Introduction to Database Systems
Efficient Sub-sequence Matching in Time Series Using a Cluster

HENGBIN LIAO, JUN XU
Boston University
May 9, 2015

1 Introduction

A time series is a sequence of data points, typically consisting of successive measurements made over a time interval. Examples of time series are ocean tides, counts of sunspots, and the daily closing value of the Dow Jones Industrial Average. Time series are used in statistics, signal processing, pattern recognition, econometrics, mathematical finance, weather forecasting, and largely in any domain of applied science and engineering which involves temporal measurements.

Given a query time series and a database of time series, subsequence matching of time series is the problem of identifying the database subsequence (i.e., some part of some time series in the database) that is the most similar to the query sequence. It's important for subsequence matching to be efficient in domains where the database sequences are much longer than the queries.

2 Using Embedding for Subsequence Matching

Given $X = (X_1, \cdots, X_{|X|})$, a database sequence that is relative long, containing for example millions of elements, we can concatenate all $X_j, \forall j = 1, \cdots, |X|$. Called it X, too.

Also given a query sequence Q, we want to find the subsequence of X that optimally matches Q under Dynamic Time Warping. Brute-force search will be too expensive to afford. We cited the method called *Approximate Embedding-based Subsequence Matching of Time Series*. This method maps every query Q into a d-dimensional vector and every element X_j of the database sequence also into a d-dimensional vector.

Define *reference sequence* R as a relative short sequence. Then the 1D embedding for the database sequence

$$F^R(X, j) = D_{|R|,j}(R, X),$$

i.e., the smallest possible cost of matching $(R_1, \cdots, R_{|R|})$ to any suffix of $(X_1, \cdots, X_j)$. And the 1D embedding for the query is

$$F^R(Q) = D_{|R|,|Q|}(R, Q),$$

i.e., the smallest possible cost of matching $(R_1, \cdots, R_{|R|})$ to any suffix of $(Q_1, \cdots, Q_{|Q|})$. If we pick multiple reference sequences to create a multidimensional embedding, we can define a d-dimensional embedding F as follows:

$$F(X, j) = (F^{R_1}(X, j), \cdots, F^{R_d}(X, j)).$$

$$F(Q) = (F^{R_1}(Q), \cdots, F^{R_d}(Q)).$$

Computing such a vector for the database sequence is an off-line preprocessing step that takes time in $O(|X| \sum_{i=1}^{d} |R_i|)$. And for a query Q, its embedding vector is computed online, by applying the DTW algorithm d times, with inputs R_i and Q. In total, this applications of DTW take time in $O(|Q| \sum_{i=1}^{d} |R_i|)$. This time is negligible compared to running the DTW algorithm between Q and X, which takes $O(|Q||X|)$ time.

Therefore, rather than brute force search, we do as follows:

- Compare $F(Q)$ to $F(X, j)$, $\forall j = 1, \cdots, |X|$.
- Choose j's such that $F(Q)$ is close to $F(X, j)$ in Euclidean distance.
- For each such j, and for some length parameter L, run dynamic time warping between Q and $X^{j-L+1:j}$ to compute the best subsequence match for Q in $X^{j-L+1:j}$.

3 Filter Step

The filter step is to provide a quick way to identify a relatively small number of candidate matches. These candidates will be evaluated by the following refine step using the DTW algorithm. Here, embeddings can be used at the filter step.

The naive way to implement the filter step is by simply comparing $F(Q)$ to every single $F(X, j)$ stored in the database. However with high-dimensional embeddings, this method turns out to be impractical. The implementation in the paper uses sampling. In detail, choose a parameter δ and sample uniformly one out of every δ vectors $F(X, j)$, because embeddings of nearby positions tend to be very similar. We only compare it with the vectors that we have samples. If, for a database position (X, j), its vector $F(X, j)$ is not sampled, The distance between $F(Q)$ and the vector sampled among $\{F(X, j - \lfloor \delta/2 \rfloor, \cdots, F(X, j + \lfloor \delta/2 \rfloor)\}$ is assigned to that position.

4 Refine Step

The filter step ranks all database positions $(X < j)$ in incrreasing order of the distance between $F(X, j)$ and $F(Q)$. The refine step is to evaluate the top p candidates. Algorithm 1 describes how this evaluation is performed. Since candidate positions (X, j) actually represent candidate *endpoints* of a subsequence match, we can evaluate each such candidate endpoint by starting the DTW algorithm from that endpoint and going backwards.

4.1 Dynamic Time Warping

For each candidate (index in the long sequence), its corresponding distance is computed using the above dynamic programming algorithm that can be visualized as below:

The height represents the length of the query. Value A is initialized to 0, then computed as the minimum of the values on its right, bottom-right, the below sides, plus cost 1 if it moves horizontally or vertically. The values in the table are computed following this algorithm until it reaches the top-left of the table, which is the distance of the candidate.

Algorithm 1 Centralized refine step

```
for $i$ = 1 to $|X|$ do
    unchecked[$i$] = 0;
for $i$ = 1 to $p$ do
    unchecked[sorted[$i$]] = 1;
distance = $\infty$;
columns = 0;
for $k$ = 1 to $p$ do
    candidate = sorted[$k$];
    if (unchecked[candidate] == 0) then
        continue;
    $j$ = candidate + 1;
    for $i$ = $|Q| + 1$ to 1 do
        cost[$i$][$j$] = $\infty$;
    while (true) do
        $j = j - 1$;
        if (candidate - $j \geq 2 * |Q|$) then
            break;
        if (unchecked[$j$] == 1) then
            unchecked[$j$] = 0;
            candidate = $j$;
            cost[$|Q| + 1$][$j$] = 0;
            endpoint[$|Q| + 1$][$j$] = $j$;
        else
            cost[$|Q| + 1$][$j$] = $\infty$;
        for $i$ = $|Q|$ to 1 do
            previous = $\{(i+1, j), (i, j+1), (i+1, j+1)\}$;
            $(p_i, p_j) = \text{argmin}_{(a,b) \in \text{previous}} \text{cost}[a][b]$;
            $\text{cost}[i][j] = |Q_i - X_j| + \text{cost}[p_i][p_j]$;
            endpoint$[i][j]$ = endpoint$[p_i][p_j]$;
        if (cost[1][$j$] < distance) then
            distance = cost[1][$j$];
            $j_{\text{start}} = j$;
            $j_{\text{end}} = endpoint[1][j]$;
        columns = columns + 1;
        if (min$\{\text{cost}[i][j]\} \geq$ distance, $\forall i = 1, \cdots, |Q|$) then
            break;
start = $j_{\text{end}} - 3 * |Q|$;
end = $j_{\text{end}} + |Q|$;
Adjust $j_{\text{start}}$ and $j_{\text{end}}$ by running the DTW algorithm between $Q$ and $X^{\text{start:end}}$;
```

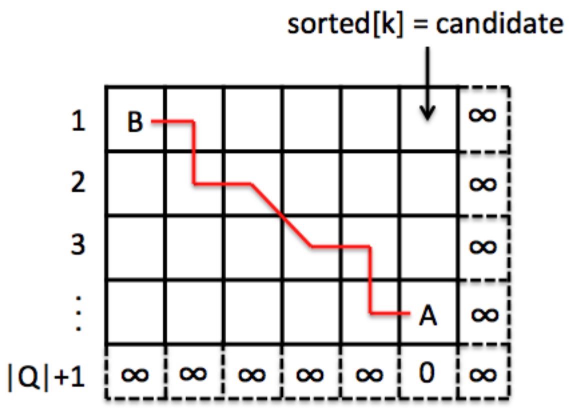

Figure 1: Dynamic Time Warping

4.2 MapReduce

Instead of a centralized algorithm (Algorithm 1), we propose here a distributed algorithm (Algorithm 2-4) via MapReduce.

Mappper

For each input $(Q, X, \text{candidate}, \text{index})$, where Q is the query, X is the database sequence, *candidate* is the candidate endpoint and *index* is that of this candidate in the candidate array, we first compute the distance between the query and this candidate subsequence. We also compute the start point j_{start} and end point j_{end} of this matching. Afterwards, we associate it with a key, which is computed as

$$\text{key} = \text{index} \ \% \ \text{total\_nodes},$$

where *total_nodes* is the number of nodes in the cloud. According to the keys, we assign every candidate to its corresponding node, candidates $(Q, X, \text{key}, \text{distance}, j_{\text{start}}, j_{\text{end}})$ with the same key will be on the same node.

Combiner

Each combiner takes candidates on the same node and computes the smallest distance among the candidates on that node, then output the results to the reducer.
input: $(Q, X, \text{key}, [(\text{distance}_1, j_{\text{start},1}, j_{\text{end},1}),\ (\text{distance}_2, j_{\text{start},2}, j_{\text{end},2}), \cdots])$
output: $(Q, X, \text{key}, \text{distance}_{\text{min}}, j_{\text{start}}, j_{\text{end}})$

Reducer

The reducer takes inputs from combiners and computer the smallest distance among these inputs. The result contains the smallest distance of all the original candidates.
input: $(Q, X, [(\text{key}_1, \text{distance}_1, j_{\text{start},1}, j_{\text{end},1}),\ (\text{key}_2, \text{distance}_2, j_{\text{start},2}, j_{\text{end},2}),\ \cdots])$
output: $(Q, X, \text{key}, \text{distance}_{\text{min}}, j_{\text{start}}, j_{\text{end}})$

Algorithm 2 Mapper

```
1: $j$ = candidate + 1;
2: for $i$ = $|Q| + 1$ to 1 do
3:     cost[$i$][$j$] = ∞;
4: while (true) do
5:     $j = j - 1$;
6:     if (candidate - $j \geq 2 * |Q|$) then
7:         break;
8:     if (candidate == $j$) then
9:         cost[$|Q| + 1$][$j$] = 0;
10:        endpoint[$|Q| + 1$][$j$] = $j$;
11:    else
12:        cost[$|Q| + 1$][$j$] = ∞;
13:    for $i$ = $|Q|$ to 1 do
14:        previous = $\{(i+1, j), (i, j+1), (i+1, j+1)\}$;
15:        $(p_i, p_j) = \text{argmin}_{(a,b)\in\text{previous}} \text{cost}[a][b]$;
16:        $\text{cost}[i][j] = |Q_i - X_j| + \text{cost}[p_i][p_j]$;
17:        $\text{endpoint}[i][j] = \text{endpoint}[p_i][p_j]$;
18: key = index % total_nodes;
19: distance = cost[1][$j$];
20: $j_{\text{start}} = j$;
21: $j_{\text{end}} = \text{endpoint}[1][j]$;
22: Emit($Q$, $X$, key, distance, $j_{\text{start}}$, $j_{\text{end}}$);
```

Algorithm 3 Combiner

```
1: $\text{distance}_{\text{min}} = \infty$;
2: for all $\text{distance}_{\text{i}} \in [\text{distance}_1, \text{distance}_2, \cdots]$ do
3:     if ($\text{distance}_{\text{i}} < \text{distance}_{\text{min}}$) then
4:         $\text{distance}_{\text{min}} = \text{distance}_{\text{i}}$;
5:         $j_{\text{start}} = j_{\text{start,i}}$;
6:         $j_{\text{end}} = j_{\text{end,i}}$;
7: Emit($Q$, $X$, key, $\text{distance}_{\text{min}}$, $j_{\text{start}}$, $j_{\text{end}}$);
```

Algorithm 4 Reducer

```
1: $\text{distance}_{\text{min}} = \infty$;
2: for all $\text{distance}_{\text{i}} \in [\text{distance}_1, \text{distance}_2, \cdots]$ do
3:     if ($\text{distance}_{\text{i}} < \text{distance}_{\text{min}}$) then
4:         key = $\text{key}_{\text{i}}$;
5:         $\text{distance}_{\text{min}} = \text{distance}_{\text{i}}$;
6:         $j_{\text{start}} = j_{\text{start,i}}$;
7:         $j_{\text{end}} = j_{\text{end,i}}$;
8: start = $j_{\text{end}} - 3 * |Q|$;
9: end = $j_{\text{end}} + |Q|$;
10: Adjust $j_{\text{start}}$ and $j_{\text{end}}$ by running the DTW algorithm between $Q$ and $X^{\text{start:end}}$;
```