Parallel Computing 114 (2022) 102984

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Efficient parallel branch-and-bound approaches for exact graph edit distance problem

Adel Dabah [a], Ibrahim Chegrane [a,b,*], Saïd Yahiaoui [a], Ahcene Bendjoudi [a], Nadia Nouali-Taboudjemat [a]

[a] CERIST Research Center on Scientific and Technical Information, Algiers, Algeria
[b] CoBIUS Lab, Department of Computer Science, University of Sherbrooke Qc, Canada

ARTICLE INFO

Dataset link: https://github.com/chegrane/ged_parallel/

Keywords:
Parallel branch-and-bound
Graph matching
Graph edit distance

ABSTRACT

Graph Edit Distance (GED) is a well-known measure used in the graph matching to measure the similarity/dissimilarity between two graphs by computing the minimum cost of edit operations needed to transform one graph into another. This process, Which appears to be simple, is known NP-hard and time consuming since the search space is increasing exponentially. One way to optimally solve this problem is by using Branch and Bound (B&B) algorithms, Which reduce the computation time required to explore the whole search space by performing an implicit enumeration of the search space instead of an exhaustive one based on a pruning technique. nevertheless, They remain inefficient when dealing with large problem instances due to the impractical running time needed to explore the whole search space. To overcome this issue, We propose in this paper three parallel B&B approaches based on shared memory to exploit the multi-core CPU processors: First, a work-stealing approach where several instances of the B&B algorithm explore a single search tree concurrently achieving speedups up to 24× faster than the sequential version. Second, a tree-based approach where multiple parts of the search tree are explored simultaneously by independent B&B instances achieving speedups up to 28×. Finally, Due to the irregular nature of the GED problem, two load-balancing strategies are proposed to ensure a fair workload between parallel processes achieving impressive speedups up to 300×. all experiments have been carried out on well-known datasets

1. Introduction

Graph Edit Distance (GED) approach is a well-known technique used in graph matching to measure the minimum distance between two graphs. The goal of the GED is to compute the amount of dissimilarity between two graphs. In other words, it represents the cost of the best set of edit operations needed to transform one graph into another [1]. The allowed operations are insertion, deletion, and substitution, which are applied on vertices and edges. This problem is known to be very challenging due to its NP-hardness nature [2], which means that the time complexity of computing the minimum distance between two graphs increases exponentially with the number of vertices. The importance of the GED comes from its multitude of use cases. It can be used to find exact and also inexact matching, where some errors are tolerated. Moreover, the GED can be used in various areas [3], especially in areas related to pattern recognition, such as, handwriting recognition [4–6], person identification and authentication (example: fingerprint recognition) [7], documents analysis [5,8,9], and in graph database search [10]. It can also be found in machine learning, nearest-neighbor classification, and in data mining area [5].

To compute optimally the GED between two graphs, often A-Star [11] based search technique is used in the literature [3]. However, this latter needs huge memory resources, making it impossible to use for large graphs. The Branch and Bound (B&B) algorithms are well-known techniques for optimally solving optimization problems via an intelligent enumeration of the search space. This method models the search space as a tree using two components: branching and bounding. The branching is a recursive process that divides the search space of a given problem into several smaller sub-problems, which are treated the same way until solutions are found. After the branching process, the bounding operator evaluates the ability of each generated sub-problem to contain good solutions. The B&B algorithm uses several techniques (elimination and selection) to avoid exploring non-promising sub-problems (branches) and accelerate the search process. Due to the complexity of the GED problem, which is NP-hard [2] for general

* Corresponding author at: CoBIUS Lab, Department of Computer Science, University of Sherbrooke Qc, Canada.
E-mail addresses: adabah@cerist.dz (A. Dabah), ibrahim.chegrane@usherbrooke.ca (I. Chegrane), syahiaoui@cerist.dz (S. Yahiaoui), abendjoudi@cerist.dz (A. Bendjoudi), nnouali@cerist.dz (N. Nouali-Taboudjemat).

https://doi.org/10.1016/j.parco.2022.102984
Received 18 September 2021; Received in revised form 16 July 2022; Accepted 27 October 2022
Available online 3 November 2022

graphs, B&B algorithms require a long time to find the optimal solution, especially when dealing with large graphs. To overcome this drawback, we consider parallel computing as an interesting way to reduce the running time of such algorithms.

In this paper, we study the impact and possible gain of exploiting the computing resources of a single machine. Indeed, most of today's computers are parallel from a hardware perspective, offering a decent computing power that is not exploited in most cases. For this reason, and to evaluate the possible gain that can be achieved, several B&B parallelization approaches are proposed. The goal here is to measure each parallel approach's impact and behavior in reducing the execution time and efficiently exploring the search space. The proposed approaches can be classified as high-level parallelization in which several instances of the B&B algorithm simultaneously explore the search space.

Two parallel approaches are proposed. The first one is based on a work-stealing strategy in which the goal is to accelerate the sequential process of exploring the search tree without increasing the amount of used memory space. Hence, we have several instances of the B&B algorithm exploring concurrently a single search tree stored in a shared memory space accessible in read/write operations by all parallel B&B instances. Experiments on reference datasets using 16 CPU cores show a relative speedup around 24 × compared to the sequential version. To explore more efficiently the search space and to study the impact of the diversification, a second parallel B&B approach is proposed. This latter, denoted by the tree-based approach, simultaneously explores multiple parts of the search tree using several independent B&B instances. Therefore, updating the Upper Bound (UB), which eventually end-up with an improvement in complexity. This approach is based on the Master–Worker paradigm, where the master splits the search tree over several B&B instances (workers). After that, each worker builds its own search tree exploiting a private memory space. The only shared information in this approach is the value of the UB, which is updated each time a better path is explored. The obtained results for this approach show the positive impact of diversifying the exploration process allowing us to reach a relative speedup around 28 × compared to our sequential B&B implementation. However, due to the irregular workload of sub-problems in the search tree, many workers finish quickly (stay idle), while others have a long-running time. The impact of load-balancing strategies on solving the GED problem was not well studied in Abu-Aisheh et al. [12,13]. For this reason, we proposed two original load-balancing strategies to prevent the idleness of workers. Our first load-balancing strategy consists of using the master as a load balancer by changing its exploration strategy to ensure the large availability of sub-problems. The idea to avoid the idleness is to give read/write access to all parallel workers to pick up a sub-problem from the load-balancer work pool whenever their work pools are empty. The second way to ensure a fair load between workers is to combine the previous two parallel approaches. The idea here is to allow the workers to perform k iterations locally and then merge their own local work-pool into one global work-pool shared between them. The results using reference datasets show the good impact of combining diversification and load-balancing strategies. Moreover, adding load-balancing techniques allowed to improve the performance of the tree-based parallel approach by a factor up to 11x.

The rest of the paper is organized as follows. Section 2 presents some basic notations and definitions related to the GED. In Section 3, we discuss related work. Section 4 describes the sequential B&B algorithm and its components. In Section 5, we detail our proposed parallel B&B approaches and load-balancing strategies. Section 6 reports computational results. Finally, conclusions and perspectives are given in Section 7.

2. Problem definition

In the following, we first define some basic concepts and then define the GED problem formally.

2.1. Graph

A graph is a structure used to model pairwise relations between objects [14]. It contains a set of vertices connected by a set of edges.

Formally, a labeled graph is denoted by $G = (V, E, \alpha, \beta)$, where:

- $V = \{v_1, v_2, \dots, v_n\}$, a set of n vertices.
- $E = \{e_1, e_2, \dots, e_m\}$, a set of m edges ($E \subseteq V \times V$).
- α is a labeling function on the vertices, i.e., $\alpha : V \rightarrow L_V$.
- β is a labeling function for the edges, i.e., $\beta : V \rightarrow L_E$.

  L_V and L_E are restricted to labels that can be given by a set of integers, the vector space R^n, or symbolic labels.

In this paper, we consider simple undirected labeled graphs without loops.

2.2. GED operations

In the GED problem, an edit path that transforms a graph g_1 into a graph g_2 consists of a set of edit operations. Each edit operation performs an action on either vertices or edges, or both vertices and edges. In this work, we consider a vertex-centric approach in which edit operations are performed on vertices, and those performed on edges are implied. Thus, only the following three basic edit operations are allowed: *insertion*, *deletion*, and *substitution*.

Let us consider a vertex $v_i \in V_1$ from g_1 and a vertex $u_j \in V_2$ from g_2, we denote:

1. $v_i \rightarrow u_j$: vertex v_i is *substituted* by vertex u_j.
2. $v_i \rightarrow \epsilon$: vertex v_i is *deleted* from g_1.
3. $\epsilon \rightarrow u_j$: vertex u_j is *inserted* into g_1.

2.2.1. Implied edges operations

In vertex-centric approach, edit operations on edges are implied. Indeed, an edge is *substituted*, *deleted* or *inserted*, depending on edit operations performed on its incident vertices [5]. Let us consider two vertices v, v' from graph g_1 and two other vertices u, u' from graph g_2. If we perform the following two edit operations $\{v \rightarrow u\}$ and $\{v' \rightarrow u'\}$ on vertices v and v', respectively, three cases of implied operations on edges can be distinguished:

1. If there is an edge $e_1 = (v, v')$ between v and v' in g_1 and there is also an edge $e_2 = (u, u')$ between u and u' in g_2, then e_1 is *substituted* by e_2, denoted $e_1 \rightarrow e_2$.
2. If there is an edge $e_1 = (v, v')$ between v and v' in g_1 and there is no edge between u and u' in g_2, then e_1 is *deleted* from g_1, denoted $e_1 \rightarrow \epsilon$.
3. If there is no edge between v and v' in g_1 and there is an edge $e_2 = (u, u')$ between u and u' in g_2, then e_2 is *inserted* into g_1, denoted $\epsilon \rightarrow e_2$.

Note that all its incident edges are automatically deleted if a vertex is deleted from g_1. Similarly, if a vertex is inserted in g_1, all its incident edges in g_2 are inserted if their other ends already exist in g_1.

2.3. Cost function

An essential parameter in GED is the cost function. This latter assigns a value (cost) to each edit operation applied to vertices or edges. Thus, the costs assigned to the edit operations affect the optimal edit path. The cost function represents an efficient way to integrate domain-specific information about object similarity. The cost $c(o)$ of a particular edit operation o is defined with respect to the underlying label alphabets. The cost for the three edit operations can be given by:

- $c(u \rightarrow \epsilon) = \theta$
- $c(\epsilon \rightarrow v) = \theta$

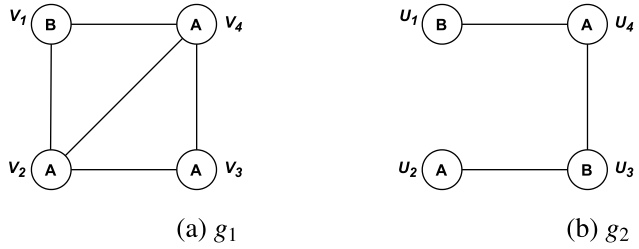


Fig. 1. Both source and target graphs.

- $c(u \rightarrow v) = c(v \rightarrow u) = \beta$

We should mention that the substitution cost is zero ($\beta = 0$) if the two vertices or edges have the same label.

2.4. Graph edit distance

Let $g_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $g_2 = (V_2, E_2, \alpha_2, \beta_2)$ be the source and the target graph respectively. The graph edit distance between g_1 and g_2, denoted by $d(g_1, g_2)$, represents the amount of dissimilarity between the two graphs. In other words, it represents the best set of edit-operations $\{e_1, e_2, \dots, e_k\}$ (in terms of cost) that transform g_1 into g_2.

Formally, the GED between g_1 and g_2 is defined by:

$$d(g_1, g_2) = min_{\{e_1, e_2, \dots, e_k\} \in \gamma(g_1, g_2)} \sum_{i=1}^{k} c(e_i)$$

where $\gamma(g_1, g_2)$ denotes the set of all edit paths transforming g_1 into g_2. Each path contains a set of edit operations $\lambda = \{e_1, e_2, \dots, e_k\}$, and $c(e_i)$ denotes the cost of the edit operation e_i [5].

2.5. Example

Let the source g_1 and the target g_2 graphs illustrated in Fig. 1. Let the cost of deletion/insertion operations be 2, and the substitution cost be 1. The optimal path that edits graph g_1 to get graph g_2 is obtained by performing four vertex substitutions, which imply the substitution of three edges and the deletion of two others. The complete edit path:

$\lambda(g_1, g_2) = \{ (v_1\{B\} \rightarrow u_1\{B\}), (v_2\{A\} \rightarrow u_2\{A\}), ((\boldsymbol{v_2}, \boldsymbol{v_1}) \rightarrow \boldsymbol{\epsilon}), (\boldsymbol{v_3}\{\boldsymbol{A}\} \rightarrow \boldsymbol{u_3}\{\boldsymbol{B}\}), ((v_3, v_2) \rightarrow (u_3, u_2)), (v_4\{A\} \rightarrow u_4\{A\}), ((v_4, v_1) \rightarrow (u_4, u_1)), ((\boldsymbol{v_4}, \boldsymbol{v_2}) \rightarrow \boldsymbol{\epsilon}), ((v_4, v_3) \rightarrow (u_4, u_3)) \}$

Thus, the cost of the optimal path that transforms g_1 into g_2 is: $d(g_1, g_2) = 5$.

3. Related work

In pattern recognition, real-world objects are generally represented by numerical vectors (Images). This representation is sensitive for translation, rotation, scaling, etc. For that, graph-based representation is used to overcome these issues. This latter comes with the power of being unchangeable and more expressive due to information that can be added in nodes/edges. Exact matching (Exact-GED) supposes a noiseless environment. Thus, it considers two graphs (g1 and g2) as a match if and only if the GED between the two graphs (GED(g1,g2)) is equal to zero. However, due to the noise factor, two graphs of the same object may not match completely. In this case, we call it inexact matching (Inexact-GED), where two graphs with an edit path below a given threshold are accepted as a match. Therefore, the GED can be used in exact and inexact matching scenarios. Several applications require an optimal solution (edit path) that transforms one graph into another. For that, they generally use optimal approaches like A-star and B&B algorithms. However, these latter approaches can also be used to obtain a sub-optimal solution by, for instance, stopping the computation after a given amount of time. In the rest of this section, we will focus on existing works that deal with the optimal GED problem, which are based on tree traversal and parallelism [15].

After the introduction of GED by A. Sanfeliu and K.-S. Fu in 1983 [16]; relatively in the same year of the final publication; Bunke and Allermann [1] were the first to implement and adapt the A-star algorithm to solve the GED problem. The authors' idea to find the optimal solution is to map all elements from the first graph into the second graph's elements. Thus, generating a search tree that models all possible solutions. The mapping uses the three edit operations: deletion, insertion, and substitution. The authors' approach was validated through some small graphs.

The A-start GED finds the optimal solution but suffers from huge computational and space complexity. In [17], the authors, proposed an approximate approach for GED problem called *A*-Beamsearch*. It limits the size of the A-star priority queue to a certain size s. Only the s partial edit paths with the lowest costs (real+estimated) are kept. The larger the size of the beam-search queue, the more accurate it becomes.

To speed up the A-star search process, Riesen et al. [4] proposed a *bipartite heuristic* that gives an estimation of the future cost h (lower bound). This heuristic is based on Bipartite matching. At first, two cost-matrices are created that contain the different assignments of vertices and edges, respectively, for the three operations (substitution, deletion, and insertion) between graph one and graph two. Then, Munkres algorithm [18,19] is used to compute the best assignment of vertices and edges separately. In the end, the h value is the sum of vertices' best assignment and edges' best assignment. In [5], the authors used their heuristic to approximately solve the GED problem. Thus, providing a good upper bound for the problem. The bipartite heuristic has been discussed and improved in [20,21] to compute more accurately lower bounds.

In [6], Abu-Aisheh et al. used a Depth First Search (DFS) approach to solve the GED problem optimally. The author's goal was to reduce the amount of used memory space compared to the A-star algorithm. The approach begins by sorting vertices using Munkres' algorithm and then explores the search space according to the depth-first search strategy. In [13], the authors proposed a shared-memory parallelization of their DF-GED algorithm denoted by PDFS. In this latter, several parallel instances of the DF-GED explore the search space simultaneously. Furthermore, the authors used a load-balancing strategy to avoid the idleness of parallel threads. The results show an improvement of 20% in execution time compared to their DF-GED algorithm. However, the experiments did not indicate the impact of the parallelization and the load-balancing strategy through the classical metrics (speedup and efficiency). In [12], the authors proposed a distributed memory version of their DF-GED algorithm. The proposed parallelization uses a master–slave paradigm and is implemented using the Hadoop framework [22]. This distributed version has shown similar results compared to their serial implementation in terms of execution time.

In [23], Gouda and Hassaan proposed an edge-based depth-first method called CSI_GED to solve the uniform GED problem, where all edit operations have the same cost. The authors' idea is to map all edges from the first graph into the second graph's edges, and the vertices are then implied. A detailed experiment section validated the proposed approach.

In [24,25], Blumenthal and Gamper adapted the $DF-GED$ algorithm in [6] for GED problem with an uniform cost for all edit operations. In this case, the complexity of computing the estimated cost can be reduced from cubic time using the Munkres algorithm to linear time. In the same paper, the authors proposed a generalization for CSI_GED method in [23] to cover non-uniform metric edit costs called CSI_GED^{nu}. This generalization consists to use Munkres' algorithm [18] to compute the future estimated cost. In [26], the same authors proposed a C++ library for solving the GED problem in exact and approximate ways called GEDLIB. The library contains several literature algorithms for the GED problem, and it can be used as a basic implementation for other new algorithms.

In [27], Chang et al. proposed new lower bounds for the GED problem, which significantly reduce the memory footprint of A-star and DFS methods. Based on these lower bounds, they developed AStar+-LSa and AStar+-BMa algorithms, which can process graphs with up to sixty vertices.

In [28], Wang et al. proposed a parallel implementation of the AStar+-LSa algorithm proposed in [27] for solving the exact GED. The main idea of their algorithm, called PGED, is to allocate the most time-consuming step of the AStar algorithm (searching the optimal vertex mapping) to several threads simultaneously. The authors' results showed a 2x speedup compared to their sequential version of the algorithm.

In inexact matching, a parallel algorithm was presented in 1997 by Allen et al. [29]. Their algorithm works on two graphs with the same number of vertices, and the distance measure is called the relational distance. The best match between g1 and g2 is obtained by finding the best permutation of the vertices that align g1 with g2. For each possible permutation, the edges of the two graphs g1 and g2 are compared, and for each unmatched edge, an error of one is added. The parallel part is based on a B&B search with heuristics to compute a good estimated distance between g1 and g2 (lower bound). An important point, the algorithm is formulated to work on the specific architecture of the single-instruction-stream/multiple-data-stream (SIMD) parallelism. The experiment was done on small and medium graphs between 10 and 27 vertices, using massive parallelism with 1024 processors.

Due to the NP-hardness nature of the GED problem, computing the minimum distance even for small graphs is time-consuming. For this reason, most of the above literature gave just a proof of concept of their approaches using typically small datasets or partially exploring the search space within a certain amount of time. In [30], we propose a tree-based approximate approach that give near-optimal results. Since exploring the whole search tree is impractical, this approach keeps only the best nodes at each tree level for further exploration. This reduces the execution time enormously without scarifying the solution quality.

To deal with larger graphs and report the impact of parallelism on optimally solving the exact GED problem, we propose several parallel B&B approaches and load-balancing strategies to accelerate the search process and efficiently exploit the computing resources available in all recent computers.

4. B&B for the GED problem

Branch and Bound (B&B) algorithms are well-known techniques for optimally solving combinatorial optimization problems. They were first proposed in 1960 by A. H. Land and A. G. Doig to solve discrete programming problems [31]. These algorithms are based on an intelligent and explicit enumeration of all the search space, which is modeled as a search tree. This latter is explored using the branching and bounding operators. The goal is to find the best edit path in terms of cost that transforms one graph into another. The branching of a search tree node generates a set of successors by splitting its search space into several smaller sub-problems. These successors are handled in the same way until reaching solutions. After the branching, the bounding operator evaluates the ability of each successor to improve the best solution found. Otherwise, the branch is pruned, and the successor is deleted. Furthermore, Algorithm 1 describes the general structure of the used B&B algorithm.

In the following, we describe the adaptation of the different B&B operations for the GED problem.

Algorithm 1: Pseudo-code of the sequential B&B algorithm

Data: Non-empty attributed graphs $g_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $g_2 = (V_2, E_2, \alpha_2, \beta_2)$ where $V_1 = \{u_1, ..., u_{|V1|}\}$ and $V_2 = \{v_1, ..., v_{|V2|}\}$
Result: *optimal_path*

```
1  OPEN ← {intial problem}; best_path ← null;
   optimal_path ← null;
2  UB = compute_upper_bound();
3  while (OPEN != ∅) do
4      node ← select_node(OPEN);
5      if LB(node) < UB then
6          if node is a leaf node then
7              optimal_path ← node;
8              UB ← LB(node);
9          else
10             Generates successors nd_j from node;
11             foreach successor(nd_j) do
12                 if LB(nd_j) < UB then
13                     OPEN ← add_node (nd_j);
14                 else
15                     prune(nd_j);
16     else
17         prune(node);
18 return optimal_path;
```

4.1. Branching

The B&B algorithms operate on a search tree that models the whole search space. The B&B search tree is generated using the branching operator. This latter decomposes a problem (tree node) into several smaller sub-problems, which are treated in the same way until reaching leaf nodes. The branching for the GED problem is based on the idea of mapping vertices from the first graph g_1 to vertices of the second graph g_2, i.e., we use a vertex-centric approach. The mapping performs the classical edit operations: substitution, deletion, and insertion. Formally, a search tree node nd is characterized by:

- A set of past edit operations known as *path* and denoted by $\lambda = \{e_1, e_2, \dots, e_k\}$, where e_i is one of the edit operations (substitution, deletion, and insertion).
- A remaining vertices from the two graphs (g_1 and g_2) denoted by R_{V1} and R_{V2} respectively.
- A processed vertices from the two graphs (g_1 and g_2) denoted by P_{V1} and P_{V2} respectively.

In this way, the root node (initial problem) is defined as $|R_{V1}| = |V_1|$, $|R_{V2}| = |V_2|$, $P_{V1} = \emptyset$, and $P_{V2} = \emptyset$.

The branching on a search tree node nd generates $|R_{V2}|$ new successors by performing the substitution of a vertex $u \in R_{V1}$ with all vertices in R_{V2}. In other words, each successor node nd_j (where $j \in [1..|R_{V2}|]$) represents the mapping of a vertex $u \in R_{V1}$ by a vertex $u' \in R_{V2}$ is defined as follows:

$$R_{V1}(nd_j) = R_{V1}(nd) \setminus \{u\}$$

$$R_{V2}(nd_j) = R_{V2}(nd) \setminus \{u'\}$$

$$P_{V1}(nd_j) = P_{V1}(nd) \cup \{u\}$$

$$P_{V2}(nd_j) = P_{V2}(nd) \cup \{u'\}$$

$$\lambda(nd_j) = \lambda(nd) \cup \{u \rightarrow u'\}.$$

In addition to the successors generated above, we add a new successor that represents the deletion of the vertex $u \in R_{V1}$ as follow:

$$R_{V1}(nd_j) = R_{V1}(nd) \setminus \{u\}$$

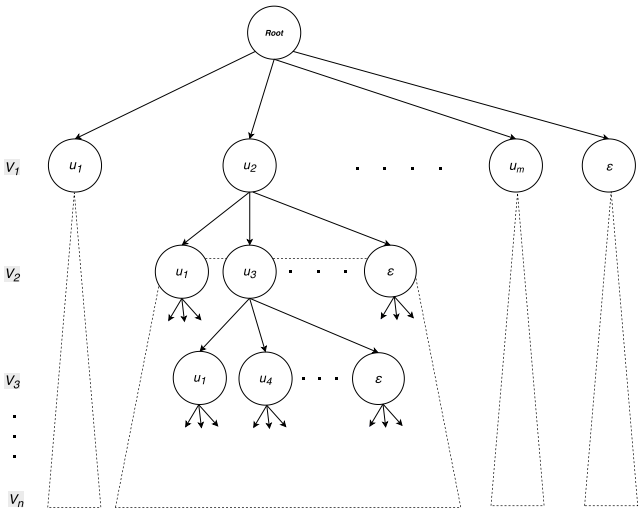


Fig. 2. General scheme of the search tree related to the GED problem.

$$R_{V2}(nd_j) = R_{V2}(nd)$$

$$P_{V1}(nd_j) = P_{V1}(nd) \cup \{u\}$$

$$R_{V2}(nd_j) = P_{V2}(nd)$$

$$\lambda(nd_j) = \lambda(nd) \cup \{u \rightarrow \epsilon\}.$$

This process is repeated until reaching leaf nodes where $R_{V1} = \emptyset$. In this case, if the set of remaining vertices related to the second graph is not empty ($R_{V2} \neq \emptyset$); we insert all the remaining vertices in R_{V2} at once. This process is described as follow:

$$R_{V1}(nd_j) = R_{V1}(nd)$$

$$R_{V2}(nd_j) = R_{V2}(nd) \setminus R_{V2}(nd)$$

$$P_{V1}(nd_j) = P_{V1}(nd)$$

$$P_{V2}(nd_j) = P_{V2}(nd) \cup R_{V2}(nd)$$

$$\forall u'_i \left[i = 1, |R_{V2}(nd)|\right] \lambda(nd_j) = \lambda(nd) \cup \{\epsilon \rightarrow u'_i\}.$$

Fig. 2 illustrates a search tree example used to solve the GED problem. At each level in this figure, one vertex v_i from the first graph g_1 is mapped with the remaining vertices u_j of g_2 and ϵ to generate deletion.

4.2. Evaluation (bounding)

After the branching process, the bounding consists of evaluating the ability of each successor to contain good solutions. Two different bounds can be distinguished: the Lower Bound (LB) and the Upper Bound (UB).

Upper Bound: The UB represents an upper limit of the evaluation of all search tree nodes. Initially, any solution to the GED problem can be considered as an initial value for the UB, which is updated as soon as a new path (solution) is found. In our case, the UB is computed using the polynomial-time algorithm proposed by Riesen et al. [5]. First, this algorithm computes the best vertices assignment of two input graphs. After that, the algorithm determines the corresponding implied edges. This creates a complete path that transforms the source graph into the target graph.

The Lower Bound: For a given sub-problem, the LB is viewed as an estimation of the lower evaluation of all its solutions. The LB represents the most consuming part of the B&B algorithm since it is computed for each successor. For this reason, it needs to be efficient and optimized as much as possible. To our knowledge, the best LB known for the GED problem is based on the bipartite heuristic proposed in [4,5]. The value of this LB is a combination of two parts: the real cost (g) and the estimated cost (h). i.e., $LB = g + h$. The real cost of a search tree node (nd) represents the cost of all edit operations performed on this path. In other words, $g(nd) = Score(\lambda(nd)) = \sum_{i=1}^{|\lambda|} cost(e_i)$. The estimated cost h represents the best mapping of the remaining vertices and edges from the two graphs using the bipartite graph matching method. It represents the sum of the optimal assignment of vertices and the optimal assignment of edges using the Munkres algorithm [18]. This algorithm operates over costs matrices defined using remaining vertices and remaining edges from the two graphs (target and source). In this way, we can give a good estimation of the LB in polynomial time complexity.

5. Proposed parallel B&B approaches for GED problem

This section presents our parallel B&B schemes for the multi-core CPU processors available in all recent computers. The proposed schemes are based on a shared memory model aiming to reduce communication costs. Before introducing our proposed approaches and load-balancing strategies, we will briefly introduce the classifications of the parallel B&B algorithm.

5.1. Taxonomies of parallel B&B

The sequential B&B algorithm is not sufficient when dealing with large problem instances. Parallel computing using High Performance Computing (HPC) architectures represents a promising way to deal with such challenging problems. The parallelization of B&B algorithm is well studied in the literature, and several classifications of this method have been proposed [32–34].

Trienekens et al. [33] classified the parallelization of the B&B algorithm as *high level* and *low level* according to the degree of the parallelization of the B&B operations. Gendron et al. [34] identified three types of parallel B&B algorithm according to the search tree parallelization degree. The first type denoted by *node based* introduces parallelism when performing operations on generated sub problems. The second type denoted by *tree-based* consists of building the search tree in parallel by performing operations on several sub-problems simultaneously. Finally, the third one (*multi-search*) implies that several search trees are built in parallel, and each tree is characterized by different operations. The most recent classification is the one proposed by Melab and Mezmaz [35]. It represents a generalization of the classification proposed by Gendron et al. [34]. In this classification, four parallel models of the B&B algorithm are identified: (1) The *parallel multi-parametric model* uses several instances of the B&B algorithm simultaneously. This model can be viewed as a coarse-grained parallelization where each instance of the B&B algorithm uses its own parameters. (2) *The parallel tree exploration model* consists of simultaneously exploring several sub-problems that define different research sub-spaces of the initial problem. This means that branching, bounding, and pruning operators are executed in parallel synchronously or asynchronously by different processes exploring these sub-spaces. (3) *The parallel evaluation of bounds model* allows the parallelization of the bounding of sub-problems generated by the branching operator. (4) *The parallel evaluation of a single bound* does not change the design of the algorithm because it is similar to the serial version except that the bounding operator is faster.

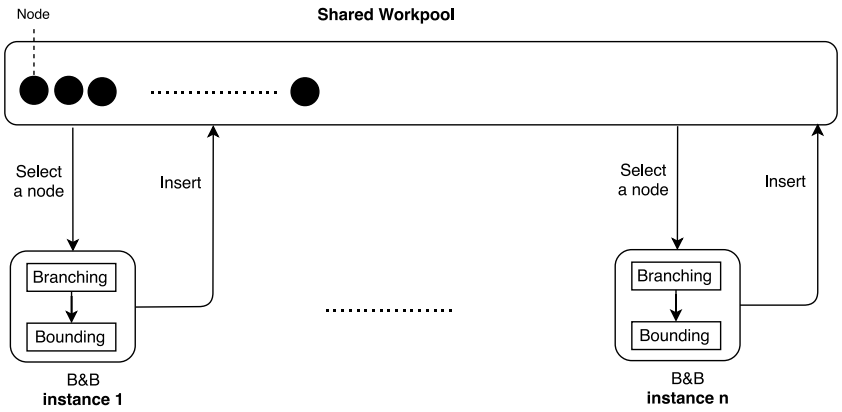


Fig. 3. Parallel B&B approach based on work-stealing paradigm.

5.2. Work-stealing parallel B&B approach ($WS_{B\&B}$)

Our first parallel B&B approach aims to accelerate the B&B execution time by accelerating the sequential process of exploring the search tree. As depicted in Fig. 3, this approach can be viewed as a high-level parallelization that aims to accelerate the exploration of a single search tree using a work-stealing strategy. In this strategy, several instances of the B&B algorithm explore concurrently a single search tree stored in a shared work-pool accessible by all B&B instances. The work-pool represents the data structure that contains a set of active sub-problems obtained after branching and bounding operations. In this first parallel approach, all the B&B instances use the same work-pool to store and pick up sub-problems. The end of this parallel version is reached when the shared work-pool is empty, and all the parallel threads become inactive.

5.2.1. Work-stealing parallel B&B functioning

This parallel approach is based on a collegial way of thinking, in which all parallel instances share the same work-pool and perform the same actions. The goal is to accelerate the sequential process of exploring the search tree. Thus, there is no distinction between all parallel processes.

Initially, we have a single instance of the B&B algorithm, referred to as the main thread, that loads the two graphs and creates the root node. The main thread performs the branching and bounding operations on the root node, creating a set of active sub-problems stored in the shared-work-pool. After that, several instances of the B&B algorithm are launched to explore simultaneously and concurrently the same search tree. In other words, each instance of the B&B algorithm picks up a sub-problem from the shared work-pool in a concurrent safe way. The resulting nodes (sub-problems) from the branching and bounding operations for each B&B instance are inserted in the shared work-pool. This process is repeated until the shared work-pool becomes empty. Hence, the end of the parallel algorithm is reached.

The key to ensuring the efficiency of this parallel approach is to provide good management of the shared work-pool since this latter is accessible by all parallel threads concurrently. To ensure no concurrency access problems, a thread-safe data structure is used to implement the shared work-pool.

Since all the B&B instances use the same work-pool with a depth-first strategy, this approach has a low memory utilization (slightly bigger than the serial version). The other interesting fact about this approach is the fair workload distribution between parallel processes, which prevents idleness, especially for these problems.

The Downside of this approach is the low impact of the diversification and the scalability issue that may occur when increasing the number of parallel B&B instances due to the concurrent access to the same work-pool. To avoid these problems, we propose a second parallel B&B approach.

5.3. Tree-based parallel B&B approach ($TB_{B\&B}$)

The goal of our B&B algorithm is to find the best path, in terms of evaluation (the path with the minimum cost), in a search tree that models the search space.

Our second parallel approach is based on the fact that each path in the search tree can be explored independently from the others. Indeed, this approach represents a tree-based parallelization in which several completely independent parallel B&B instances simultaneously explore several paths (parts) of the search tree. As depicted in Fig. 4, each instance of the B&B algorithm independently explores a part of the search tree using a private work-pool to store and pick up sub-problems. Compared to the first parallel approach, multiple work-pools are used, each containing a sub-search tree. The only shared information between parallel B&B instances is the UB value. This latter is updated each time a B&B instance finds a better path that transforms the source graph into the target graph. The end of this parallel version is reached when all private work-pools are empty, and all the parallel threads become inactive.

5.3.1. Tree-based parallel B&B functioning

This parallel approach aims to diversify the search process by exploring simultaneously several parts of the search tree, i.e., several independent B&B instances operate on their own local work-pools. This parallel approach is implemented using the Master/Worker paradigm. Indeed, we have a single instance of the master process (controller) in the system, and the others are workers. Initially, the workers are blocked, waiting for nodes to explore. The controller loads the problem data and creates the root node. After that, it explores several search tree levels, using its own B&B algorithm, which creates a set of active sub-problems stored in its own work-pool. At this point, each sub-problem represents a sub-search tree. After that, the master wakes blocked workers, where each one explores a sub search tree obtained from the master in its private local work-pool. The local work-pool evolves constantly, and when it becomes empty, its owner becomes idle. The end of the parallel approach is reached when all parallel processes are idle, including the master process.

The primary benefit of this approach is: (1) Reducing the synchronization points, which allows exploiting the available multi-core CPU processors more efficiently. (2) Diversifying the search process which leads to improving the UB more efficiently. This allows an efficient pruning process that significantly reduces the explored search space. Hence, the overall complexity in terms of execution time. However, the major limitation of this approach is the idleness of parallel processes due to the unfair workload distribution in each branch.

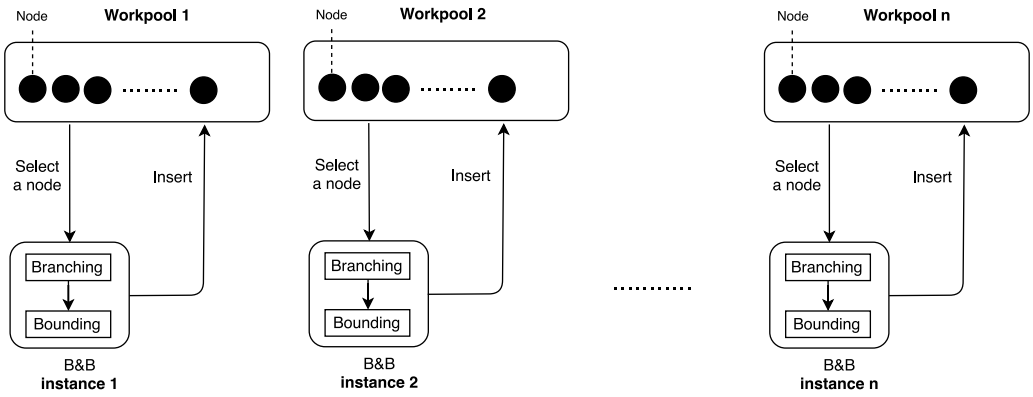


Fig. 4. Tree-based parallel B&B approach.

5.4. Exploration strategies for our parallel B&B approaches

Another important aspect that influences the efficiency of our parallel approaches is the exploration strategy used to explore the search tree. Indeed, two main exploration strategies exist in the literature: Breadth-First Strategy (BFS) and Depth First Strategy (DFS).

The BFS begins by exploring all the sub-problems of a given level before starting the exploration of lower levels. Unfortunately, this strategy often results in a huge memory footprint without even reaching leaf nodes, making it inefficient and impossible to use for large graphs. In addition to the huge memory footprint of this strategy, it also implies a poor pruning process which increases the B&B complexity dramatically, i.e., the UB stays the same since we reach leaf nodes only in the last level of the search tree.

In our approaches, we opted for the DFS. After the branching process, this latter strategy explores the most recently created sub-problem added to the work-pool. Hence, a global strategy that explores leaf nodes (solutions) first, which improves the UB. Thus, avoiding the exploration of a huge number of non-promising branches. In addition, the choice of this strategy is motivated by its small memory footprint allowing us to treat large graphs. Our proposed parallel approaches are viewed as high-level parallelization where several instances of the B&B algorithm explore the search space using one or several work-pools.

In our First parallel approach ($WS_{B\&B}$), a standard DFS strategy is used. i.e., Each instance of the B&B algorithm picks up from the shared work-pool the most recent added sub-problem. After that, the selected sub-problem will be the subject of the branching and bounding operations which creates a set of actives sub-problems added directly to the shared work-pool.

In our second parallel approach ($TB_{B\&B}$), two exploration strategies are used according to the role of the process. A DFS for the workers and a mixed exploration strategy for the master. Indeed, the master explores the first levels of the tree using the BFS to generate a large number of sub-problems divided between the workers. After that, the master uses a standard DFS exploration to prevent memory saturation.

5.5. Load-balancing strategies

Due to the complexity of the GED problem and the irregular workload of B&B branches, a load-balancing strategy must be used. This strategy aims to increase the efficiency of our tree-based parallel approach ($TB_{B\&B}$) by avoiding the idleness of parallel threads. In the following, we describe two load-balancing strategies we propose.

5.5.1. The use of load-balancer (LB_1)

Our first load-balancing strategy uses the master process as a load balancer. This strategy avoids the idleness by giving a read/write access to all parallel B&B instances (workers) to pick up a sub-problem from the load-balancer work-pool whenever their own work-pools are empty. Furthermore, the load-balancer uses an adaptive exploration strategy to ensure the wide availability of sub-problems. This adaptive strategy uses the BFS exploration model whenever the size of the load-balancer work-pool is less than a certain limit. Beyond this limit, the load-balancer adapts its exploration strategy and uses the DFS to explore the search tree. This strategy aims to enable wide availability of sub-problems without generating a huge memory footprint. Whenever a thread work-pool is empty, the corresponding thread picks up a sup-problem from the load-balancer work-pool and completes its exploration in its own work-pool. Moreover, Fig. 5 describes our tree-based parallel B&B algorithm using this load-balancing strategy.

To minimize the communication cost, each thread whose work-pool is empty chooses a sub-problem from the head of the load-balancer work-pool since it has a large number of unprocessed vertices and edges. The end of this parallel version is reached whenever the private and load-balancer work-pools are both empty.

5.5.2. K-iterations parallel approaches (LB_2)

The second way to ensure a fair workload among all parallel processes is to combine both work-stealing and tree-based approaches (WS_B&B and TB_B&B). Therefore, each parallel B&B instance has access to two work-pools as shown in Fig. 6. A private local work-pool owned by the process (master or workers) and one global work-pool shared between all parallel processes. The idea here is that each parallel process picks up a sub-problem from the shared work-pool and explores it recursively in its own work-pool. When the number of performed iterations reaches a certain limit denoted by k, the corresponding process inserts all sub-problems from its work-pool into the global shared work-pool. This process is repeated until the shared and private work-pools are empty. In this way, we guarantee a fair workload distribution between all parallel processes. As depicted in Fig. 6, the main idea here is to allow the parallel processes to perform k-iterations at a time and then merge their own local work-pool into one global shared work-pool. To ensure a depth-first exploration of the search tree, all parallel processes use a standard DFS strategy. In addition, the global shared work-pool is sorted automatically after each insertion according to the number of processed vertices. In other words, sub-problems with a higher number of processed vertices are chosen first (at the head of the list).

6. Performance evaluation

In this section, we investigate the ability of our proposed parallel approaches and load-balancing strategies to efficiently reduce the running time for optimally solving the GED problem. To this aim, two sets of experiments are conducted using well-known datasets and cost functions. The first one aims to find the best parameters for our two parallel approaches and measures our proposed load-balancing strategies' impact on the execution time and the number of explored

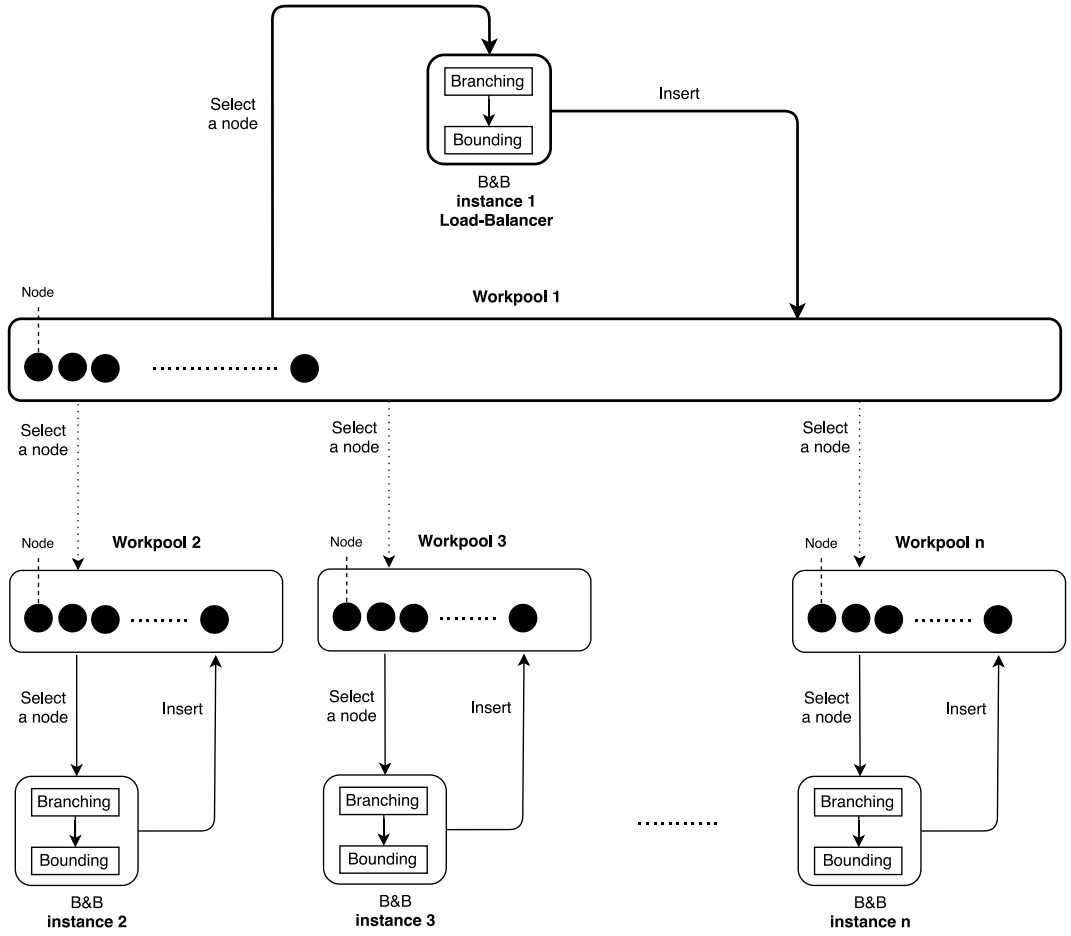


Fig. 5. Parallel B&B approach using a load-balancer.

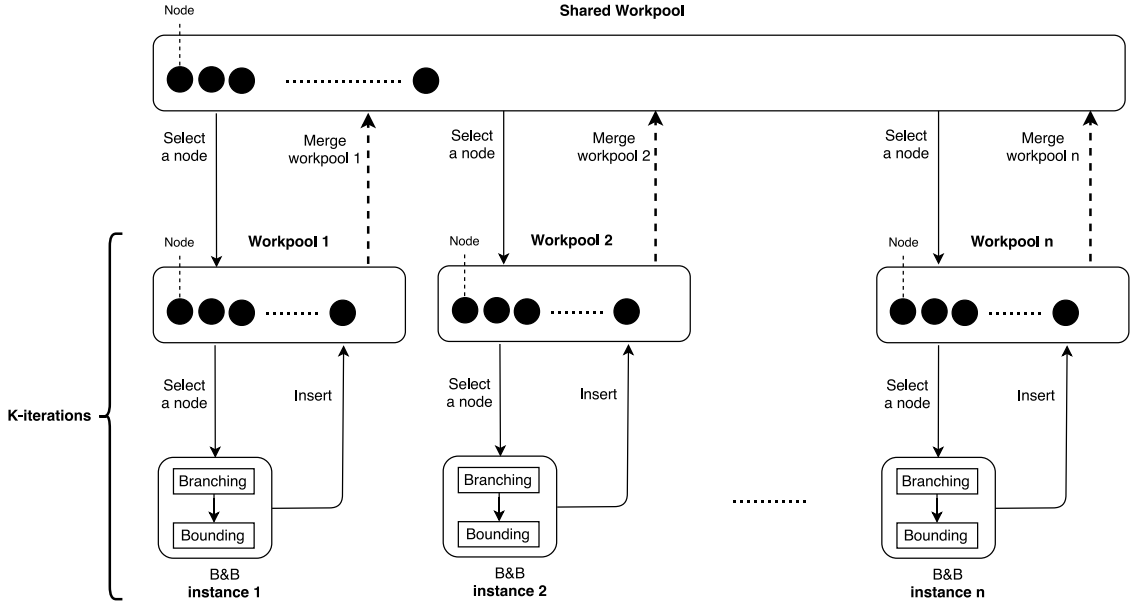


Fig. 6. Parallel B&B approach using our second load-balancing strategy.

Table 1
Datasets information.

Dataset	NB graphs	mean #nodes	mean degree	min #nodes	max #nodes
Alkane	150	8.9	1.8	1	10
Acyclic	185	8.2	1.8	3	11
PAH	94	20.7	20.4	10	28
MAO	68	18.4	2.1	11	27

Table 2
Cost Setting for vertex and edge edit operation.

	Vertex			Edge		
	Sub	Del	Ins	Sub	Del	Ins
Setting 1	2	4	4	1	1	1
Setting 2	2	4	4	1	2	2
Setting 3	6	2	2	3	1	1

nodes. The second set of our experiments evaluates and compares the performance of the different parallel methods in accelerating the time needed to solve well-known datasets.

Computing Platform: We performed our experiments on a single HPC compute node from the IBNBADIS cluster located at the CERIST research center. This compute node contains two Intel Xeon processors (E5-2650) with 8 CPU cores of 2 GHz speed each and 32 Go of memory, running under Linux operating system. Our parallel code is written in JAVA.

Datasets: We performed our experiments using two large datasets. The first dataset contains PAH, MAO, Alkane, Acyclic, and GREC.

- PAH, MAO, Alkane, and Acyclic are part of the GREYC's Chemistry dataset, which contains several archives and links to various chemical databases of molecules. These datasets can be obtained using the following link: https://brunl01.users.greyc.fr/CHEMISTRY/.
- GREC is a subset of the IAM graph database repository proposed by Riesen K. and Bunke H. in [36]. This dataset can be obtained using the following link: http://www.fki.inf.unibe.ch/databases/iam-graph-database.

All datasets are in Graph exchange Language (GXL) format[1] and defined as follows: The GREC dataset is decomposed into several subsets, each one contains graphs that have the same size, groups of 5, 10, 15, and 20 vertices. For the other remaining datasets, Table 1 summarizes some important information about these datasets.

The second dataset is the TUDataset [37],[2] where we used the following dataset: benzene, MUTAG, AIDS, Aspirin, TRIANGLES, MSRC_21. These datasets are transformed into GXL format.

Cost Function: The cost function represents the weighting of each edit operation (substitution, deletion, or insertion) on both vertices and edges. Therefore, the cost function affects the amount of the processing time and the memory space needed by the B&B algorithm to solve the GED problem optimally, i.e., the algorithm's complexity differs significantly by changing the cost of different edits operations. The difference in complexity is explained by the fact that the search tree generated varies according to the user costs, which implies different optimal paths and, thus, different complexity. For this reason, it is essential to use several cost settings. Three cost settings have been used in our experiments to test the robustness of our parallel approaches. These settings were used in the competition on graph matching organized in 2016 (https://gdc2016.greyc.fr/#ged). Moreover, Table 2 describes the used settings. Each setting favorites either substitution (Sub) or deletion/insertion (Del/Ins) edit operations.

[1] GXL is defined as an XML sub-language, which offers support for exchanging instance graphs together with appropriate information in a uniform format (http://www.gupro.de/GXL/).

[2] TUDataset: https://chrsmrrs.github.io/datasets/

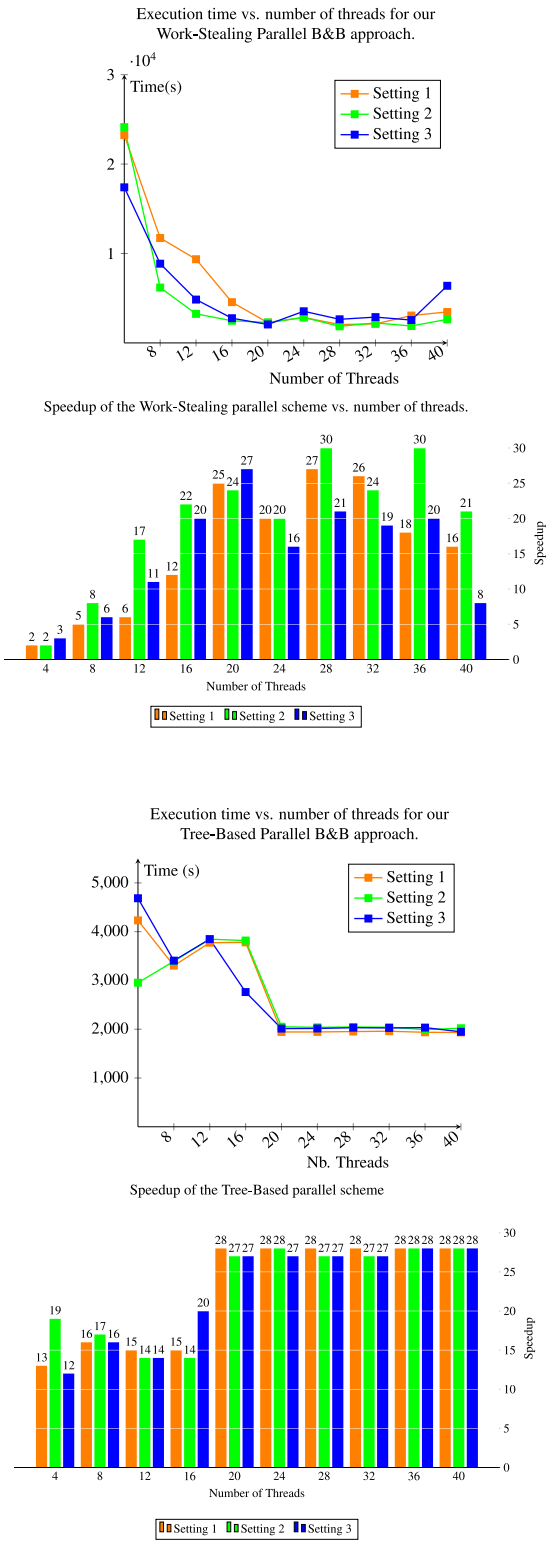


Fig. 7. Variation of the execution time and speedup according to the number of used threads for our work-stealing and tree-based parallel approaches using the three cost-settings in Table 2.

6.1. Finding the best parameters

In this section, we aim to find the required number of parallel threads that maximize the performance of our proposed parallel approaches. In addition, we will present in this section the results of our

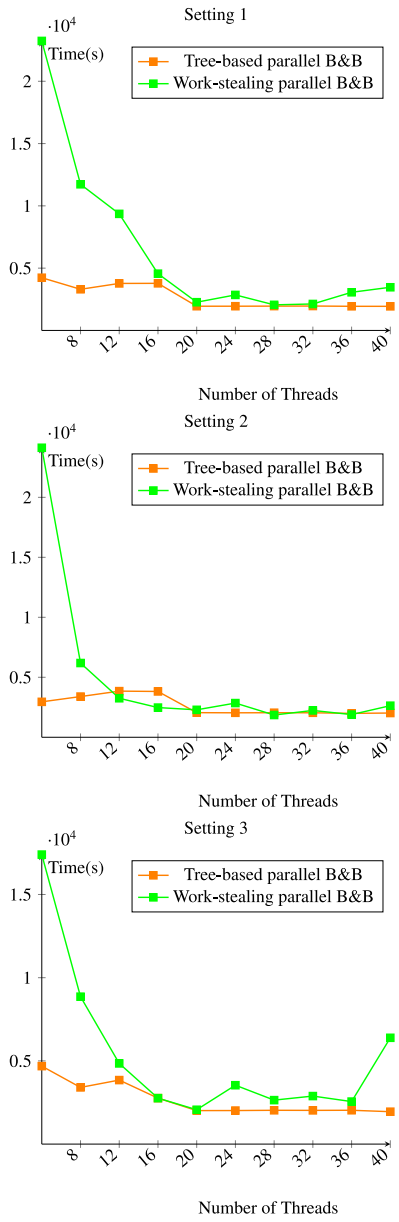


Fig. 8. Comparing the performance of our two proposed parallelization approaches for each setting in Table 2.

two load-balancing strategies. For this reason, all experiments in this section are performed using only two medium size graphs from the PAH dataset.

Fig. 7 shows the variation of the execution time and speedup of our two parallel B&B approaches using the cost settings in Table 2. The first interesting observation from this figure is the positive impact of parallelization in improving the complexity of the B&B algorithm. Indeed, using twenty parallel threads, our two parallel approaches improved the complexity by a factor of 27 × faster than the serial version of the B&B algorithm executed under the same test configuration.

We can notice from Fig. 7 that the curves of our parallel B&B approaches, for all settings, have the same behavior. In other words, both parallel B&B approaches have two phases. A first phase where increasing the number of parallel threads improves the running time. i.e., Increasing the speedup. After reaching the threshold of twenty parallel threads, a second phase begins where adding new parallel threads does not affect much the execution time and, thus, the speedup. This behavior is closely related to the number of computing cores

Table 3
Comparison between the Work-stealing parallel B&B and the Tree-based parallel B&B approaches.

Evaluation criteria	Work-stealing parallel B&B	Tree-based parallel B&B
Memory Utilization	Low	High
Scalability	Low	Embarrassingly Parallel
Diversification Gain	Low	High
Load Balancing Problem	none	Very high

available in our station, which is sixteen CPU cores. In more detail, the number of parallel threads in the first phase is below the number of CPU cores available, which explains the decrease in complexity. While in the second phase, the number of parallel threads is greater than the number of computing cores. Thereby, the system schedule sequentially the additional work-load, which slows down the parallel approaches.

As indicated by the same figure, the best performance for both parallel approaches is reached for a number of parallel threads equal to twenty. This number represents the number of parallel threads that fully exploit the computing resources of our workstation. For this reason, and for all incoming experiments, we fixed the number of parallel threads for all parallel B&B versions to twenty.

Fig. 8 shows, for all settings, a comparison between the performance of the work-stealing parallel B&B and the performance of the tree-based parallel B&B approaches when increasing the number of parallel threads. We can notice from Fig. 8 that the complexity of both parallel B&B approaches decreases when increasing the number of parallel threads before reaching approximately the same results. However, the slope of the decrease is not the same. Indeed, the curve of the work-stealing parallel version decreases gradually, which induces a sharper slope compared to the tree-based parallel version. The improvement in this latter is made in two steps, the first step between four and sixteen threads and a second step between twenty and forty threads. The slope of the tree-based parallel version is not as sharp as the slope of the work-stealing version. This can be explained by the efficient improvement in the complexity of the tree-based version, even for a low number of parallel threads. Indeed, Unlike the work-stealing version, the tree-based version explores several regions of the search tree simultaneously, which allows to take advantage of the diversification gain. In other words, the tree-based parallel version has more chance to encounter better solutions, improving the UB. Furthermore, this allows to prune many branches in the search tree explored in the work-stealing version, which explains the gain in complexity and the super-linear speedup.

Table 3 shows a comparison between our two proposed parallel B&B approaches to identify the strengths and the weaknesses of each approach.

The exact GED problem is known in the literature to have compute-bound and memory-bound limitations simultaneously. For this reason, our first discussed criteria is memory usage in both approaches. Our work-stealing parallel approach uses less memory space than the tree-based parallel approach. Indeed, the work-stealing parallel approach involves exploring only one search tree in depth using several threads, which induces a fixed memory footprint that does not change much when increasing the number of parallel threads. On the other hand, unlike the work-stealing approach, the tree-based approach involves the exploration of several search trees in parallel, which induces a huge memory footprint that increases according to the number of used parallel threads.

Scalability refers to the possibility of improving parallel algorithms' performance when the number of parallel threads reaches hundreds. Our work-stealing parallel approach does not perform well when increasing the number of parallel threads to reach hundreds. This behavior is explained by the fact that all parallel threads operate on a single search tree. The access to this latter, to insert or peek-up a node, is done sequentially. Thereby, it induces a bottleneck effect on

the performance. This explains why the complexity of this approach increases when reaching 40 threads, as shown in Fig. 8 (setting 1 and setting 3). Unlike the work-stealing approach, the tree-based parallel B&B approach is embarrassingly parallel due to the low communication and synchronization costs between the parallel threads since each thread operates on its own search tree.

The diversification (gain from expanding the search) represents an essential technique to tackle NP-hard optimization problems. Moreover, it represents an essential technique to consider, especially when exploring a search tree, due to its intuitive parallel nature. On the one hand, our work-stealing approach explores the search tree in one direction (approximately the same as the sequential version but faster), which induces a low diversification gain. On the other hand, our tree-based approach simultaneously explores several parts (directions) of the search tree, allowing a faster improvement of the upper bound. It avoids the exploration of a huge number of branches explored in serial and parallel work-stealing versions. This results in a super-linear speedup since the explored parts of the search tree are not the same between the serial and parallel tree-based versions. Moreover, the huge difference in complexity in favor of our tree-based approach in Fig. 8 (between 4 and 16 threads) results from the diversification gain. When the number of parallel threads exceeds 16, another problem that neutralizes the diversification gain occurs. This problem is called *load-balancing problem*.

The load-balancing problems refer to the unfair distribution of the work-load between parallel threads, inducing the idleness of some threads and a long-running time for others. This is due mainly to the imbalanced work in search tree nodes. Our work-stealing version has no load-balancing problem since parallel threads operate on a single shared work-pool. However, unlike the work-stealing version, the performance of our tree-based approach is seriously influenced by the load-balancing problem, especially for a large number of parallel threads. This neutralizes the diversification gain as shown in Fig. 8.

6.1.1. Impact of our load balancing strategies

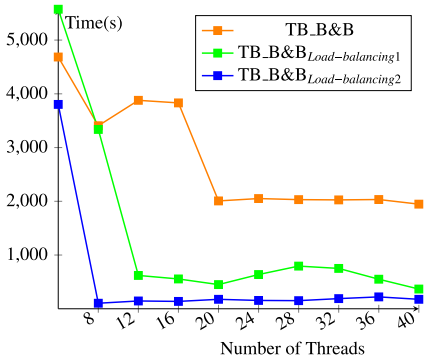
Two load-balancing strategies have been proposed to overcome the load-balancing problem in the tree-based parallel B&B approach. The first load-balancing strategy (LB_1) uses the master process as a load-balancer. i.e. Avoid the idleness of threads by giving to them a read/write access to the load-balancer work-pool whenever their own work-pools are empty. In comparison, the second load-balancing strategy (LB_2) merges all private work-pools into a global shared one after each k iterations.

In the following, we will show the impact of each load balancing strategy on the performance of the tree-based parallel B&B approach.

Fig. 9(a) shows the performance of the tree-based parallel B&B approach, with and without load-balancing strategies, when increasing the number of parallel threads. Similarly, Fig. 9(b) shows the variation of the number of explored nodes by the main thread when increasing the number of parallel threads. The first conclusion that can be made from the figure is the positive and significant impact of our proposed load-balancing strategies on the performance of the tree-based parallel B&B approach. The improvement in the execution time is the result of the fair work-load distribution between the parallel threads, which avoids the idleness of threads and thus, maximizing the diversification gain. Moreover, the curves of our two load-balancing strategies in Fig. 9(b) confirm this theory and show a constant decrease in the number of explored nodes by the master when increasing the number of parallel threads. This is not the case with the standard tree-based parallel version, especially for a large number of parallel threads in which the number of explored nodes stays the same.

Fig. 10(a) shows the relative speedup of our tree-based parallel B&B approach (with/without load-balancing strategies) compared to our serial implementation of the B&B algorithm executed under the same test configuration. Moreover, Fig. 10(b) shows the obtained improvement factor when adding our proposed load-balancing strategies to the tree-based parallel B&B algorithm.

(a) Execution Time vs the number of threads.



(b) Explored Nodes vs. the number of threads.

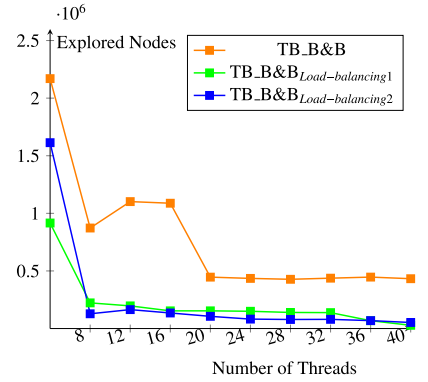


Fig. 9. Variation of the execution time and mean explored nodes according to the number of used threads.

Fig. 10 shows that the first load-balancing strategy improved the performance of the tree-based parallel approach by a factor of 4x. The second load-balancing strategy achieved even more significant performance by reporting an improvement factor around 11 times faster. By computing the relative speedup (Fig. 10(a)) against our serial B&B version executed in the same test configuration, we can see that adding load-balancing strategies to the tree-based parallel approach allowed us to achieve a speedup around 124 times faster when using the first load-balancing strategy, and 318 times faster when using our second load-balancing strategy. The speedup reported in the figure shows the huge gain of using parallelism to solve the GED problem. Indeed, by using only 16 CPU cores, we have reported a speedup around 318 times faster due to the high impact of the diversification gain boosted by the proposed load-balancing strategies. Indeed, exploring several parts of the search tree simultaneously allows a faster improvement of the upper bound. Thus, avoiding the exploration of a huge number of branches explored in the serial version of the B&B algorithm. By ensuring a fair work-load between the parallel processes, the impact of the diversification gain is higher, which explains the super-linear speedup.

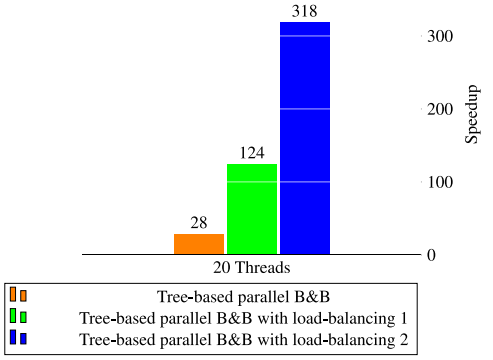
6.2. Performance evaluation of our proposed approaches

In the following, we will focus mainly on the computation time of the different proposed approaches and load-balancing strategies. All approaches have twenty parallel threads and are tested using the chosen graphs in Table 4. The idea behind selecting just some graphs is: (1) Performing all combinations is not practical due to a large number of graphs in each dataset. (2) To allow other researchers to fairly compare theirs approaches against ours due to the huge lack in this aspect in the literature. Table 4 contains fifteen combinations where each one is identified by a codification (Column Cod.). For example,

Table 4
Graphs combinations used to test our parallel B&B approaches.

		Cod.	G_1	G_2
Small	Acyclic	Ac11_1	di-tert-butyl_peroxide.gxl	1,1-dimethoxyhexane.gxl
		Ac11_2	1,1-diisopropoxyethane.gxl	2,4-dimethoxy-2-methylpentane.gxl
		Ac11_3	1,1-dipropoxyethane.gxl	1,4-diethoxybutane.gxl
Medium	Mao	Ma17_1	molecule33.gxl	molecule18.gxl
		Ma17_2	molecule34.gxl	molecule20.gxl
		Ma17_3	molecule36.gxl	molecule21.gxl
	Grec	Gr15_1	image11_24.gxl	image11_22.gxl
		Gr15_2	image11_22.gxl	image18_16.gxl
		Gr15_3	image11_30.gxl	image18_14.gxl
Large	Grec	Gr20_1	image11_42.gxl	image11_50.gxl
		Gr20_2	image11_46.gxl	image11_50.gxl
		Gr20_3	image5_45.gxl	image5_50.gxl
	PAH	Pa18_1	triphenylene.gxl	benzo[a]anthracene.gxl
		Pa18_2	naphthacene.gxl	benzo[c]phenanthrene.gxl
		Pa18_3	chrysene.gxl	triphenylene.gxl

(a) Relative Speedup (Against our serial version).



(b) Improvement factor against the tree-based parallel approach.

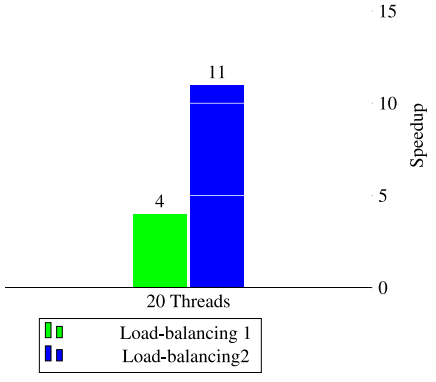


Fig. 10. Improvement factor and relative speedup of our proposed load-balancing strategies using twenty threads.

Ac11_1 denotes the combination of two graphs with eleven vertices obtained from the Acyclic dataset i.e. di-tert-butylperoxide.gxl (G1) and 1,1-dimethoxyhexane.gxl (G2). These combinations are divided into small, medium, and large. The first set (small) contains three combinations of graphs obtained from the Acyclic dataset. The second set (medium) contains six combinations obtained from Mao and Grec datasets. Finally, The last set (Large) contains also six combinations of graphs obtained from Grec and PAH datasets.

Table 5, Table 6, and Table 7 show the performance, in terms of execution time, of our proposed parallel B&B approaches and load-balancing strategies using the three cost-settings in Table 2 and the graph combinations in Table 4.

Table 5
Execution time of our proposed parallel B&B approaches and load-balancing strategies using cost Setting 1.

		Cod.	Serial B&B	$WS_{B\&B}$	$TB_{B\&B}$	$TB_{B\&B}$ $LB1$	$TB_{B\&B}$ $LB2$
Small	Acyclic	Ac11_1	58	6	6	2	2
		Ac11_2	15	1	2	0	0
		Ac11_3	7	0	1	0	0
Medium	Mao	Ma17_1	13444	908	600	78	26
		Ma17_2	12206	478	869	295	337
		Ma17_3	22197	746	2405	318	22
	Grec	Gr15_1	19681	821	1422	433	2
		Gr15_2	344204	11035	12281	1352	490
		Gr15_3	344055	13744	12332	850	548
Large	Grec	Gr20_1	–	–	–	–	527
		Gr20_2	–	–	–	–	1124
		Gr20_3	–	–	–	–	3130
	PAH	Pa18_1	6978	751	343	107	278
		Pa18_2	55877	2820	1918	492	107
		Pa18_3	28584	3366	1272	115	138

Table 6
Execution time of our proposed parallel B&B approaches and load-balancing strategies using cost Setting 2.

		Cod.	Serial B&B	$WS_{B\&B}$	$TB_{B\&B}$	$TB_{B\&B}$ $LB1$	$TB_{B\&B}$ $LB2$
Small	Acyclic	Ac11_1	74	2	7	2	2
		Ac11_2	15	0	1	0	0
		Ac11_3	7	0	1	0	0
Medium	Mao	Ma17_1	14968	2177	1543	1892	185
		Ma17_2	13756	1690	1476	601	330
		Ma17_3	18610	1949	2778	345	46
	Grec	Gr15_1	108547	3655	17499	1728	117
		Gr15_2	806310	15068	115674	24832	1774
		Gr15_3	806144	15035	116064	24724	2804
Large	Grec	Gr20_1	–	–	–	–	1851
		Gr20_2	–	–	–	–	5090
		Gr20_3	–	–	–	–	79552
	PAH	Pa18_1	6998	565	355	170	372
		Pa18_2	56021	1609	1994	625	26
		Pa18_3	28601	3018	1311	230	168

The first three columns from these tables report respectively the dataset's name and size, in-addition to the codification of the chosen graphs. Column Serial B&B indicates the execution time of our sequential B&B algorithm under the same test configuration. Column $WS_{B\&B}$ reports the results of our work-stealing parallel B&B approach using twenty parallel threads. Column $TB_{B\&B}$ reports the results of our tree-based parallel B&B approach using twenty parallel threads. Finally,

(a) Increase in number of optimally solved combinations compared to serial approach.

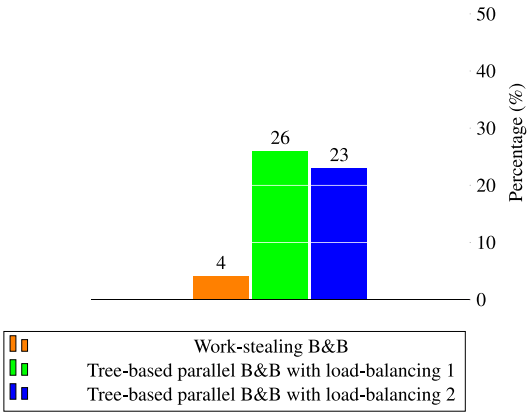


Fig. 11. Improvement percentage in the number of optimally solved combinations within one hour execution time when using our proposed parallel approaches.

Table 7
Execution time of our proposed parallel B&B approaches and load-balancing strategies using cost Setting 3.

		Cod.		$WS_{B\&B}$	$TB_{B\&B}$	$TB_{B\&B}$ $LB1$	$TB_{B\&B}$ $LB2$
Small	Acyclic	Ac11_1	58	3	7	2	2
		Ac11_2	14	0	2	0	0
		Ac11_3	5	0	1	0	0
Medium	Mao	Ma17_1	47986	4029	5046	1892	25
		Ma17_2	40619	3478	4248	1064	2784
		Ma17_3	10668	1014	2099	500	25
	Grec	Gr15_1	408	28	70	20	20
		Gr15_2	3	3	3	3	3
		Gr15_3	3	3	3	3	3
Large	Grec	Gr20_1	–	–	–	–	23
		Gr20_2	–	–	–	–	12
		Gr20_3	–	–	–	–	97
	PAH	Pa18_1	6998	361	350	219	546
		Pa18_2	55992	3317	1994	560	59
		Pa18_3	28589	1627	1309	288	109

(a) Speedup of our parallel approaches using MUTAG dataset.

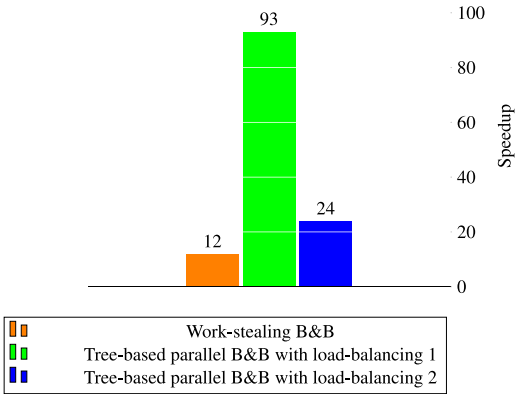


Fig. 12. The average speedup obtained by our parallel B&B approaches.

Columns $TB_{B\&B}$ with $LB1$ and $TB_{B\&B}$ with $LB2$ report the results of adding our two load-balancing strategies to the tree-based parallel B&B approach.

The first observation from Tables 5, 6, and 7 is the ability of our proposed approaches and load-balancing strategies to reduce the execution time of our serial B&B version efficiently. Indeed, this improvement is noticed for all tested cost settings, even for the small-size test combinations. Moreover, the graph combinations Gr15_2 and Gr15_3 from Tables 6 and 7, show the impact of the cost settings on the time needed for optimally solving the GED problem. Indeed, the serial execution time varies from hundreds of thousands to only a few hundred or even less. For this reason, it is important to test the proposed approaches under different cost settings. To compare the results of our proposed approaches, it is interesting to observe the *statewise dominance*. This latter represents the simplest case of stochastic dominance [38,39], and it is defined as follows: Approach A is statewise dominant over Approach B if A gives at least as good as B for every tested benchmark, and a strictly better results for at least one benchmark. In the following, we will first compare the results of our two parallel approaches, and then we will evaluate the impact of our proposed load-balancing strategies. Table 5, Table 6, and Table 7 show that both work-stealing and tree-based parallel B&B approaches are statewise dominant over the serial B&B version. However, there is no dominance between the two parallel approaches. For some test combinations, the work-stealing approach is better and for others, the tree-based version is better. But globally, we can say that the work-stealing version is better since it outperforms the results of the tree-based approach in *21* cases against 12 cases for the tree-based version in all cost settings. When adding load-balancing strategies to our tree-based approach, this latter becomes statewise dominant over the work-stealing parallel approach.

Indeed, the unbalanced work-load between threads in the tree-based version reduces enormously the diversification gain, which explains why the work-stealing version was better. This also explains why the results of the tree-based parallel approach with load-balancing strategies are much better compared to the basic version of the tree-based approach. If we compare the results of our two load-balancing strategies, we cannot establish a dominance relation between the two strategies. However, we can say that the results of our second load-balancing strategy are mostly better than the first load-balancing strategy. In addition, the tree-based parallel B&B using our second load-balancing strategy is the only approach that reported optimal results for large Grec combinations, which indicates the need for efficient load-balancing strategies.

6.3. Performance results for TUdataset

Table 8 shows the mean edit distance obtained by our parallel approaches during one hour execution time. The first two columns show the dataset's name and its average number of vertices. For each dataset, we tested 57 graph combinations. Each approach is represented by three columns indicating the mean edit distance using the three cost

Table 8
Mean edit distance results of parallel approaches using TUDataset and the three settings in Table 1.

GED-mean 1 h		Comb.	$WS_{B\&B}$			$TB_{B\&B}LB1$			$TB_{B\&B}LB2$		
Datasets	avg_nodes		ST1	ST2	ST3	ST1	ST2	ST3	ST1	ST2	ST3
Benzene	12	57	1.26	2.52	1.26	1.26	2.52	1.26	1.26	2.52	1.26
AIDS	15.7	57	32.68	32.89	39.78	31.52	40.42	**39.05**	**30.78**	**39.10**	39.15
MUTAG	17.93	57	26.26	33.89	23.73	24.91	31.78	**21.94**	**24.73**	**30.42**	22.36
Triangles	20.85	57	13	26.84	13	**11.73**	24.10	**12.36**	11.94	**22.21**	12.78
Aspirin	21.00	57	48.42	97.89	75.36	**13.57**	**27.15**	**13.47**	14.15	28.21	14.63
MSRC_21	77.52	57	415.31	675.47	**509.42**	635.31	920.94	509.73	**378.47**	**623.26**	564.68

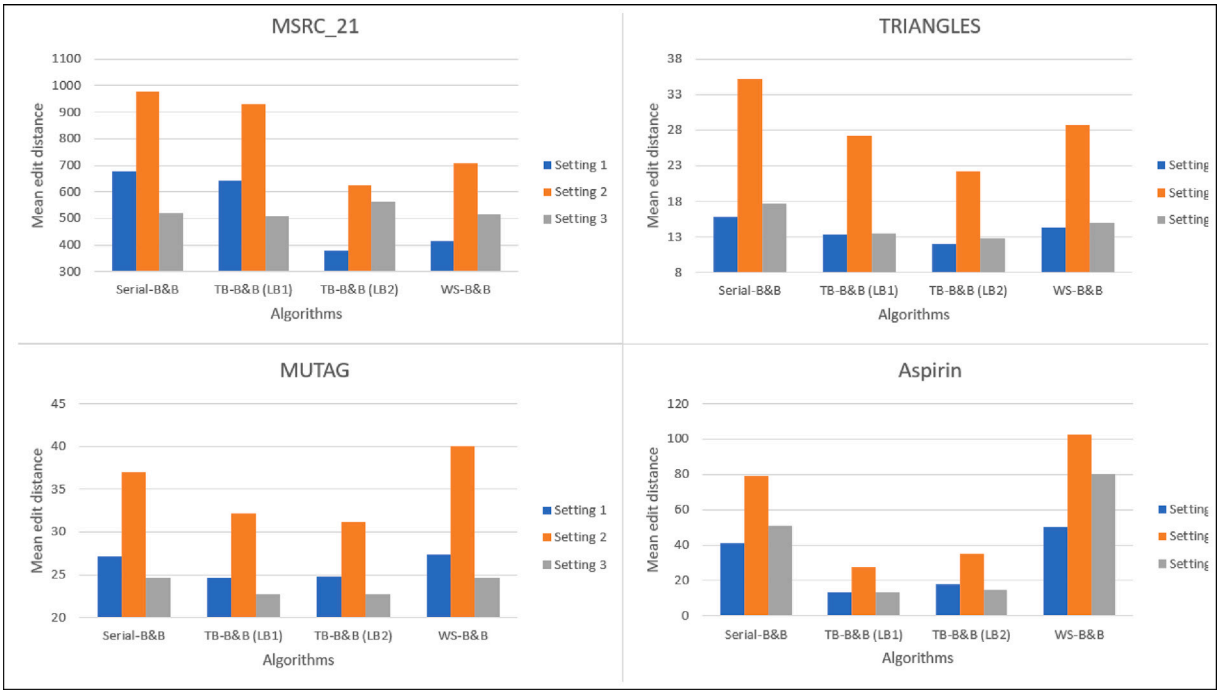


Fig. 13. Mean edit distance of serial and parallel B&B approaches using four datasets (MSR_21, TRIANGLES,MUTAG, and Aspirin).

settings in Table 1. We limited the execution time for each combination to one hour because each combination may take several days to finish since our approaches are optimal. Thus, limiting the time allows us to test many graph combinations.

The results obtained for the TUDataset confirm the results reported earlier. The mean edit distance obtained by our TB approaches (LB1 and LB2) is better than that obtained by the $WS_{B\&B}$ version for all the datasets and cost settings. As explained earlier, this is the result of diversifying the search process in the TB approaches that allows the explore different parts of the search tree simultaneously. Thus, more chance to explore better paths and improve the edit distance. Moreover, the TB approaches (LB1 and LB2) are similar in mean edit distance, with slightly better results when using the k-level technique to ensure fair work-load distribution.

Fig. 11 shows that our TB approaches increase the number of optimally solved combinations by 27% compared to the serial B&B version. On the other hand, the work-stealing version improved the number of solved combinations by only 4%. Moreover, the tree-based (LB1) approach was the only approach able to solve 23 over 57 graph combinations in the Aspirin dataset.

Fig. 12 shows the average speedup obtained by our parallel B&B approaches considering only the instances solved optimally by the serial version in the MUTAG dataset. The speedup of our parallel approaches for each instance varies between 5× and 250×. The best performance is reached when using $TB_{B\&B}$ with *LB1* with an average speedup of 93×, followed by the $TB_{B\&B}$ with *LB2* with 24× and finally $WS_{B\&B}$ with 13×.

Fig. 13 shows the mean edit distance obtained by our serial and parallel approaches in one hour execution time. The first observation is the ability of parallel approaches to reduce the mean edit distance compared to our serial B&B version. This is not always the case, as we can see for Aspirin and MUTAG datasets in which the serial version has better results than the WS version. Moreover, the tree-based approaches show a significant improvement in reducing the mean edit distance for all datasets and cost settings. The performance of the proposed parallel approaches varies from one dataset to another. Therefore, there is no strict superiority for one approach over the others, especially for the two tree-based approaches.

7. Conclusion and perspectives

This work considers the parallelization of the B&B algorithm to solve the Exact Graph Edit Distance problem referred to as Exact GED. This problem measures the amount of dissimilarity between two graphs by finding the best set of edit operations to transform one graph into another. The GED is widely used in various applications, especially in areas related to pattern recognition. However, the use of exact GED is limited in practice due to its prohibitive complexity when dealing with large graphs. To overcome this drawback, we proposed two efficient shared-memory parallel B&B schemes. The first one is based on a work-stealing strategy in which several B&B instances operate on the same work-pool. The second one aims to build the B&B search tree in parallel. Therefore, several B&B instances operate on their private work-pool. Furthermore, we proposed two load-balancing strategies to avoid the

idleness of parallel threads. Experiments on several reference datasets showed the efficiency of our proposals in reducing the execution time and exploiting the power of multi-core CPU processors.

It turns out that even when using parallelism, the running time and space needed for solving the problem of exact GED is still very high. For this reason, our future works will investigate using tree-based approximate approaches to solve this challenging problem.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.parco.2022.102984.

Data availability

Link to our project (source code/ data): https://github.com/chegrane/ged_parallel/.

Acknowledgment

This work was supported by the DGRSDT, Algeria grant FNRSDT N° 253.

References

[1] H. Bunke, G. Allermann, Inexact graph matching for structural pattern recognition, Pattern Recognit. Lett. 1 (4) (1983) 245–253.
[2] Z. Zeng, A.K. Tung, J. Wang, J. Feng, L. Zhou, Comparing stars: On approximating graph edit distance, Proc. VLDB Endow. 2 (1) (2009) 25–36.
[3] K. Riesen, Structural Pattern Recognition with Graph Edit Distance: Approximation Algorithms and Applications, in: Advances in Computer Vision and Pattern Recognition, Springer International Publishing, 2016, URL: https://books.google.fr/books?id=2PJeCwAAQBAJ.
[4] K. Riesen, S. Fankhauser, H. Bunke, Speeding up graph edit distance computation with a bipartite heuristic, in: MLG, 2007.
[5] K. Riesen, H. Bunke, Approximate graph edit distance computation by means of bipartite graph matching, Image Vis. Comput. 27 (7) (2009) 950–959.
[6] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, P. Martineau, An exact graph edit distance algorithm for solving pattern recognition problems, in: 4th International Conference on Pattern Recognition Applications and Methods 2015, 2015.
[7] B. Gaüzere, S. Bougleux, K. Riesen, L. Brun, Approximate graph edit distance guided by bipartite matching of bags of walks, in: Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition, SSPR, Springer, 2014, pp. 73–82.
[8] M. Eshera, K.-S. Fu, A graph distance measure for image analysis, IEEE Trans. Syst. Man Cybern. (3) (1984) 398–408.
[9] H. Bunke, K. Riesen, Recent advances in graph-based pattern recognition with applications in document analysis, Pattern Recognit. 44 (5) (2011) 1057–1067.
[10] W. Zheng, L. Zou, X. Lian, D. Wang, D. Zhao, Graph similarity search with edit distance constraint in large graph databases, in: Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management, ACM, 2013, pp. 1595–1600.
[11] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.
[12] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, P. Martineau, A distributed algorithm for graph edit distance, in: DBKDA 2016, 2016, p. 76.
[13] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, P. Martineau, A parallel graph edit distance algorithm, Expert Syst. Appl. 94 (2018) 41–57.
[14] D.B. West, et al., Introduction to Graph Theory, vol. 2, Prentice hall Upper Saddle River, 2001.
[15] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, Int. J. Pattern Recognit. Artif. Intell. 18 (03) (2004) 265–298.
[16] A. Sanfeliu, K.-S. Fu, A distance measure between attributed relational graphs for pattern recognition, IEEE Trans. Syst. Man Cybern. (3) (1983) 353–362.
[17] M. Neuhaus, K. Riesen, H. Bunke, Fast suboptimal algorithms for the computation of graph edit distance, in: Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition, SSPR, Springer, 2006, pp. 163–172.
[18] J. Munkres, Algorithms for the assignment and transportation problems, J. Soc. Ind. Appl. Math. 5 (1) (1957) 32–38.
[19] R. Burkard, M. Dell'Amico, S. Martello, Assignment Problems: Revised Reprint, SIAM, 2012.
[20] K. Riesen, A. Fischer, H. Bunke, Computing upper and lower bounds of graph edit distance in cubic time, in: N. El Gayar, F. Schwenker, C. Suen (Eds.), Artificial Neural Networks in Pattern Recognition: 6th IAPR TC 3 International Workshop, ANNPR 2014, Montreal, QC, Canada, October 6-8, 2014. Proceedings, Springer International Publishing, Cham, 2014, pp. 129–140, http://dx.doi.org/10.1007/978-3-319-11656-3_12.
[21] D.B. Blumenthal, J. Gamper, Correcting and speeding-up bounds for non-uniform graph edit distance, in: Data Engineering (ICDE), 2017 IEEE 33rd International Conference on, IEEE, 2017, pp. 131–134.
[22] T. White, Hadoop-The Definitive Guide: Storage and Analysis at Internet Scale (revised and updated), O'Reilly, 2012.
[23] K. Gouda, M. Hassaan, Csi_Ged: An efficient approach for graph edit similarity computation, in: Data Engineering (ICDE), 2016 IEEE 32nd International Conference on, IEEE, 2016, pp. 265–276.
[24] D.B. Blumenthal, J. Gamper, Exact computation of graph edit distance for uniform and non-uniform metric edit costs, in: International Workshop on Graph-Based Representations in Pattern Recognition, Springer, 2017, pp. 211–221.
[25] D.B. Blumenthal, J. Gamper, On the exact computation of the graph edit distance, Pattern Recognit. Lett. 134 (2020) 46–57.
[26] D.B. Blumenthal, S. Bougleux, J. Gamper, L. Brun, GEDLIB: A C++ library for graph edit distance computation, in: International Workshop on Graph-Based Representations in Pattern Recognition, Springer, 2019, pp. 14–24.
[27] L. Chang, X. Feng, X. Lin, L. Qin, W. Zhang, Efficient graph edit distance computation and verification via anchor-aware lower bound estimation, 2017, arXiv preprint arXiv:1709.06810.
[28] R. Wang, Y. Fang, X. Feng, Efficient parallel computing of graph edit distance, in: 2019 IEEE 35th International Conference on Data Engineering Workshops, ICDEW, IEEE, 2019, pp. 233–240.
[29] R. Allen, L. Cinque, S. Tanimoto, L. Shapiro, D. Yasuda, A parallel algorithm for graph matching and its MasPar implementation, IEEE Trans. Parallel Distrib. Syst. 8 (5) (1997) 490–501.
[30] A. Dabah, I. Chegrane, S. Yahiaoui, Efficient approximate approach for graph edit distance problem, Pattern Recognit. Lett. (2021) http://dx.doi.org/10.1016/j.patrec.2021.08.027, URL: https://www.sciencedirect.com/science/article/pii/S0167865521003251.
[31] A.H. Land, A.G. Doig, An automatic method of solving discrete programming problems, Econometrica (1960) 497–520.
[32] T.G. Crainic, B. Le Cun, C. Roucairol, Parallel branch-and-bound algorithms, in: Parallel Combinatorial Optimization, Vol. 1, Wiley, 2006, pp. 1–28.
[33] H.W. Trienekens, A.d. Bruin, Towards a Taxonomy of Parallel Branch and Bound Algorithms, Technical Report, Erasmus School of Economics (ESE), 1992.
[34] B. Gendron, T.G. Crainic, Parallel branch-and-branch algorithms: Survey and synthesis, Oper. Res. 42 (6) (1994) 1042–1066.
[35] N. Melab, Contributions À La Résolution De Problèmes D'optimisation Combinatoire Sur Grilles De Calcul (Ph.D. thesis), 2005.
[36] K. Riesen, H. Bunke, IAM graph database repository for graph based pattern recognition and machine learning, in: Structural, Syntactic, and Statistical Pattern Recognition, Springer, 2008, pp. 287–297.
[37] C. Morris, N.M. Kriege, F. Bause, K. Kersting, P. Mutzel, M. Neumann, Tudataset: A collection of benchmark datasets for learning with graphs, in: ICML 2020 Workshop on Graph Representation Learning and beyond, GRL+ 2020, 2020, arXiv:2007.08663, URL: http://www.graphlearning.io.
[38] J. Hadar, W.R. Russell, Rules for ordering uncertain prospects, Am. Econ. Rev. 59 (1) (1969) 25–34.
[39] V.S. Bawa, Optimal rules for ordering uncertain prospects, J. Financ. Econ. 2 (1) (1975) 95–121.