Graph Edit Distance Compacted Search Tree

Ibrahim Chegrane[1,3(✉)], Imane Hocine[1,2], Saïd Yahiaoui[1], Ahcene Bendjoudi[1], and Nadia Nouali-Taboudjemat[1]

[1] CERIST, Research Center on Scientific and Technical Information, 16306 Ben Aknoun, Algiers, Algeria
{syahiaoui,abendjoudi,nnouali}@cerist.dz
[2] Ecole Nationale Supérieure d'Informatique, BP 68M, 16309 Oued-Smar, Alger, Algeria
bi_hocine@esi.dz
[3] CoBIUS Lab, Department of Computer Science, University of Sherbrooke, Sherbrooke, QC, Canada
ibrahim.chegrane@usherbrooke.ca

Abstract. We propose two methods to compact the used search tree during the graph edit distance (GED) computation. The first maps the node information and encodes the different edit operations by numbers and the needed remaining vertices and edges by BitSets. The second represents the tree succinctly by bit-vectors. The proposed methods require 24 to 250 times less memory than traditional versions without negatively influencing the running time.

Keywords: Graph Edit Distance (GED) · Compacted GED search space

1 Introduction

The Graph Edit Distance (GED) is a well-known metric used to compute the degree of dissimilarity between two graphs g_1 and g_2. It is generally used in pattern recognition [12], such as handwriting recognition [9] and document analysis [4]. The GED is defined as the minimum-cost sequence of edit operations needed to transform graph g_1 into graph g_2 [3]. The allowed operations are insertion, deletion, and substitution, which are applied on vertices and their corresponding edges. The GED computation is an NP-hard problem [13]. It has an exponential time complexity due to the exponential size of the generated search tree.

Bunke and Allermann were the precursors for solving the GED problem [3]. The authors used an A* based algorithm where the search tree is generated dynamically. In [8], the authors proposed an approximation for the GED problem called *A*-Beamsearch*. By limiting the size of the A* priority queue to a certain size s. To speed up the A* search process, [10] presents an effective heuristic that gives the estimated cost h and concludes a lower bound. This heuristic, called bipartite heuristic [9], has been discussed and improved in [11] and [2] to compute a more accurate lower bound. Authors in [1] proposed an approach called (DF_GED)

T. Skopal et al. (Eds.): SISAP 2022, LNCS 13590, pp. 181–189, 2022.
https://doi.org/10.1007/978-3-031-17849-8_14

based on the Depth-First Search (DFS) to reduce the amount of used memory space. It proposes an alternative to the A* algorithm without addressing the underlying data structure and data representation. In [6], a tree-based approximate approach that gives near-optimal results is proposed. Gouda and Hassaan in [7] proposed an edge-based DFS method called CSI_GED.

Existing methods [1,6,10] mainly focus on reducing the time complexity by using parallel techniques or heuristics which provide approximate results without addressing the used data structure. In this work we attempt to reduce the memory space inherent to the GED computation by proposing two methods to compact the GED search tree. First, we compact each field used in its nodes based on an efficient mapping to encode the information of the different edit operations. A single number encodes the hole edit operation including its type and involved vertices, and a BitSet encodes the needed remaining vertices and edges. This mapping allows us to represent each path as a sequence of numbers. Second, the search tree is represented succinctly by bit-vectors where only the active nodes are stored and all the ancestor parents are deleted. Our work is independent of the search tree algorithms. Therefore, the proposed approach represents a general framework that can be used with both algorithms based on best first search (A*) or on depth-first search (B&B). Experiments on well-known benchmarks show the efficiency of our proposed methods. It confirms that our methods reduce the memory used by a factor of 24x to 250x. Moreover, they do not negatively influence the running time. It generally gets the same processing time as traditional algorithms, and may achieve twice faster times with some benchmarks.

The rest of this paper is organized as follows. In Sect. 2, we present the search tree used to solve the GED problem. In Sect. 3, we describe our methods. Section 4 outlines the obtained results. Finally, conclusions are given in Sect. 5.

2 The Data Structure Used to Solve the GED Problem

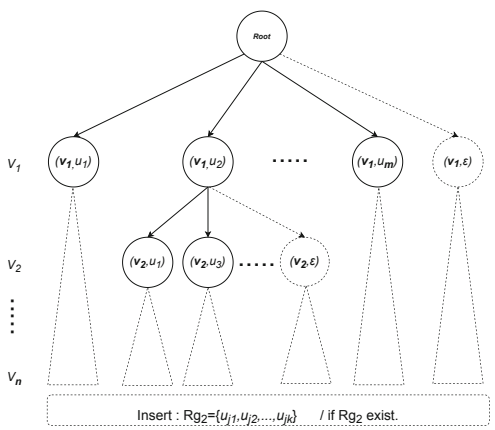


Fig. 1. Vertex edit operations in the search tree.

The GED search tree represents the mapping of the vertices of the first graph g_1 with the vertices of the second graph g_2 using the three edit operations: substitution, deletion and insertion. It generates for each vertex $v_i \in V_1$ all the possible substitutions with other vertices $u_j \in V_2$, and also the deletion of v_i. At the end, if there are still vertices in V_2, we insert them in one single operation. In addition to the list OPEN that is used in the tree exploration process.

The following information is associated with each node:

1. A pointer to keep relationships with its parent (to construct the edit path).
2. The vertex edit operation: The two involved vertex from g_1 and g_2; (v_i, u_j), (v_i, ϵ) or (ϵ, u_j). The vertex information may contain a weight or labels.
3. The implied edges operation which are either added or calculated directly.
4. The real cost g: the sum of costs from the root to this intermediate node.
5. The estimated cost h from this intermediate node to the leaf.
6. The remaining vertices of g_1: At each level l, each outgoing path has the same vertex v_l from g_1 and a different w_j from g_2 plus one deletion node. So it suffices to store the last index of the processed vertex (See Fig. 1).
7. The remaining vertices of g_2: We keep a list of non-processed vertices.
8. The remaining edges of g_1 and g_2: If we compute them, the processing time will increase. In contrast, we increase memory space needed if we store them.

3 Compacted Search Tree for the GED Problem

To compact the data structure presented in Sect. 2, we present two methods:

3.1 Compacted Method 1: GED Compacted Search Tree (_CT)

The idea is to compact each separate field needed (the vertex edit operations). Our work is inspired by a cost matrix proposed in [9,10]. The cost matrix represents all the combinations of edit operations between the vertices of the two graphs g_1 and g_2. Using this matrix model, we propose the *Edit Operations Matrix* to index all possible operations. The proposed matrix is divided into four regions (See Table 1). The top left region represents substitutions between g_1 and g_2. The far-right column represents the deletions from g_1, while the bottom line represents the insertions in g_2. The last region of the bottom-right cell is useless. We assign to each cell a unique number. We begin from the top left of the matrix by 0 and, each time, we increment by 1 till we finish at the bottom right of the matrix. Therefore, instead of manipulating the vertex edit operations, we use the unique *id* assigned to each edit operation (See Table 1). Each node in the tree uses the *ids* from the edit operation matrix. This avoids manipulating the entire vertices of edit operations. We do not need to store this matrix. We only generate each vertex edit operation *id* based on the indices of the vertices from g_1 and g_2.

Table 1. Vertex operation matrix id.

	0	1	2	. . .	(m-1)	m
0	0	1	2	. . .	m-1	m
1	$(1 \times m) + 1$	$(1 \times m) + 2$	$(1 \times m) + 3$	. . .	$(1 \times m) + m$	$(1 \times m) + (m + 1)$
2	$(2 \times m) + 2$	$(2 \times m) + 3$	$(2 \times m) + 4$		$(2 \times m) + (m + 1)$	$(2 \times m) + (m + 2)$
.	.					.
.	.					.
.	.					.
(n-1)	$((n-1) \times m) + (n-1)$	$((n-1) \times m) + (n-1) + 1$	$((n-1) \times m) + (n-1) + 2$	. . .	$((n-1) \times m) + m + (n-1) - 1$	$((n-1) \times m) + m + (n-1)$
n	$(n \times m) + (n)$	$(n \times m) + (n) + 1$	$(n \times m) + (n) + 2$	. . .	$(n \times m) + m + (n) - 1$	$(n \times m) + m + (n)$

The ids in the Edit Operations Matrix: Let n,m be the number of vertices of g_1, g_2 respectively. The id of an edit operation that involves the vertex i form g_1 and the vertex j from g_2 (the indices begin from 0) can be given by the following equation: $id(i,j) = (i \times m) + (i + j)$, so we can write it in the following form: $id(i,j) = (m+1) \times i + j$. In Table 1, it is clear that the column m concerns operations of the deletion, and the row n concerns those of the insertion. Hence, in the case of deletion a vertex v_i, we set $j = m$. In contrast, we set $i = n$ if we insert a given v_j. Therefore, the vertex edit operations are given as follow:

- Substitution: $get\_id\_sub(i,j) = (m+1) \times i + j$
- Deletion: $get\_id\_del(i) = (m+1) \times i + m$
- Insertion: $get\_id\_ins(j) = (m+1) \times n + j$

Get i and j the Involved Vertices form the Edit Operation $id(i,j)$: From the id of a given edit operation, we need to find the type of that operation and its involved vertices. Note that, we only have the following three values: id, n and m. The index i of the vertex in g_1 is given by:

$$i = \left\lfloor \frac{id(i,j)}{(m+1)} \right\rfloor = \left\lfloor \frac{(m+1) \times i + j}{(m+1)} \right\rfloor$$

The index j of a vertex in g_2 is given by:

$$j = id(i,j) \ mod \ (m+1) = ((m+1) \times i + j) \ mod \ (m+1)$$

After getting i and j values, we deduct the edit operation by checking if:

- $i < n$ & $j < m$, then it is a substitution between v_i from g_1 and u_j from g_2.
- $i < n$ & $j = m$, then it is a deletion of the vertex v_i from g_1.
- $i = n$ & $j < m$, then it is an insertion of the vertex u_j in g_2.

Get the Complete Edit Path: When the search process finds a solution, we need to reconstruct the whole path of edit operations including vertices and edges. For each edit operation id in the edit path $\lambda(g_1, g_2) = \{id_{e1}, id_{e2}, ..., id_{ek}\}$, we retrieve the type of the edit operation and the involved vertices. Then, we extract the implied edges and add them to the final path solution.

The List of the Remaining Vertices and Edges: The remaining vertices and edges at each node are represented by a separate BitSet. A bitvector is created for each remaining list. Initially, all the bits are set to 1. Each node is assigned a copy of its parent BitSet with a 0 in the processed node position.

3.2 Compacted Method 2: Path Representation Using BitSets (_CB)

We benefit from the mapping of the first method where paths are represented by suits of numbers. We represent these paths by BitSets. Each path will be compacted. Moreover, we keep only the active nodes in the search tree, and

we delete all the ancestor parents. A path is the succession of nodes related by pointers that contain *ids*. Assuming that b the size of the pointer equals the size of the edit operation *id*, we need $2b \times k$ bits to represent a path of k operations. This can be reduced to only a few bits when representing a path by a BitSet.

In the first method, each path is represented as follows $\lambda(g_1, g_2) = \{id_{e1}, id_{e2}, ..., id_{ek}\}$. For example, $path(g_1, g_2) = \{2, 6, 8, ...\}$. We encode this path by a BitSet, where each edit operation *id* is represented in the BitSet by 1 in its position. For the *ids* $(2, 6, 8)$, in the BitSet we put 1 at the 2^{nd},6^{th}, and 8^{th} positions; and all other bits are set to 0. Hence, the $path(g_1, g_2) = \{2, 6, 8, ...\}$ is represented as BitSets like $path(g_1, g_2) = [0, 0, 1, 0, 0, 0, 1, 0, 1, ...]$. Each partial or complete path (at inner or leaf node) is represented by its own BitSet.

During the searching process, we keep only the nodes (paths) that are not treated yet. We do not need the ancestor nodes from the root to the final treated node (inner or leaf node), because all the *ids* of the edit operations are in the BitSet of the final treated node. Figure 2 illustrates the search tree using BitSets.

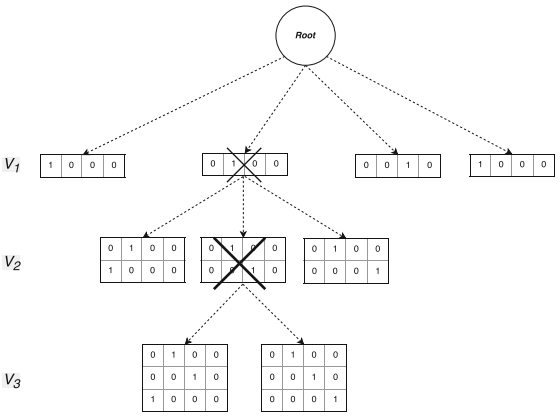


Fig. 2. The search tree using bites array.

The BitSet Size Needed by Nodes at Each Level of the Tree: Each node in the search tree generates at most $m + 1$ child (m is the number of vertices of g_2), where the outgoing are composed by only one vertex from g_1 combined with the other vertices of g_2 plus its own deletion (see Fig. 1). As shown in Table 1, in each row of the edit operations matrix, one vertex v_i from g_1 is substituted with all the vertices of g_2, and deleted at last. The first row contains the *ids* from 0 to m, the second from $m+1$ to $2m+1$, etc.

At the first level of the tree, each node, can so be represented by a BitSet having the same size as the first column in the matrix which is $m + 1$. At the second level, each node, requires the size of the second column of the matrix $(m + 1)$ that allows to index the *ids* of the second vertex from g_1, combined with the rest of vertices from g_2 (m) plus one deletion. We also need to keep all the path in the second level. Thus, we add the BitSet of the first level. As a result, the size is $2 \times (m + 1)$. As a general rule, each node has to keep the information of its previous parents. Therefore, we need a BitSet of $l \times (m + 1)$ bits in each node at the level l. We should notice that we have a sparse BitSet. All the positions contain the value 0, except the one representing the *id* of the edit operation associated with the node. Hence, if desired, one can compress the

Table 2. Setting cost for vertex and edge edit operations.

	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

Table 3. Datasets information.

Dataset	NB graphs	mean #nodes	mean degree	min #nodes	max #nodes
PAH	94	20.7	20.4	10	28
Mao	68	18.4	2.1	11	27

BitSet [5]. The other information needed during the searching process, such as g, h values and remaining vertices and edges, are kept at each node.

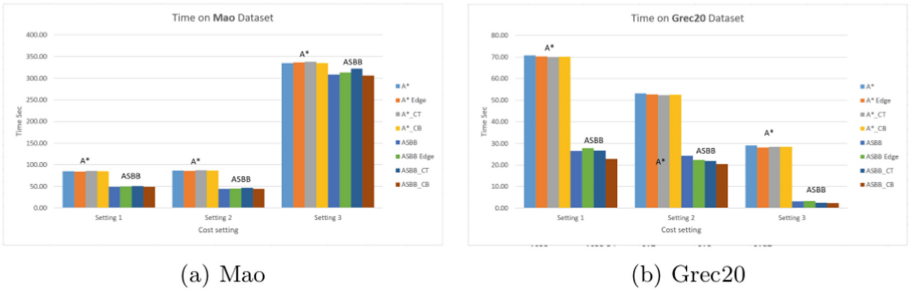
4 Tests and Experiments

In this section, we investigate the ability of our methods to efficiently reduce the used memory space, and see their impact on the running time. Our program is written in JAVA 1.8. Our tests are conducted using one core of one computing node of IBNBADIS Cluster, *with a RAM memory limited to 16 GB*. The two proposed methods implemented with A*GED and ASBB (based on B&B) approaches are compared using the datasets[1] given in Table 3. We have a total of 8 methods. Each basic algorithm (A*GED and ASBB) is implemented: (1) computing the implied edges (A*), (2) storing the implied edges (A*Edge), (3) using compacting method 1 on A* (A*_CT), and (4) using compacting method 2 on A* (A*_CB). The same goes for ASBB method. We have used three cost settings[2] (Table 2).

Time Processing Results: The processing time is relatively close for different methods tested for A*, ASBB and, for their variants. Figure 3 illustrates results for Mao and Grec20 benchmarks. It is clear that the compaction does not negatively influence the processing time. Since the PAH Benchmark is difficult to solve, we do our experiment with only ASBB algorithm using only setting 1. The results are in hours as follow: *basic ASBB: 42.50 h*, *ASBB_CT: 21.92 h* and *ASBB_CB: 20.98 h*. We notice clearly that the compacted methods speed up the processing time to twice because they manipulate only ids with elementary operations (number and bits).

Memory Space Results: To measure the space used by the program, we use the open source *MemoryMeter*[3]. Figure 4 illustrates the memory space used by A*, ASBB and their four implementations with compaction to solve the problem of exact GED for Mao and Grec20 datasets. Storing the implied edges (A*/ASBB

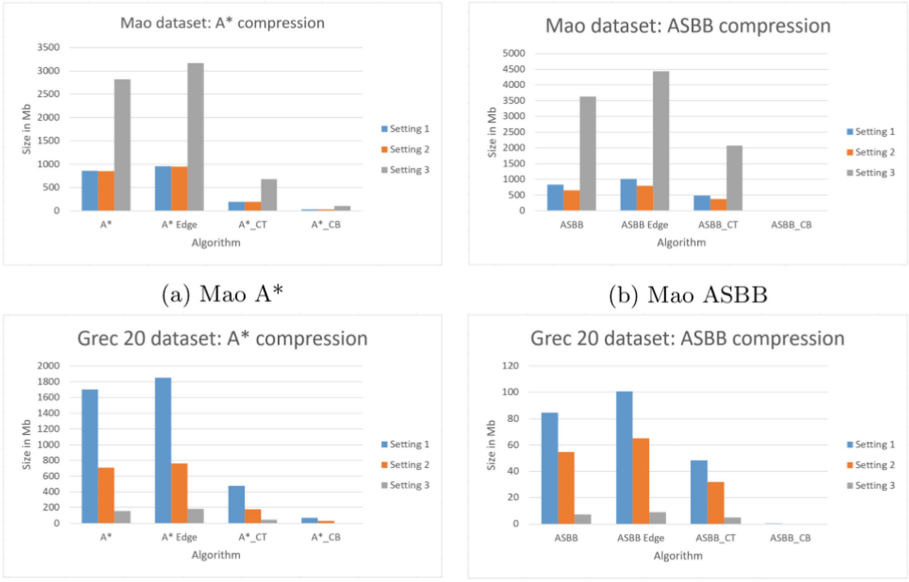
[1] Download from https://gdc2016.greyc.fr/#ged.
[2] These settings were used in the competition https://gdc2016.greyc.fr/#ged.
[3] MemoryMeter: https://github.com/jbellis/jamm.



(a) Mao (b) Grec20

Fig. 3. The running time for A*, ASBB with compaction on Mao, and Grec20.

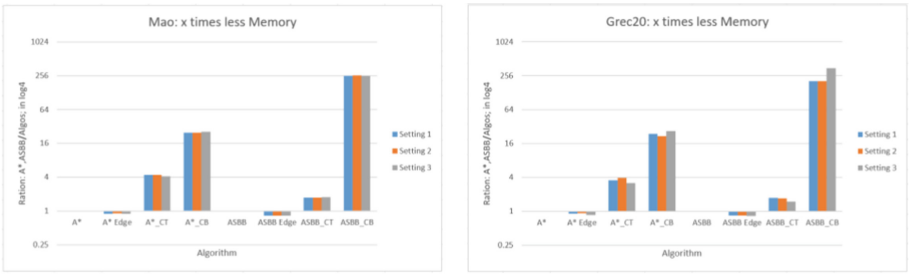
Edge), as expected, increases a little bit the used memory space. It is very clear that the compaction improves the used memory space. Our first method A*_CT achieves an average gain of 4.33, 3.57 less memory with respectively Mao and Grec20. Whereas, ASBB_CT method gets an average gain of 1.73, 164 for the same datasets. Our second method (A*/ASBB)_CB achieves remarkable good results. We clearly see the power of this method to reduce the memory space better than basic algorithms, where it gains an average of 24 less memory with A*_CB, and 250x less with ASBB_CB. Figure 5 illustrates the gain factor of the used memory space between basic and compacted algorithms.



(a) Mao A* (b) Mao ASBB

(c) Gr20 A* (d) Gr20 ASBB

Fig. 4. The memory space occupied by each method on the used datasets.



(a) Mao (b) Gr20

Fig. 5. The ratio of used memory space between basic and compacted algorithms.

5 Conclusion

The exact GED problem has exponential space and time complexity. This work focuses on the memory space of the search tree used to solve this problem. We proposed to compact this tree using an intelligent mapping to represent the different edit operations, and BitSets to represent the needed remaining vertices and edges. Moreover, instead of storing the traditional search tree, we represent it succinctly by bit-vectors. The experiments on several datasets show that these methods decrease significantly the search space about 24 to 250 times compared to the A*GED and ASBB algorithms. This allows solving larger instances compared to the reference algorithm without influencing the running time.

References

1. Abu-Aisheh, Z., Raveaux, R., Ramel, J.Y., Martineau, P.: An exact graph edit distance algorithm for solving pattern recognition problems. In: 4th International Conference on Pattern Recognition Applications and Methods 2015 (2015)
2. Blumenthal, D.B., Gamper, J.: Correcting and speeding-up bounds for non-uniform graph edit distance. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 131–134. IEEE (2017)
3. Bunke, H., Allermann, G.: Inexact graph matching for structural pattern recognition. Pattern Recogn. Lett. **1**(4), 245–253 (1983)
4. Bunke, H., Riesen, K.: Recent advances in graph-based pattern recognition with applications in document analysis. Pattern Recogn. **44**(5), 1057–1067 (2011)
5. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with roaring bitmaps. Softw. Pract. Exp. **46**(5), 709–719 (2016)
6. Dabah, A., Chegrane, I., Yahiaoui, S.: Efficient approximate approach for graph edit distance problem. Pattern Recognit. Lett. **151**, 310–316 (2021)
7. Gouda, K., Hassaan, M.: Csi_ged: an efficient approach for graph edit similarity computation. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE), pp. 265–276. IEEE (2016)

8. Neuhaus, M., Riesen, K., Bunke, H.: Fast suboptimal algorithms for the computation of graph edit distance. In: Yeung, D.-Y., Kwok, J.T., Fred, A., Roli, F., de Ridder, D. (eds.) SSPR /SPR 2006. LNCS, vol. 4109, pp. 163–172. Springer, Heidelberg (2006). https://doi.org/10.1007/11815921_17
9. Riesen, K., Bunke, H.: Approximate graph edit distance computation by means of bipartite graph matching. Image Vis. Comput. **27**(7), 950–959 (2009)
10. Riesen, K., Fankhauser, S., Bunke, H.: Speeding up graph edit distance computation with a bipartite heuristic. In: MLG (2007)
11. Riesen, K., Fischer, A., Bunke, H.: Computing upper and lower bounds of graph edit distance in cubic time. In: El Gayar, N., Schwenker, F., Suen, C. (eds.) ANNPR 2014. LNCS (LNAI), vol. 8774, pp. 129–140. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11656-3_12
12. Vento, M.: A long trip in the charming world of graphs for pattern recognition. Pattern Recogn. **48**(2), 291–301 (2015)
13. Zeng, Z., Tung, A.K., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. Proc. VLDB Endow. **2**(1), 25–36 (2009)