An optimal algorithm for MoJo distance

Zhihua Wen and Vassilios Tzerpos
York University
Toronto, Ontario, Canada
{zhihua,bil}@cs.yorku.ca

Abstract

Effective comparison of different decompositions of a software system is integral for the evaluation of software clustering algorithms. One of the approaches that has been developed to tackle this issue is the MoJo distance measure.

The original algorithm for the computation of MoJo distance was a heuristic one that only approximated the exact value. In this paper, we introduce an algorithm that calculates the exact distance in polynomial time. We also present experiments that showcase that the new algorithm calculates MoJo distance more accurately and efficiently than before.

1. Introduction

A common approach to the problem of understanding a large software system is to decompose it into smaller, easier to comprehend subsystems. Software clustering techniques attempt to perform this task automatically [12, 4, 11, 9, 1, 17, 2].

An important property of automatically-created decompositions of large software systems is that they are difficult to evaluate. Manual comparison against a "gold standard" is time-consuming and far from trivial for large clusterings. Neither textual nor visual representations of the decompositions are particularly helpful when dealing with software systems comprised of thousands of entities.

As described in more detail in section 6, several researchers have attempted to tackle this difficult problem [8, 2, 7, 10]. One of the first approaches to be presented in the literature, was the MoJo distance measure[1] [14]. MoJo distance between two clusterings A and B of the same software system is defined as the minimum number of *Move* or *Join* operations one needs to perform in order to transform either A to B or vice versa.

[1]Initially presented as the MoJo distance metric. However, since triangle inequality does not hold for MoJo distance, we will refer to it as the MoJo distance measure.

The above implies the following two features of MoJo distance:

1. The two operations (*Move* and *Join*) are of equal weight, i.e. 1.
2. The *Split* operation is disallowed (or effectively assigned a weight equal to the cardinality of the smaller of the two new clusters).

These choices are surprising at first, but at closer inspection, there are good reasons for them:

- From a mathematical point of view, *Move* and *Join* are operations that can only be done in one way (as opposed to the operation of splitting a cluster of size n into two, which can be done in $2^{n-1} - 1$ ways).
- From a software clustering point of view, the fact that we need to join two clusters means that we performed more detailed clustering than required, hardly a big problem. Having to split a cluster, however, indicates that our clustering was rather crude.
- From an information theory point of view, there is always a constant amount of information one needs to provide in order to distinctly identify a *Move* or a *Join* operation (the name of the object to be moved, and the names of the clusters involved are sufficient). However, a *Split* operation requires a list of all objects to be placed in each cluster, something that can vary depending on the size of the cluster in question.

The MoJo distance measure has been used to study the stability [16], as well as the quality of particular software clustering algorithms [15]. It has also been extended to fit the needs of particular situations, as in the case of MoJoPlus distance [6].

However, a shortcoming of the original presentation of MoJo distance, was the fact that the algorithm developed to calculate it was a heuristic one. As a result, the value computed was only an approximation of the exact distance.

This paper introduces an algorithm that calculates the exact distance in polynomial time.

The structure of the rest of this paper is as follows: Section 2 describes the new MoJo algorithm and discusses its complexity. Section 3 presents a proof that the value calculated is the exact MoJo distance. Section 4 describes our implementation of the new algorithm, while section 5 presents experiments that demonstrate its improved performance. Related and future work is discussed in section 6. Finally, section 7 summarizes the paper.

2. The new algorithm

Assume S is a software system containing n objects, and A and B are two decompositions of S, $A = A_1, A_2, ..., A_l$ and $B = B_1, B_2, ..., B_m$. An example is shown in figure 1, where A and B are two partitions of the same system containing 16 objects.

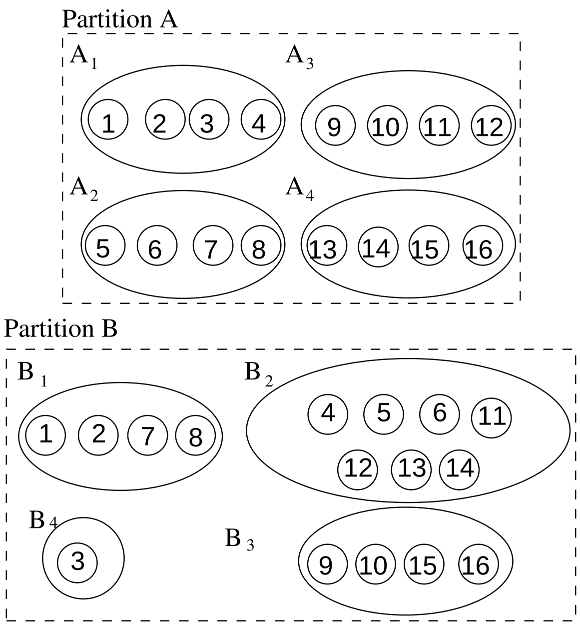

Figure 1. Example partitions A and B.

For each object in A, we find its corresponding subsystem in B. If an object in A_i belongs in B_j in partition B, we give this object a tag T_j. The tags are shown in figure 2 as a rectangle next to each object. For example, since object 4 is in B_2 in partition B, it is assigned tag T_2 (shown as $\boxed{2}$).

Let us denote the intersection between A_i and B_j as v_{ij}. Thus, $v_{ij} = |A_i \cap B_j|$ or $v_{ij} = |A_i(T_j)|$. For example, in figure 2, $v_{11} = 2, v_{12} = 1, v_{14} = 1, v_{21} = 2, v_{22} = 2$, etc.

Our algorithm considers only the tags of each object from now on. Figure 3 presents partition A including only the tags of each object.

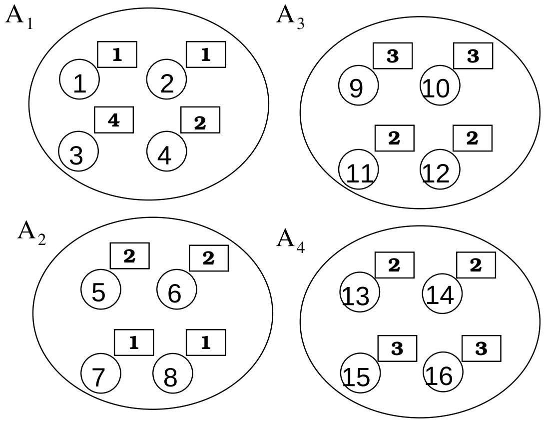

Figure 2. Partition A with the tags.

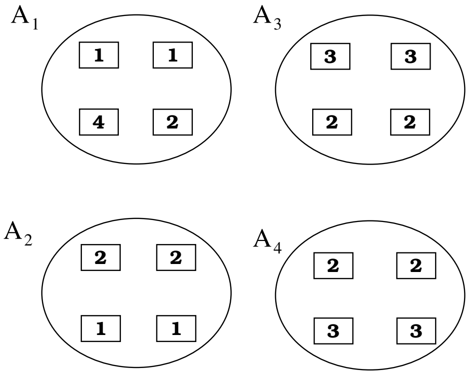

Figure 3. Partition A containing only tags.

Next, we create m sets G_k, one for each cluster in B (we will refer to set G_k as group G_k). For every A_i, we say that A_i belongs in group G_k if $v_{ik} = max_{j=1}^{m}(v_{ij})$. For example, in figure 3, A_1 contains two $\boxed{1}$s, one $\boxed{2}$ and one $\boxed{4}$. Thus, A_1 belongs in group G_1.

It is a requirement for our algorithm, that each cluster belong to exactly one group. However, the above definition of group assignment might create some ambiguity. In our example, A_2 contains two $\boxed{1}$s and two $\boxed{2}$s, which means it may belong in either G_1 or G_2. A_3 contains two $\boxed{2}$s and two $\boxed{3}$s, which means it may belong in either G_2 or G_3. A_4 contains two $\boxed{2}$s and two $\boxed{3}$s, which means it may belong in either G_2 on G_3. Our algorithm chooses the group for these clusters, so that the number of non-empty groups is maximized.

Let g be the number of non-empty groups. We will use the maximum bipartite matching algorithm to maximize g. The following two definitions are necessary:

1. *Bipartite graph*. A bipartite graph is a graph $Gr = (V, E)$, such that the vertex set can be partitioned into $V = L \cup R$ where L and R are disjoint sets and all edges in E go between L and R.

2. *Maximum matching*. Given an undirected graph $Gr = (V, E)$, a matching is a subset of edges $F \subseteq E$ such that for all vertices $v \in V$, at most one edge of F is incident on v. We say that a vertex $v \in V$ is matched, if some edge in F is incident on v; otherwise v is unmatched. A maximum matching is a matching of maximum cardinality, i.e. a matching F such that for any matching F', we have $|F| \geq |F'|$.

It has been shown that the problem of finding a maximum matching in a bipartite graph can be solved in $O(VE)$ [5], where V is the number of vertices and E is the number of edges. The problem of maximizing the number of non-empty groups can be transferred to the problem of finding a maximum bipartite matching. We do this in the following way:

We start by creating a bipartite graph. For every cluster A_i, we add a vertex in the left side of the graph. For every group G_k, we add a vertex in the right side of the graph. For any cluster A_i, if $v_{ik} = max_{j=1}^{m}(v_{ij})$, we draw an edge between A_i and G_k. The graph for our example is shown in figure 4.

Since all edges of the graph go between the left side and the right side, our graph is clearly a bipartite graph. It is possible that two or more edges are incident on a single cluster A_i. We will show that the cardinality of a maximum matching is equal to the maximum number of non-empty groups.

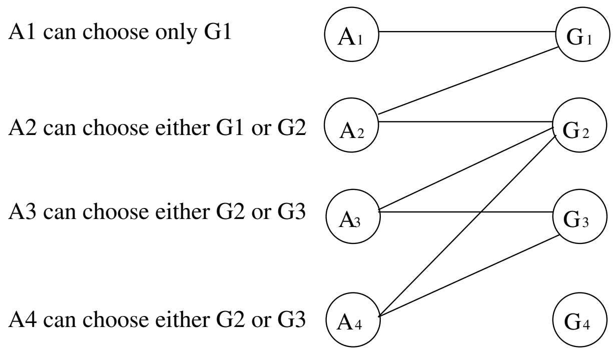

Figure 4. The bipartite graph for our example indicates the relationship between clusters and their possible groups.

Let T be any maximum matching in our bipartite graph. We want to show that assigning groups so that $g > |T|$ is impossible. Suppose there exists a group assignment that creates $|T| + 1$ non-empty groups. Each of these groups must contain at least one cluster in A. Let us randomly pick a cluster from each group. Let T' be the subset of edges that go between these clusters and their groups. Because each cluster may belong in only one group, each edge in T' is incident on only one cluster in A. Also, because we chose only one cluster in each group, each edge in T' is incident on only one group. By definition, T' is a matching. Because $|T'| > |T|$, we have a contradiction.

At this point, our algorithm employs the maximum bipartite matching algorithm to obtain a maximum matching T in the bipartite graph it created. Every matched vertex in A is assigned its matching vertex as its group. Every unmatched vertex in A may choose any possible group since that doesn't affect the total number of non-empty groups. Thus the number of non-empty groups will be $|T|$.

Figure 5 illustrates how the above translates to our example. The number of non-empty groups is 3, which is the maximum possible. Figure 6 shows the group assignment for partition A.

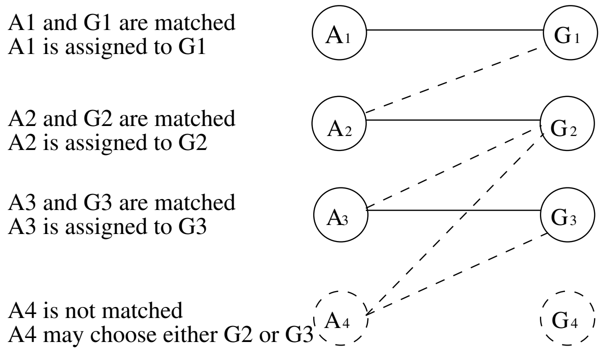

Figure 5. Conversion of a maximum matching to group assignment. Matched edges and vertices are solid, while unmatched ones are dashed.

We are now ready to perform *Move* and *Join* operations. As a preliminary step, we create and assign an empty cluster to each empty group. Next, we perform the following *Move* operations:

For each group G_k, we move all objects tagged with T_k that belong to clusters in other groups to any cluster in G_k. The result of this process to our example is shown in figure 7. G_1, G_2, and G_3 were non-empty, so we moved tags $\boxed{1}$, $\boxed{2}$ and $\boxed{3}$ to clusters in those groups. G_4 was an empty group, so we first created a new cluster A_5, then moved the lone tag $\boxed{4}$ to it.

The total number of *Move* operations is as follows: Each cluster needs to move out all objects except those belonging to its own group. The moving cost for clus-

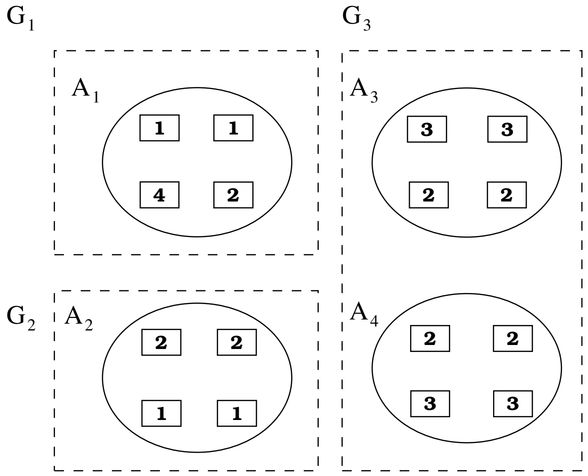

Figure 6. The group assignment for partition A. Cluster A_4 was assigned to group G_3 (assigning it to group G_2 would also work).

ter A_i is $|A_i| - max_{j=1}^{m}(v_{ij})$. Thus the total cost is $n-\sum_{i=1}^{l} max_{j=1}^{m}(v_{ij})$. We use M to denote the total number of *Move* operations (in our example M is $16-(2+2+2+2)=8$).

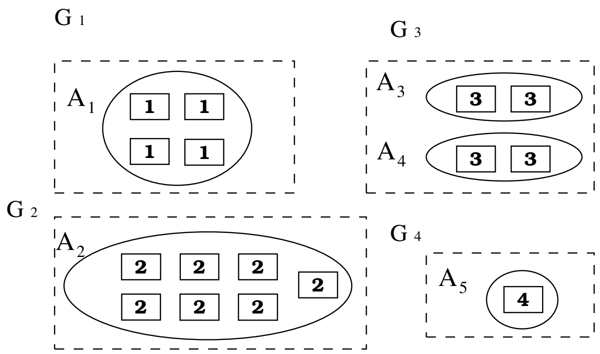

Figure 7. Partition A after all the *Move* operations.

After all the *Move* operations have been performed, any cluster in group G_k will contain only tags T_k. Then, we join all clusters belonging to the same group. After all the *Join* operations, every group is non-empty and contains exactly one cluster. Thus, our transformation is complete.

In our example, since only G_3 contains two or more clusters, we need only join A_3 and A_4. The resulting partition (shown in figure 8) is isomorphic to partition B.

The number of *Join* operations within group G_k is $|G_k|-1$. Since *Join* operations happen only within original non-empty groups (i.e. non-empty before any *Move* and *Join* operations), the total cost of *Join* operations is $l-g$, where l is the number of clusters in partition A and g is the number of non-empty groups (in our example we have $l=4, g=3$, and $l-g=1$, which is consistent with the one *Join* operation already performed). Therefore, the total cost of our algorithm is $M+l-g$ ($8+4-3=9$ in our example). This gives us some intuition on why we attempted to maximize g. The more non-empty groups, the smaller the cost.

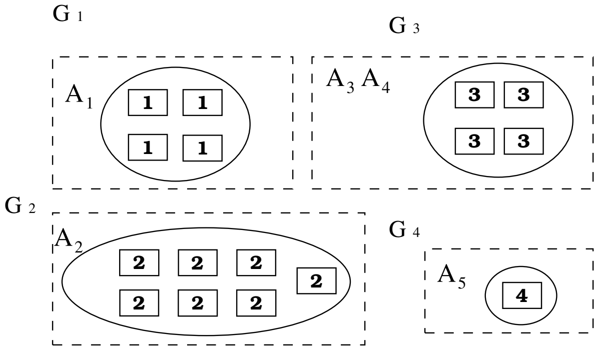

Figure 8. Partition A after all the *Join* operations.

Our algorithm computes MoJo distance in polynomial time. Tag assignment can be done in $O(n\log n)$ time, and so is the construction of the bipartite graph. The group assignment requires $O(VE)$ [5]. Because $V=l+m$ and $E \leq l*m$, the total computational complexity of our algorithm is $O(n\log n+(l+m)*lm)$.

3. Proof of correctness

In this section, we will prove that our algorithm guarantees the exact value of the MoJo distance.

In the following, we will denote by $\mathbb{M}_{A\rightarrow B}$ a *Move* operation that moves an object from cluster A to cluster B. Also, $\mathbb{J}_{A+B}$ will denote a *Join* operation that merges clusters A and B. We will also represent any algorithm that attempts to transform partition A to partition B by its corresponding sequence of operations. We will start by showing that two operations in a minimum-length sequence can be swapped.

Lemma 3.1. *Any two Move operations can be swapped.*

Proof. This is obvious when the two *Move* operations refer to different objects. However, two *Move* operations in a minimum-length sequence cannot refer to the same object. □

Lemma 3.2. *Any two Join operations can be swapped.*

Proof. This is trivial to see when the two *Join* operations refer to different clusters. If they refer to the same clusters (as in $\mathbb{J}_{A+B}\mathbb{J}_{AB+C}$) the associativity of the *Join* operation allows us to perform these operations in any order. □

Lemma 3.3. *Any two Move and Join operations can be swapped.*

Proof. There exist three distinct relevant possibilities for an $\mathbb{M}\mathbb{J}$ sequence (note that the $\mathbb{M}_{A\to B}\mathbb{J}_{A+B}$ sequence is not possible since it is equivalent to $\mathbb{J}_{A+B}$). All three cases are listed below with their corresponding $\mathbb{J}\mathbb{M}$ sequences. Equivalence is easy to show in all three cases.

$$\begin{aligned} \mathbb{M}_{A\to B}\mathbb{J}_{C+D} &\equiv \mathbb{J}_{C+D}\mathbb{M}_{A\to B} \\ \mathbb{M}_{A\to B}\mathbb{J}_{B+C} &\equiv \mathbb{J}_{B+C}\mathbb{M}_{A\to BC} \\ \mathbb{M}_{A\to B}\mathbb{J}_{A+C} &\equiv \mathbb{J}_{A+C}\mathbb{M}_{AC\to B} \end{aligned}$$

□

From the above lemmas, we can deduce that any two operations in a minimum-length sequence can be swapped. Because of the commutativity of *Move* and *Join* operations, we only need to consider competing sequences that perform all the *Move* operations first.

We define the notion of groups for any sequence of operations as follows: After all the *Move* operations have been performed, each cluster must contain only one kind of tag (*Join* operations cannot separate objects with different tags). We assign A_i in group G_k iff A_i contains only T_k tags after all the *Move* operations. Clearly, this definition is consistent with the way we defined groups for our algorithm.

Lemma 3.4. *For any A_i, at least $|A_i| - max_{j=1}^{m}(v_{ij})$ objects must be moved out using Move operations.*

Proof. Any two objects in A_i that have different tags, i.e. they belong to different clusters in partition B, must be separated before the transformation is complete. No matter how many times A_i joins with other clusters, any two objects can not be separated because they will always be moved together. The only way to separate them is to move one object out. Since for any cluster A_i, only one kind of tag can remain, the cost of *Move* operations for each cluster is at least $|A_i| - max_{j=1}^{m}(v_{ij})$. □

As a result, the total cost to separate different tags is at least

$$\sum_{i=1}^{l}(|A_i| - max_{j=1}^{m}(v_{ij})) = n - \sum_{i=1}^{l} max_{j=1}^{m}(v_{ij}) = M$$

which is the number of *Move* operations used by our algorithm. Thus, our algorithm uses the fewest *Move* operations possible.

Next, we will prove the following lemma:

Lemma 3.5. *A sequence that performs M Move operations cannot perform fewer Join operations than our algorithm.*

Proof. Any sequence that uses M *Move* operations will have to assign groups in a manner consistent with the bipartite graph built by our algorithm. Let g' be the number of initially non-empty groups for a competing sequence. Since our algorithm has maximized the number of non-empty groups, we have $g \geq g'$.

Also, let us assume that the *Move* operations create a new clusters. Since a sequence that uses the minimum number of *Move* operations cannot delete any clusters (since it would then use $|A_i|$ moves for that cluster as opposed to $|A_i| - max_{j=1}^{m}(v_{ij})$), the number of clusters after the *Move* operations will be $l + a$. Therefore, the number of *Join* operations required will be $l + a - m$.

Since, any competing sequence will have to create a cluster for each initially empty group G_e (since objects of tag T_e cannot remain in any other group), we have $a \geq m - g'$. Therefore, the number of *Join* operations required will be at least $l - g'$. However, since $g \geq g'$, we know that our algorithm performs the fewest possible *Join* operations ($l - g$) for a sequence that uses M *Move* operations. □

The only remaining possible problem for our proof is that there exists a sequence that uses more than M *Move* operations but might defeat our algorithm by performing significantly fewer *Join* operations. The following lemma concludes the proof:

Lemma 3.6. *For any sequence S that includes more than M Move operations, there exists a sequence S' of equal or smaller length that includes exactly M Move operations.*

Proof. Since sequence S includes more than M *Move* operations, there exists a cluster A_x which used more than $|A_x| - v_{xz}$ *Move* operations, where $v_{xz} = max_{j=1}^{m}(v_{xj})$. A_x must be pure (containing only one kind of tag) or empty after all the *Move* operations.

If A_x is empty, the proof of the lemma is trivial: The series of *Move* operations involving objects of tag T_z can be replaced by a *Join* operation with any of the clusters in group G_z.

If A_x is pure and contains only tag T_y, we know that $v_{xy} < v_{xz}$, or equivalently $v_{xz} \geq v_{xy} + 1$.

The *Move* and *Join* operations of S that involve A_x designate a sub-sequence S_x. Because of the commutativity of *Move* and *Join* operations, we can concentrate on this subsequence knowing that the rest of the operations will not be affected.

We can write S_x as $\mathbb{M}'\mathbb{M}_{A_x\to G_z}\mathbb{J}_{A_x+G_y}$, where:

- $\mathbb{M}'$ denotes a series of *Move* operations involving objects with tags other than T_y or T_z.

- $\mathbb{M}_{A_x \to G_z}$ denotes a series of *Move* operations involving objects tagged with T_z. Thus, $|\mathbb{M}_{A_x \to G_z}| = v_{xz}$.

- $\mathbb{J}_{A_x + G_y}$ denotes the *Join* operation of A_x with the other clusters in G_y. Since G_y may not include other clusters $|\mathbb{J}_{A_x + G_y}|$ is either 0 or 1.

The following sequence (similarly defined) will also accomplish the transformation of A to B and will use the minimum number of *Move* operations for cluster A_x:

$$S'_x = \mathbb{M}' \mathbb{M}_{A_x \to G_y} \mathbb{J}_{A_x + G_z}$$

It suffices to show that $|S'_x| \leq |S_x|$. We have: $|S'_x| = |\mathbb{M}'| + |\mathbb{M}_{A_x \to G_y}| + |\mathbb{J}_{A_x + G_z}| \leq |\mathbb{M}'| + v_{xy} + 1 \leq |\mathbb{M}'| + v_{xz} \leq |\mathbb{M}'| + |\mathbb{M}_{A_x \to G_z}| + |\mathbb{J}_{A_x + G_y}| = |S_x|$.

□

4. Implementation

The new MoJo Algorithm has been implemented in Java. Since the calculation of the quantity $M + l - g$ is sufficient, the new implementation does not have to simulate the transformation, as was the case with the heuristic algorithm. This results in a much faster calculation of the MoJo distance.

A hash table was used to perform the tag assignment. We start by reading the target partition, i.e. partition B. We make a new hash entry for each object and the cluster in which it belongs. In our example, object ⑫ belongs in cluster B_2. When we read this record from the file of partition B, we store ⑫ as the hash key and B_2 as the content, an operation that costs $O(1)$. Next, we read the source partition, i.e. partition A. When we encounter object ⑫ , which belongs in cluster A_3 in partition A, we get its corresponding cluster in B from the hash table in $O(1)$ on average. Thus, each object requires $O(1)$ time, which shows that the tag assignment can be done in linear time (note that this is average-case analysis).

Using a hash table is a practical solution for the problem at hand. However, by using a different structure such as a balanced search tree, we can guarantee that tag assignment can be done in worst-case time complexity $O(n\text{log}n)$.

The calculation of the total number of objects (n), the number of clusters in partition A and B (l and m), and the maximum value of v_{ij} for all i can also be done during the process of reading the input files. As a result, M can be simply computed as $n - \sum_{i=1}^{l} max_{j=1}^{m}(v_{ij})$.

Because l is already known, the only thing left to compute is the maximum number of non-empty groups, i.e. g. The maximum bipartite matching algorithm [5] has been implemented for this calculation. The MoJo distance is finally computed as $M + l - g$.

The implementation is available for download at `http://www.cs.yorku.ca/~bil/downloads`. It also includes an option that will output the actual sequence of *Move* and *Join* operations that one could perform in order to transform partition A to B.

It is interesting to note that the algorithm presented in this paper can also be used for the exact calculation of MoJoPlus distance [6]. In MoJoPlus, a multi-move operation is added. This operation can be used to move a group of instances from one cluster to another as a single operation that costs only 1.

In our example, cluster A_1 has two $\boxed{1}$s, one $\boxed{2}$ and one $\boxed{4}$. Using MoJo distance, if one wants to move both $\boxed{1}$s to another cluster, one needs to execute two *Move* operations. In MoJoPlus, a multi-move can move both objects in a single step.

In order to adjust the MoJo algorithm to calculate MoJoPlus distance, we only need to change the way M and g are calculated. Because we can move multiple objects with one multi-move operation, the number of tags in each cluster does not matter any more. In MoJo, M is calculated as $\sum_{i=1}^{l} |A_i| - \sum_{i=1}^{l} max_{j=1}^{m}(v_{ij})$, where $|A_i|$ is the number of objects in cluster A_i. In MoJoPlus, $|A_i|$ is replaced by the number of different tags in A_i. We use $T(A_i)$ to denote it. At the same time, $v_{ij} = 1$ for all tags in A_i. Thus M can be computed as $\sum_{i=1}^{l} T(A_i) - l$.

The bipartite graph created for the group assignment will also change. In MoJo, for a group G_k to be considered as a possible group for cluster A_i, it would have to hold that $v_{ik} = max_{j=1}^{m}(v_{ij})$. In MoJoPlus, since moving groups of tags costs only 1, every tag in A_i indicates a possible group selection. In our example, A_1's possible groups change from only G_1 to G_1, G_2 and G_4. However, the maximum bipartite algorithm can still be applied to calculate g.

After calculating M and g, the exact MoJoPlus distance can be calculated as $M + l - g$. In fact, our implementation contains an option to calculate MoJoPlus distance.

5. Experiments

In this section, we present experiments that compare the new MoJo algorithm to the heuristic algorithm presented earlier. We will refer to the old algorithm as HAM (for Heuristic Algorithm for MoJo distance), while the new algorithm will simply be referred to as the MoJo algorithm.

The expected result of our experiments was that the output of MoJo should always be less or equal to that of HAM if tested on the same data files. After thousands of tests, we never found a counterexample where HAM would output a smaller distance than MoJo.

In order to observe the difference between the two algorithms, we developed a random partition generator. For each combination of l, m, and n mentioned below, we created 1000 random pairs of partitions. We run both HAM and MoJo on them and recorded the difference. We then

n	Average difference	Maximum difference	% of HAM=MoJo
20	2.64%	18.18%	75
30	3.77%	19.05%	47
40	3.99%	13.33%	32
50	3.88%	10.81%	26
60	3.71%	11.36%	23
70	2.85%	9.09%	25
80	2.95%	9.68%	23
90	2.66%	7.14%	16
100	2.78%	8.75%	19
200	1.77%	5.45%	13
300	1.34%	4.65%	10
400	1.1%	4.02%	15
500	0.87%	3.09%	11
600	0.75%	2.51%	13
700	0.83%	2.66%	13
800	0.70%	3.05%	10
900	0.60%	2.81%	15
1000	0.61%	2.41%	14
2000	0.39%	1.25%	13
3000	0.34%	1.76%	12
4000	0.28%	1.16%	11
5000	0.24%	0.9%	8

Table 1. Summary of results for l = 10, m = 10.

calculated the maximum and average difference between the two algorithms. These differences are indicated below as percentages on the actual value of MoJo distance. We also recorded the percentage of experiments where the two algorithms output the same result.

We conducted two sequences of experiments. In the first sequence, both l and m were fixed at 10, while the number of objects changed from 20 to 5000. Table 1 presents our findings.

It is interesting to note that the average difference between HAM and MoJo was never higher than 4% and that the output of HAM and MoJo was identical on a large portion of test cases. However, the maximum difference observed was 19.05%, which indicates that HAM had significant problems in some cases. Moreover, when the number of objects is large, HAM rarely finds the optimal value even though it comes close on average.

Figure 9 presents the relationship between the number of objects and the average difference between the two algorithms in graphical form.

In the second sequence of experiments, the number of objects was fixed at 100, while l increased from 2 to 50, and m decreased from 50 to 2. Table 2 summarizes the obtained results, while figure 10 depicts the relationship between l, m and the average difference between HAM and MoJo.

l	m	Average difference	Maximum difference	% of HAM=MoJo
2	50	0.89%	2.17%	32
3	33	1.99%	5.88%	5
4	25	3.03%	4.55%	0
5	20	3.62%	6.74%	0
6	16	4.02%	7.95%	1
7	15	4.09%	8.14%	3
8	12	4.40%	8.33%	0
9	11	3.72%	9.30%	4
10	10	2.78%	8.75%	19
11	9	1.26%	6.33%	32
12	8	0.94%	3.75%	43
15	7	0.78%	2.74%	54
16	6	0.47%	2.74%	70
20	5	0.27%	2.78%	84
25	4	0.16%	1.54%	89
33	3	0.12%	1.67%	92
50	2	0.02%	1.56%	99

Table 2. Summary of results for experiments containing 100 objects.

Again, we observe that the average difference between the two algorithms was never higher than 4.5%. However, we can see that in some cases even after running 1000 experiments, HAM never found the optimal value. This indicates that the new MoJo can provide significant improvement in the calculation of MoJo distance and more accurate results for the purpose of evaluating software clustering algorithms.

We also experimented with two real software systems. The first one was a proprietary system with an authorita-

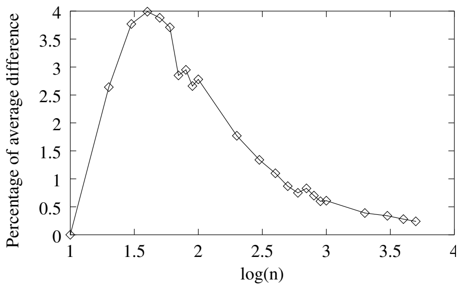

Figure 9. The relationship between the number of objects and the average difference between the two algorithms. The x-coordinate is the logarithm of the number of objects.

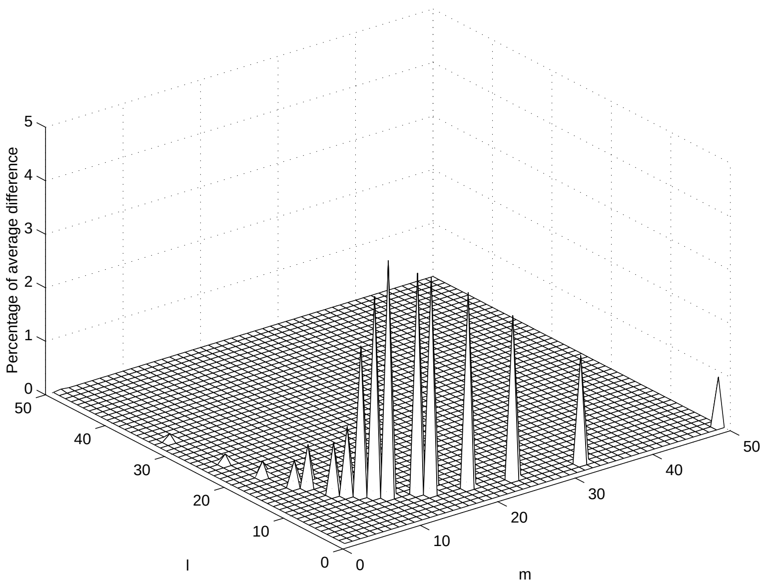

Figure 10. The relationship between the number of clusters in A and B and the average difference between HAM and MoJo.

tive clustering known to us. The second was Linux, a well-known open-source operating system. Its authoritative clustering was available as well [3].

We clustered both systems using ACDC [15] and calculated the MoJo distance of the obtained clustering to the authoritative clustering using both HAM and MoJo. The values for Linux were 551 and 536 respectively, while for the proprietary system they were 325 and 320. This indicated clearly that the new algorithm computes MoJo distance more accurately than HAM.

Finally, it is interesting to note that for all the experiments described above, the MoJo algorithm outputs the result instantaneously. A further experiment was conducted with very large input. We created several large random partitions to test the running speed. Our testing environment was a PC with an Intel Celeron 800 MHz processor and 256MB RAM running Microsoft Windows 2000 Pro and JDK 1.4.01. When l and m were set to 400, and n to 160,000, the total running time was still below 20 seconds. Most of that time was spent in file handling. The same input files were used with HAM. Table 3 shows the results of our performance experiments. It is clear that the new algorithm is significantly faster than the old one.

l	m	n	FH time	TA time	GA time	HAM time
100	100	10000	7381	70	40	64
200	200	40000	13659	171	101	968
300	300	90000	16039	440	40	4683
400	400	160000	19148	671	181	16802

Table 3. The running time of the MoJo algorithm with large input. Time is shown in milliseconds. HAM time is shown in seconds. Legend: FH = File Handling, TA = Tag Assignment, GA = Group Assignment.

6. Related and future work

One of the early works on experimental evaluation of software clustering techniques was presented by Lakhotia and Gravely [8]. Their evaluation metric, however, can only be applied to hierarchical clustering algorithms that produce dendrograms, a feature that limits its applicability.

More recently, Anquetil and Lethbridge [2] used the well-known Information Retrieval measures of Precision and Recall in order to measure similarity between different clusterings of the same software system. Though their work produced many interesting results, the limitations of the two measures are well-documented [10, 13].

Koschke and Eisenbarth [7] presented an elaborate framework that extends and removes the limitations of the approach taken by Lakhotia and Gravely. They incorporate the fact that one may have to cope with approximate matches between clusters in real life. Their approach includes an overall recall rate that captures the quality of a particular technique. One of our future interests would be to investigate how this recall rate compares to the MoJo quality metric [14].

Finally, Mitchell and Mancoridis [10] presented a similarity (*EdgeSim*) and a distance (*MeCl*) measurement that consider relations between components as an important factor for the comparison of software system decompositions. Their approach has definite merit. As a result, we are currently investigating an extension to the MoJo distance that will accommodate this factor.

Another future direction for our work is the extension of MoJo to measure distance between nested decompositions. The approaches presented above (including MoJo) assume either flat decompositions (only one level of clusters) or hierarchical ones (each cluster containing two other clusters/objects). One would expect however, that decompositions of industrial-strength software systems contain several levels of clusters, each containing a significant number of other clusters or objects. Since evaluation techniques geared towards flat decompositions do not translate to nested ones without loss of information, this issue requires further research.

7. Summary

This paper introduced an optimal algorithm for the calculation of MoJo distance between two decompositions of a software system. We presented a proof that the value calculated is indeed the minimum number of *Move* and *Join* operations one needs to perform in order to transform one decomposition to the other. Finally, we presented experiments that demonstrated the effectiveness of our algorithm, and indicated directions for further research.

References

[1] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of CASCON 1997*, pages 184–195, Nov. 1997.

[2] N. Anquetil and T. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255, Oct. 1999.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21th International Conference on Software Engineering*, May 1999.

[4] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, Jan. 1990.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. McGraw-Hill Higher Education, 2001.

[6] M. El-Ramly, P. Iglinski, E. Stroulia, P. Sorenson, and B. Matichuk. Modeling the system-user dialog using interaction traces. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 208–217, Oct. 2001.

[7] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 201–210, June 2000.

[8] A. Lakhotia and J. M. Gravley. Toward experimental evaluation of subsystem classification recovery techniques. In *Proceedings of the Second Working Conference on Reverse Engineering*, pages 262–269, July 1995.

[9] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, 1999.

[10] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of the International Conference on Software Maintenance*, pages 744–753, Nov. 2001.

[11] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, Dec. 1993.

[12] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[13] V. Tzerpos. Comprehension-driven software clustering. *Ph.D. Thesis, Department of Computer Science, University of Toronto*, July 2001.

[14] V. Tzerpos and R. C. Holt. Mojo: A distance metric for software clusterings. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 187–193, Oct. 1999.

[15] V. Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, Nov. 2000.

[16] V. Tzerpos and R. C. Holt. On the stability of software clustering algorithms. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 211–218, June 2000.

[17] A. van Deursen and T. Kuipers. Identifuing objects using cluster and concept analysis. In *Proceedings of the 21th International Conference on Software Engineering*, pages 246–255, May 1999.