A Parallel Algorithm for Subgraph Isomorphism

Vincenzo Carletti(✉), Pasquale Foggia, Pierluigi Ritrovato, Mario Vento, and Vincenzo Vigilante

Department of Information Engineering, Electrical Engineering and Applied Mathematics, University of Salerno, Fisciano, Italy
{vcarletti,pfoggia,pritrovato,mvento,vvigilante}@unisa.it
http://mivia.unisa.it

Abstract. In different application fields, such as biology, databases, social networks and so on, graphs are a widely adopted structure to represent the data. In these fields, a relevant problem is the detection and the localization of structural patterns within very large graphs; such a problem, formalized as subgraph isomorphism, has been proven to be NP-Complete in the general case. Moreover, the continuously growing size of the graphs to face, actually of hundred thousands of nodes, is making the problem even more challenging also for the most efficient algorithms in the state of the art, requiring days or weeks of computational time. This huge amount of time is also consequence of the fact that most of the algorithms do not exploit any kind of parallelism, even if the problem is suitable to be solved adopting parallel approaches. In this paper we present a new parallel algorithm for subgraph isomorphism, namely VF3P, based on a redesign of the well known algorithm VF3. The effectiveness of VF3P has been experimentally proven on a publicly available dataset of very large graphs, confirming that the algorithm is able to efficiently scale w.r.t. the number of used CPUs without affecting the memory usage.

Keywords: Exact graph matching · Subgraph isomorphism · Parallel algorithms · VF3

1 Introduction

Graphs are discrete mathematical structures representing objects in terms of their parts and the relationships among these parts, using abstractions called nodes and edges respectively. Such a representation is much more expressive than the vector-based one, but it requires more complex algorithms, and thus a higher computational effort, also to perform simple operations like evaluating the similarity between two objects. Nevertheless, there are several cases where graphs are preferred to vectors because the latter are ineffective to model the complexity of the objects especially when these are composed by parts suitably

D. Conte et al. (Eds.): GbRPR 2019, LNCS 11510, pp. 141–151, 2019.
https://doi.org/10.1007/978-3-030-20081-7_14

interconnected each other and the application at hand exploits this relevant structural information [16,18,26].

Nowadays, the field of social networks [15,27], databases, semantic web and biology require to use bigger and bigger structural information, typically represented in term of graphs [3,21]. Among them, the latter is undoubtedly the most promising and challenging area [5,9,13,14], where many biological entities are naturally represented as graphs; moreover, the quantity of data generated every year and needed to be analyzed, is enormous. Noteworthy examples are molecular and protein structures, interaction networks and more recently the genome, that for many years has been represented as a string of bases [4,23].

In this context graph matching algorithms play an important role because they allow to perform the basic operations required to apply pattern recognition methods, such as the computation of the similarity (i.e. the distance) or the search for a structural pattern.

It is important to say that all the previously cited fields provide, year by year, new challenges to graph matching algorithms, due to the continuously growing size of the graphs they require to deal on. It it important to note that, actually, even a graph of thousand nodes is considered small in many cases, for instance when working with the genome of an individual that is composed of billions of bases. Therefore, even a frequent operation like searching for a pattern structure inside a graph, namely the subgraph isomorphism problem, becomes very time expensive also for the most efficient algorithms in the state of the art.

The current graph matching algorithms have been generally designed according to a sequential computational paradigm, even if many operations required by them can be potentially done in parallel; indeed, analyzing the literature it is possible to find extremely efficient algorithms, such as VF3 [8,11,12,14] and RI [5], able to work with graphs of thousands of nodes using a very limited quantity of memory and CPU, but requiring a large amount of time when the size and the density of the graphs increase (e.g. weeks of computational time).

Another important evidence of the need for parallel graph matching algorithms, is the growing interest in parallelizing the computation of the graph edit distance recently arisen in the scientific community [1,2,6,24].

Realizing efficient parallel graph matching algorithms is not as easy as it would appear. Indeed, starting from a sequential algorithm and making in parallel some of its steps and procedures is, in most of the case, useless. The big challenge is to design the algorithm so as to take full advantage form a specific parallel architecture, such as multicore systems using CPU, GPU, clusters and so on. Distributed subgraph isomorphism methods to deal with very large graphs have been proposed in [7,28], but there still remains the problem of reducing the high communication cost among the nodes of the cluster. Concerning the GPUs, some papers [20,29] have proposed interesting performance analysis on graph matching algorithms for GPU highlighting the bottlenecks and the reasons why graph algorithms are not able to exploit the architecture CPU-GPU at the best. A parallel approach on multicore CPUs has been recently proposed by McCreesh et al. [22]. The authors have proposed a simple parallel constrain programming

approach based on LAD [25]; they have focus the attention on specific parallelizable steps and have presented an analysis limited to the execution time, without discussing efficiency and speedup.

In this paper we propose a parallel algorithm for multicore CPUs obtained from VF3-Light [11] by realizing a state-level parallelization. The effectiveness of our proposal have been proved by analyzing the memory requirements, the speed-up and the efficiency with respect to the original sequential algorithm.

2 VF3-Light: The Sequential Algorithm

In this section, we briefly present some fundamental concepts on graph matching and VF3-Light required to understand the design choices discussed in the successive sections, the reader who is interested in deepening the algorithm is referred to [8,11].

2.1 Graphs and Graph Matching

A graph is an ordered pair $G = (V, E)$ where V and $E \subset V \times V$ are the set of nodes and edges respectively. Given a node $u \in V$, the set of its *successors* (the nodes connected to u by outgoing edges) is denoted as $\mathcal{S}(u)$, while the set of its *predecessors* (the nodes connected by incoming edges) is denoted as $\mathcal{P}(u)$. In a more general definition, graphs can carry also *attributes* or *labels* attached to their nodes and edges. Hence, two additional sets are considered: the set of node labels L_v and the set of edge labels L_e; two labeling functions, $\lambda_v : V \to L_v$ and $\lambda_e : E \to L_e$, are used to associate each node or edge to the corresponding label.

Considering two graphs, namely $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, graph matching is the problem of finding a function $M : V_1 \to V_2$, namely the *mapping function*, satisfying some structural constraints. In the case of subgraph isomorphism [16], the constraints are that M is injective and *structure preserving*, i.e. the nodes put in correspondence must have the same structure considering both the presence and the absence of edges. It is important to note that the mapping function is not unique, but the problem can have several distinct solutions where the nodes in V_1 are mapped to different subsets of nodes in V_2. In general, we are not interested in finding the first solution only, but all the possible ones.

2.2 VF3-Light

VF3-Light is the most recent successor of the well-know algorithm for subgraph isomorphism VF2 [17]; it has been proposed in [11] as a lightened version of VF3 [8] where some of the heuristics, required to deal with large and dense graphs, have been relaxed to reduce the overall computational time when facing small or sparse graphs. All the algorithms, designed from VF2, share the same structure and use a depth-first approach search (DFS) over a tree-structured search space of states. Each *state* s_n represents a partial mapping $M(s_n)$ between

the nodes in V_1 and those in V_2. Two additional sets $M_1(s_n) \subseteq V_1$ and $M_2(s_n) \subset V_2$ are used to represent the nodes of V_1 and V_2, respectively, that are in $M(s_n)$.

That said, the algorithm starts from a *root state* s_0 where the mapping $M(s_0)$ is empty and proceed until it reaches a *leaf state* s_l whose mapping $M(s_l)$ is *complete*, i.e. it involves all the nodes in V_1. The exploration from the root to a leaf proceed by extending the nodes involved in the mapping M; at each state s_n a new one is generate by adding to the mapping $M(s_n)$, a new couple of nodes $u_n \in V_1$, $v_n \in V_2$ that are not yet in $M(s_n)$. Only some of the couples are feasible to generate a state that is *consistent* with the constrains of the subgraph isomorphism and so are used to generate a new state. Finally, the leaves are also consistent, namely the *goal states*, represent the solutions of the problem.

In order to avoid an expensive bind search over the whole search space, VF3-Light uses the *feasibility rule* (Eq. 1) to explore the sub-space composed of only *consistent* states.

$$\texttt{IsFeasible}(s_n, u_n, v_n) = F_s(s_n, u_n, v_n) \wedge F_c(s_n, u_n, v_n) \tag{1}$$

The function F_s verifies the *semantic consistency* of node and edge labels (or attributes), while F_c verifies the *structural consistency*. Such a rule aims at ensuring that the addition of a couple (u_n, v_n) to a consistent state s_n will not produce an inconsistent state.

3 Parallel Algorithms

Designing a parallel algorithm starting from a sequential one is not an immediate task, but it requires to analyze different aspects. A widely adopted methodological approach, proposed by Ian Foster in [19], organizes the design of a parallel algorithm in four steps: Partitioning, Communication, Agglomeration and Mapping. The first is the arrangement of the data into discrete chunk of work that can be distributed to multiple tasks. Two basic ways to perform this process are: *domain decomposition*, aimed at decomposing the data into many small partitions to which parallel computation may be applied, and *functional decomposition* where the problem is decomposed in terms of operations that can be performed simultaneously. Once the decomposition have been defined, it is necessary to define how the task communicate each other; for instance if they work in a coordinated way or asynchronously. Then the agglomeration step aims at reducing the number of tasks generated during the partitioning in order to reduce communication costs, that is the most influencing factor in parallel algorithms performance. Finally, in the mapping step we define where each task is to run in order to minimize the execution time. For instance, we can map on the same processors tasks that need a strong communication while on different processors tasks able to run concurrently.

The exploration of the search space is a problem suitable to effectively exploit the data parallelism, thus, according to the partitioning strategies proposed by Foster, we have designed a parallel algorithm, namely $VF3P$, based on a domain decomposition where each task is responsible to explore a single state. It is worth

to point out that we have considered each task as performed by a single thread, therefore we will use equivalently the terms thread and task. The communication is realized through a *global state stack*; a task extracts the state to explore from the stack, then puts the feasible states generated in it. Each task stops when no further states have to be explored, such a condition is verified when the global stack is empty and all the other tasks have finished to explore their last extracted state, thus no other state is going to be put in the stack. Adopting such a strategy, the agglomeration is implicitly realized by the way the tasks communicate each other through the global structure. An outline of the procedure performed by each task in $VF3P$ is shown in Fig. 1. On the one hand, the used strategy allows to reach a high efficiency because all the threads work for most of the time; but, on the other hand, it requires a high level high level of synchronization among the threads that can affect the efficiency when the number of threads grows. To deal with this problem, we have designed a further optimization of $VF3P$, namely $VF3P_{LS}$, aimed at reducing the synchronization through the use of a side state stack used privately by the tasks. Each task has its *local state stack* where it puts the generated states to be successively explored. Therefore, until a task does not need to access the global stack it is able to work independently from the others. It is worth noting that using the local stack only does not guarantees that the workload is balanced among all the thread, this can cause a loss of efficiency due to the fact that some tasks are unoccupied. To avoid this problem, the local stack has a limited size, thus, when it is full the task is force to put the exceeding

```
1: function VF3P-Task(s, G1, G2, Sg, out Results)
2:    if IsEmpty(Sg) then
3:       return CheckActiveTasks()
4:    s := PullFromStack(Sg)
5:    if IsGoal(s) then
6:       append M(s) to Results
7:    else
8:       for (un, vn) ∈ NextCandidates(s, G1, G2)
9:          if IsFeasible(s, un, vn) then
10:             sn := ExtendState(s, un, vn)
11:             PushInStack(sn, Sg)
12:       NotifyEndOfExploration()
13:    return True
14: end
```

Fig. 1. Outline of $VF3P$ task procedure. Each task pull the next state to process from the global stack S_g. If the state is not a goal one, the task explores all of its descendant and put in S_g only those are feasibile. Since the condition **IsEmpty**(S_g) is not sufficient to guarantees that no more states have to be explored, each task notifies the start of the exploration when it pulls a state from the stack and uses the procedure **NotifyEndOfExploration** to communicates to the others when it finished. When S_g is empty and no more tasks are involved in exploring a state, than all the tasks will stop working.

states in the global stack. It is worth to note that, if the size of local state stack is configured taking into account the maximum depth of the search space (the size of the pattern graph) and the density of the two graphs, each task will be able to explore the space, from the root to leaves, without picking states from the global stack. Moreover, considering how the DFS works, each task maintains in its local stack the states corresponding to the higher levels of the state space, while it tends to put in the global stacks the states belonging to the lower levels.

The two algorithms, $VF3P$ and $VF3P_{LS}$, differ in the procedures **PullFromStack** and **PushInStack** (see Fig. 1). Indeed, while the tasks of $VF3P$ work directly using the global stack, in $VF3P_{LS}$ each a task checks firstly if the local stack is empty (full) before accessing the global stack to pull (put) a state.

4 Experiments

The benchmark of parallel algorithms is not limited to time and memory requirements; indeed, two relevant performance measures (see Eq. 2) are *speed-up* (Sp) and *efficiency* (Ef). The first represents the improvement of the execution time evaluated as the ratio between the run time T_s of the most efficient sequential algorithm and T_p, the one of the parallel algorithm. The second characterizes how the parallel algorithm is efficient in exploiting the available hardware resources and is obtained dividing the speed-up by the number of CPUs.

$$Sp = \frac{T_s}{T_p} \qquad Ef = \frac{Sp}{\#CPU} \tag{2}$$

The goal for a parallel algorithm is to reach a *linear speed-up*, where the value of the ratio is exactly the number of used CPUs. Very rarely, it is also possible to witness a *superlinear speed-up*, when the speed-up ratio is higher than the number of the CPUs. In general, a linear speed-up is very difficult to achieve because it requires that all the CPUs have always the same amount of workload and are able to execute their task independently or with few interactions. Unfortunately, not for all the problems it is possible to design algorithms exposing a linear speed-up. This is the case of graph algorithms, where this difficulty is confirmed by the fact that, until now, they have not been implemented effectively on modern GPU architectures, that are designed to exploit algorithms suitable to exhibit a linear speed-up such as low level image processing ones.

As previously introduced, computing the speed-up and the efficiency requires to choose a reference sequential algorithm, that is usually selected among the most efficient ones solving the problem under analysis. In our case, since the proposed parallel algorithms have been realized starting from VF3-Light, it is the most suited to this purpose, even because it has been proved to be one of the most efficient subgraph isomorphism algorithms. Therefore, in our experiments we have computed the aforementioned performance measures by executing $VF3P$ and $VF3P_{LS}$ with 2, 4 and 8 working threads respectively, in order to evaluate how the performance measures evolve when the number of thread grows.

Table 1. Speed-up of the parallel algorithms over different target graph size and number of CPU cores employed.

Dataset		Target size	Speed-up					
			$VF3P$			$VF3P_{LS}$		
			2 core	4 core	8 core	2 core	4 core	8 core
$\eta = 0.2$	Uniform	1000	0.76	0.75	0.70	0.99	0.84	0.76
		2000	1.59	2.34	3.42	1.55	2.63	3.74
		4000	1.56	2.96	5.16	1.69	3.14	5.61
		10000	1.79	3.44	6.36	1.82	3.53	6.63
	Non-Uniform	1000	0.70	0.75	0.61	1.06	0.76	0.57
		2000	1.54	2.16	2.98	1.51	2.33	3.11
		4000	1.70	2.98	4.77	1.72	3.12	4.97
		10000	1.77	3.37	6.11	1.80	3.47	6.41
$\eta = 0.3$	Uniform	1000	1.72	2.63	3.46	1.79	2.69	3.59
		2000	1.46	3.00	5.19	1.54	3.24	5.64
		4000	1.78	3.36	6.05	1.81	3.47	6.39
		6000	1.85	3.56	6.68	1.88	3.65	6.89
		8000	1.88	3.64	6.82	1.90	3.69	6.96
	Non-Uniform	1000	1.65	2.53	2.93	1.77	2.55	3.17
		2000	1.59	2.93	4.91	1.66	3.15	5.30
		4000	1.76	3.27	5.69	1.77	3.38	6.28
		8000	1.86	3.60	6.74	1.88	3.66	6.91

The experimental environment has been properly configured so as to collect unbiased measures and ensure that each thread run on the same core during all the execution time. The experiments have been performed on a Ubuntu 18.04 server where all the unnecessary services and the swap area have been deactivated. The server is equipped with two Intel(R) Xeon(R) CPU E5-2650 v2 and 256 Gb of Ram. Each Xeon E5-2650 has 8 physical core and three level of cache, in particular 256 Kb of L2 cache dedicated to the each single core and 20 Mb of L3 cache shared by all the cores laying on the same CPU. Hyperthreading has been deactivated to let the operating system to run one thread per physical core. One of the CPUs hosted the threads of the operating system and the experimental environment, while the other has been completely dedicated to run the working threads of the algorithms; thus, by setting the affinity we have ensured that each thread was executed on a dedicated core. In this way, we are able to properly measure how the algorithms improves speed-up and efficiency w.r.t. the number of cores by setting the wanted number of running threads.

The experiments have been performed over a subset of the MIVIA LDG, a standard dataset firstly used in [8,10,11] to benchmark VF3. The dataset is composed of very large and dense random Erdős and Rényi graphs, both labelled

Table 2. Efficiency of the parallel algorithms over different target graph size and number of CPU cores employed.

Dataset		Target size	Efficiency					
			$VF3P$			$VF3P_{LS}$		
			2 core	4 core	8 core	2 core	4 core	8 core
$\eta = 0.2$	Uniform	1000	0.38	0.18	0.08	0.49	0.21	0.09
		2000	0.79	0.58	0.42	0.77	0.65	0.47
		4000	0.78	0.74	0.64	0.84	0.78	0.70
		10000	0.89	0.86	0.79	0.91	0.88	0.83
	Non-Uniform	1000	0.35	0.18	0.07	0.53	0.19	0.07
		2000	0.77	0.54	0.37	0.75	0.58	0.38
		4000	0.85	0.74	0.59	0.86	0.78	0.62
		10000	0.88	0.84	0.76	0.90	0.86	0.80
$\eta = 0.3$	Uniform	1000	0.86	0.65	0.43	0.90	0.67	0.45
		2000	0.73	0.75	0.64	0.77	0.81	0.71
		4000	0.89	0.84	0.75	0.90	0.87	0.80
		8000	0.94	0.91	0.85	0.95	0.92	0.87
	Non-Uniform	1000	0.83	0.63	0.38	0.89	0.63	0.40
		2000	0.79	0.73	0.61	0.83	0.78	0.66
		4000	0.88	0.82	0.71	0.89	0.85	0.79
		8000	0.93	0.90	0.84	0.94	0.92	0.86

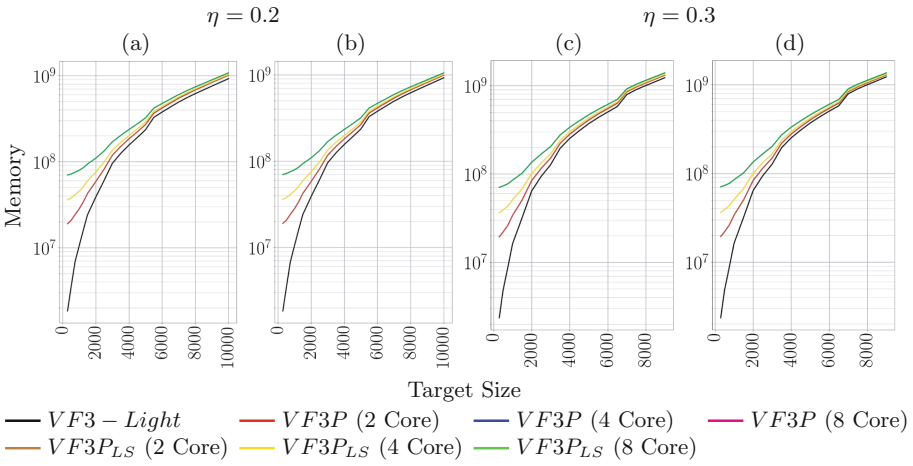

Fig. 2. Memory usage on $\eta = 0.2$ and $\eta = 0.3$ random graphs of both $VF3P$ and $VF3P_{LS}$, by varying the number of used cores.

and unlabelled, having densities (η) of 0.2, 0.3 and 0.4 respectively. The target graphs size ranges from 300 to 10,000 nodes, while the size of the pattern graphs is 20% with respect to that of the corresponding target. As discussed in [10], although in the MIVIA LDG dataset there is only one solution for each pair of graphs, the computational effort required to search for the solution and explore the whole search space to confirm the absence of other solutions is very high. Therefore, due to its complexity such a dataset is very suitable to stress the algorithm in terms of CPU usage, so as to highlight possible loss of efficiency.

In Tables 1 and 2 we present the results of the experiments in terms of speed-up, efficiency on the considered datasets. Analyzing the speed-up it is possible to note that both the parallel algorithms are able to scale w.r.t. the number of cores. In particular, in the case of $\eta = 0.2$, the best speed-up is achieved by $VF3P_{LS}$ for graphs larger than 8,000 node, and it is of 1.7, 3.5 and 6.6 when using 2, 4 and 8 cores respectively. On $\eta = 0.3$ graph, the best results is still obtained by $VF3P_{LS}$; it is worth to note that, in this case, the speed-up using 8 cores is about 7. The effectiveness of the proposed algorithms is also confirmed by the achieved efficiency, especially when the CPUs is more stressed, i.e. for graphs larger than 4,000 nodes; as expected, both the algorithm are able to obtain values higher than 0.8 irrespective from the number of cores. Of course, using less cores the efficiency is higher due to the lower time lost in synchronization. Anyway, when the number of cores increases, the benefit of the local stack, adopted by $VF3P_{LS}$, in reducing the amount of synchronization time lost, is more evident both on the speed-up and on the efficiency.

Differently happens for the memory, there are not notable benefits in using the local stack because both the solutions requires the same amount of memory to manage the communication; indeed looking at the Fig. 2 the curves are completely overlapped. It is worth to note that, the higher is the number of cores the higher is the memory usage because of the higher is the number of states generated and buffered in the communication stacks; even if the growth in the usage of memory is very limited if compared with that of VF3-Light.

5 Conclusions

In this paper we have proposed $VF3P$, a parallel algorithm to solve subgraph isomorphism. The effectiveness of the proposed algorithm has been proved using very large and dense graphs considering three performance measures: the speed-up, the efficiency and the memory usage. On the base of the achieved results we have demonstrated that the proposed algorithm is very efficient and able to scale w.r.t. the number of used CPUs. Nevertheless, a deeper analysis can be performed to explore other aspects impacting the performance and further improvements to the efficiency can be achieved by adopting different communication schemas and agglomeration.

References

1. Abu-Aisheh, Z., et al.: Graph edit distance contest: results and future challenges. Pattern Recogn. Lett. **100**, 96–1103 (2017)
2. Abu-Aisheh, Z., Raveaux, R., Ramel, J.Y., Martineau, P.: A parallel graph edit distance algorithm. Expert Syst. Appl. **94**, 41–57 (2018)
3. Aittokallio, T., Schwikowski, B.: Graph-based methods for analysing networks in cell biology. Briefings Bioinform. **7**, 243–255 (2006)
4. Computational pan-genomics: status, promises and challenges. Oxford J. Brief. Bioinf. **19**, 118–135 (2016)
5. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinform. **14**, 1–13 (2013)
6. Bougleux, S., Brun, L., Carletti, V., Foggia, P., Gazre, B., Vento, M.: Graph edit distance as a quadratic assignment problem. Pattern Recogn. Lett. **87**, 38–46 (2017)
7. Broecheler, M., Pugliese, A., Subrahmanian, V.S.: COSI: cloud oriented subgraph identification in massive social networks. In: 2010 International Conference on Advances in Social Networks Analysis and Mining (2010)
8. Carletti, V., Foggia, P., Saggese, A., Vento, M.: Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. IEEE Trans. Pattern Anal. Mach. Intell. **40**, 804–818 (2018)
9. Carletti, V., Foggia, P., Vento, M., Jiang, X.: Report on the first contest on graph matching algorithms for pattern search in biological databases. In: Liu, C.-L., Luo, B., Kropatsch, W.G., Cheng, J. (eds.) GbRPR 2015. LNCS, vol. 9069, pp. 178–187. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18224-7_18
10. Carletti, V., Foggia, P., Greco, A., Saggese, A., Vento, M.: Comparing performance of graph matching algorithms on huge graphs. Pattern Recogn. Lett. (2018)
11. Carletti, V., Foggia, P., Greco, A., Saggese, A., Vento, M.: The VF3-light subgraph isomorphism algorithm: when doing less is more effective. In: Bai, X., Hancock, E.R., Ho, T.K., Wilson, R.C., Biggio, B., Robles-Kelly, A. (eds.) S+SSPR 2018. LNCS, vol. 11004, pp. 315–325. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-97785-0_30
12. Carletti, V., Foggia, P., Saggese, A., Vento, M.: Introducing VF3: a new algorithm for subgraph isomorphism. In: Foggia, P., Liu, C.-L., Vento, M. (eds.) GbRPR 2017. LNCS, vol. 10310, pp. 128–139. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58961-9_12
13. Carletti, V., Foggia, P., Vento, M.: Performance comparison of five exact graph matching algorithms on biological databases. In: Petrosino, A., Maddalena, L., Pala, P. (eds.) ICIAP 2013. LNCS, vol. 8158, pp. 409–417. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41190-8_44
14. Carletti, V., Foggia, P., Vento, M.: VF2 plus: an improved version of VF2 for biological graphs. In: Liu, C.-L., Luo, B., Kropatsch, W.G., Cheng, J. (eds.) GbRPR 2015. LNCS, vol. 9069, pp. 168–177. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18224-7_17
15. Coffman, T., Greenblatt, S., Marcus, S.: Graph-based technologies for intelligence analysis. Commun. ACM **47**, 45–47 (2004)
16. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. Int. J. Pattern Recogn. Artif. Intell. **18**, 265–298 (2004)

17. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**, 1367–1372 (2004)
18. Foggia, P., Percannella, G., Vento, M.: Graph matching and learning in pattern recognition on the last ten years. J. Pattern Recogn. **28**, 1450001 (2014)
19. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
20. Jenkins, J., Arkatkar, I., Owens, J.D., Choudhary, A., Samatova, N.F.: Lessons learned from exploring the backtracking paradigm on the GPU. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6853, pp. 425–437. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23397-5_42
21. Lacroix, V., Fernandez, C., Sagot, M.: Motif search in graphs: application to metabolic networks. Trans. Comput. Biol. Bioinf. **3**, 360–368 (2006)
22. McCreesh, C., Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 295–312. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_21
23. Paten, B., Novak, A.M., Eizenga, J.M., Garrison, E.: Genome graphs and the evolution of genome inference. Genome Res. **27**, 665–676 (2017)
24. Rodenas, D., Serratosa, F., Solé-Ribalta, A.: Parallel graduated assignment algorithm for multiple graph matching based on a common labelling. In: Jiang, X., Ferrer, M., Torsello, A. (eds.) GbRPR 2011. LNCS, vol. 6658, pp. 132–141. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20844-7_14
25. Solnon, C.: Alldifferent-based filtering for subgraph isomorphism. Artif. Intell. **174**, 850–864 (2010)
26. Vento, M.: A long trip in the charming world of graphs for pattern recognition. Pattern Recogn. **48**, 291–301 (2014)
27. Wasserman, S., Faust, K.: Social Network Analysis: Methods and Applications. Cambridge University Press, Cambridge (1994)
28. Xie, X., Li, Z., Zhang, H.: Efficient subgraph matching in large graph with partitioning scheme. In: 13th Web Information Systems and Applications Conference (2016)
29. Xu, Q., Jeon, H., Annavaram, M.: Graph processing on GPUs: where are the bottlenecks. In: 2014 IEEE International Symposium on Workload Characterization (2014)