<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>SparseHC: a memory-efficient online hierarchical clustering algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Nguyen, Thuy-Diem; Schmidt, Bertil; Kwoh, Chee-Keong</td>
</tr>
<tr>
<td><strong>Date</strong></td>
<td>2014</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10220/20325">http://hdl.handle.net/10220/20325</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>© 2014 The Author(s). This paper was published in Procedia Computer Science and is made available as an electronic reprint (preprint) with permission of the Author(s). The paper can be found at the following official DOI: <a href="http://dx.doi.org/10.1016/j.procs.2014.05.001">http://dx.doi.org/10.1016/j.procs.2014.05.001</a>. One print or electronic copy may be made for personal use only. Systematic or multiple reproduction, distribution to multiple locations via electronic or other means, duplication of any material in this paper for a fee or for commercial purposes, or modification of the content of the paper is prohibited and is subject to penalties under law.</td>
</tr>
</tbody>
</table>
SparseHC: a memory-efficient online hierarchical clustering algorithm

Thuy-Diem Nguyen¹, Bertil Schmidt², and Chee-Keong Kwoh³

¹ School of Computer Engineering, Nanyang Technological University, Singapore
thuy1@e.ntu.edu.sg
² Institut für Informatik, Johannes Gutenberg University, Mainz, Germany
bertil.schmidt@uni-mainz.de
³ School of Computer Engineering, Nanyang Technological University, Singapore
asckkwoh@ntu.edu.sg

Abstract
Computing a hierarchical clustering of objects from a pairwise distance matrix is an important algorithmic kernel in computational science. Since the storage of this matrix requires quadratic space with respect to the number of objects, the design of memory-efficient approaches is of high importance to this research area. In this paper, we address this problem by presenting a memory-efficient online hierarchical clustering algorithm called SparseHC. SparseHC scans a sorted and possibly sparse distance matrix chunk-by-chunk. Meanwhile, a dendrogram is built by merging cluster pairs as and when the distance between them is determined to be the smallest among all remaining cluster pairs. The key insight used is that for finding the cluster pair with the smallest distance, it is unnecessary to complete the computation of all cluster pairwise distances. Partial information can be utilized to calculate a lower bound on cluster pairwise distances that are subsequently used for cluster distance comparison. Our experimental results show that SparseHC achieves a linear empirical memory complexity, which is a significant improvement compared to existing algorithms.

Keywords: hierarchical clustering, memory-efficient clustering, sparse matrix, online algorithms

1 Introduction
Clustering is an important unsupervised machine learning technique to group similar objects in order to uncover the inherent structure of a given dataset. Depending on the output, clustering algorithms are broadly divided into two main categories: hierarchical clustering and partitional (or flat) clustering [2, 12, 26]. The structured output produced by hierarchical clustering algorithms is often more informative than the unstructured set of clusters returned by partitional clustering algorithms [16, 24]. Thus, hierarchical clustering is a crucial data analysis tool in many fields including computational biology and social sciences [9]. Nonetheless, the quadratic time and especially the quadratic memory complexity have limited the use of hierarchical clustering software to rather small datasets [24]. Since many areas of computational science face a
Table 1: The parameters of the Lance-Williams recurrence formula for 7 popular linkage schemes

<table>
<thead>
<tr>
<th>Linkage</th>
<th>$\alpha_1$</th>
<th>$\alpha_2$</th>
<th>$\beta$</th>
<th>$\gamma$</th>
<th>Alternative formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>-0.5</td>
<td>$d_{ij} = \min_{x \in C_i, y \in C_j} d_{xy}$</td>
</tr>
<tr>
<td>Complete</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>0.5</td>
<td>$d_{ij} = \max_{x \in C_i, y \in C_j} d_{xy}$</td>
</tr>
<tr>
<td>Average</td>
<td>$\frac{</td>
<td>C_i</td>
<td>}{</td>
<td>C_i</td>
<td>+</td>
</tr>
<tr>
<td>Weighted</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Centroid</td>
<td>$\frac{</td>
<td>C_i</td>
<td>}{</td>
<td>C_i</td>
<td>+</td>
</tr>
<tr>
<td>Median</td>
<td>0.5</td>
<td>0.5</td>
<td>-0.25</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Ward</td>
<td>$\frac{</td>
<td>C_i</td>
<td>+</td>
<td>C_m</td>
<td>}{</td>
</tr>
</tbody>
</table>

Hierarchical clustering can be divided into two categories: the agglomerative “bottom-up” approach and the divisive “top-down” approach [24]. We focus on the former category: agglomerative hierarchical clustering (AHC). AHC algorithms can be characterized as sequential, agglomerative, hierarchical, and non-overlapping [7, 23]. In AHC algorithms, objects or data points are first treated as singletons and subsequently merged one pair of clusters at a time until there is only one cluster left. There are seven commonly used linkage schemes: single, complete, average (UPGMA), weighted (WPGMA), centroid (UPGMC), median (WPGMC) and Ward’s method. The properties of each scheme are discussed in [8]. The merging criteria used by all these schemes can be neatly represented with the recurrence formula by Lance and Williams [14]. Given that two clusters $C_i$ and $C_j$ have previously been merged into cluster $C_k$, the distance between cluster $C_k$ and any unmerged cluster $C_m$ is defined as:

$$d_{km} = d(C_i \cup C_j, C_m) = \alpha_1 d_{im} + \alpha_2 d_{jm} + \beta d_{ij} + \gamma |d_{im} - d_{jm}|$$

The specific parameters for each scheme are defined in Table 1.

Depending on the input data, AHC algorithms can be divided into the “stored data approach” and the “stored matrix approach” [1, 18]. The stored data approach requires the recalculation of pairwise distance values for each merging step. Since only data points are stored in the main memory, algorithms in this approach can achieve $O(N)$ space complexity often at the expense of $O(N^3)$ time complexity [19], where $N$ is the number of input data points. One notable algorithm in the stored data approach is the nearest-neighbor chain algorithm, which achieves $O(N)$ space complexity and $O(N^2)$ time complexity for the Ward’s method linkage scheme. However, this algorithm is not applicable to the centroid and median linkage schemes because these schemes do not fulfill the required reducibility criterion i.e. $d(C_i \cup C_j, C_m) \geq \min(d(C_i, C_m), d(C_j, C_m))$ [18, 19]. For the single-, complete- and average-linkage schemes, this algorithm requires $O(N^2)$ space and time complexity [10]. On the contrary, in the stored matrix approach an all-against-all pairwise distance matrix of size $N^2$ is first computed and then used for clustering. As a result, this approach requires $O(N^2)$ time and memory complexity [26].

To overcome the low memory efficiency of classical AHC algorithms, new techniques perform either data reduction by random sampling (e.g. data sampling and partitioning in CURE [11])
or data summarization by using a new data structure to represent the original data (e.g. the CF tree in BIRCH [27]). Although these algorithms have linear memory complexity [26], the dendrograms produced by these algorithms are indeterministic and are dissimilar those produced by standard AHC tools because of the random procedures being used.

In this paper, we focus on reducing the primary memory consumption of the AHC stored matrix approach. We introduce SparseHC, a general-purpose memory-efficient AHC algorithm for single-, complete- and average-linkage schemes. SparseHC is an online algorithm. Borodin and El-Yaniv [5] defined online algorithms as algorithms that focus on scenarios where “the input is given one piece at a time and upon receiving an input, the algorithm must take an irreversible action without the knowledge of future inputs”. Because online algorithms only require partial input in the main memory for processing, they are often used to target problems with high space complexity. To our knowledge, there are only a few existing online hierarchical clustering algorithms for the stored matrix approach including MCUPGMA [15] for the average scheme and ESPRIT hcluster [25] for single and complete schemes.

SparseHC employs a similar strategy as in MCUPGMA and hcluster where the input distance matrix is first sorted and then processed in a chunk-by-chunk manner. SparseHC incorporates two new techniques in order to achieve significantly better performance:

1. Compression of the information in the currently loaded chunk of the input matrix into the most compact form.
2. Usage of an efficient graph representation to store unmerged cluster connections, which allows constant access to these connections for faster speed.

2 Background and Concepts

SparseHC and other online AHC algorithms work based on the observation that once the values of an input distance matrix are sorted in ascending order and loaded chunk-by-chunk from the top, the merge order and the dendrogram distances can be accurately determined using only the loaded part i.e. without any knowledge about the unseen portion.

Figure 1: Illustration of the dendrogram and the corresponding complete binary tree produced by applying average-linkage clustering to a full distance matrix computed from 10 data points.

SparseHC takes a sorted distance matrix $D$ as input and iteratively builds a dendrogram
from reading only a part of $D$ in each iteration step as shown in Figure 1. Depending on the available main memory, a sequence of values $0 = \lambda_0 < \lambda_1 < \ldots < \lambda_T = \theta$ is built on-the-fly. In each iteration step $1 \leq t \leq T$, all distances $d_{xy}$ with $\lambda_{t-1} \leq d_{xy} < \lambda_t$ are read from $D$. Starting from the a tree consisting of only $N$ leaves where a leaf node $i$ ($1 \leq i \leq N$) represents the singleton cluster $C_i = \{i\}$, a binary tree (which is the dendrogram) is built from bottom up. Since only two clusters are merged at a time, the full binary tree has a height of $N - 1$ and consists of $2N - 1$ nodes (see Figure 1).

In offline AHC algorithms, $D$ has to be a full pairwise distance matrix. However, in online AHC algorithms such as SparseHC, $D$ can be either full or sparse. A sparse distance matrix $D_\theta$ uses a predefined distance cutoff $\theta$ ($0 \leq \theta < 1$) and stores only distance values up to $\theta$ ($0 \leq d_{xy} \leq \theta, \forall d_{xy} \in D$). For sparse matrix clustering, the output dendrogram has a height in the range of $[1, N - 1]$ and a size in the range of $[N, 2N - 1]$ as shown in Figure 2.

![Dendrogram and Incomplete Binary Tree](image)

**Figure 2:** Illustration of the dendrogram and the corresponding incomplete binary tree produced by applying average-linkage clustering to a sparse distance matrix computed from 10 data points with a distance cutoff $\theta = 0.4$.

The input to SparseHC is a sorted full or sparse distance matrix stored in a list of tuples $(i, j, d_{ij})$ format (similar to the MATLAB sparse matrix external format: [http://www.mathworks.com/help/matlab/ref/spconvert.html](http://www.mathworks.com/help/matlab/ref/spconvert.html)). The maximum element of a full matrix is 1.0 while that of a partial matrix is a pre-defined distance cutoff $\theta < 1.0$. The ability to process sparse distance matrices is particularly useful in applications like taxonomic studies in bioinformatics [4, 25] where only the lower part of the final dendrogram is of interest. In these situations, runtime and memory usage are further reduced depending on the sparsity of the input matrix. The memory efficiency and ability of SparseHC to process sparse matrices come at the cost of pre-sorting the input matrices. Nonetheless, the memory performance of SparseHC is not affected if an external merge sort algorithm [13] is used for the sorting stage.

Similar to offline AHC algorithms, during the clustering process, SparseHC needs to store all the connections amongst unmerged clusters to figure out which cluster pair will be merged next. However, same as other online AHC algorithms, SparseHC only stores the connections amongst active clusters. A cluster pair is called *active* in iteration step $t$ when (1) both clusters do not have a parent and (2) at least one distance value between the member data points has been read from the input file during the first $t$ iteration steps. We observe that active clusters contribute to only a small subset of unmerged clusters. The memory efficiency of online AHC algorithms is determined by their ability to store active connections in a compact way.
Table 2: The time and memory complexity of different graph representations. We derive the adjacency map from the adjacency list to facilitate edge operations required by SparseHC.

<table>
<thead>
<tr>
<th>Representation</th>
<th>Storage</th>
<th>Add edge</th>
<th>Remove Edge</th>
<th>Query edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incidence matrix</td>
<td>(O(</td>
<td>V</td>
<td>+</td>
<td>E</td>
</tr>
<tr>
<td>Adjacency matrix</td>
<td>(O(</td>
<td>V</td>
<td>^2))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Incidence list</td>
<td>(O(</td>
<td>V</td>
<td>+</td>
<td>E</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>(O(</td>
<td>V</td>
<td>+</td>
<td>E</td>
</tr>
<tr>
<td>Adjacency map</td>
<td>(O(</td>
<td>V</td>
<td>+</td>
<td>E</td>
</tr>
</tbody>
</table>

Table 3: Distance \(d_{ij}\) between cluster \(C_i\) and \(C_j\) for clustering sparse matrices

<table>
<thead>
<tr>
<th>Linkage</th>
<th>Edge definition</th>
<th>Cluster distance</th>
<th>Complete condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>(e_{ij}^{(t)} = ())</td>
<td>(d_{ij}^{(t)} = 1.0)</td>
<td>(n_{ij}^{(t)} = 1)</td>
</tr>
<tr>
<td>Complete</td>
<td>(e_{ij}^{(t)} = (n_{ij}^{(t)}))</td>
<td>(d_{ij}^{(t)} = d_{xy})</td>
<td>(n_{ij}^{(t)} =</td>
</tr>
<tr>
<td>Average</td>
<td>(e_{ij}^{(t)} = (s_{ij}^{(t)}, n_{ij}^{(t)}))</td>
<td>(d_{ij}^{(t)} = \frac{s_{ij}^{(t)} + \lambda (</td>
<td>C_i</td>
</tr>
</tbody>
</table>

In SparseHC, we use an undirected weighted graph to model the connections amongst active cluster pairs. This graph consists of a set of vertices \(V\) and a set of edges \(E\). The vertices are the nodes of the binary tree i.e. \(V = \{C_1, C_2, \ldots, C_{2N-1}\}\). SparseHC uses a fixed size array to store all possible vertices, hence allowing \(O(1)\) vertex query and update. The undirected weighted edges are the active connections amongst the clusters.

Graphs are typically implemented using an adjacency matrix, an adjacency list, an incidence matrix or an incidence list [21]. The time and space complexity of each representation are shown in Table 2. To facilitate its cluster merging process, SparseHC prefers a graph representation that requires minimum storage for the graph and allows constant time to perform edge insertion, edge deletion, and edge update. Therefore, we have modified the standard adjacency list to assist these operations. We call this graph representation the adjacency (hash) map. The adjacency map is a collection of unordered hash maps, one for each vertex of the graph. Each hash map records the set of neighbors of its vertex using the neighbor vertex identification number as the key. Because of this adjacency map representation, SparseHC can use \(O(|V|+|E|)\) space to store all the clusters and their active connections. Moreover, these connections can be accessed and updated in \(O(1)\) time.

3 SparseHC

3.1 Algorithm

The definition of the edge \(e_{ij}^{(t)}\) between two active clusters \(C_i\) and \(C_j\) in iteration step \(t\) is defined in Table 3 depending on the clustering scheme. \(d_{ij}^{(t)}\) is the minimum possible distance between \(C_i\) and \(C_j\) and is computed according to Table 3. \(s_{ij}^{(t)} (n_{ij}^{(t)})\) is the sum (number) of distance values between any member of \(C_i\) to any member of \(C_j\) that has been read from the input file so far. \(\lambda\) is the maximum distance value loaded from the input matrix so far.

In each iteration step \(t\), active edges are partitioned into two sets: a set of complete edges \(K^{(t)}\) and a set of incomplete edges \(I^{(t)}\) (both sets are stored in the adjacency map). A complete edge is a connection between two active clusters that are ready to be merged. An incomplete edge is
a connection between two active clusters that are yet to be merged. For complete- and average-
linkage schemes, an edge is complete when $n_{ij}^{(t)} = |C_i||C_j|$. Otherwise, when $n_{ij}^{(t)} < |C_i||C_j|$, the
edge is considered incomplete. For single-linkage scheme, an edge is complete when $n_{ij}^{(t)} = 1$
i.e. the connection between two clusters is complete as soon as the first distance value between
any member reads has been read from the input.

Let $\min(I^{(t)})$ (min($K^{(t)}$)) denote the smallest distance value in $I^{(t)}$ ($K^{(t)}$). The high-level
description of the SparseHC algorithm in each iteration $t$ ($1 \leq t \leq T$) consists of three steps:

1. Read the distance values $d_{xy}$ from matrix $D$ in ascending order until the adjacency map
   is full and determine the value $\lambda_t$.
2. Update/create the edges for all active cluster pairs with the new distances and partition
   them into $I^{(t)}$ and $K^{(t)}$.
3. Retrieve the edge $e_{ij}^{(t)}$ for which $d_{ij}^{(t)} = \min(K^{(t)}) \leq \min(I^{(t)})$. Merge the cluster pair $C_i$
   and $C_j$ into cluster $C_k$. Delete $e_{ij}^{(t)}$ from $K^{(t)}$ and combine existing edges to either cluster
   $C_i$ or $C_j$ into new edges to cluster $C_k$. Repeat until $\min(K^{(t)}) > \min(I^{(t)})$.

Algorithm 1 shows the details of SparseHC.

3.2 Correctness

To show the dendrogram produced by SparseHC is correct, we need to prove that up to the
distance cutoff $\theta$ both the merge distance values and the merge order are preserved.

_Merge distances:_ Let $d_{ij}^{(t)}$ be the merge distance between two clusters $C_i$ and $C_j$ assuming
that they are being merged by SparseHC in an iteration $t$. Let $d_{ij}$ be the merge distance
between $C_i$ and $C_j$ produced by a traditional AHC algorithm. We need to show that $d_{ij}^{(t)} = d_{ij}$.

Indeed, when $C_i$ and $C_j$ are merged by SparseHC, the edge $e_{ij}^{(t)}$ is complete. By definitions of
$d_{ij}$ in Table 1 and in $d_{ij}^{(t)}$ when $e_{ij}^{(t)}$ is complete in Table 3, it holds that $d_{ij}^{(t)} = d_{ij}$. Therefore,
the merge distance values are preserved.

_Merge order:_ To prove that SparseHC preserves the merge order, we show that if $C_i$ and $C_j$
are merged before $C_k$ and $C_m$, then $d_{ij} \leq d_{km}$. At the time when $C_i$ and $C_j$ are being merged
in an iteration $t$, we have $d_{ij}^{(t)} = d_{ij} = \min(K^{(t)})$. In step $t$, after $C_i$ and $C_j$ are merged, the
status of the edge $e_{km}^{(t)}$ is one of the followings:

1. $e_{km}^{(t)}$ is active and complete $\Rightarrow e_{km}^{(t)} \in K^{(t)}$. For a complete edge, it holds that $d_{km}^{(t)} = d_{km}$.

   Besides, $e_{km}^{(t)} \in K^{(t)} \Rightarrow \min(K^{(t)}) \leq d_{km}^{(t)}$. Therefore, $d_{ij} \leq d_{km}$
2. $e_{km}^{(t)}$ is active and incomplete $\Rightarrow e_{km}^{(t)} \in I^{(t)}$. For an incomplete edge, it holds that $d_{km}^{(t)} \leq d_{km}$.

   In SparseHC, it always holds that $\min(K^{(t)}) \leq \min(I^{(t)}) \Rightarrow d_{ij} \leq \min(I^{(t)})$.

   Besides, $e_{km}^{(t)} \in I^{(t)} \Rightarrow \min(I^{(t)}) \leq d_{km}^{(t)}$. Therefore, $d_{ij} \leq d_{km}$
3. $e_{km}^{(t)}$ is inactive $\Rightarrow e_{km}^{(t)} \notin \{K^{(t)} \cup I^{(t)}\}$. For an inactive edge, it holds that $\lambda_t < d_{km}^{(t)} \leq d_{km}$.

   Since $C_i$ and $C_j$ have been merged in iteration $t$, $d_{ij} \leq \lambda_t$. Therefore, $d_{ij} < d_{km}$

For all cases, we have $d_{ij}^{(t)} = d_{ij}$ and $d_{ij} \leq d_{km}$ i.e. both the merge distances and the merge
order are preserved in SparseHC.

3.3 Memory efficiency

While standard offline AHC algorithms store all the connections amongst unmerged clusters in
memory (i.e. $|C_i| \times |C_j|$ values for a cluster pair $C_i$, $C_j$), SparseHC uses at most two values
Algorithm 1 SparseHC algorithm for a sorted input matrix $D$ from $N$ data points stored as a list of tuples $(x, y, d_{xy})$.

$C_i \leftarrow \{i\} \quad \forall i = 1, \ldots, N$

$E$.max_size $\leftarrow$ $N$ \{E is the adjacency map $E = K \cup I$\}

$k \leftarrow N; t \leftarrow 0; \lambda_0 \leftarrow 0$ \{initialize cluster id $k$, iteration $t$, distance threshold $\lambda$\}

while $D \neq \emptyset$ do

$t \leftarrow t + 1$

while $D \neq \emptyset$ and $E$.size $\leq$ $E$.max.size do

$d_{xy} \leftarrow D$.get_next(); $D = D \setminus \{d_{xy}\}$

$C_i \leftarrow C_i$.get_ancestor(); $C_j \leftarrow C_j$.get_ancestor()

$e_{ij}^{(t)}$.update($d_{xy}$) \{create $e_{ij}^{(t)}$ if it does not exist\}

compute $d_{ij}^{(t)}$ \{use the cluster distance formula in Table 3\}

if $e_{ij}^{(t)}$ is complete then

$C_i$.minK $\leftarrow$ min($C_i$.minK, $d_{ij}^{(t)}$); $C_i$.merge.candidate $\leftarrow$ $C_j$

else

$C_i$.minI $\leftarrow$ min($C_i$.minI, $d_{ij}^{(t)}$)

end if

end while

$\lambda_t \leftarrow d_{xy}$ \{$\lambda_t$ is the largest distance in an iteration\}

while $d_{ij} = \min(K^{(t)}) \leq \min(I^{(t)})$ and $k \leq 2N - 1$ do

$k \leftarrow k + 1$; $C_k \leftarrow C_i \cup C_j$ \{merge clusters $C_i$ and $C_j$ into cluster $C_k$\}

for all $C_m$ such that $e_{im}^{(t)} \in E \cup e_{jm}^{(t)} \in E$ do

$e_{km}^{(t)} \leftarrow$ merge($e_{im}^{(t)}, e_{jm}^{(t)}$) \{$s_{km}^{(t)} \leftarrow s_{im}^{(t)} + s_{jm}^{(t)}$, $n_{km}^{(t)} \leftarrow n_{im}^{(t)} + n_{jm}^{(t)}$\}

$E = E \cup \{e_{km}^{(t)}\} \setminus \{e_{im}^{(t)}, e_{jm}^{(t)}, e_{ij}^{(t)}\}$

compute $d_{km}^{(t)}$ \{use the cluster distance formula in Table 3\}

if $e_{km}^{(t)}$ is complete then

$C_k$.minK $\leftarrow$ min($C_k$.minK, $d_{km}^{(t)}$); $C_k$.merge.candidate $\leftarrow$ $C_m$

else

$C_k$.minI $\leftarrow$ min($C_k$.minI, $d_{km}^{(t)}$)

end if

end for

end while

if $E$.size $\geq$ $E$.max.size then

$E$.max.size $\leftarrow$ $2 \times E$.max.size \{dynamically increase the adjacency map size\}

if $E$.size $\geq$ RAM.size then

return partial result \{when the memory limit is reached\}

end if

end if

end while

return full result
per cluster pair: the number of connections $n_{ij}^{(t)}$ and the sum of distances $s_{ij}^{(t)}$ (see Table 3). Specifically, SparseHC maintains only one value per cluster pair ($n_{ij}^{(t)}$) for complete-linkage clustering, two values per pair ($n_{ij}^{(t)}, s_{ij}^{(t)}$) for average-linkage clustering and none for single-linkage clustering.

Compared to offline AHC tools, SparseHC uses less primary memory because of two reasons: (1) SparseHC stores only the information from the currently loaded chunks and (2) It stores a compact version of the seen information: at most two values per active cluster pair.

Compared to existing online AHC tools such as hcluster and MCUPGMA, SparseHC is better because of three reasons. Firstly, SparseHC uses an array of hash maps to store the compact cluster connections. This efficient data structure allows $O(1)$ query, insert and delete, which contributes to the compute efficiency of SparseHC. Secondly, for average-linkage clustering, SparseHC uses two values instead of four values per cluster connection as in MCUPGMA. More importantly, SparseHC dynamically allocates the amount of memory needed and returns partial results if all the available memory is consumed. MCUPGMA and hcluster require the user to specify the amount of memory beforehand and return error if the allocated amount is insufficient. Thirdly, SparseHC supports three linkage types while ESPRIT hcluster supports only single- and complete-linkage clustering and MCUPGMA supports only average-linkage. ESPRIT has another sub-module called aveclust which performs fast average-linkage clustering. However, aveclust is not memory-efficient and still requires quadratic memory complexity. Finally, SparseHC stops after performing $N - 1$ merges. This termination condition is particularly useful for single-linkage clustering where the clustering process converges early.

4 Empirical Results

4.1 Experiment setup

We compare the performance of SparseHC against two offline AHC implementations: MATLAB linkage, fastcluster [17] and two online AHC implementations: ESPRIT hcluster and MCUPGMA. These tools are chosen for their compute and/or memory efficiency as well as the availability of executable source codes.

The experiments in this section are conducted on a 64-bit Linux operating system using a Dell T3500 PC with a quad-core Intel Xeon W3540 2.93 GHz processor and 8GB of RAM. The runtime is measured using the Linux time command and the peak memory usage is measured with the Valgrind Massif profiler [20].

4.2 Empirical complexity

Since online AHC algorithms have a heuristic nature, their theoretical complexity is often hard to estimate. As a result, we use the regression model of space and running time [6] to calculate the empirical complexity [22] instead of the theoretical values to compare the algorithms of interest. Assuming the runtime and memory usage follow the power rule i.e. $f(n) \approx Cn^k$ where $n$ is the input size, the constant factor $C$ and the order $k$ can be estimated using regression on the log-transformed model where $\epsilon$ is the error term:

$$\log f(n) = k \log n + \log C + \epsilon$$

Table 4 reports the average empirical runtime and memory growth of the tested AHC clustering implementations of interest. We use full pre-sorted pairwise Euclidean distance matrices as inputs in this experiment. These matrices are computed from 1000 - 20000 randomly-generated data points. Although the values of $C$ and $k$ in Table 4 are only representative of the performance of these algorithms on the tested random datasets, our results on larger datasets in Table
Table 4: The empirical runtime and memory growth ($f(n) = Cn^k$) of SparseHC versus popular offline and online AHC implementations. This experiment uses 20 matrices computed from 1000 to 20000 data points. Runtime $f_r(n)$ is measured in seconds using the Linux `time` command. Memory usage $f_s(n)$ is measured in megabytes using the Valgrind Massif profiler. The input size $n$ is measured in thousand data points.

<table>
<thead>
<tr>
<th>AHC tool</th>
<th>Single-linkage</th>
<th>Complete-linkage</th>
<th>Average-linkage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SparseHC</td>
<td>$0.003 \times n^{1.855}$</td>
<td>$0.190 \times n^{2.047}$</td>
<td>$0.216 \times n^{2.030}$</td>
</tr>
<tr>
<td>hcluster/aveclust</td>
<td>$0.340 \times n^{2.015}$</td>
<td>$0.378 \times n^{2.000}$</td>
<td>$0.216 \times n^{2.047}$</td>
</tr>
<tr>
<td>MATLAB linkage</td>
<td>$0.352 \times n^{1.996}$</td>
<td>$0.344 \times n^{1.996}$</td>
<td>$0.336 \times n^{2.003}$</td>
</tr>
<tr>
<td>fastcluster</td>
<td>$0.221 \times n^{2.085}$</td>
<td>$0.306 \times n^{1.955}$</td>
<td>$0.236 \times n^{2.073}$</td>
</tr>
<tr>
<td>MCUPGMA</td>
<td>not available</td>
<td>not available</td>
<td>$1.313 \times n^{2.120}$</td>
</tr>
</tbody>
</table>

The empirical memory growth $f_s(n)$:

<table>
<thead>
<tr>
<th>AHC tool</th>
<th>Single-linkage</th>
<th>Complete-linkage</th>
<th>Average-linkage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SparseHC</td>
<td>$0.886 \times n^{0.456}$</td>
<td>$1.272 \times n^{0.848}$</td>
<td>$1.155 \times n^{0.962}$</td>
</tr>
<tr>
<td>hcluster/aveclust</td>
<td>$0.242 \times n^{0.482}$</td>
<td>user-defined</td>
<td>$1.007 \times n^{1.982}$</td>
</tr>
<tr>
<td>MATLAB linkage</td>
<td>$7.674 \times n^{1.998}$</td>
<td>$7.673 \times n^{1.998}$</td>
<td>$7.674 \times n^{1.998}$</td>
</tr>
<tr>
<td>fastcluster</td>
<td>$79.166 \times n^{1.995}$</td>
<td>$78.343 \times n^{2.001}$</td>
<td>$78.336 \times n^{2.001}$</td>
</tr>
<tr>
<td>MCUPGMA</td>
<td>not available</td>
<td>not available</td>
<td>user-defined</td>
</tr>
</tbody>
</table>

6 and on biological sequence datasets in Table 5 further confirm and strengthen the validity of the regression model for evaluating empirical complexity and the estimated values in Table 4. The upper subtable of Table 4 shows that all algorithms have quadratic runtime with $k \approx 2$ as expected. Nevertheless, if we plot these functions in the domain $[0, 10^6]$ data points, we observe that SparseHC is the fastest amongst them. Especially for single-linkage clustering, the constant factor $C$ of SparseHC is two orders of magnitude smaller than other tools. For the complete- and average-linkage schemes, the main reason for the fast runtime of SparseHC is the efficiency of edge operations of the adjacency map data structure. For single-linkage scheme, the significant improvement in speed is due to the edge completion condition ($n_{ij}^{(k)} = 1$). This condition allows two clusters to be merged as soon as the connection between them becomes active, making it unnecessary for SparseHC to store and query active connections of unmerged clusters. Moreover, because of this condition, the merging process for the single-linkage scheme often completes before all values of the input file are loaded, effectively reducing the amount of runtime spent for file input.

The lower subtable of Table 4 shows that offline algorithms have quadratic memory complexity with $k \approx 2$ as anticipated. Python clustering modules such as `fastcluster` or SciPy `cluster` function are less memory-efficient than MATLAB `linkage` since they require additional intermediate data besides the input matrix. On the contrary, the memory usage of SparseHC grows sublinearly/linearly with the input size. SparseHC mainly uses memory to store the adjacency map of unmerged cluster connections. For the “user-defined” cases in Table 4, our experiments show that SparseHC uses less memory than `hcluster` and MCUPGMA. For example, to cluster a 4GB matrix, SparseHC consumes 16MB while `hcluster` uses up 192MB of main memory. Similarly, to cluster a 2.2GB matrix, SparseHC consumes 21MB while MCUPGMA uses up 312MB of main memory. Therefore, SparseHC is the most space-efficient for complete- and average-linkage clustering. For single-linkage, SparseHC and `hcluster` achieve similarly good memory performance. The reasons behind SparseHC memory efficiency are discussed in details in Section 3.3.
Table 5: Using SparseHC, aveclust and MCUPGMA for clustering sparse matrices computed from DNA datasets with sparsity = 50%

<table>
<thead>
<tr>
<th>Number of sequences</th>
<th>Sparse matrix size (in MB)</th>
<th>Runtime (in seconds)</th>
<th>Memory usage (in MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SparseHC</td>
<td>aveclust</td>
<td>MCUPGMA</td>
</tr>
<tr>
<td>10000</td>
<td>483</td>
<td>13.3</td>
<td>15.0</td>
</tr>
<tr>
<td>20000</td>
<td>2035</td>
<td>54.2</td>
<td>67.3</td>
</tr>
<tr>
<td>30000</td>
<td>4706</td>
<td>126.0</td>
<td>174.8</td>
</tr>
<tr>
<td>40000</td>
<td>8415</td>
<td>229.8</td>
<td>321.1</td>
</tr>
</tbody>
</table>

Table 6: The memory efficiency of SparseHC, presented by the ratio \( \frac{\text{matrix size}}{\text{memory usage}} \)

<table>
<thead>
<tr>
<th>Number of data points</th>
<th>Matrix size (in GB)</th>
<th>Memory usage (in MB)</th>
<th>Memory efficiency of SparseHC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
<td>Complete</td>
<td>Average</td>
</tr>
<tr>
<td>50000</td>
<td>14</td>
<td>7.0</td>
<td>30.5</td>
</tr>
<tr>
<td>100000</td>
<td>56</td>
<td>12.2</td>
<td>60.0</td>
</tr>
<tr>
<td>150000</td>
<td>126</td>
<td>17.7</td>
<td>89.6</td>
</tr>
<tr>
<td>200000</td>
<td>224</td>
<td>22.9</td>
<td>119.4</td>
</tr>
</tbody>
</table>

4.3 SparseHC for clustering DNA datasets

To demonstrate the usage of SparseHC for bioinformatics applications, we use SparseHC for average-linkage clustering of sparse matrices computed from DNA sequence datasets. This experiment uses four sparse matrices computed from DNA sequence datasets of size 10000 - 40000 sequences. The sparsity of these matrices is about 50%.

The matrices are computed using the sequence embedding approach as used in the popular Clustal-Omega multiple sequence alignment tool [3]. Each DNA sequence is converted into a vector of real coordinates by computing the \( k \)-mer distances between that sequence and a set of seeds (seeds are representative sequences chosen from the same datasets). The pairwise distances amongst these DNA sequences are then computed by the Euclidean distances of their corresponding embedding vectors. Subsequently, the pairwise distance matrix is sorted and its lower half is written to disk for clustering. We report the runtime and memory usage of SparseHC and demonstrate its efficiency against other sparse clustering tools in Table 5.

4.4 SparseHC for clustering large matrices

To highlight the memory efficiency of SparseHC, we report the \( \frac{\text{matrix size}}{\text{memory usage}} \) ratio for four representative large datasets in Table 6. These datasets are 2 - 28 times bigger than the amount of RAM on the test platform. This table shows that SparseHC can process distance matrices three to four orders of magnitude larger than the memory capacity.

5 Conclusion

Producing dendograms by performing a hierarchical clustering of objects is a crucial data analysis tool in computational science. In this paper we have addressed the problem of finding a memory-efficient and fast approach (SparseHC) to compute such dendograms, which is of high importance to research since many scientific areas are facing a data explosion. SparseHC is a new online AHC tool which can perform accurate single-, complete- and average-linkage hierarchical clustering with linear empirical memory complexity, making it particularly useful to cluster large datasets using computers with limited memory resources.

References

