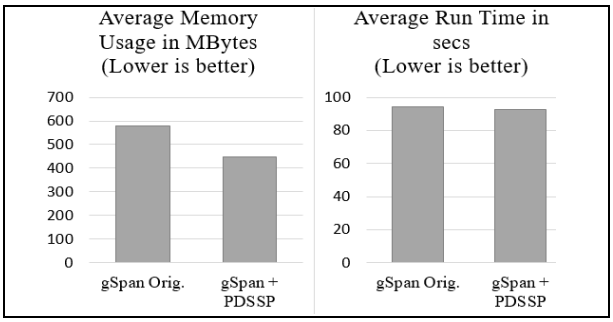

Figure 7. Avg. Memory usage (MBytes) and avg. Run Time durations (secs) comparison of gSpan to gSpan + PDSSP.

Table 13. 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**

The results of our second set of experiments are shown in Table 13. As the previous experiment with gSpan with PDSSP, the memory usage is also significantly reduced in Gaston with PDSSP. The reduction ranges from 38.42% to 14.03% on the 6 of the 6 benchmark datasets. The average improvement is 26.65%. The average memory usage and run time durations for all cases of Gaston and Gaston + PDSSP are given in Fig.8.

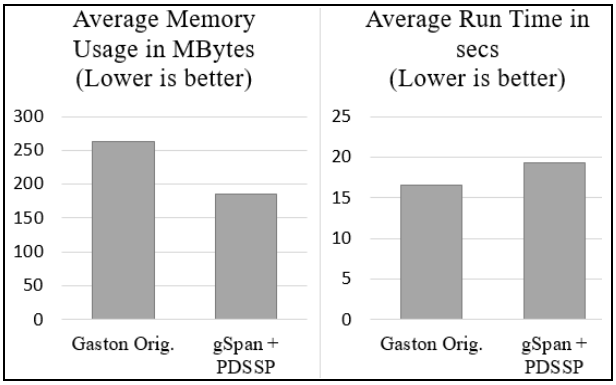

Figure 8. Avg. Memory usage (MBytes) and avg. Run Time durations (secs) comparison of Gaston to Gaston + PDSSP.

In this case, total run time on all experiments are slightly worse than the original implementation. The average run time is 15.16% longer than the original. gSpan uses adjacency list for graph representation, whereas Gaston uses a hash table. This approach makes the Gaston the fastest out of 4 algorithms named MoFa, gSpan, FFSM and Gaston [30]. We have used gSpan as "fairly optimized" representative, whereas the Gaston has been chosen as "well optimized" representative to demonstrate our approach. Our PDSSP approach may cause some delay in Gaston which is the worst case for our approach. We may say that PDSSP approach may cause delay at most 15% in the worst case. In general, an actual delay will be much smaller. A delay of 15% is not bad, since Gaston algorithm may operate up to 50% faster than gSpan. The corresponding lines of Table 10 and Table 11 may be used to compare 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. Almost all results show that our proposed PDSSP implementation consumes less memory on each support level. The experimental results show that FSM_PDSSP can save memory and the peak memory usage decreases dramatically up to 38% depending on the dataset.

5. RESULTS AND DISCUSSION

Frequent subgraph mining is one of the most challenging problems in the graph-mining domain. This article

provides a novel approach to minimize the memory consumption of FSM algorithms. We call our approach as Predictive Dynamic Sized Structure Packing (PDSSP). In order to demonstrate the efficiency of PDSSP, a number of experiments have been carried out on both real-life datasets and large synthetic datasets. Total run times and the maximum memory consumption (peak memory) are compared with the original implementations. The experimental results clearly stated that PDSSP can significantly decrease memory usage. There may be some delay in the total run time of some well-optimized FSM implementations such as Gaston.

Our PDSSP approach has two contributions. The first one is the Dynamic Sized Integer Type (ds_Int) which is a newly designed unsigned integer data type. The second contribution is "Data Structure packaging" component that uses a data structure packing technique which changes the behaviour of the compiler.

As future work, we are planning to use Map/Reduce and Message Passing Interface (MPI) in order to improve the overall performance of the PDSSP embedded FSM algorithms.

ETİK STANDARTLARIN BEYANI (DECLARATION OF ETHICAL STANDARDS)

Bu makalenin yazar(lar)ı çalışmalarında kullandıkları materyal ve yöntemlerin etik kurul izni ve/veya yasal-özel bir izin gerektirmediğini beyan ederler.

REFERENCES

[1] Burkhardt P., Waring C., "An NSA big graph experiment". ***In presentation at the Carnegie Mellon University SDI/ISTC Seminar***, Pittsburgh, USA, (2013).

[2] Fleury E., Lattanzi S., Mirrokni V., Perozzi B., "ASYMP: fault-tolerant mining of massive graphs." ***arXiv preprint*** arXiv:1712.09731, (2017).

[3] Muttipati A. S., Padmaja P., "Analysis of large graph partitioning and frequent subgraph mining on graph data". ***International Journal of Advanced Research in Computer Science***, 1: 6-7, (2015).

[4] Carlos G. V., Esteban M., "Comparative Analysis of de Bruijn Graph Parallel Genome Assemblers", ***2018 IEEE International Work Conference on Bioinspired Intelligence (IWOBI),*** 1-8, (2018).

[5] Talukder N., Zaki, M. J., "A distributed approach for graph mining in massive networks". ***Data Mining and Knowledge Discovery,*** 30: 1024-1052, (2016).

[6] Di Fatta G., Berthold M. R., "High performance subgraph mining in molecular compounds". ***In International Conference on High Performance Computing and Communications***, 866-877, (2005).

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

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

[9] Anastasiu D. C., Iverson J., Smith S., Karypis G., "Big data frequent pattern mining". In: Frequent Pattern Mining. ***Switzerland: Springer International Publishing***, 225–259, (2014).

[10] Yan X., Han J., "gSpan: Graph-based substructure pattern mining". ***In: IEEE 2003 International Conference on Data Mining,*** 721–724, (2003).

[11] Nijssen S., Kok J. N., "The Gaston tool for frequent subgraph mining". ***Electronic Notes in Theoretical Computer Science,*** 127: 77–87, (2005).

[12] Yan X., Han J., "CloseGraph: Mining closed frequent graph patterns". ***In: ACM SIGKDD 2003 International Conference on Knowledge Discovery and Data Mining,*** 286–295, (2003).

[13] Lakshmi K., Meyyappan D.T., "A comparative study of frequent subgraph mining algorithms". ***IJITCS 2012,*** 2: 2, (2012).

[14] Borgelt C., Berthold M. R., "Mining molecular fragments: Finding relevant substructures of molecules". ***In: IEEE 2003 International Conference on Data Mining,*** 51–58, (2003).

[15] Guan B., Zan X. Z., Xiao B.Y., Ma R. N., Zhang F.Y., and Liu W. B., "Detecting dense subgraphs in complex networks based on edge density coefficient", ***Chinese Journal of Electronics***, 22: 517–520, (2013).

[16] Kuramochi M., Karypis G., "Frequent subgraph discovery". ***In: IEEE 2001 International Conference on Data Mining,*** 313–320, (2001).

[17] Vanetik N., Gudes E., Shimony S. E., "Computing frequent graph patterns from semistructured data". ***In: IEEE 2002 International Conference on Data Mining,*** 458–465, (2002).

[18] Rehman S. U., Asghar S., and Fong S. J., "Optimized and Frequent Subgraphs: How Are They Related?". ***IEEE Access,*** 6: 37237-37249, (2018).

[19] Yang S., Guo R., Liu R., Liao X., Zou Q., Shi B. and Peng S., "cmFSM: a scalable CPU-MIC coordinated drug-finding tool by frequent subgraph mining". ***BMC bioinformatics***, 19: 98, (2018).

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

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

[22] Bryant R. E., David Richard O. H., "Computer systems: a programmer's perspective." ***Upper Saddle River: Prentice Hall***; (2003).

[23] https:/docs.microsoft.com/en-us/previous-versions/ms253935

[24] "Valgrind and Massif Visualizer tools", http://valgrind.org/info/tools.html, (2017).

[25] Wale N., Ian A. W., and Karypis G., "Comparison of descriptor spaces for chemical compound retrieval and classification." ***Knowledge and Information Systems***, 347-375, (2008).

[26] Dobson P. D., Doig A. J., "Distinguishing enzyme structures from non-enzymes without alignments". ***Journal of Molecular Biology***, 330: 771–783, (2003).

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

[28] Thoma M., Cheng H., Gretton A., Han J., Kriegel H. P., Smola A., Song L., Yu P. S., Yan X., Borgwardt K. M., "Discriminative frequent subgraph mining with optimality guarantees". ***Statistical Analysis and Data Mining. The ASA Data Science Journal;*** 3: 302–318, (2010).

[29] http://liacs.leidenuniv.nl/~nijssensgr/gaston/index.html

[30] Wörlein M., Meinl T., Fischer I., and Philippsen, M., "A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston.", ***In: European Conference on Principles of Data Mining and Knowledge Discovery***, 392-403, (2005).