TSP-Graph for agglomerative clustering

Sami Sieranoja∗, Pasi Fränti

"School of Computing, University of Eastern Finland, Box 111, Joensuu FIN-80101, Finland"

ABSTRACT

Agglomerative clustering, and Ward's method in particular, provides good clustering accuracy for most applications. However, its use has been limited by quadratic time complexity which makes it slow for large datasets. It also consumes $O(N^2)$ memory in the case of non-vectorial data. In this work, we propose a fast variant of Ward's method which reduces the number of distance calculations and only needs O(N) memory. It works by constraining the merges to neighbor nodes on a graph. The graph used in the method is a novel fully connected graph which consists of multiple approximate travelling salesman problem (TSP) solutions. This avoids the problems caused by disconnected graph components which may happen with a kNN graph. The method is general and works for all types of data where a distance function can be defined. For a dataset of size 1.28 million, the proposed method has a speedup factor of 25:1 for identical results with the best exact implementation of Ward's method. The algorithm allows a flexible compromise between clustering result quality and running time. For a moderate degradation in quality (NMI 0.90 to 0.80), the speedup factor improves further to 625:1. The proposed TSP-graph can also be used in other applications where a connected neighborhood graph is needed.

1. Introduction

Agglomerative clustering is a hierarchical clustering method which iteratively merges clusters until all points are in the same cluster [1]. Compared to k-means type of partitional clustering, it has the benefit that it produces a result with all possible number of clusters. It has been used widely for many different types of applications, including version identification of malware programs [2], estimating the number of wolves by clustering their howls [3], vector quantization [4], language processing [5] and in the biological sciences [6,7].

However, the use of agglomerative clustering has been limited by high time complexity and memory use. For example, a straightforward implementation of Ward's agglomerative clustering method has $O(N^3)$ time complexity. Speedup methods exist that can improve the processing time to $O(N^2)$, but they either have a high $O(N^2)$ memory requirement [8] or they are only applicable for data in Euclidean space [9, 10]. Approximation methods can provide results even faster, but they have thus far been limited to only vector data [11, 12].

In this work we present a new method to speedup Ward's agglomerative clustering in a way which is applicable for any type of data for which a distance function can be formulated. It works by constraining the merges to neighbor nodes on a graph.

We use a novel *TSP-graph* for the graph. It is built from multiple approximate travelling salesman path (TSP) solutions which makes it fully connected. It contains only the most relevant links which are in most cases the same that an exact Ward's method would have used in the merge process. At the same time, it disregards most of the pairwise distances which are usually between faraway points. It requires only $O(TN)$ memory where T is the number of TSP solutions (typically less than 100).

2. Agglomerative clustering

In this section we shortly discuss how agglomerative clustering works and recall the previous improvements on its performance. We use the following notations:

x_{mi}	The i'th point in cluster m
$d(x,y)$	Distance between points x and y
C_m	The m'th cluster
c_m	Centroid of cluster m
n_m	Size of cluster m
S_m	Sum of squared pairwise distances inside cluster m
S_{AB}	Sum of squared pairwise distances between clusters A and B

The sums are calculated as follows:

$$S_m = \sum_{i=1}^{n_m} \sum_{j=i+1}^{n_m} d(x_{mi}, x_{mj})^2$$

∗ Corresponding author.
Email-addresses: samisi@cs.uef.fi (S. Sieranoja), franti@cs.uef.fi (P. Fränti)

$$S_{AB} = \sum_{i=1}^{n_A} \sum_{j=i}^{n_B} d(x_{Ai}, x_{Bj})^2$$

Agglomerative clustering is a bottom-up hierarchical method which starts by putting every point into its own cluster. Clusters are then iteratively merged by joining the two clusters that are closest to each other according to some linkage criteria (Table 1). In this work we focus on Ward's method which minimizes the sum of squared errors (SSE). Compared to k-means type of clustering, agglomerative clustering has the benefit that it produces a result with all possible number of clusters (2...*N*).

Originally the Ward's method was presented only for numerical data [1]. Later Wishart [6] gave values for Ward's method that allow updating cluster distances after the merge using the Lance-Williams formula [13]. This effectively allows running Ward's method for any kind of data where initial pairwise distances between all points are defined. The generality of Ward's method has also been discussed in [5, 14].

Table 1. Different ways of calculating merge cost between clusters *A* and *B*.

SSE	$\frac{2\cdot(S_A+S_B+S_{AB})}{n_A+n_B} - \frac{2\cdot S_A}{n_A} - \frac{2\cdot S_B}{n_B}$
Average link	$\frac{2}{n_A \cdot n_B} \sum_{i=1}^{n_A} \sum_{j=i}^{n_B} d(x_{Ai}, x_{Bj})$
Complete link	$\max_{x \in C_A, y \in C_B} d(x,y)$

The use of Ward's method has been limited by its high time complexity and memory requirements. A straightforward implementation requires O(N^3) processing time [15]. Several techniques can make the method faster without compromising quality. Kurita uses heap data structure and a distance matrix to reduce the time complexity to O(N^2log$N$) but it increases memory requirement to O(N^2) [16]. The NN-chain algorithm reduces time complexity to O(N^2) [17, 8]. This algorithm is fastest according to Müllner [18] when input is stored as a distance matrix.

Fränti&Kaukoranta [9] proposed to store only the nearest neighbor pointers, and in this way, reduce the memory requirement to O(N). They refer to as Fast Exact PNN. In case of vector data, the cluster means (centroids) can be used to calculate merge cost between two clusters in O(1) time [19]. Further speed-up is achieved by the lazy update strategy [10], which delay the updates of the nearest neighbor pointer until it is picked from the heap. Müllner uses these same ideas in their Generic linkage algorithm [8]. However, all these variants are lower limited by the same time complexity of $\Omega(N^2)$.

A further improvement for Ward's method is possible for certain types of data. In [4] mean-distance ordered partial search (MPS) was used to speed up the search for the nearest cluster. The MPS works for datasets that have low intrinsic dimensionality. These include for example, 2D data or high dimensional data distributed along the diagonal.

Different ways of approximation can make agglomerative clustering even faster. In [11] O(N log$N$) time complexity was achieved by constraining merges between nodes in a kNN graph. Bouguettaya et al. [12] used k-means to generate a reduced dataset and then clustered that using the agglomerative method. However, these speedup approaches limit the applicability of the method to only Euclidean data.

3. TSP-graph

The TSP-graph (Algorithms 1-2) consists of multiple linear orderings of the dataset combined into one graph (Figure 3). In the process of building the linear orders (Figures 1-2), we aim to minimize the sum of distances between consecutive points. Therefore, they can be seen also as crude solutions to the Travelling Salesman Problem (TSP).

3.1. Motivation

The TSP-graph is related to other neighborhood graphs, such as *k* nearest neighbor graph (kNN), and thus, it can potentially be used for other applications beyond agglomerative clustering. As the TSP graph is built from linear orderings of the data, it guarantees that the graph is fully connected. This is an advantage over *k*NN graph, which can contain disconnected components especially when using low values of *k*. The consequence of this is an occasional need for full searches for finding new nearest clusters when the *k*NN list exhausts [27].

TSP-graph has also the benefit of having a fixed upper bound for the number of neighbors. In TSP path, the nodes have only two links. Constructing T paths can therefore produce at most $g=2T$ neighbors. Conversely, in the *k*NN graph one node may be the nearest neighbor for several other nodes, and the number of neighbors is upper bound only by the kissing number [25], which grows exponentially with the dimensions. This affects recommendation systems where the same items tend to become recommended repeatedly [26].

3.2. Construction of the graph

The construction of a single TSP path starts by selecting two random points from the data. The dataset is then split into two groups based on which of these two points are closest. After splitting, a connection is added between the groups. The process continues hierarchically until each group has only one point.

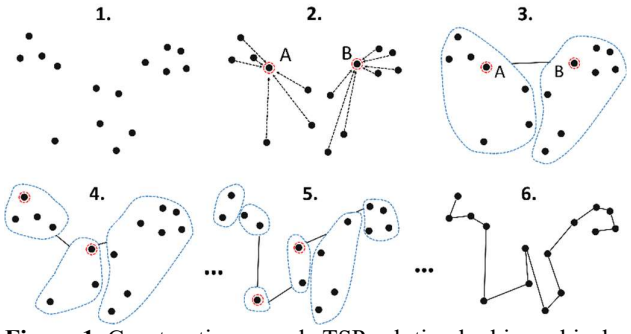

Figure 1. Constructing a crude TSP solution by hierarchical random point division.

After splitting a group, we need to solve the connections to the adjacent cluster. To achieve this, we keep track of the connections between the groups and analyze them. In Figure 2, suppose we split the group AB, containing A and B, and suppose the previous connection was C-(AB)-D. After the split, we have two options: either the connection C-A-B-D or C-B-A-D. We choose the one with the smallest path between the representatives of C, A, B and D.

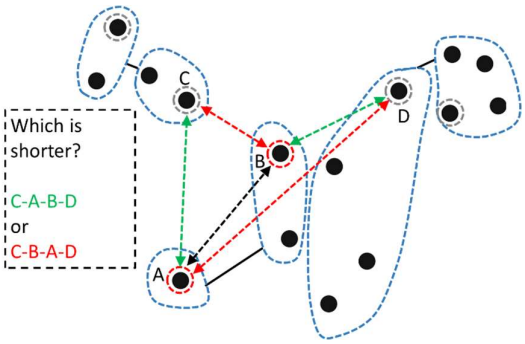

Figure 2. To connect to groups after split, the algorithm selects the shortest path.

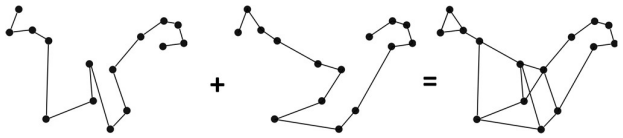

Figure 3. Multiple TSP solutions combined to the TSP-graph.

```
Algorithm 1: createTSPGraph(X,T)
1 g = new Graph();
2 for i=1:T
3   tsp = [1,2...N]
4   llnode = new linkedList();
5   tsp = randomPairDivision(X,tsp,llnode)
6   for j=1:N-1
7     g.addLink(j,j+1)
8 return g
```

```
Algorithm 2: randomPairDivision(X,tsp,llnode)
 1 if size(tsp) <= 1:
 2   return tsp
 3 A = rand(1,size(tsp))
 4 B = rand(1,size(tsp))
 5 tspA=[];tspB=[]
 6 for x in tsp:
 7   if d(X[x],X[A]) < d(X[x],X[B]):
 8     tspA.add(x)
 9   else:
10     tspB.add(x)
11 LA = d(X[A],X[llnode.left.id])
12 LB = d(X[B],X[llnode.left.id])
13 RA = d(X[A],X[llnode.right.id])
14 RB = d(X[B],X[llnode.right.id])
15 (nodeLeft,nodeRight) = llnode.split()
16 if LA+RB < LB+RA:
17   nodeLeft.id=A; nodeRight.id=B;
18   tspL = randomPairDivision(X,tspA,nodeLeft)
19   tspR = randomPairDivision(X,tspB,nodeRight)
20 else:
21   nodeLeft.id=B; nodeRight.id=A;
22   tspL = randomPairDivision(X,tspB,nodeLeft)
23   tspR = randomPairDivision(X,tspA,nodeRight)
24 tsp = tspL+tspR
25 return tsp
```

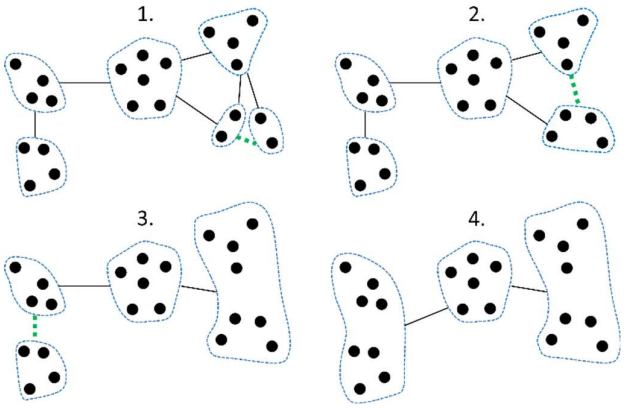

Figure 4. In graph constrained agglomerative clustering merges are allowed only between neighbors in the graph.

4. Clustering using TSP-graph

The main bottleneck of agglomerative clustering is the need to calculate all $O(N^2)$ distances between clusters when selecting the pair of clusters to merge. Instead of calculating distances between all clusters, we allow merges only between nodes that are neighbors in the graph. The TSP-graph has a maximum of $g=2T$ neighbors for each node, which reduces the possible merges from $O(N^2)$ to $O(gN)$. We consider values $T=2...100$ in our experiments.

4.1. Merge Cost

We follow the Ward's method and optimize the *sum of squared errors* (SSE) when selecting the two nearest clusters to merge. The SSE is most often calculated in centroid based clustering as the sum of distances to the nearest centroid. This limits its use to vectorial data as the centroid cannot be calculated for all types of data. However, the SSE can also be formulated in a more general way by using the sum of all squared pairwise distances inside the cluster. This allows to use it for all types of data for which a distance function can be formulated:

$$\begin{aligned} SSE &= \sum_{m=1}^{k}\sum_{i=1}^{n_m} d(x_{mi}, c_m)^2 \\ &= \sum_{m=1}^{k}\sum_{i=1}^{n_m} E[d(x_{mi}, y \in C_m)^2] \\ &= \sum_{m=1}^{k}\sum_{i=1}^{n_m}\frac{1}{n}\sum_{j=1}^{n_m} d(x_{mi}, x_j)^2 = \sum_{m=1}^{k}\frac{2\cdot \mathrm{S}_m}{n_m} \end{aligned} \quad (1)$$

The key point here is to notice that the squared distance to the centroid is the same as the expected squared distance to any point in the cluster. This in turn can be calculated as the average of squared distances to all points in the cluster. The equation is true in case of Euclidean distance. It does not apply generally to all distance functions because the concept of centroid cannot always be defined. In practice, optimizing the same (sum of pairwise distances based) cost function works also in case of other distance functions, as we will show in the experiments section.

By using Equation 1 we can also formulate the merge cost by using the sum of pairwise distances. The merge cost for the clusters A and B is the SSE of the new cluster AB minus the SSE values of the clusters A and B:

$$\begin{aligned} \mathrm{MergeCost}(A,B) &= SSE_{AB} - SSE_A - SSE_B \\ &= \frac{2\cdot(\mathrm{S}_A+\mathrm{S}_B+\mathrm{S}_{AB})}{n_A+n_B} - \frac{2\cdot \mathrm{S}_A}{n_A} - \frac{2\cdot \mathrm{S}_B}{n_B} \end{aligned} \quad (2)$$

4.2. The TSPg-clu algorithm

The TSPg-clu algorithm (Algorithm 3) takes as input the dataset X, number of clusters k, and the number of TSP-solutions to use in the graph (T). As output, it returns k non-overlapping clusters. In case a the dendogram or all possible clusterings are wanted as result, they can be constructed from the merge order which is also returned as result from the algorithm. In this case the algorithm should be run for parameter k=1.

The algorithm starts by creating the TSP-graph from multiple random travelling salesman path (TSP) solutions. The algorithm can also be run on other types of graphs if they are fully connected.

A heap is created from the graph nodes (Algorithm 3, line 2) so that the graph node with the smallest distance (merge cost) to the nearest neighbor is maintained at the top of the heap. There are two nodes (the closest pair) with the same distance to the nearest. It does not matter which one of them appears at the top.

The merge process starts by picking the top node from the heap (Algorithm 3, line 7). This node is then merged with its nearest cluster. This merge process (lines 6-10) continues until the specified number of clusters (k) is reached.

4.3. Merge operation

The merge operation (Algorithm 4) joins given clusters A and B (represented by graph nodes) together and updates both the graph (lines 5-24) and the heap (lines 25-35) data structures accordingly.

The graph is modified by updating the distances (merge costs) to other clusters from the newly merged cluster. When merging clusters A and B, the new cluster C=A∪B inherits all connections that the individual clusters had to other nodes. For all of those, the merge cost needs to be updated.

```
Algorithm 3: TSPg-clu(X,T,k)
 1 g = createTSPGraph(X,T)
 2 h = createHeapFromNodes(g)
 3 N = size(X)
 4 num_clu=N
 5 mergeOrder =[]
 6 while num_clu > k:
 7   node=h.pop()
 8   mergeNodes(node, node.nearest)
 9   mergeOrder.add([node.id,node.nearest.id])
10   num_clu--
11 Cluster = []
12 FOR i=1..k
13   node=h.pop()
14   Cluster[i] = node.points
15 RETURN Cluster,mergeOrder
```

Updating the merge cost (Eq. 2) requires calculating S_{CX} (pdist in the pseudocode), the sum of squared pairwise distances to all neighbor clusters X (lines 5-19). Also the internal sums of pairwise distances (line 21) need to be calculated.

Calculation of the sum of squared pairwise distances (S_{CX}) is the bottleneck here (lines 5-19). It takes $O(n_A \cdot n_X)$ steps to calculate it by brute force in the general case. However, when merging clusters A and B, if both have a connection to cluster x in the graph (lines 5-9), then the sum of squared pairwise distances to X, (S_{CX}) can be calculated simply by $S_{CX} = S_{AX} + S_{BX}$. If only A has a connection to X (lines 10-17), then we estimate S_{BX} by sampling 50 random pairwise distances and multiplying their mean value by $n_B \cdot n_X$.

For the special case of Euclidean data, we can further speed up the S_{CX} calculation to O(1) by using centroids. For this purpose, we derived the following equation using the Equation 2 and the Equation 13 in [19]. It allows to calculate S_{CX} based on the distance of centroids of cluster C and X:

$$S_{CX} = d(c_C, c_X) \cdot n_C \cdot n_X + \frac{S_C \cdot n_X}{n_A} + \frac{S_X \cdot n_A}{n_X} \qquad (3)$$

This requires updating the centroid of the new cluster based on the old centroids for clusters A and B, as follows:

$$c_C = c_a \cdot w + c_b \cdot (w-1), \text{ where } w = n_A/(n_A+n_B) \qquad (4)$$

Also, the sets that contain points belonging to the cluster need to be merged (line 24).

After modifying the graph, also the heap must be updated. The heap is ordered based on the distance to nearest neighbor and the nearest neighbor may change for those nodes which consider either A or B as their nearest neighbor. For each of them, finding the new nearest requires looping all $O(t)$ adjacent nodes. This would add considerably to the computational costs of the merge. Therefore, we use lazy evaluation instead, and mark each of those nodes as dirty (line 27). If a dirty node happens to be at the top of the heap, its nearest neighbor would be updated (lines 31-35). This continues until the top node of the heap is not dirty. The lazy evaluation is possible because the merge costs only increase during the process due to the monotony property [10].

```
Algorithm 4: mergeNodes(nodeA,nodeB)
1 common = nodeA.neighbors ∩ nodeB.neighbors
 2 onlyA = nodeA.neighbors \ common
 3 onlyB = nodeB.neighbors \ common
 4 newneigh = {}
Update graph:
 5 FOR x IN common
 6   S_AX = nodeA.neighbors[x].pdist
 7   S_BX = nodeB.neighbors[x].pdist
 8   a.pdist=SAX+ SBX; a.id=x
 9   newneigh.add(a)
10 FOR x IN onlyA
11   SAX = nodeA.neighbors[x].pdist
12   IF data.type == euclidean
13     SBX = CalcUsingCentroid(nodeA,x) // Eq. 3
14   ELSE
15     SBX = EstimateBySampling(nodeB,x)
16   a.pdist= SAX+SBX; a.id=x
17   newneigh.add(a)
18 FOR x IN onlyB
19   // Similarly as above
20 nodeA.neighbors = newneigh
21 nodeA.internalSum = nodeA.internalSum
22 + nodeB.internalSum + nodeA.nearest.pdist
23 nodeA.centroid = // Eq. 4
24 nodeA.points = nodeA.points ∪ nodeB.points
Update heap:
25 FOR x IN nodeA.neighbors
26   IF x.nearest == nodeA OR x.nearest == nodeB
27     heap.nodes[x.id].outdated = TRUE
28   x.cost = calcCost(nodeA,x) // Eq. 2
29 h.remove(nodeB);
30 nodeA.outdated = true;
31 WHILE heap.topNode.outdated == TRUE
32   node=h.pop()
33   node.nearest = ARGMIN{x.cost: x ∈ node.neighbors}
34   node.outdated = FALSE
35   h.add(node)
```

5. Experimental setup

5.1. Datasets

We have selected a collection of datasets that are suitable for the SSE cost function of the Ward's method. We generated new numerical datasets (RC-var and RC-N) and string datasets (Cntr-v and Cntr-N) for this study. Other datasets are previously published. We use edit distance in case of the string datasets. The new datasets are published online[1].

Basic benchmark: These are well-known [20] 2-dimensional clustering benchmark datasets (s1-s4,unbalance,birch1,birch2). They are used to show that the tested algorithms can solve easier datasets, which can still be challenging to many algorithms such as k-means.

RC-v: 100 random gaussian clusters. Dimensionality D=64, N=200,000. The dataset has 13 different variants with variance of the clusters varied from 0.20 to 0.50 with 0.025 steps. Centroids of the clusters were uniformly distributed in random positions with each attribute in the range [10..11].

RC-N: Otherwise same as RC-v, but the size of clusters is varied. Dataset has 9 different variants with size N=[5k, 10k, 20k, 40k, 80k, 160k, 320k, 640k, 1280k]. Other parameters were fixed as follows: D=64, variance=0.425.

Cntr-v: String dataset with 48 clusters where names of european countries represent the centroids of the clusters and objects are randomly transformed strings formed from the centroid. For example: [urnaine, okraino, kpanev, krae, ukrainef, uxkkaine, xukraine,...] in case of Ukraine cluster. The dataset has 8 variants where amount of transformation is gradually increased. Each cluster has 750 strings and total dataset size is N=36,000.

Cntr-N: Dataset is otherwise same as Cntr-v, but in this case the size of the dataset (size of clusters) is varied, and

[1] http://cs.uef.fi/sipu/datasets

transformation amount is kept fixed. The dataset has six different variants with sizes N=[2400, 4800, 9600, 19200, 38400, 76800].

5.2. Algorithms

The focus of our work is to provide a faster alternative to Ward's method. The benchmark therefore focuses on speed comparison with Ward's method. We do not aim for comprehensive quality comparison with other algorithms as the good performance of Ward's method is well known. We mainly compare with the two fastest known exact Ward implementations (*fclu* and *pnn-mps*) [4,18]. We also selected two other algorithms (*fclu-avg* and *genie*) for comparison to show that the quality of Ward's method is competitive also in cases like string data where it is rarely used. We aimed to select also other Ward approximation methods to compare with, but to the best of our knowledge, no such methods exist which would also maintain the generality of the Ward's method.

fclu: The fastcluster software[2] by [18]. This is currently the fastest implementation of several (exact) agglomerative linkage methods. It has $O(N^2)$ time complexity for Ward's method, average linkage and several others. We use the ward's method (**fclu-ward**) and average linkage (**fclu-avg**) from this package.

pnn-mps: Implementation by Virmajoki [4]. This includes similar Ward algorithm as *fclu* but includes also mean-ordered partial distance search which speeds up the algorithm in case of data with high real dimensionality, and in case of low intrinsic dimensionality. It applies only to vector data.

genie: The genie algorithm from [21]. Graph based agglomerative clustering algorithm that uses a minimum spanning tree and gini-index based cost function.

TSPg: The proposed method[3]. The number after the name signifies how many TSP paths were used. For example, TSPg-5 used five.

5.3. Quality

Clustering quality is measured against ground truth using two metrics: normalized mutual information (NMI) [22] and centroid index (CI) [23]. The NMI values are within the range [0,1], where 1.0 indicates correct clustering. The CI value represents the number of incorrectly detected clusters so that a value of 0 indicates a correct clustering. The CI gives an intuitive cluster-level overview of the correctness of the clustering, whereas NMI provides a more fine-grained measurement.

6. Results

The results for the more challenging datasets (RC- and Cntr-variants) are shown in Figure 5. A summary of the results is shown in Table 2, which includes selected cases from the RC- and Cntr-sets as well as results for the easier basic benchmark sets (s1-B2).

The basic benchmark results (s1-B2 in Table 2) show only small differences between the tested algorithms. All algorithms can solve (CI=0) almost all these datasets. The genie algorithm has slightly better NMI results than others for most of these datasets, although it fails in case of the unbalance dataset.

There are larger differences in case of the more challenging datasets (RC- and Cntr-variants). The fclu-ward and TSPg-100 methods provide equally good results for all datasets although TSPg-100 is much faster for the larger datasets. The speedup factor was 243372/9423 = 26:1 for the largest case (N=1,280,000) of the RC-N sets. As can be seen in Figure 5 for RC-N and Countries-N cases, TSPg scales much better than fclu-ward, so for larger datasets the speedup factor would increase further. The other methods (fclu-avg and genie) had significantly worse results for these sets.

For the Countries-N string datasets a smaller number of TSP solutions was enough to provide good results and the TSPg-10 variant had equal (NMI within 0.01) results with fclu-ward, but with a speedup factor 17:1. For the case of string data we could not do a comparison with larger datasets because the fclu-ward method requires $O(N^2)$ memory from the use of a distance matrix. The algorithm consumed 23.7GB memory for the largest N=76,800 dataset. In comparison, the TSPg-20 variant consumed only 720MB (3%) of memory for the same dataset. The numerical datasets did not have the same problem with memory usage as the fastcluster package had a memory efficient implementation for the case of Euclidean data which did not require the full distance matrix. The pnn-mps is the fastest exact Ward method for the 2D datasets of the basic benchmark. For the high-dimensional datasets the MPS speedup method does not work and pnn-mps is roughly 3-4 times slower than fclu-ward.

If fast running time is more important than quality, the TSPg-5 variant provides good results (mean NMI 0.89 vs. 0.91) compared to fclu-ward much faster. For the RC-N case the speedup factor was 614:1 for a moderate quality reduction of NMI from 0.90 to 0.80. For easier datasets, such as RC-var with variance=0.3, there were no difference in quality compared to fclu-ward.

7. Conclusions

We proposed a new clustering method that approximates Ward's method while providing results several orders of magnitude faster. Unlike other approaches to speed up Ward's method, it also maintains its generality and can be applied to any type of data for which a distance function can be formulated.

A key part of the method is a structure called TSP-graph which can be constructed in $O(N \log N)$ time. The TSP-graph has many potential applications in addition to speeding up Ward's method. For example, it could be used as an index structure for nearest neighbor search in the same way as kNN graph has been used [24]. It could also be used to improve other graph-based clustering algorithms [21]. The single TSP solutions could be used as a fast way to sort objects like images or strings while preserving their proximity.

Acknowledgments

This study is a part of the consortium project "Improving the Information Base and Optimising Service Solutions to Support Social Welfare and Healthcare Reform (IMPRO)". The research was funded by the Strategic Research Council at the Academy of Finland, funding decision numbers 336325 and 336330.

References

[1] J. H. Ward Jr, Hierarchical grouping to optimize an objective function, Journal of the American statistical association, 58 (1963), pp. 236—244.
[2] L. H. Park, J. Yu, H.-K. Kang, T. Lee, and T. Kwon, Birds of a feature: Intrafamily clustering for version identification of packed malware, IEEE Systems Journal, 14 (2020), pp. 4545—4556.
[3] S. Sadhukhan, H. Root-Gutteridge, and B. Habib, Identifying unknown indian wolves by their distinctive howls: its potential as a non-invasive survey method, Scientific reports, 11 (2021), pp. 1—13.
[4] O. Virmajoki, P. Fränti, and T. Kaukoranta, Practical methods for speeding-up the pnn method, Optical Engineering, 40 (2001), pp. 2495—2504.
[5] T. Strauss and M. J. von Maltitz, Generalising ward's method for use with manhattan distances, PloS one, 12 (2017), p. e0168288.
[6] D. Wishart, 256. note: An algorithm for hierarchical classifications, Biometrics, (1969), pp. 165—170.
[7] C. Ambroise, A. Dehman, P. Neuvial, G. Rigaill, and N. Vialaneix, Adjacency-constrained hierarchical clustering of a band similarity matrix

[2] http://danifold.net/fastcluster.html

[3] https://github.com/uef-machine-learning/tspgclu

with application to genomics, Algorithms for Molecular Biology, 14 (2019), pp. 1—14.
[8] D. Müllner, Modern hierarchical, agglomerative clustering algorithms, arXiv preprint arXiv:1109.2378, (2011).
[9] P. Fränti, T. Kaukoranta, D.-F. Shen, and K.-S. Chang, Fast and memory efficient implementation of the exact pnn, IEEE transactions on image processing, 9 (2000), pp. 773—777.
[10] T. Kaukoranta, P. Fränti, and O. Nevalainen, Vector quantization by lazy pairwise nearest neighbor method, Optical Engineering, 38 (1999), pp. 1862—1868.
[11] P. Fränti, O. Virmajoki, and V. Hautamaki, Fast agglomerative clustering using a k-nearest neighbor graph, IEEE Transactions on Pattern Analysis and Machine Intelligence, 28 (2006), pp. 1875—1881.
[12] A. Bouguettaya, Q. Yu, X. Liu, X. Zhou, and A. Song, Efficient agglomerative hierarchical clustering, Expert Systems with Applications, 42 (2015), pp. 2785—2797.
[13] G. N. Lance and W. T. Williams, A general theory of classificatory sorting strategies: 1. hierarchical systems, The computer journal, 9 (1967), pp. 373—380.
[14] V. Batagelj, Generalized ward and related clustering problems, Classification and related methods of data analysis, (1988), pp. 67—74.
[15] J. Shanbehzadeh and P. O. Ogunbona, On the computational complexity of the lbg and pnn algorithms, IEEE Transactions on Image Processing, 6 (1997), pp. 614—616.
[16] T. Kurita, An efficient agglomerative clustering algorithm using a heap, Pattern Recognition, 24 (1991), pp. 205—209.
[17] F. Murtagh, Multidimensional clustering algorithms, Compstat lectures, 1985.
[18] D. Müllner, fastcluster: Fast hierarchical, agglomerative clustering routines for r and python, Journal of Statistical Software, 53 (2013), pp. 1—18.
[19] W. H. Equitz, A new vector quantization clustering algorithm, IEEE transactions on acoustics, speech, and signal processing, 37 (1989), pp. 1568—1575.
[20] P. Fränti and S. Sieranoja, K-means properties on six clustering benchmark datasets, Applied Intelligence, 48 (2018), pp. 4743—4759.
[21] M. Gagolewski, M. Bartoszuk, and A. Cena, Genie: A new, fast, and outlier-resistant hierarchical clustering algorithm, Information Sciences, 363 (2016), pp. 8—23.
[22] L. Danon, A. Diaz-Guilera, J. Duch, and A. Arenas, Comparing community structure identification, Journal of Statistical Mechanics: Theory and Experiment, 2005 (2005), p. P09008.
[23] P. Fränti and M. Rezaei, Generalizing centroid index to different clustering models, in Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR), Springer, 2016, pp. 285—296.
[24] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, Fast approximate nearest-neighbor search with k-nearest neighbor graph, in IJCAI Proceedings-International Joint Conference on Artificial Intelligence, vol. 22, 2011, p. 1312.
[25] C. Bachoc, F. Vallentin, New upper bounds for kissing numbers from semidefinite programming, Journal of the American Mathematical Society 21(3), 909–924, 2008.
[26] A. Flexer, J. Stevens, Mutual proximity graphs for improved reachability in music recommendation , Journal of new music research 47(1), Taylor & Francis, 17–28, 2018
[27] J.Z.C. Lai, T-J Huang, "An agglomerative clustering algorithm using a dynamic *k*-nearest-neighbor list", *Information Sciences*, 181 (9), 1722-1734, May 2011.

Table 2. Results for the basic benchmark sets and selected cases from the countries and random clusters sets. From the Cntr-N and RC-N sets we selected the largest cases (N=76,800 and N=128,000), from the RC-v set the case with variance=0.4, and from the Cntr-v sets the case with transformation=44%. The highest NMI values are highlighted with bold. The smallest time values that achieved equivalent results (within 0.01 NMI) with exact ward implementation are also highlighted.

NMI

	s1	s2	s3	s4	unb	a1	a2	a3	B1	B2	RC-v	RC-N	Cntr-v	Cntr-N	mean
TSPg-2	0.98	0.92	0.77	0.70	**1.00**	0.97	0.97	**0.98**	0.92	**1.00**	0.62	0.62	0.69	0.79	0.85
TSPg-5	0.98	**0.93**	0.77	0.70	**1.00**	0.96	0.97	**0.98**	0.92	**1.00**	0.84	0.80	0.79	0.86	0.89
TSPg-10	0.98	**0.93**	0.77	0.70	**1.00**	0.96	0.97	**0.98**	0.92	**1.00**	0.89	0.85	0.81	**0.88**	0.90
TSPg-20	0.98	**0.93**	0.77	0.70	**1.00**	0.96	0.97	**0.98**	0.92	**1.00**	0.91	0.87	**0.82**	**0.88**	**0.91**
TSPg-100	0.98	**0.93**	0.77	0.69	**1.00**	0.96	0.97	**0.98**	0.92	**1.00**	**0.93**	**0.90**	0.81	**0.88**	**0.91**
pnn-mps	0.98	0.93	0.78	0.69	**1.00**	0.96	0.97	**0.98**	0.92	**1.00**	**0.93**	-	-	-	-
fclu-ward	0.98	**0.93**	0.77	0.69	**1.00**	0.96	0.97	**0.98**	0.92	**1.00**	**0.93**	**0.90**	0.81	0.87	**0.91**
fclu-avg	0.98	**0.93**	0.75	0.67	**1.00**	0.96	0.97	**0.98**	0.92	**1.00**	0.77	-	0.05	0.05	-
genie	**0.99**	**0.93**	**0.78**	**0.72**	0.75	**0.97**	**0.98**	**0.98**	**0.94**	**1.00**	0.75	0.02	0.64	0.61	0.79

CI

	s1	s2	s3	s4	unb	a1	a2	a3	B1	B2	RC-v	RC-N	Cntr-v	Cntr-N	mean
TSPg-2	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0.24
TSPg-5	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0.16
TSPg-10	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0.20
TSPg-20	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0.21
TSPg-100	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0.23
pnn-mps	0	0	0	1	0	0	0	0	0	0	0	-	-	-	-
fclu-ward	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0.21
fclu-avg	0	0	3	4	0	0	0	0	1	0	67	-	46	46	-
genie	0	0	0	0	4	0	0	0	0	0	31	71	14	17	9.79

Time (seconds)

	s1	s2	s3	s4	unb	a1	a2	a3	B1	B2	RC-v	RC-N	Cntr-v	Cntr-N
TSPg-2	**0.03**	**0.03**	**0.03**	**0.03**	**0.04**	**0.01**	**0.03**	**0.05**	**1**	**1**	12	120	14	21
TSPg-5	0.06	0.06	0.06	0.06	0.08	0.03	0.06	0.10	3	2	40	396	25	45
TSPg-10	0.10	0.10	0.10	0.09	0.15	0.04	0.11	0.17	4	4	100	857	**41**	**75**
TSPg-20	0.19	0.19	0.19	0.18	0.28	0.08	0.20	0.32	8	7	213	1797	81	126
TSPg-100	0.90	0.90	0.91	0.89	1.32	0.47	0.90	1.42	33	27	**1004**	**9423**	385	646
pnn-mps	0.08	0.08	0.08	0.08	0.13	0.03	0.09	0.17	31	28	25356	-	-	-
fclu-ward	0.13	0.18	0.13	0.14	0.25	0.10	0.20	0.27	57	57	6906	243372	366	1306
fclu-avg	0.42	0.47	0.42	0.47	0.61	0.17	0.46	0.72	257	196	1978	-	298	1229
genie	0.13	0.13	0.13	0.13	0.20	0.05	0.14	0.28	49	49	2153	73099	60	107

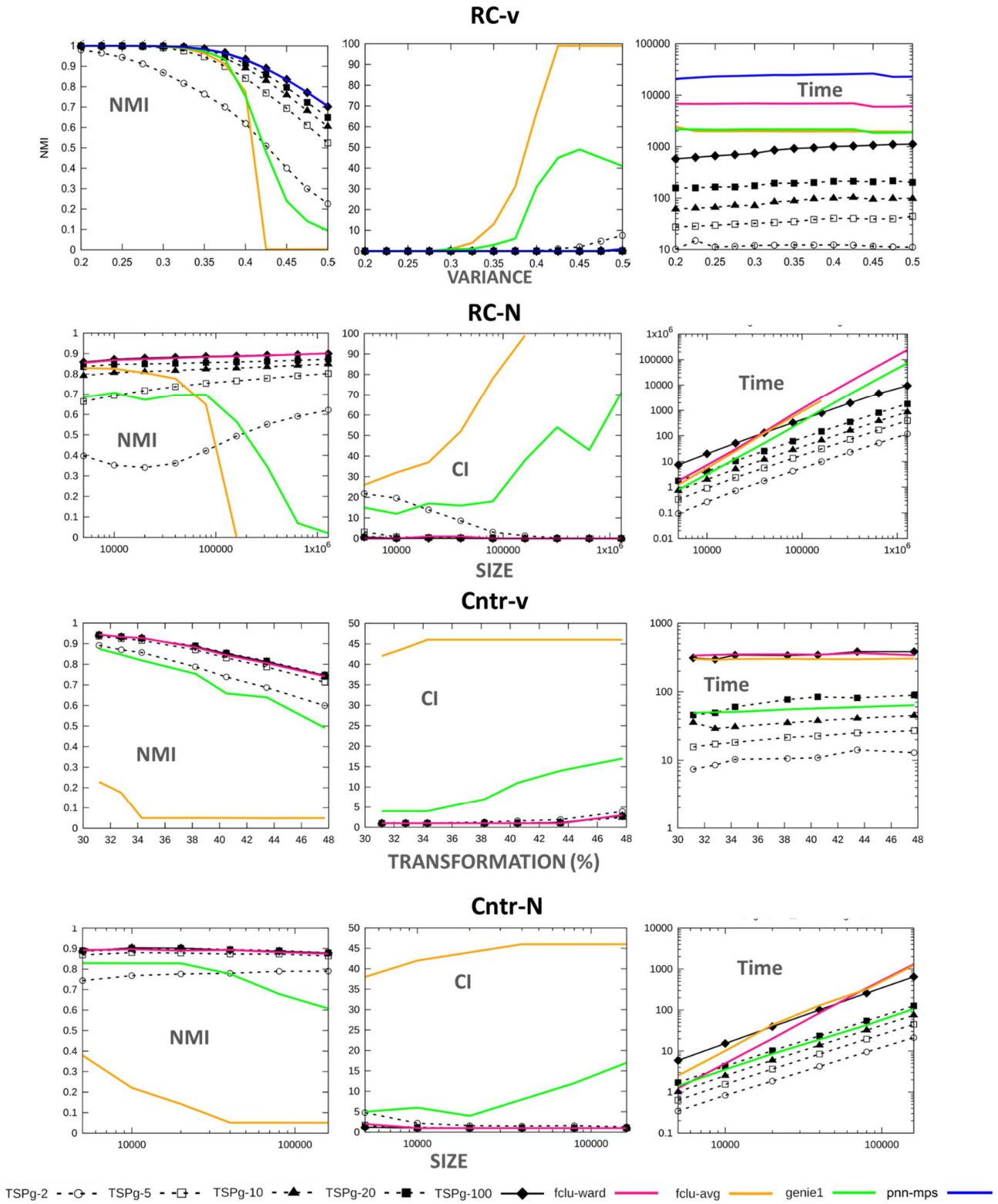

Figure 5. Results for the datasets of random gaussian clusters (RC-v and RC-N) and country names (Cntr-v and Cntr-N).