Revisiting Volgenant-Jonker for Approximating Graph Edit Distance

William Jones(✉), Aziem Chawdhary, and Andy King

University of Kent, Canterbury, CT2 7NF, UK
{wrj2,a.m.king}@kent.ac.uk, aziem@chawdhary.co.uk

Abstract. Although it is agreed that the Volgenant-Jonker (VJ) algorithm provides a fast way to approximate graph edit distance (GED), until now nobody has reported how the VJ algorithm can be tuned for this task. To this end, we revisit VJ and propose a series of refinements that improve both the speed and memory footprint without sacrificing accuracy in the GED approximation. We quantify the effectiveness of these optimisations by measuring distortion between control-flow graphs: a problem that arises in malware matching. We also document an unexpected behavioural property of VJ in which the time required to find shortest paths to unassigned nodes decreases as graph size increases, and explain how this phenomenon relates to the birthday paradox.

1 Introduction

Graph edit distance (GED) [5] measures the similarity of two graphs as the minimum number of edit operations needed to convert one graph to another. More precisely, suppose $G = \langle V, E, \ell \rangle$ is a labelled directed graph where $E \subseteq V \times V$ and $\ell : V \to \Sigma$ assigns each vertex to a label drawn from an alphabet Σ. (In the general case, edges can also be similarly attributed.) An edit operation on a graph G_1 inserts or deletes an isolated vertex, inserts or deletes an edge, or relabels a vertex, to obtain a new graph G_2. Applying a sequence of $n - 1$ edit operations gives a sequence of n graphs $G_1, G_2, \ldots, G_n$. Since the cost of edit operations is not necessarily uniform, in the more general form, each edit operation has an associated edit cost as defined by a cost function. The GED between two graphs G and G' is the minimum sum of edit operation costs. GED has proven to be useful [2,4,7,8] because it is an error tolerant measure of similarity.

However, computing GED is equivalent to finding an optimal permutation matrix [11], which is NP-hard. Fast but suboptimal approaches have thus risen to prominence [8], in which GED is approximated by solving a linear sum assignment problem. Of those algorithms proposed for solving this problem, the Volgenant-Jonker (VJ) algorithm [6] is the most efficient. This paper takes VJ as the starting point, and explores how it can be improved for the specific task of GED computation. Several similar works have been attempted before [9,10]. Our work differs from these previous attempts because our changes attack the highly regular structure of the cost matrix and the redundancy that this implies for the

C.-L. Liu et al. (Eds.): GbRPR 2015, LNCS 9069, pp. 98–107, 2015.
DOI: 10.1007/978-3-319-18224-7_10

VJ algorithm instead of approaching the problem by using a refined concept of edit distance. Our paper makes the following contributions:

- It shows how the VJ algorithm can be tuned to GED computation;
- It quantifies the ensuing speedup and decrease in memory requirements;
- It reports an emergent behaviour in which the time taken on the shortest path calculations decreases as the problem size increases;
- It gives an explanation to the above phenomenon based on the well known birthday problem.

The paper is structured as follows: To keep the paper self-contained, section 2, explains how GED is related to the linear sum assignment problem, and section 3 describes the classical VJ algorithm. Section 4 introduces the proposed optimisations, and Section 5 presents the experimental results, comparing the improved algorithm with the original. Finally, Section 6 conclusions.

2 The Linear Assignment Problem and GED

Given an $n \times n$ cost matrix C, the linear assignment problem [3] is that of finding a bijection $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ which minimises $\sum_{i=1}^{n} C_{i,f(i)}$. When $|V_1| = n = |V_2|$ the GED between $G_1 = \langle V_1, E_1, \ell_1 \rangle$ and $G_2 = \langle V_2, E_2, \ell_2 \rangle$ can be approximated by solving a linear assignment problem using an $n \times n$ matrix C where $C_{i,j}$ denotes the cost of substituting vertex i for vertex j. Approximating GED with this minimum requires $|V_1| = |V_2|$ and is imprecise since it only considers vertex substitutions. A more general approach [8] addresses both these problems by working on an extended cost matrix defined as follows:

$$C = \left[\begin{array}{cccc|cccc}
c_{1,1} & c_{1,2} & \cdots & c_{1,m} & c_{1,\epsilon} & \infty & \cdots & \infty \\
c_{2,1} & c_{2,2} & \cdots & c_{2,m} & \infty & c_{2,\epsilon} & \ddots & \vdots \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \infty \\
c_{n,1} & c_{n,2} & \cdots & c_{n,m} & \infty & \cdots & \infty & c_{n,\epsilon} \\
\hline
c_{\epsilon,1} & \infty & \cdots & \infty & 0 & 0 & \cdots & 0 \\
\infty & c_{\epsilon,2} & \cdots & \infty & 0 & 0 & \ddots & 0 \\
\vdots & \ddots & \ddots & \infty & \vdots & \ddots & \ddots & 0 \\
\infty & \cdots & \infty & c_{\epsilon,m} & 0 & \cdots & 0 & 0
\end{array}\right]$$

where $n = |V_1|$ and $m = |V_2|$. The top left hand corner of $c_{i,j}$ describes the cost of vertex substitution. The top right hand corner $c_{i,\varepsilon}$ the cost of vertex deletion u_i. The bottom left hand corner $c_{\varepsilon,j}$ denotes the cost of vertex insertion v_j. The bottom right hand corner is uniformly zero (henceforth called the null quadrant).

A further extension [2] uses E_k and ℓ_k to compute a lower bound on GED, by finding labels on the incoming neighbours of a given vertex j in G_k using $In_k(j) = \{\ell_k(i) \mid \langle i, j \rangle \in E_k\}$. The matrix is defined $c_{i,j} = d_{i,j} + e_{i,j}$ where $d_{i,j} = 1$ if $\ell_1(i) \neq \ell_2(j)$ otherwise 0, and $e_{i,j} = \max(|In_1(i) - In_2(j)|, |In_2(i) - In_1(i)|)$. Then $c_{i,j}$ accounts for any difference in labelling between vertex i and vertex j and also removing edges and relabelling their incoming neighbours. The diagonals $c_{i,\epsilon}$ and $c_{\epsilon,i}$ are degenerative and defined as above with $In_k(\epsilon) = \emptyset$.

$$\underset{(a)}{\begin{bmatrix} 1 & 7 & 6 \\ 5 & 2 & 3 \\ 8 & 9 & 4 \end{bmatrix}} \qquad \underset{(b)}{\begin{bmatrix} 0 & 5 & 3 \\ 4 & 0 & 0 \\ 7 & 7 & 1 \end{bmatrix}} \qquad \underset{(c)}{\begin{bmatrix} 3 & 5 & 3 \\ 7 & 0 & 0 \\ 10 & 7 & 1 \end{bmatrix}} \qquad \underset{(d)}{\begin{bmatrix} 0 & 2 & 0 \\ 7 & 0 & 0 \\ 10 & 7 & 1 \end{bmatrix}}$$

Fig. 1. (a) Example cost matrix; (b) After column reduction; (c) After anti-column reduction; (d) After row reduction (reduction transfer)

3 The Classical VJ Algorithm

The VJ algorithm [6] is a shortest path algorithm solved by a dual method. We describe the essence of the algorithm (though not the detail). The algorithm consists of two main steps, which are outlined in the sub-sections that follow:

1. Initialisation in three stages: (a) column reduction; (b) reduction transfer; and (c) augmenting row reduction.
2. Augmentation until complete, in which alternating paths are found where each path is from an unassigned row to an unassigned column.

3.1 Initialisation

Column reduction The first step of initialisation is a column reduction, in which a positive value is subtracted from each element of a column. Starting at the last column, the VJ algorithm reduces each column by its minimum element, so that each column contains a zero. Figure 1(b) illustrates the result of column reduction. As the matrix is scanned right-to-left, each column is assigned, whenever possible, to a unique row that contains a zero in that column. Column 3 is assigned to row 2 (and vice versa), and column 1 is assigned to row 1 (and vice versa), but column 2 will remain unassigned (as does row 3).

Reduction Transfer. The second step of initialisation is reduction transfer, which is applied to enable row reduction, in which a positive value is subtracted from each element of a row. Row reduction cannot be applied to row 1 of Figure 1(b), without introducing a negative entry. Thus an inverse of column reduction is applied to row 1, to give the matrix depicted in Figure 1(c). Row reduction is then applied, the result of which is illustrated in Figure 1(d), albeit at the expense of column reduction. This exchange in reduction value between a column and a row, in this case by 3, is called reduction transfer.

Augmenting Row Reduction. In the third phase of initialisation, an attempt is made to find a set of (alternating) paths where each path starts in an unassigned row and ends in an unassigned column. For a given unassigned row i, VJ finds a column j_1 that contains the minimum entry e_1 and another column j_2 that contains the least entry e_2 such that $e_2 \geq e_1$. Row i is then reduced by e_2. If $e_2 > e_1$ then this incurs a negative value in column j_1, in which case, anti-column reduction is applied to column j_1 to eliminate the negative entry. Row

$$\left[\begin{array}{ccc|ccc} 3 & 1 & 4 & 1 & \infty & \infty \\ 5 & 9 & 2 & \infty & 6 & \infty \\ 5 & 3 & 5 & \infty & \infty & 8 \\ \hline 9 & \infty & \infty & 0 & 0 & 0 \\ \infty & 7 & \infty & 0 & 0 & 0 \\ \infty & \infty & 9 & 0 & 0 & 0 \end{array}\right] \qquad \left[\begin{array}{c} \begin{array}{ccc|c} 3 & 1 & 4 & 1 \\ 5 & 9 & 2 & 6 \\ 5 & 3 & 5 & 8 \end{array} \\ \hline \begin{array}{c|ccc} 9 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 \end{array} \end{array}\right] \qquad \left[\begin{array}{ccc|c} 3 & 1 & 4 & 1 \\ 5 & 9 & 2 & 6 \\ 5 & 3 & 5 & 8 \\ \hline 9 & 7 & 9 & \end{array}\right]$$

$$(a) \qquad\qquad\qquad (b) \qquad\qquad\qquad (c)$$

Fig. 2. (a) Original cost matrix. (b) Row-by-row representation with zeroes. (c) Row-by-row representation without zeroes.

i is then assigned to column j_1 regardless of whether this column is already assigned or not. If j_1 was previously assigned to a row k, then row k becomes unassigned and the procedure continues from row k. This repeats until either row k is matched to an unassigned column, or it becomes impossible to transfer reduction to the selected row k. Observe how reduction transfer provides a vehicle for constructing a path that alternates between rows and columns, hence the name.

3.2 Augmentation

For each unassigned row, the augmentation phase finds a shortest alternating path (of the type previously described) to an unassigned column. VJ modifies Dijkstra's algorithm to search for these shortest paths, where the notion of distance between a row and a column is the entry in the cost matrix. Search starts at an unassigned row, say row i, and a shortest edge is found from row i to a column j. If column j was previously assigned to row k, then row k becomes unassigned (though no changes are made until a complete path to an unassigned column is found) and search resumes from row k. Unlike classical Dijkstra, search continues in this fashion until such a column is found. After augmentation, the assignments to the cost matrix are updated so that all assignments in the current solution correspond to minimum entries in each row of the cost matrix.

4 The Improved VJ Algorithm

This section explores several improvements to the classical VJ algorithm, most of which follow from the regular structure of the cost matrix.

4.1 Representation

A brief foreword to this section. It should be noted that while we change the representation in memory, we do not just naïvely iterate over it. Our change is simply to simplify many operations; we are still essentially calculating assignments and solutions in their "real" positions.

$$C = \left[\begin{array}{cccc|c} c_{1,1} & c_{1,2} & \cdots & c_{1,m} & c_{1,\epsilon} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} & c_{n,\epsilon} \\ \hline c_{\epsilon,1} & c_{\epsilon,2} & \cdots & c_{\epsilon,m} & \end{array}\right]$$

(a)

$$C = \left[\begin{array}{cccc|c} c_{1,1} & c_{1,2} & \cdots & c_{1,m} & c_{1,\epsilon} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} & c_{n,\epsilon} \\ \hline c_{\epsilon,1} & c_{\epsilon,2} & \cdots & c_{\epsilon,m} & \end{array}\right]$$

(b)

Fig. 3. (a) The split for the left hand and right hand block (the left hand block is highlighted). (b) The split for the top and bottom blocks (the top is highlighted).

Given two reasonably-sized graphs G_1 and G_2, the largest data structure by far is the cost matrix requiring $(n+m)^2$ entries for $n = |V_1|$ and $m = |V_2|$. However, most of these entries are either zero or infinity, as is illustrated in Figure 2(a). Given the operations that must be applied to the cost matrix, there are two natural compressed representations: a row-by-row representation in which each row of the matrix is represented by only storing non-infinite values, as depicted in Figure 2(b); and an analogous column-by-column representation.

There is also the question of whether to explicitly store the zeros in the bottom right of Figure 2(b). We choose to discard them as we found no algorithmic benefit in retaining them. With this change, the row-by-row representations reduces to Figure 2(c). The net effect is that the cost matrix is represented in space $(n+1) \times (m+1)$. Although row equality is no longer preserved, operations on the effected rows can still be performed in constant time since there is only one variable entry per row. Moreover, this representation homogenises the column-by-column and row-by-row representations, which means that it simultaneously benefits both row- and column-based operations.

4.2 Column Reduction

This representation simplifies column reduction in two ways: First, almost half of the costs in the matrix are infinite and so will never be chosen as a minimum in a column. Second, nearly a quarter of the costs will be 0, and these zeros dictate that the column minimum will be zero. Hence only the position of the minimum need be computed in column reduction (rather than its position and value). To take advantage of this, the matrix is considered as two separate blocks with different operations provided for each. see Figure 3a. The leftmost entries are handled as before thanks to the new data-structure. The rightmost entries that are stored in a single column, see Figure 2(c), correspond to the top-right diagonal above the null quadrant. These entries are only compared to zeros and hence the reduction value will always be zero and thus only the position of the zero need be found. This can be further simplified in the case that deletion costs are non-zero because this removes the need for position computation too.

4.3 Reduction Transfer

Reduction transfer actually becomes slightly more complicated to accommodate the new data-structure. However, this is a worthwhile trade as reduction transfer typically takes up a minute fraction of the total run time.

4.4 Augmenting Row Reduction

Augmenting row reduction can be optimised in a very similar fashion to column reduction. Once again, for each row being considered there is no point considering infinite costs and similarly the presence of the null quadrant simplifies many of the calculations. The only difference is that since augmenting row reduction considers rows instead of columns, the most effective way to operate is a top-to-bottom split (see Figure 3b), where the top block and the bottom block are handled separately. We can consider similar simplifications as before if we can assume that vertex addition will always have a non-zero cost.

4.5 Augmentation

Augmentation is slightly quicker with these improvements, particularly over very large graphs. There are a number of operations that can be simplified by clever use of the new data-structure, but these refinements have little to no benefit. The only significant changes are those that simply replace variable lookups when the outcome is known, for example, looking up an entry in the null quadrant.

5 Experimental Results

To empirically assess the proposed improvements to VJ, two versions were implemented: the original version (VJ-ORG) and an improved version (VJ-IMP). Both were compared against a version developed by Jonker himself (VJ-CTRL), which was used as a control. All three versions were implemented in C++.

5.1 Evaluation on Random Data

Initially random square ($n = m$) cost matrices were used to provide a large corpus of data for comparing all versions of VJ. The improvements have least effect on square matrices and thus, if anything, the setup is biased against VJ-IMP. Costs and matrix sizes were chosen to approximate what might typically be encountered in malware matching. In this context a control-flow graph (CFG) is extracted from a binary for comparison against a database of CFGs derived from malware. BinSlayer [2] was used to derive CFGs from several medium-sized binaries. These possessed between 465 to 6984 vertices (basic blocks), and produced matrices where the costs rarely exceeded 2000 and never exceeded 3000. To cover a range of scenarios, matrices were populated with random values from cost ranges varying over 1-500 to 1-3000. Matrix sizes were also varied between

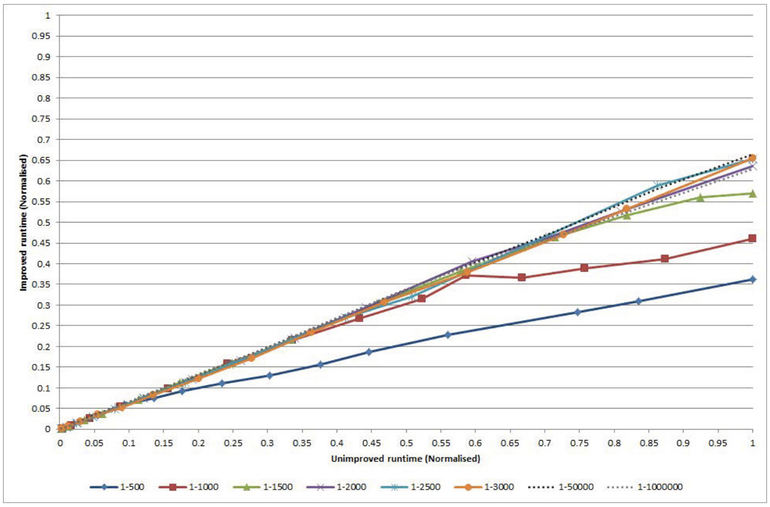

Fig. 4. VJ-IMP against of the fastest of VJ-ORG and VJ-CTRL for various cost ranges and matrix sizes

1000 and 14000, again to simulate CFGs. We also tested the resilience of the improvements over some much larger cost ranges (1-50000 and 1-10^9) for the full range of matrix sizes.

Figure 4 gives a series of plot lines, one for each cost range. Each data point on a line corresponds to the runtimes for a different matrix size, though the horizontal axis is normalised by the runtime for solving a 14000×14000 matrix. The vertical axis is normalised relative to the unimproved times, so that the instantaneous gradient quantifies the improvement over a range of costs and matrix sizes. Note that VJ-IMP is uniformly faster than both VJ-ORG, and the third party implementation by Jonker, VJ-CTRL. While we find the improved version of algorithm to be about twice as fast; the difference being more striking

CFG 1	**CFG 2**	**Size Ratio**	VJ-ORG	VJ-CTRL	VJ-IMP	**Speedup**
bash	BinSlayer	0.27	201	205	125	62%
BinSlayer	bash	3.75	2944	3164	1820	68%
comaker	BinSlayer	0.11	623	627	264	137%
BinSlayer	comaker	8.83	13221	13729	8905	51%
comaker	bash	0.42	3763	3901	3080	24%
bash	comaker	2.35	54308	58333	43576	29%
GB	BinSlayer	0.07	1862	1656	588	199%
BinSlayer	GB	15.02	34020	35468	22127	57%
GB	bash	0.25	4555	5087	3155	53%
bash	GB	4.00	151457	160074	114091	37%
GB	comaker	0.59	17987	18495	15291	19%
comaker	GB	1.70	209562	228467	164442	33%

Fig. 5. Runtimes in milliseconds, where the size ratio is $|V_1|/|V_2|$

at lower costs and higher sizes. Although not represented by this data, there is a small performance advantage to VJ-ORG over VJ-CTRL, which suggests that our implementations and experiments are robust.

5.2 Evaluation on CFGs

Figure 5 summaries some CFG comparisons for four different binaries, where the CFGs were derived using BinSlayer. Comparing $CFG_1 = \langle V_1, E_1, \ell_1 \rangle$ against $CFG_2 = \langle V_2, E_2, \ell_2 \rangle$ does not necessarily take the same time as comparing CFG_2 against CFG_1. This is because if $|V_1| < |V_2|$ then cost matrix will have a large deletion block (in its top right) and a small addition block (in its bottom left). It is notable that while all versions of VJ are faster when $|V_1| < |V_2|$, the benefits to VJ-IMP are more significant. Excluding augmentation, the runtime of each component of each algorithm is almost constant no matter whether $|V_1| < |V_2|$. However, the runtime of augmentation is smaller when $|V_1| < |V_2|$. Since column reduction is faster in VJ-IMP extra benefits follow from $|V_1| < |V_2|$ because column reduction is faster and the cost of augmentation is less dominant. We have found that the number of iterations in augmentation does not depend on $|V_1| < |V_2|$, and so the decreased time in augmentation is entirely a byproduct of an decrease in the runtime of each iteration.

5.3 Component Analysis

Figure 6 shows the time proportion spent in each component of VJ-ORG and VJ-IMP. Column reduction and augmenting row reduction benefit most from the improvements; augmentation is faster with VJ-IMP (in absolute terms).

We also see an interesting behaviour in the runtimes of augmentation and column reduction, especially at low cost ranges. Column reduction quickly increases as a percentage of total runtime as cost matrix size increases eventually

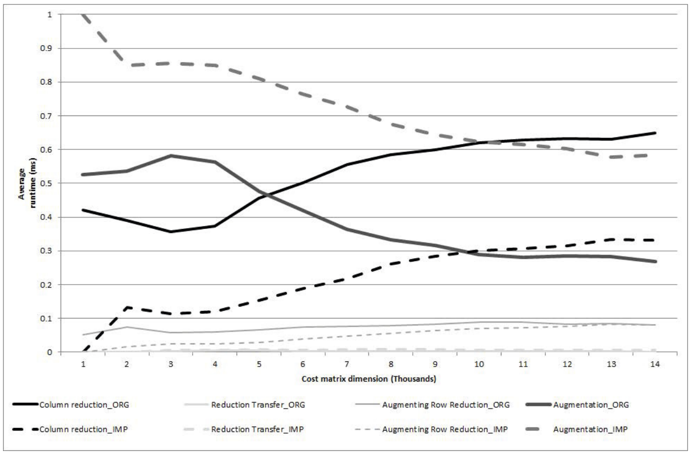

Fig. 6. Normalised runtime of components, for VJ-ORG and VJ-IMP, over range 1-500

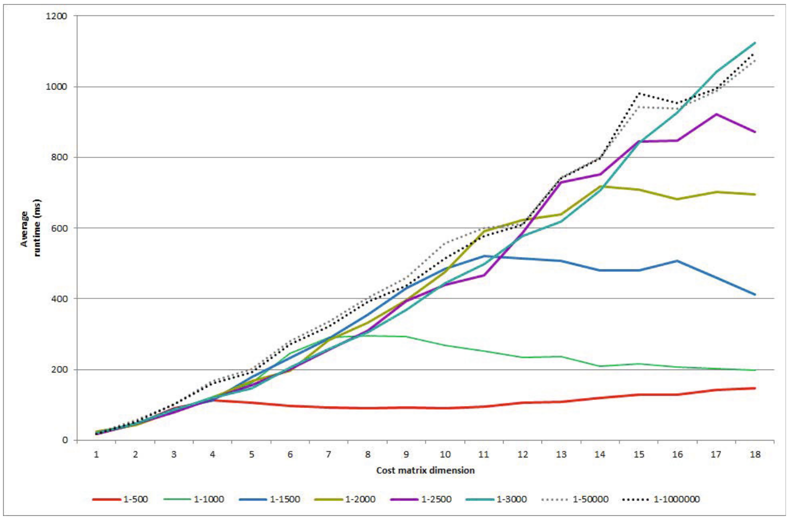

Fig. 7. Runtime of each iteration of augmentation (averaged over all three algorithms)

overtaking augmentation. On closer examination, it is apparent that this is the result of a reduction in the growth of the runtime of augmentation instead of a sharp increase in the runtime of column reduction. Furthermore this happens across all cost ranges (but is more visible for smaller ranges). This is surprising as augmentation is merely an implementation of Dijkstra's algorithm. Figure 7 suggests that this stems from an effect in which the growth in runtime of each iteration of augmentation actually tails off as the size of cost matrix increases.

We conjecture that this is because of a statistical property related to the birthday paradox. During initialisation the column reduction step works backward (from the highest index to the lowest), so low index columns have a higher chance of involving a collision and having a lowest element in the same position as another column. For a randomly generated matrix of total size $t \times t$, column index i will have a $(\frac{t}{t+1})^{t-i}$ probability of not having a minimum element in the same row as another column, and thus low i are very likely to be unassigned. Thus a column with a given index is more likely to be unassigned as cost matrix size increases. Consequently not only are low indexed columns more likely to be unassigned, but across all columns this effect will increase disproportionally as matrix size increases. Since Dijkstra's algorithm scans from low indexed columns to high indexed ones, it will find assignments for most of its rows more quickly as cost matrix size increases, even though worse-case complexity remains in $O(n^3)$.

6 Conclusions

We have examined the VJ algorithm and studied improvements for approximating GED. We have shown that our improved algorithm is uniformly faster than its unimproved counterparts, both across randomly generated matrices and data sets that arise in call-graph comparison. The speedups, which are almost 200% in one case, suggest the that refinements are truly worthwhile. Moreover, the improved version has a smaller memory footprint, and incurs no loss of accuracy whatsoever. Finally, we have also documented and explained an anomaly in the runtime of the Dijkstra's shortest path search component of VJ. Future work will, among other things, empirically investigate how the relative sizes of the two graphs under comparison effect the overall runtime, and also explore the prospects for parallelisation [1].

Acknowledgements. We thank Earl Barr and David Clark for discussions on malware matching that motivated this work, and the reviewers of this paper who had several important insights into our work, and Laetitia Pelacchi for her support.

References

1. Balasn, E., Miller, D., Pekny, J., Toth, P.: A parallel shortest augmenting path algorithm for the assignment problem. JACM 38(4), 985–1004 (1991)
2. Bourquin, M., King, A., Robbins, E.: BinSlayer: Accurate Comparison of Binary Executables. In: Proceedings of Program Protection and Reverse Engineering Workshop. ACM (2013)
3. Burkard, R.E., Cela, E.: Linear Assignment Problems and Extensions. Springer (1999)
4. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty Years of Graph Matching in Pattern Recognition. International Journal of Pattern Recognition and Artificial Intelligence 18(3), 265–298 (2004)
5. Gao, X., Xiao, B., Tao, D., Li, X.: A Survey of Graph Edit Distance. Pattern Analysis and Applications 13(1), 113–129 (2010)
6. Jonker, R., Volgenant, A.: A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems. Computing 38(4), 325–340 (1987)
7. Myers, R., Wison, R.C., Hancock, E.R.: Bayesian graph edit distance. IEEE Transactions on Pattern Analysis and Machine Intelligence 22(6), 628–635 (2000)
8. Riesen, K., Bunke, H.: Approximate Graph Edit Distance computation by means of Bipartite Graph Matching. Image and Vision Computing 27(7), 950–959 (2009)
9. Serratosa, F.: Fast computation of bipartite graph matching. Pattern Recognition Letters 45, 244–250 (2014)
10. Serratosa, F., Cortés, X.: Edit Distance Computed by Fast Bipartite Graph Matching, pp. 253–262 (2014)
11. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing Stars: On Approximating Graph Edit Distance. VLDB Endowment 2(1), 25–36 (2009)