SUBSEQUENCE AND RUN HEURISTICS FOR SORTING BY TRANSPOSITIONS

SCOTT A. GUYER, LENWOOD S. HEATH, AND JOHN PAUL C. VERGARA

ABSTRACT. *Sorting by transpositions* is the problem of finding the minimum number of transpositions required to sort a permutation π. A transposition involves repositioning a contiguous sequence (block) of elements by inserting it elsewhere in the permutation. The problem has applications in the study of genome rearrangements and phylogeny reconstruction.

In this paper, several heuristics based on analyses of subsequences and runs in a permutation are employed. Experimental results are provided. The algorithm based on the longest increasing subsequence in a permutation appears most promising.

1. INTRODUCTION

In molecular biology, two genomes are compared by analyzing their respective gene sequences. Phylogenetic trees of genomes are constructed based on information about how closely two genomes are related. A genome typically evolves by mutations that are insertions, deletions, duplications, inversions, and transpositions of its fragments. When the operations are such that fragments are never introduced nor removed, one may think of these mutations as operations applied to a string or permutation of genes. In studying the distance between two genomes, the biologist wants to determine the minimum number of operations required to transform one into the other.

Some algorithmic results on this problem have already been obtained. These results typically involve restricting the operations on a genome to a subset of the operations identified above. Wagner [8] and Lowrance and Wagner [7] investigate *string-to-string correction*, a similar problem on strings involving edit operations such as insertion, deletion, and adjacent swaps of single symbols in the string. They show that the problem is NP-complete in general but solvable in polynomial time when only certain operations are allowed. Kececioglu and Sankoff [5, 6] and Bafna and Pevzner [1] have results on *sorting by reversals*. Here, the operation is a reversal, i.e., a fragment of the genome is inverted. Notice that the problem can be restated as a sorting problem since the destination genome can be considered as the sorted permutation of genes and one seeks the minimum number of reversals required to sort a given genome. They identify upper and lower bounds for the reversal distance between permutations and devise corresponding approximation algorithms. Bafna and Pevzner's algorithm, in particular, has a performance guarantee of 1.75. Heath and Vergara [4] test some conjectures on this problem and provide a mechanism to generate counterexamples. It is not yet known whether this problem is NP-complete or has a polynomial time algorithm. Hannenhalli and

1991 *Mathematics Subject Classification.* 68R00; Secondary 06-04.

Pevzner [3] show that a variant of the problem, sorting *signed* permutations by reversals, is solvable in $O(n^4)$ time.

Given a permutation $\pi = \pi_1\pi_2\ldots\pi_n$, a *transposition* $\rho(i,j,k)$, where $1 \leq i < j < k \leq n+1$, has the effect of repositioning $\pi_i\pi_{i+1}\ldots\pi_{j-1}$ immediately to the left of element π_k. Equivalently, it has the effect of repositioning $\pi_j\pi_{j+1}\ldots\pi_{k-1}$ immediately to the left of π_i. Consider the permutation $\pi = 236745981$, for example. Applying $\rho(3,5,8)$ to π transforms π to $\pi \cdot \rho = 234596781$. In this paper, we investigate *sorting by transpositions*, the problem of determining the transposition distance of a permutation π, i.e., the minimum number of transpositions required to sort π.

Bafna and Pevzner [2] study sorting by transpositions and provide a metric on a permutation that counts cycles in a graph representation of the permutation. They derive upper and lower bounds and present an approximation algorithm with a 1.5 performance guarantee. We considered implementing these algorithms to compare to our heuristics but were unable to provide the implementation details needed to do so.

In this paper, we devise heuristics and formulate corresponding algorithms for the problem. We analyze subsequences in the permutations. Given a permutation $\pi = \pi_1\pi_2\ldots\pi_n$, a *subsequence* of length k is a sequence $\pi_{i_1}\pi_{i_2}\ldots\pi_{i_k}$ such that $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. One heuristic involves longest increasing subsequences. Given a permutation and a transposition, we measure the length of the longest increasing subsequence in the resulting permutation. Among all possible transpositions, we select one that yields the longest increasing subsequence.

Our remaining heuristics are based on runs. A run (*increasing strips* in previous literature [1, 2, 5, 6]) is defined as a maximal subsequence of consecutive, contiguous elements in the permutation. We employ algorithms that repeatedly select a run and then merge it with other runs using appropriate transpositions.

We also implement branch and bound algorithms to determine the exact transposition distances of permutations of small size. We use the same technique to determine the correct run selections for our second heuristic.

The paper is structured as follows. Section 2 discusses the longest increasing subsequence heuristic and a corresponding algorithm. Section 3 presents several algorithms based on runs. In Section 4, we describe two branch and bound algorithms: one that finds the exact transposition distance of a permutation and another that finds the transposition distance of a permutation assuming only transpositions that place runs are allowed. In Section 5, we describe our methods for generating permutations for the experiments. Section 6 summarizes the results of our experiments. We conclude in Section 7 where we also speculate about possible improvements in our algorithms.

2. Longest Increasing Subsequence

Let $\pi = \pi_1\pi_2\ldots\pi_n$ be a permutation. An increasing subsequence of π is a subsequence $\pi_{i_1}\pi_{i_2}\ldots\pi_{i_k}$ such that for all j, where $1 \leq j < k$, we have $\pi_{i_j} < \pi_{i_{j+1}}$. A *longest increasing subsequence* $LIS(\pi)$ is an increasing subsequence of π of maximum length. There may be several longest increasing subsequences; we denote any one of these by $LIS(\pi)$. For example, if $\pi = 354726819$, $LIS(\pi) = 34689$. Clearly, π is sorted if and only if the length $|LIS(\pi)|$ of its longest increase subsequence is n.

TSLIS(π) Sorting by transpositions using the LIS metric.

INPUT: A permutation π.
OUTPUT: A sequence of transpositions $\rho_1, \rho_2, \ldots, \rho_k$ such that $\pi \cdot \rho_1 \cdot \rho_2 \cdot \ldots \cdot \rho_k$ is sorted.

$k \leftarrow 0$
while ($|LIS(\pi)| < n$) **do**
    $\triangleright$ while π is not sorted
    begin
        $k \leftarrow k + 1$
        $\rho_k \leftarrow$ the transposition such that $|LIS(\pi \cdot \rho_k)|$ is maximum
        $\pi \leftarrow \pi \cdot \rho_k$
    end
return $\rho_1, \rho_2, \ldots, \rho_k$

FIGURE 1. Algorithm TSLIS.

We present a straightforward algorithm based on longest increasing subsequences. Algorithm TSLIS is presented in Figure 1. The algorithm iteratively chooses a transposition based on the following criteria. Given the current permutation π, the algorithm selects, from all possible transpositions, the transposition ρ such that $|LIS(\pi \cdot \rho)|$ is maximum. The algorithm stops when the LIS of the current permutation reaches the value n, indicating that the permutation is sorted. Since there are $O(n^3)$ different transpositions, and it takes $O(nlogn)$ time to determine a longest increasing subsequence of a permutation, TSLIS runs in $O(n^5 logn)$ time (at worst n iterations).

3. RUNS

A run in a permutation $\pi = \pi_1\pi_2 \ldots \pi_n$ is defined as a maximal contiguous subsequence of consecutive elements. In particular, a run $R(\pi, a, b)$, where $1 \leq a \leq b \leq n$, is the subsequence $\pi_a\pi_{a+1} \ldots \pi_b$ such that for all i, where $a \leq i < b$, we have $\pi_i = \pi_{i+1} - 1$ and R cannot be extended at either end to obtain a longer such subsequence. For example, the permutation $\pi = 345786912$ has runs 345, 78, 6, 9, and 12. We can "place" the run 345 by inserting it right before element 6. This results in the permutation 783456912. So, given a run $R(\pi, a, b)$, let π_j be the element whose value is $\pi_b + 1$. The *placement* of a run is defined as the transposition $\rho(a, b, j)$ if $j > b$ or the transposition $\rho(j, a, b)$ if $j < a$. Notice that the run is merged with one or two other runs when a placement occurs, resulting in a longer run and fewer total runs.

There are a few exceptions we need to discuss. First, whenever the run ends with n, the placement of the run involves moving it to the right side of the permutation. For example, placing the run 789 in the permutation 237896145 results in the permutation 236145789. Second, a run on the left side of the permutation that includes element 1 need not be placed. Likewise, a run on the right side of the permutation that includes element n need not be placed. For example, in the permutation 126453789, the runs 12 and 789 are not considered for placement.

TSRun(π) Sorting by transpositions using runs.

INPUT: A permutation π.
OUTPUT: A sequence of transpositions $\rho_1, \rho_2, \ldots, \rho_k$ such that $\pi \cdot \rho_1 \cdot \rho_2 \cdot \ldots \cdot \rho_k$ is sorted.

```
k ← 0
while (π is not sorted) do
    begin
        k ← k + 1
        R ← run of π
        ▷ the run depends on some selection process.
        ρ_k ← the transposition that places R in π
        π ← π · ρ_k
    end
return ρ_1, ρ_2, ..., ρ_k
```

FIGURE 2. Algorithm TSRun.

We propose different algorithms, each of which iteratively selects a run according to some criterion. The general framework of the algorithms is shown in Figure 2. The first three algorithms select the shortest, longest, and a random run, respectively. After a run has been selected, it is placed in the permutation. The algorithm stops when the permutation is already sorted, i.e., there is only one run left. For the three algorithms mentioned, every iteration decreases the total number of runs by at least one. Occasionally, there is a decrease of two or three; that is, placing a run may cause separate runs to be linked. This observation leads to a fourth selection criterion, which is to select the run whose placement yields the best net decrease in the total number of runs. We name the four algorithms TSRunLong, TSRunShort, TSRunRand and TSRunBest. All these algorithms run in $O(n^2)$ time, because there are at most n iterations and identifying the runs in the current permutation requires a single scan of the permutation implying $O(n)$ time per iteration.

4. Branch and Bound

Algorithm TSBnB shown in Figure 3 is a branch and bound algorithm to determine the minimum number of transpositions required to sort a permutation.

This algorithm is patterned after the branch and bound algorithm for sorting by reversals presented by Kececioglu and Sankoff [5]. Notice that the algorithm eliminates from consideration those transpositions that could not lead to an optimal solution. Unfortunately, the worst-case time complexity of this algorithm is exponential, which makes it impractical for large n. However, we use algorithm TSBnB for small n so that we can compare the results of the algorithms of the previous sections with optimal results.

Recall that the algorithms in Section 3 are based on runs. We have devised another branch and bound algorithm, Algorithm TSRunBnB, an algorithm similar

TSBnB(π) Sorting by transpositions using branch and bound.

INPUT: A permutation π.
OUTPUT: A sequence of transpositions $\rho_1, \rho_2, \dots, \rho_k$ such that $\pi \cdot \rho_1 \cdot \rho_2 \cdot \dots \cdot \rho_k$ is sorted.

```
1   GLOBAL bound, current[1..n], best[1..n]
2   bound, best ← UpperBound(π)
3   Search(π, 0)
4   return bound, best
```

Search(π, *depth*) Recursive search routine used by TSBnB.

```
1   if π is sorted then
2       if depth < bound then
3           bound, best ← depth, current
4   else
5       foreach transposition ρ do
6           if LowerBound(π · ρ) + depth + 1 < bound then
7               begin
8                   current[depth + 1] ← ρ
9                   Search(π · ρ, depth + 1)
10              end
```

Figure 3. Algorithm TSBnB.

to TSBnB (Figure 3) with line 5 changed to read "**foreach** transposition ρ that places a run **do**". Algorithm TSRunBnB determines the minimum number of transpositions required to sort a permutation, where transpositions are restricted to those that place a run. We use Algorithm TSRunBnB to determine whether any of the run-based algorithms in Section 3 produce comparable results (for relatively small n).

The algorithms in this section require the computation of upper and lower bounds for the transposition distance of a permutation π. These are computed as follows. Let r be the number of runs in π. UpperBound(π) $= r - 1$ and LowerBound(π) $= \lceil r/3 \rceil$. The upper bound computation was obtained by observing that the number of runs in a sorted permutation is 1 and that placing any run decreases the total number of runs by at least 1. For the lower bound, we notice that any transposition would cause a decrease in total number of runs of at most 3.

5. Generating permutations

Two ways of generating permutations were employed. The first method involves randomly selecting numbers from $\{1, 2, \dots, n\}$ in sequence to form a permutation of size n. Bafna and Pevzner [1] prove that permutations generated this way have an expected *reversal* distance close to $n - 1$, which is the worst-case distance for size-n permutations. We suspect that the same is true for the transposition distance of randomly generated permutations. Bafna and Pevzner [2] conjecture that the worst case value for the transposition distance of size-n permutations is $\lfloor n/2 \rfloor + 1$.

The second method generates a permutation by applying m random transpositions to the identity permutation. Clearly, for this method, the transposition distance of the permutation obtained is at most m. In our experiments, we use the following values for m: $\lfloor n/4 \rfloor$, $\lfloor \sqrt{n} \rfloor$, and $\lfloor log(n) \rfloor$. The intention is to determine whether the algorithms perform differently when the expected distances are smaller in value.

6. Experimental results

In this section, we describe the results of our experiments. Section 6.1 discusses the first set of tests, where the algorithms were tried on randomly generated permutations. In Section 6.2, a second set of experiments is discussed; here, the permutations were generated using the second method discussed in Section 5. The machine used in these experiments is a DEC Alpha 3000 running OSF/1 V3.2. All algorithms were implemented in C.

6.1. **Random permutations.** There are four tests in this first set of experiments.

The first test was designed to compare the number of transpositions in the solutions found by each algorithm. Fifty permutations of size 12 were randomly generated. The limit of 12 is imposed by the exponential time complexity of our branch and bound algorithms. We tested all seven algorithms on this set of permutations and compared the number of transpositions in the obtained solutions. We also determined the number of times the algorithms matched the optimal result obtained by TRBnB, the exact algorithm. Table 1 reports the results. Algorithms TSLIS and TSRunBest produced much more accurate results than Algorithms TSRunShort, TSRunLong and TSRunRand. It is also interesting to note that Algorithm TRunBest performed almost as well as Algorithm TSRunBnB which is the one that chooses the run that leads to a best solution (assuming only run-placing transpositions are used).

The second test (results shown in Table 2) reinforces our observations on the first test. Here, we used an even smaller n, $n = 6$, but we ran the algorithm for all permutations of that size.

The third test compares the polynomial-time algorithms on larger permutations. For this test, we used 100 randomly generated permutations of sizes 20, 30, 40, and 50. Table 3 shows how the algorithms compared. Comparing only the run-based heuristics, we notice that Algorithm TSRunBest produces significantly better results for these permutations. Algorithm TSLIS is consistently better than these run-based heuristics and it appears that TSLIS performs increasingly better as the permutation size increases. Furthermore, note that the results obtained with Algorithm TSLIS are fairly close to the worst-case $\lfloor n/2 \rfloor + 1$ transposition distance.

The final test was aimed at testing permutations of even larger sizes. One hundred random permutations each of size 1000 and 5000 were generated for the four run-based algorithms to process. The other algorithms were too slow for permutations of these sizes. Table 4 reports the results for the permutations of size 1000 and Table 5 reports the results for the permutations of size 5000. Run time measurements are included in Table 5 illustrating the relative speeds of the algorithms. From the results in the tables, we can see that Algorithms TSRunShort,

Algorithm	Average # of Transpositions	Standard Deviation	# of Correct Results
TSRunRand	7.00	1.74	3
TSRunShort	6.60	1.74	6
TSRunLong	6.22	1.58	9
TSRunBest	5.00	1.20	38
TSLIS	4.94	1.06	37
TSRunBnB	4.90	1.04	39
TSBnB	4.64	0.87	50

Table 1. All algorithms on 50 permutations of size 12.

Algorithm	Average # of Transpositions	Standard Deviation	# of Correct Results
TSRunShort	3.44	1.07	264
TSRunRand	3.17	1.12	420
TSRunLong	2.93	0.91	503
TSRunBest	2.70	1.20	637
TSRunBnB	2.68	0.75	673
TSLIS	2.67	0.74	675
TSBnB	2.61	0.70	720

Table 2. All algorithms on all 6! permutations of size 6.

Algorithm	Size 20	Size 30	Size 40	Size 50
TSRunRand	13.46	21.88	30.59	39.46
TSRunShort	12.68	20.90	29.79	38.11
TSRunLong	12.08	20.49	28.84	37.93
TSRunBest	9.97	17.11	24.72	32.68
TSLIS	9.52	15.76	22.22	28.77

Table 3. Average number of transpositions for 100 permutations.

TSRunLong and TSRunRand perform similarly. Algorithm TSRunBest produced better results although the difference in its performance against the other run-based heuristics appear less noticeable for permutations of larger sizes. Note that although the TSRunShort and TSRunLong algorithms produce slightly better results, they are not significantly better than picking any run at random. Also, note that the values obtained by all algorithms are not anywhere near the worst-case $\lfloor n/2 \rfloor + 1$, indicating that run-based algorithms perform poorly, particularly for larger n. Finally, Algorithm TSRunBest, as expected, ran slower than the other algorithms since it entailed more evaluation on the runs to be considered for placement.

6.2. **Smaller transposition distances.** In this section, we repeat most of the tests in the Section 6.2 but use permutations with smaller expected transposition distances. The permutations were generated using the second method described in

Algorithm	Average # of Transpositions	Standard Deviation
TSRunRand	963.4	9.29
TSRunShort	954.1	13.86
TSRunLong	955.2	8.54
TSRunBest	927.0	15.32

Table 4. Size-1000 Permutations.

Algorithm	Average # of Transpositions	Standard Deviation	Average Time	Standard Deviation
TSRunRand	4929.5	13.78	20.8	0.95
TSRunShort	4908.9	21.19	14.6	0.56
TSRunLong	4913.9	14.49	17.5	1.52
TSRunBest	4847.8	30.00	35.6	0.60

Table 5. Size-5000 Permutations.

Algorithm	Size 20	Size 30	Size 40	Size 50
TSRunRand	7.71	11.28	17.55	21.98
TSRunShort	7.66	11.97	17.82	22.23
TSRunLong	6.51	10.21	15.81	19.90
TSRunBest	5.33	8.42	13.23	17.20
TSLIS	5.29	8.06	12.57	15.85
m	5	7	10	12

Table 6. Average transposition distance; $m = \lfloor n/4 \rfloor$.

Algorithm	Size 20	Size 30	Size 40	Size 50
TSRunRand	6.44	9.04	11.43	13.52
TSRunShort	6.48	8.88	11.53	13.64
TSRunLong	5.23	7.48	9.74	11.69
TSRunBest	4.34	6.08	7.64	9.39
TSLIS	4.35	6.14	7.70	9.19
m	4	5	6	7

Table 7. Average transposition distance; $m = \lfloor \sqrt{n} \rfloor$.

Section 5; recall that this method requires an additional parameter m representing the maximum transposition distance for the permutations.

Table 6 shows the results for 100 permutations of size $n = 20$, 30, 40, and 50, where $m = \lfloor n/4 \rfloor$. Table 7 shows the results for permutations for the same values of n but with $m = \lfloor \sqrt{n} \rfloor$. We observe that Algorithm TSRunBest performs almost as well as Algorithm TSLIS particularly when m is smaller ($\lfloor \sqrt{n} \rfloor$).

The results we obtained for permutations of larger sizes appear consistent with the previous observation. Permutations of size 1000 and 5000 were used for $m =$

Algorithm	Random	$m = 250$ ($\lfloor n/4 \rfloor$)	$m = 31$ ($\lfloor \sqrt{n} \rfloor$)	$m = 6$ ($\lfloor log(n) \rfloor$)
TSRunRand	963.4	519.2	82.6	13.9
TSRunShort	954.1	520.2	82.7	14.2
TSRunLong	955.2	516.3	79.4	12.0
TSRunBest	927.0	511.1	75.2	9.4

Table 8. Average transposition distance; $n = 1000$.

Algorithm	Random	$m = 1250$ ($\lfloor n/4 \rfloor$)	$m = 70$ ($\lfloor \sqrt{n} \rfloor$)	$m = 8$ ($\lfloor log(n) \rfloor$)
TSRunRand	4929.5	2624.9	198.9	19.4
TSRunShort	4908.9	2625.9	199.2	19.8
TSRunLong	4913.9	2620.9	195.0	17.3
TSRunBest	4847.8	2613.5	189.6	14.0

Table 9. Average transposition distance; $n = 5000$.

$\lfloor n/4 \rfloor$, $\lfloor \sqrt{n} \rfloor$, and $\lfloor log(n) \rfloor$. Tables 8 and 9 shows the results. When $m = \lfloor log(n) \rfloor$, the measurements obtained are much closer to the expected distance m particularly for Algorithm TSRunBest.

7. Conclusions

Our experiments suggest that our algorithm based on longest increasing subsequences, TSLIS, has the potential to produce near-optimal results. To make the algorithm a practical alternative, it is essential to reduce its time complexity dramatically. We have some ideas on how to do this. TSLIS guarantees a solution consisting of at most $n - |LIS(\pi)|$ transpositions. This is because in the worst case, the longest increasing subsequence will improve by one element every time; equivalently, a single element is inserted in the current LIS on every iteration. Note that ties often occur when choosing a transposition; this means that the resulting LIS for different transpositions may yield the same lengths. This suggests that a possible improvement would be to determine the LIS (select an arbitrary one) and monitor the elements not in the LIS. The algorithm could then favor the transposition whose resulting non-LIS elements are less scattered so that succeeding transpositions are less likely to increase the length of the LIS by just one. There may also be some advantages to a hybrid algorithm that combines the approach of merging runs with that of choosing the transposition that creates the longest increasing subsequence.

There are permutations with which the LIS heuristic will not lead to an optimal solution. Consider the permutation $\pi = 54126378$ where $|LIS(\pi)| = 4$. The transposition that leads to an optimal solution in this case would be that which inserts 63 before element 4 ($\rho(2, 5, 7)$). However, this yields $\pi \cdot \rho = 56341278$, where $|LIS(\pi \cdot \rho)|$ remains 4, implying no increase in the longest increasing subsequence. Future work could certainly involve determining performance guarantees of the LIS heuristic.

On the other hand, our run-based algorithms guarantee a solution consisting of at most $r - 1$ transpositions, where r is the number of runs in the permutation.

Algorithm TSRunBest, the most promising of these algorithms, performed well compared to the other run-based algorithms because it deliberately chose the transposition that resulted in the greatest net decrease in the number of runs. The other algorithms, TSRunShort, TSRunLong, and TSRunRand, only infrequently decreased the number of runs by more than 1 as their run selection had nothing to do with the potential to join other runs.

We observe that there are transpositions which yield a net decrease of 3 in total number of runs that do not involve placing just a single run. Consider $\pi = 128943675$. Applying the transposition $\rho(3,6,9)$ (yielding 123678945) decreases the total number of runs by 3 and places both run 3 and 67. This suggests an improvement for Algorithm TSRunBest. Instead of considering only those transpositions that involve a single run, the algorithm could also inspect those that move more than one run.

References

1. Vineet Bafna and Pavel A. Pevzner, *Genome rearrangements and sorting by reversals*, Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science, 1993, pp. 148–157.
2. ______, *Sorting permutations by transpositions*, Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 614–623.
3. Sridhar Hannenhalli and Pavel A. Pevzner, *Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals)*, Twenty-Seventh Annual ACM Symposium on the Theory of Computing, 1995, pp. 178–189.
4. Lenwood S. Heath and John Paul C. Vergara, *Some experiments on the sorting by reversals problem*, Tech. Report TR 95-16, Department of Computer Science, Virginia Polytechnic Institute and State University, 1995.
5. J. Kececioglu and D. Sankoff, *Exact and approximation algorithms for the inversion distance between two chromosomes*, Fourth Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 684, 1993, pp. 87–105.
6. ______, *Efficient bounds for oriented chromosome inversion distance*, Fifth Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 807, 1994, pp. 307–325.
7. Roy Lowrance and Robert A. Wagner, *An extension of the string-to-string correction problem*, Journal of the ACM **22** (1975), no. 2, 177–183.
8. Robert A. Wagner, *On the complexity of the extended string-to-string correction problem*, Proceedings of the 7th Annual ACM Symposium on Theory of Computing, 1975, pp. 218–222.

Virginia Polytechnic Institute & State University, Blacksburg, VA 24061-0106
E-mail address: sguyer@cs.vt.edu

Virginia Polytechnic Institute & State University, Blacksburg, VA 24061-0106
E-mail address: heath@cs.vt.edu

Ateneo De Manila University, Manila 0917, Philippines
E-mail address: jpv@admu.edu.ph