Context-Aware Graph-Based Algorithm for Merging Rows (Entities) with Selected-Match Gain and Path Penalty

Roman Pleshkov*, Alexander Voronkov †

*dmgcodevil@gmail.com

†alex_voronkov@spmk72.com

***Abstract*—We present a graph driven algorithm for merging partially overlapping rows into a coherent table, designed for scenarios where entities extracted from unstructured or semi-structured data sources need to be consolidated into a unified structure. Each row contains a set of label-value tuples, and the algorithm resolves missing labels by traversing a graph of connected rows based on shared tuples. Our method introduces a custom scoring function with three components: *local strength, selected-match gain, tuple weight*, and *path penalty*, ensuring fine-grained control over the merging process. By leveraging a weighted BFS, we guarantee optimal label assignments under this scoring framework.**

This algorithm provides a cost-effective, fine-tunable alternative to Retrieval-Augmented Generation (RAG) for entity consolidation tasks, especially in domains requiring interpretable, deterministic outputs and minimal reliance on pre-trained models. Its two-phase approach—prioritizing "perfect rows" (those fully matching partial merges) before exploring other candidates—achieves efficient and high-quality merges. We include examples, pseudo-code, and a proof of correctness, demonstrating its practical applicability in use cases such as entity resolution, log analysis, and structured data enrichment.

I. Introduction

Merging partially overlapping data ("rows") is a fundamental task in *data integration, entity resolution*, and *record linkage*. Each row may contain a set of label-value tuples (e.g., (L_1, V_1)), but many labels are missing or incomplete. The goal is to fill these missing labels with values from other rows that share overlapping tuples. For example:

```
R1=[(L1,V1)]
R2=[(L1,V2)]
R3=[(L2,V2), (L3,V3), (L4,V5)]
R4=[(L2,V2), (L3,V4), (L4,V6)]
R5=[(L1,V1), (L2,V2), (L3,V3)]
R6=[(L1,V2), (L2,V2), (L3,V4)]
```

To calculate L_4 for R_1, we observe the following connections:

- R_1 is connected to R_5 via (L_1, V_1).
- R_5 is connected to R_3 via (L_2, V_2) and (L_3, V_3).

Thus, L_4 for R_1 should come from R_3, resulting in the expected output:

```
R1 = [(L1,V1,R5),(L2,V2,R5),(L3,V3,R5),(L4,V5,R3)]
R2 = [(L1,V2,R6),(L2,V2,R6),(L3,V4,R6),(L4,V6,R4)]
```

This process builds a graph where nodes represent rows, and edges represent label-value pairs that connect rows. Figure 1 illustrates the graph for this example:

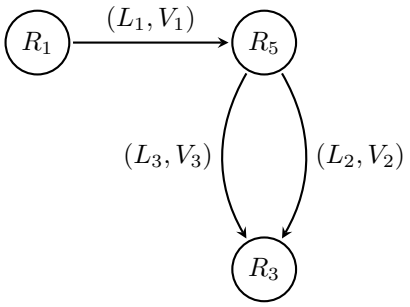

Fig. 1. Graph of rows connected by label-value pairs. Each edge is labeled by the shared tuple (L, V).

A. Key Scoring Components

Our approach employs a flexible scoring system with the following components:

- **Local Strength**: This measures the overlap between tuples of two rows, incorporating tuple weights.
- **Selected-Match Gain**: A boost is applied to rows consistent with already-selected label-values.
- **Path Penalty**: A linear cost per step in the BFS path discourages long, convoluted routes unless they offer higher scores.
- **Tuple Weight**: Each tuple can have an associated weight, allowing fine-grained control over the importance of specific tuples.

B. Handling Ties in Scores

When two paths yield the same score, the algorithm returns all possible values. For example:

```
R1=[(L1,V1)]
R2=[(L1,V1), (L2,V2)]
R3=[(L1,V1), (L2,V3)]
```

To calculate L_2 for R_1, both paths $R_1 \to R_2 \to (L_2, V_2)$ and $R_1 \to R_3 \to (L_2, V_3)$ have equal scores. The result is:

```
R1 = [(L1,V1,R2),(L2,V2,R2),(L2,V3,R3)]
```

C. Guaranteeing Optimal Merges

We prove that a *Weighted BFS*, exploring paths in descending order of "score so far," guarantees globally optimal merges under this scoring system. This ensures that missing labels are filled with the best possible values, balancing consistency, and path efficiency.

II. Definitions and Graph Model

A. Rows and Their Connections

Let $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ be the set of rows, where each row R_i is represented as a set of label-value tuples:

$$R_i = \{(L_1, V_1), (L_2, V_2), \ldots, (L_k, V_k)\}.$$

Here, L_j denotes a *label*, and V_j denotes its corresponding *value*.

To facilitate efficient search and merging, we construct a graph $G = (\mathcal{V}, \mathcal{E})$:

- $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, where each v_i corresponds to a row R_i.
- $\mathcal{E} = \{(v_i, v_j) \mid R_i \cap R_j \neq \emptyset\}$, indicating an edge exists if rows R_i and R_j share at least one label-value tuple.

B. Graph Construction: Buckets and Nodes

To improve search efficiency, we organize tuples into *buckets* based on their labels, as follows:

$$\text{Bucket} = \begin{cases} \{\text{label: } L \mapsto \text{nodes:} \\ \{\text{Hashcode} \mapsto \text{List of Nodes}\}\} \end{cases}$$

Each node in a bucket corresponds to a token hash, uniquely identifying a tuple across rows. Formally:

$$\text{Node} = \{\text{row\_id}, \text{tuple\_index}, \text{token\_hash}\}$$

The overall graph structure is:

$$\text{Graph} = \{\text{buckets: } L \mapsto \text{Bucket}, \text{rows: RowID} \mapsto \text{Row}\}$$

C. Scoring Function: Local, Selected, and Penalty

To determine the optimal row for completing a missing label, we define a scoring function for paths in the graph:

$$\text{Score}(\pi) = \sum_{i=1}^{k} \Big[w_{\text{local}}(v_{i-1}, v_i) + w_{\text{selected}}(v_i) \Big] - \alpha \cdot k,$$

where:

- $w_{\text{local}}(v_{i-1}, v_i)$: Measures the *local strength*, proportional to the number of shared label-value tuples between R_{i-1} and R_i and adjusted by tuple weights.
- $w_{\text{selected}}(v_i)$: Adds a *selected-match gain* for consistency with previously selected tuples.
- $\alpha \cdot k$: Imposes a *path penalty* proportional to the number of edges in the path.

D. Perfect Rows (Optional First Phase)

We define a *perfect row* as one that matches all currently selected label-value pairs. For a row R_j and selected tuples Selected, the match strength is:

$$\text{strength}(R_j, \text{Selected}) = |\{(L, V) \in \text{Selected} \mid (L, V) \in R_j\}|.$$

Perfect rows are given an initial BFS score equal to the size of the selected set:

$$\text{Score}(\text{Perfect Row}) = |\text{Selected}|.$$

III. Algorithm

A. Pseudo Code for BFS-based Tuple Search and Merge

Algorithm 1 Weighted BFS to Find Best Tuple (`bfs_find_best`)

Initialize a max-heap priority queue pq
Initialize an empty set $visited$
Insert $start\_row$ into $visited$
Push $(-initial\_score, curr\_row\_id, 0)$ onto pq ▷ Path length starts at 0
while pq is not empty **do**
    Pop $(neg\_score, curr\_row\_id, path\_length)$ from pq
    $current\_score \leftarrow -neg\_score$
    if $curr\_row\_id \in visited$ **then**
        continue
    end if
    Insert $curr\_row\_id$ into $visited$
    for $t \in$ tuples of row $curr\_row\_id$ **do**
        if $t.label == label$ **then**
            return `SearchResult`($current\_score$, t)
        end if
    end for
    $candidates \leftarrow$ get_connected_rows_with_scores($curr\_row\_id$)
    for $next\_row\_id, local\_score \in candidates$ **do**
        $selected\_score \leftarrow$ strength_selected($next\_row\_tuples, selected$)
        $path\_penalty \leftarrow path\_length \cdot$ PATH_PENALTY
        $total\_score \leftarrow current\_score + local\_score + selected\_score - path\_penalty$
        Push $(-total\_score, next\_row\_id, path\_length + 1)$ onto pq
    end for
end while
return `None` ▷ No valid tuple found

Algorithm 2 Find the Best Tuple: `find_best_tuple(label, source_row_id, selected)`

Initialize $best\_result \leftarrow$ `SearchResult`$(0.0, [])$
$perfect\_rows \leftarrow \{\}$ ▷ Stores rows fully matching the selected dictionary
for $candidate \in$ graph.rows **do**
    if $candidate.id \neq source\_row\_id \wedge$ is_perfect$(candidate.tuples, selected)$ **then**
        $perfect\_rows[candidate.id] \leftarrow$ strength_selected$(candidate.tuples, selected)$
    end if
end for
for $row\_id, score \in perfect\_rows$ **do**
    $res \leftarrow$ `bfs_find_best`$(source\_row\_id, row\_id, label, selected, score)$
    if $res.score > best\_result.score$ **then**
        $best\_result \leftarrow res$
    else if $res.score == best\_result.score$ **then**
        Append $res.tuples$ to $best\_result.tuples$
    end if
end for
if best_result.tuples $\neq \emptyset$ **then**
    return $best\_result$
end if
$candidates \leftarrow$ get_connected_rows_with_scores($source\_row$)
for $row\_id, score \in (candidates)$ **do**
    $res \leftarrow$ `bfs_find_best`$(source\_row\_id, row\_id, label, selected, score +$ strength_selected(graph.rows[row_id].tuples, $selected$))
    if $res.score > best\_result.score$ **then**
        $best\_result \leftarrow res$
    else if $res.score == best\_result.score$ **then**
        Append $res.tuples$ to $best\_result.tuples$
    end if
end for
return $best\_result$

Algorithm 3 Merge Rows into a Cohesive Table (`merge`)

Initialize an empty dictionary $table$
$row\_filter \leftarrow$ kwargs.get($'row\_filter'$, {})
for $row \in$ graph.rows **do**
  if $row.id \notin row\_filter \wedge row\_filter \neq \emptyset$ **then**
    continue
  end if
  Initialize $total\_score \leftarrow 0.0$ and $selected \leftarrow \{\}$
  for $col \in columns$ **do**
    if $col \notin selected$ **then**
      $res \leftarrow$ find_best_tuple($col, row.id, selected$)
      if $res \neq$ `None` **then**
        $total\_score \leftarrow total\_score + res.score$
        $selected[col] \leftarrow res.tuples$
      else
        Use placeholder tuple for col if no match is found
      end if
    end if
  end for
  Form a unique key for $table$ using the resolved column values
  if Key not in $table$ or $total\_score > table[key].score$ **then**
    $table[key] \leftarrow$ `DataRow`($row.id, total\_score, selected$)
  end if
end for
return List of values from $table$

B. Algorithm Explanation

The algorithm consists of three primary components: `merge`, `find_best_tuple`, and `bfs_find_best`, which work together to resolve missing labels in rows by leveraging a graph-based search.

The process begins with the `merge` function, which iterates through all rows in the graph. For each row, it attempts to resolve each missing label by calling `find_best_tuple`. This function ensures that the most relevant values for the missing label are identified based on a scoring system that incorporates *local strength*, *selected-match gain*, and a *path penalty*.

`find_best_tuple` operates in two phases. In the first phase, it identifies "perfect rows," which fully match the currently selected label-value pairs. These rows are prioritized and used as the starting points for a weighted Breadth-First Search (BFS). In the second phase, the algorithm explores rows that are directly connected to the current row, expanding paths in descending order of their scores.

The actual BFS logic is implemented in `bfs_find_best`. This function performs a weighted BFS starting from a given row. It evaluates candidate rows based on their local strength, consistency with previously selected values, and the path penalty to discourage unnecessarily long paths. The BFS terminates when it finds a row that provides the desired label, ensuring that the selected path maximizes the defined scoring function.

By combining these components, the algorithm efficiently resolves missing labels in rows while maintaining consistency with existing data, leveraging both direct and indirect relationships in the graph. This two-phase approach ensures that the algorithm is both computationally efficient and capable of achieving high-quality results.

IV. Distributed Algorithm Design (Optional Consideration)

While the primary goal of this paper is not to design a distributed version of the algorithm, we outline a potential approach to enable scalability and fault tolerance in large-scale scenarios. The algorithm can be decentralized by leveraging a distributed architecture where machines are grouped based on labels, allowing parallel processing of data within each `Bucket`.

Cluster Organization:

- *Label-Based Clusters:* Each cluster of machines is responsible for handling a specific label. This ensures that operations related to a particular label are confined to a single group of machines, minimizing cross-cluster communication.
- *Hash Ranges:* Within each cluster, machines are assigned ranges of hash values, dividing the `nodes: Dict[HashcodeT, List[Node]¿` evenly. For example, Machine A handles hashes $[1, 10]$, Machine B handles $[11, 20]$, and so on.

Fault Tolerance:

- Each hash range is replicated across at least three machines (replication factor of 3). If one machine becomes unavailable, another machine in the replica group can take over its responsibilities.
- Leader-election mechanisms (e.g., using Zookeeper) can manage primary/secondary replicas and ensure consistent updates.

Graph and Metadata Management:

- *Graph Storage:* The `rows: Dict[RowIdT, Row]` can be stored in a distributed, in-memory key-value store (e.g., Redis, Hazelcast, or Apache Ignite). Each machine fetches relevant rows based on its assigned label and hash range.
- *Metadata Management:* Metadata about the graph (e.g., bucket ownership, hash range assignments, replication groups) is stored in Zookeeper, ensuring

consistent coordination and dynamic reassignment of responsibilities when machines join or leave.

Enhancements:

- *Compound Hashing:* To improve load balancing, a compound hash of $(\text{label}, \text{token})$ can be used for node distribution. This ensures that both label and token influence machine assignment.
- *Dynamic Scaling:* The architecture allows for dynamic scaling by adding machines to the cluster and redistributing hash ranges as needed.
- *Parallel Bucket Processing:* Operations such as `find_best_tuple` or BFS traversal can be parallelized across machines processing different buckets, minimizing inter-machine communication by focusing on local processing within hash ranges.

This distributed design provides scalability and fault tolerance, ensuring the algorithm can handle large datasets in real-world applications. However, the implementation of such a system is beyond the scope of this paper, which focuses on the core algorithm for merging rows efficiently.

V. Conclusion

This paper introduces a simple yet effective BFS-based algorithm for merging partially overlapping rows into a cohesive table. The algorithm's design emphasizes stability and flexibility, providing users with various parameters to fine-tune its behavior based on specific use cases. Key adjustable parameters include:

- `DEFAULT_TUPLE_MATCH_GAIN`: Controls the influence of tuple overlap on the score (default: 1.0).
- `SELECTED_MATCH_GAIN`: Adds a bonus for rows that align with already-selected tuples (default: 1.5).
- `PATH_PENALTY`: Penalizes longer paths to discourage overly complex routes (default: 0.1).

The algorithm also supports assigning higher weights to tuples originating from trusted or high-confidence sources, enabling a domain-aware merging process. This feature further enhances the algorithm's applicability to real-world data integration tasks.

One of the strengths of this approach is its minimalistic design, which makes it straightforward to adapt for distributed and parallel execution. For instance, the graph's buckets can be processed by different machines based on labels, and hash ranges within buckets can be further distributed across nodes in a cluster. Such a setup ensures scalability and fault tolerance, with redundancy achievable through mechanisms like consistent hashing or replication.

Compared to expensive solutions like Large Language Models (LLMs) or Retrieval-Augmented Generation (RAG), this algorithm offers a cost-effective and deterministic alternative. Its simplicity and interpretability make it suitable for applications where deterministic outputs and low operational costs are prioritized.

In summary, this algorithm provides a practical and robust solution to row-merging tasks, balancing performance, adaptability, and ease of implementation. Its inherent flexibility and potential for distributed deployment make it a compelling choice for modern data integration challenges.