DEIM Forum 2018 I5-5

# A Schema Extraction Algorithm for External Memory Graphs Based on Novel Utility Function

Yoshiki SEKINE† and Nobutaka SUZUKI††

† Graduate School of Library, Information and Media Studies, University of Tsukuba
1-2, Kasuga, Tsukuba-shi, Ibaraki, 305-8550, Japan
†† Faculty of Library, Information and Media Science, University of Tsukuba
1-2, Kasuga, Tsukuba-shi, Ibaraki, 305-8550, Japan
E-mail: †ysekine@klis.tsukuba.ac.jp, ††nsuzuki@slis.tsukuba.ac.jp

**Abstract** In recent years, the size of graph data is drastically growing. In contrast to relational databases, most of graphs do not have their own schemas. If we can extract a schema from a graph efficiently, we can take advantage of the extracted schema for query optimization, structure browsing, query formulation, and so on. In this paper, we propose an external memory algorithm for extracting a schema from a graph. The algorithm is designed so that each file is read sequentially in most cases and very few random accesses are required for schema extraction, which makes our algorithm I/O efficient.

**Key words** graph schema, schema extraction, external memory algorithm

## 1. Introduction

In recent years, graph data has been widely used and various kinds of new graph data is actively being created. In contrast to other databases such as relational databases and XML, most of graphs do not have their own schemas. Therefore, in many cases we cannot make use of schema to manage graphs effectively. Here, if we can extract a schema from a graph efficiently, we can take advantage of the extracted schema for query optimization [1], structure browsing, query formulation [2], and so on. However, extracting schemas from large graphs are difficult due to the following reasons. Firstly, most of schema extraction algorithms proposed so far are in-memory algorithms and thus cannot deal with large graphs that do not fit in main memory. Secondly, schema extraction is a complex and time-consuming task. The utility function, which is a popular function used in schema extraction, requires a large amount of computation cost as the number of unique edge labels in a graph becomes larger. To address these problems, we propose a novel schema extraction algorithm for large graphs. This is designed as an external memory algorithm using parallel processing and our novel utility function. Our utility function is designed so that less computation cost is required while schemas are extracted as appropriately as the original utility function. Experimental results suggest that our algorithm can extract schemas from graphs more efficiently and appropriately than the algorithm using the previous utility function, and that the parallelization of the class extraction makes the execution time faster for a real-world graph with many unique edge labels.

**Related Work**

A number of schema extraction algorithms for graphs have been proposed. DataGuide [2] extracts a schema by grouping nodes reachable from the root via the same label path into the same class. ApproximateDataguite [3] is the approximate version of DataGuide. Nestorov et al. proposes an algorithm for extracting a set of classes by using a clustering approximation method [4]. Wang et al. proposes an algorithm that extracts a schema by an incremental clustering method [5]. These algorithms are in-memory algorithms and cannot handle large graphs that do not fit in main memory. Navlakha et al. proposes a graph summarization algorithm [6]. This is an in-memory algorithm designed for unlabeled undirected graphs, while our algorithm is designed for labeled directed graphs. Luo et al. proposes an external memory algorithm for $k$-bisimulation [7]. However, the notion of $k$-bisimulation is too strong to extract classes from usual graphs, since under the condition of $k$-bisimulation, any two nodes in the same class must have same label paths whose length is $k$. On the other hand, our algorithm assumes a weaker condition under which nodes having a "similar" set of edges are grouped into the same class.

Several external memory algorithms have been proposed in database research field, e.g., graph triangulation [8], strongly connected components [9], graph reachability [10], and regular path query [11]. To the best of our knowledge, how-

ever, no external memory algorithm for schema extraction has been proposed so far.

## 2. Preliminaries

Let $L$ be a set of labels. A *labeled directed graph* (*graph* for short) is denoted $G = (V, E)$, where $V$ is a set of *nodes* and $E \subset V \times L \times V$ is a set of *labeled directed edges* (*edges* for short). Let $e \in E$ be an edge labeled by $l \in L$ from a node $v \in V$ to a node $u \in V$. Then $e$ is denoted $(v, l, u)$, $v$ is called *source*, $u$ is called *target* of $e$, and we say that $v$ has the edge $e$. The set of outgoing edge labels of $v$, namely the set of labels which $v$ has, is denoted $L(v)$. A *schema* is a summarization of a graph and it is also represented as a graph. A node in a schema is called a *class*. Any node in a graph is mapped to a class in a schema. We assume that every text leaf node belongs to a particular class called $LEAF$ and that every non-text leaf node belongs to another particular class called $LEAF2$. For a node $v$ in a (instance) graph, by $class(v)$ we mean the class that $v$ belongs to. A schema is denoted $S = (C, E_s)$, where $C$ is a set of classes and $E_s$ is a set of edges between classes.

In this paper, we assume that a graph is stored in a file like N-Triples format, which is a container for Resource Description Framework (RDF) data. Each line of a file corresponds to an edge, namely, a line consists of $(source, label, target)$. We assume that a schema is composed of two files denoted schema_classes and schema_edges. The former stores pairs of a node and its class, namely a line consists of $(node, class)$. The latter stores edges between classes, namely a line consists of $(source\ class, label, target\ class)$.

## 3. The Algorithm

In this section, we first define our utility function, then we describe the schema extraction algorithm.

### 3.1 Utility Function

Wang et al. proposes an algorithm that extracts a graph schema by grouping nodes having a similar set of edge labels into the same class using the utility function [5]. However, the utility function requires a large amount of calculation cost for graphs containing a large number of unique edge labels. To cope with the problem, we define a new utility function, called *light utility function*, so that we can extract schemas from such graphs more efficiently.

Let $c \in C$ be a class. In this paper, we ignore incoming edges of $v$ to make our algorithm simple and to reduce calculation cost. The set of edge labels of $v$ is denoted $L(v)$. By $L(c)$ we mean the set of edge labels of $c$, that is, $L(c) = \bigcup_{v \in c} L(v)$. Let $|c|$ be the number of nodes in $c$ and $nodes(c, l)$ be the set of nodes in $c$ having an edge labeled by $l$. By $nodes(C, l)$ we mean the set of nodes in $C$ having an edge labeled by $l$, that is, $nodes(C, l) = \bigcup_{c \in C} nodes(c, l)$. Then the *light utility function* (*utility function*, for short), denoted $U(C, v, c_i)$, is defined as the product of the Dice coefficient and the mean of the ratio of $|nodes(c, l)|$ to $|nodes(C, l)|$, that is,

$$U(C, v, c_i) = Dice(L(v), L(c_i))^{\alpha} \frac{1}{|L(v)|} \sum_{l \in L(v)} \frac{|nodes(c_i, l)|}{|nodes(C, l)|},$$

where $\alpha$ is a parameter to control which of $Dice$ and the latter ratio is emphasized when extracting classes. $U$ becomes higher if nodes having similar edge labels are grouped into the same class. By this definition we intend to balance how labels of a node $v$ are similar with labels of a class $c_i$ and how much labels of $c_i$ occupy the entire schema. We need to extract classes so that the classes bring a high value of the utility function.

### 3.2 Schema Extraction Algorithm

In-memory schema extraction algorithms assume that the entire graph is stored in main memory. However, since recent large graphs are too large to fit in main memory, we need another approach to handling such large graphs.

In order to deal with such large graphs, we take the following approach. First, our algorithm sequentially reads a graph, extracts classes with maintaining minimum information to extract classes in main memory, and outputs a class file consisting of the classes of all the nodes. Next our algorithm creates an edge file having information required for extracting edges between classes. Note that since the information for edge extraction must include the all edges, the files cannot be fit in main memory. Then the algorithm extracts edges by reading these files sequentially. An outline of our algorithm is as follows (see Fig. 1).

**Input:** graph file. Each line of the file represents an edge. In the following, we call the input graph file file 1.

**Output:** schema_classes and schema_edges.

(1) **Preprocessing**

(a) Sort file 1 externally. Let file 1' be the resulting file.

(2) **Class Extraction**

(a) Read file 1' sequentially and extract the class of each node based on the utility function.

(b) Each time the class of a node is extracted, output the node and the class to schema_classes.

(3) **Edge Extraction**

(a) Read file 1' and schema_classes concurrently and sequentially, and output the outgoing edges of nodes (source nodes are replaced by their classes) to another file tmp_file1. The file stores edges between classes and nodes.

(b) Sort tmp_file1 externally. Let tmp_file1' be the resulting file.

(c) Read schema_classes and tmp_file1' concurrently and sequentially, and replace the target node of each edge in tmp_file1' with its class. This results in edges between classes, which are written into schema_edges.

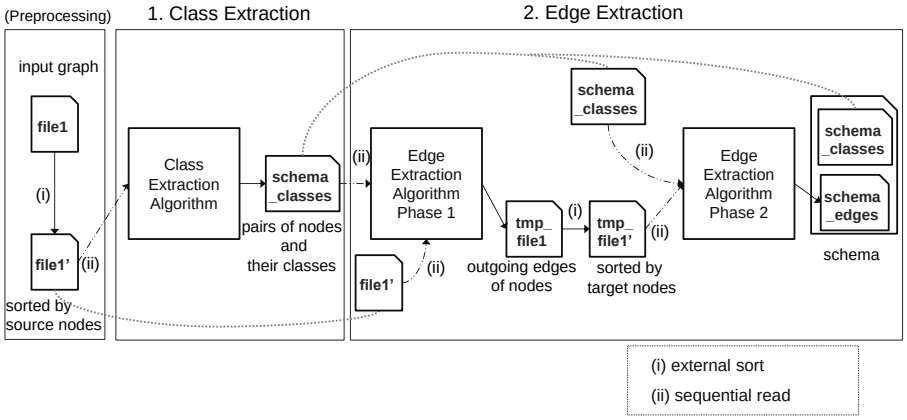


Figure 1 Outline of our schema extraction algorithm

In the following, we give the details of our class extraction and edge extraction algorithms.

### 3. 3 Class Extraction

This algorithm is designed as (1) an external memory algorithm to handle large graphs that cannot fit in main memory, and (2) a parallel processing algorithm to reduce the calculation time of utility function. In the class extraction, calculating utility function requires a large amount of computation cost. Therefore we design the class extraction as a parallel process algorithm. We first show the non-parallel version next show the parallel version. Both versions are also designed as an external memory algorithm.

#### 3. 3. 1 Data Maintained in Main Memory

Values $|nodes(c, l)|$ for each class $c \in C$ and $|nodes(C, l)|$ are kept in memory to calculate utility function until the classes of all nodes are extracted. That is, we store the number of nodes having an edge labeled by $l$ (1) in $L(c)$ for each class $c \in C$, (2) and in $C$. On the other hand, for a node $v$, $v$'s name and $v$'s edge labels $L(v)$ is kept in memory until the algorithm outputs the class of $v$ to schema_classes. When the class of $v$ is extracted, $v$ and $L(v)$ are discarded.

#### 3. 3. 2 Class Extraction Algorithm (non-parallel version)

Algorithm 1 shows the procedure of the single process (non-parallel) class extraction. An input file, file 1' is a sorted file obtained in the preprocessing. Since file 1' is sorted, edges having the same source appear consecutively in file 1'. Therefore, we can obtain the outgoing edges of each node by one sequential read only.

Let $v$ be the node that is currently read. By reading file 1' sequentially, we obtain the edges $v$ has and extract the class of $v$ based on the utility function in Algorithm 2. This process is repeated until file 1' reaches the end of file.

#### 3. 3. 3 Class Extraction Algorithm (parallelized version)

Algorithm 3 shows the procedure of the parallelized class extraction. We parallelize the process of calculating the utility function. Like the non-parallel algorithm, an input file, file 1', is a sorted file obtained in the preprocessing. By reading file 1' sequentially, we read $k$ nodes, obtain the edges of them. Then we extract classes of $k$ nodes based on the utility function in $k$ parallel processes. After the parallel processes, some nodes are assigned to existing classes $c \in C$, and the other nodes assigned to the new class $c_v$. While we confirm that nodes assigned to existing classes belong to the classes, we must determine whether some of nodes assigned to the new class $c_v$ should be merged into or remain. We call this

**Algorithm 1** Class Extraction Algorithm (non-parallel version)

**Input:** file 1'
**Output:** schema_classes

1: $C \leftarrow \emptyset$.
2: **while** file 1' does not reach EOF **do**
3: $L(v) \leftarrow$ the set of outgoing edge labels of $v$ obtained by reading file 1'.
4: $class(v) \leftarrow$ ClassDetermination$(C, v, L(v))$.
5: Add a pair $(v, class(v))$ to schema_classes.
6: $C \leftarrow C \cup \{class(v)\}$.
7: **end while**

**Algorithm 2** Class Determination

1: **procedure** ClassDetermination$(C, v, L(v))$
2: **for each** class $c_i \in C$ **do**
3: Calculate $U(C, v, c_i)$.
4: **end for**
5: Let $c_v$ be the new class having the same set of outgoing edges as $v$ (edges having the same label are merged into one).
6: Calculate $U(C, v, c_v)$.
7: Let $class(v)$ be the class such that the value of $U$ is the highest among $C \cup \{c_v\}$.
8: **return** $class(v)$.
9: **end procedure**

**Algorithm 3** Class Extraction Algorithm (parallelized version)

**Input:** file 1'
**Output:** schema_classes

1: $C \leftarrow \emptyset$.
2: $k$ is set to the parallel number.
3: **while** file 1' does not reach EOF **do**
4: $V_{tmp} \leftarrow \emptyset$.
5: **Repeat** $k$ times
6: $L(v_i) \leftarrow$ the set of outgoing edge labels of $v_i$ obtained by reading file 1'.
7: $V_{tmp} \leftarrow V_{tmp} \cup \{v_i\}$.
8: **End Repeat**
9: $R \leftarrow \emptyset$.
10: **Parallel** for each $v_i \in V_{tmp}$
11: $class(v_i) \leftarrow$ ClassDetermination$(C, v_i, L(v_i))$.
12: $R \leftarrow R \cup \{(v_i, class(v_i))\}$.
13: **End Parallel**
14: $V_{one} = \{v \mid (v, class(v)) \in R$ such that $class(v) = c_v\}$.
15: **if** $|V_{one}| > 1$ **then** ▷ conflict occurs
16: $R' \leftarrow$ ConflictResolution$(C, R, V_{one})$.
17: **else**
18: $C \leftarrow C \cup \{class(v_1), \ldots, class(v_k)\}$.
19: $R' \leftarrow R$
20: **end if**
21: For each node $v_i \in V_{tmp}$, add a pair $(v_i, class(v_i)) \in R'$ to schema_classes.
22: **end while**

recalculation process *conflict resolution*. By doing that, we finally obtain the classes of $k$ nodes completely. This process is repeated until file 1' reaches the end of file.

Let us explain the procedure of ConflictResolution. We select the class of each node belonging to existing classes and store them in $R'$ in line 3. ($C$, $|nodes(c, l)|$ for each $c \in C$, and $|nodes(C, l)|$ are updated by these results in line 7.) The first node whose class is $c_v$ is selected in line 4 and add it to $R'$ in line 5. Then in lines 6 to 7 we update $C$, $C_{tmp}$, $|nodes(c, l)|$ for each $c \in C$ and each $c \in C_{tmp}$, and $|nodes(C, l)|$ in main memory assuming that the node $v$ selected in line 4 belongs to $class(v)$. For each node $v \in V_{one}$, namely each node in $V_{tmp}$ assigned to $c_v$ except the first node, we recalculate the utility function. We call ClassDetermination2 (Algorithm 5) instead of ClassDetermination so that the existing classes are computed to $C_{tmp}$ only. ClassDetermination2 extracts the class of $v \in V_{one}$ as follows. We calculate the following utility function and choose the class for which the maximum value is obtained.

- The utility function assuming that $v$ belongs to $c \in C_{tmp}$, for each class $c$ extracted so far in a conflict resolution.
- The utility function assuming that $v$ belongs to a new class $c_v$ having the same edges as $v$.

**Algorithm 4** Conflict Resolution

1: **procedure** ConflictResolution$(C, R, V_{one})$
2: $C_{tmp} \leftarrow \emptyset$.
3: $R' = \{(v, class(v)) \in R \mid v \notin V_{one}\}$. ▷ conflict resolution result
4: Remove the first node of $V_{one}$. Let $v$ be the first node.
5: $R' \leftarrow R' \cup \{(v, class(v))\}$.
6: $C_{tmp} \leftarrow C_{tmp} \cup \{class(v)\}$.
7: $C \leftarrow C \cup \{class(v) | (v, class(v)) \in R'\}$.
8: **for each** $v \in V_{one}$ **do**
9: $class(v) \leftarrow$ ClassDetermination2$(C, C_{tmp}, v, L(v))$.
10: $R' \leftarrow R' \cup \{(v, class(v))\}$.
11: $C_{tmp} \leftarrow C_{tmp} \cup \{class(v)\}$.
12: $C \leftarrow C \cup \{class(v)\}$.
13: **end for**
14: **return** $R'$.
15: **end procedure**

**Algorithm 5** Class Determination in Conflict Resolution

1: **procedure** ClassDetermination2$(C, C', v, L(v))$
2: **for each** class $c_i \in C'$ **do**
3: Calculate $U(C, v, c_i)$.
4: **end for**
5: Let $c_v$ be the new class having the same set of outgoing edges as $v$ (edges having the same label are merged into one).
6: Calculate $U(C, v, c_v)$.
7: Let $class(v)$ be the class such that the value of $U$ is the highest among $C' \cup \{c_v\}$.
8: **return** $class(v)$.
9: **end procedure**

Note that we recalculate $c_v$ to obtain the utility value from the latest schema because the schema is updated and the resulting utility value may be different. Now the class extraction of $v$ is completed. Each time the class is selected, we update $C$, $C_{tmp}$, $|nodes(c, l)|$ for each $c \in C$ and each $c \in C_{tmp}$, and $|nodes(C, l)|$ in main memory assuming that the node $v$ belongs to $class(v)$ in lines 10 to 12. Extracting the class of every node in $V_{one}$ completely, we return $R'$ and back to Algorithm 3.

### 3.4 Edge Extraction

In this edge extraction step, we replace nodes in the input graph file by their extracted classes. To do that sequentially, first we use file 1' and schema_classes, and create an intermediate file, in which source nodes are replaced by their classes (Phase 1). Then by using the intermediate file and schema_classes, we replace target nodes by their classes (Phase 2).

Algorithm 6 shows the procedure of the edge extraction phase 1. file 1' is a sequence of triples (*source*, *label*, *target*), and the edge extraction phase 1 is done by replacing *source*

**Algorithm 6** Edge Extraction Phase 1

**Input:** file 1' and schema_classes
**Output:** tmp_file1 ▷ sources are replaced by their class

Read a line from schema_classes. Let $v_s$ be the node and $class(v_s)$ be the class of the line.
**while** file 1' does not reach EOF **do**
  Read a line from file 1'. Let $v, l, u$ be the source, the label, and the target of the line, respectively.
  **if** $v = v_s$ **then**
    Add a triple $(u, l, class(v_s))$ to tmp_file1 ($u$ is replaced by "leaf" if the target $u$ is a leaf node).
  **else**
    Read schema_classes sequentially and find a line $(v_s, class(v_s))$ such that $v_s = v$.
    Add a triple $(u, l, class(v_s))$ to tmp_file1 ($u$ is replaced by "leaf" if the target $u$ is a leaf node).
  **end if**
**end while**

of each triple by $class(source)$. Since file 1' is sorted, the edges having the same source appear consecutively in file 1', which enables source nodes to be replaced consecutively. Let $v$ be the source node of the "current" edge read from file 1', and suppose that $class(v)$ is obtained from schema_classes. Then we can replace the source of every edge whose source is $v$ by $class(v)$, which can be done by a sequential read from file 1'.

Algorithm 7 shows the procedure of the edge extraction phase 2. The phase 2 replaces the target node of each edge obtained in Phase 1 by the class of the target node. Actually, this phase is done in a similar way to the phase 1.

### 3.5 I/O Cost

We consider the I/O cost of our algorithm. Let $G = (V, E)$ be a graph. We assume that data is transferred between external memory and main memory in blocks of size $B$. $\mathcal{O}(sort(|E|))$ represents the I/O complexity of external merge sort. The I/O cost of each step is as follows.

(1) **Preprocessing**

Sorting file 1 externally: $\mathcal{O}(sort(|E|))$

(2) **Class Extraction**

(a) Reading file 1': $\mathcal{O}(|E|/B)$

(b) Writing pairs of a node and its class to schema_classes: $\mathcal{O}(|V|/B)$

(3) **Edge Extraction Phase 1**

(a) Reading file 1': $\mathcal{O}(|E|/B)$

(b) Reading schema_classes: $\mathcal{O}(|V|/B)$

(c) Writing outgoing edges to tmp_file 1: $\mathcal{O}(|E|/B)$

(4) **Edge Extraction Phase 2**

(a) Sorting tmp_file 1 externally: $\mathcal{O}(sort(|E|))$

(b) Reading tmp_file 1': $\mathcal{O}(|E|/B)$

(c) Reading schema_classes: $\mathcal{O}(|V|/B)$

**Algorithm 7** Edge Extraction Phase 2

**Input:** tmp_file 1, schema_classes
**Output:** schema_edges

Sort tmp_file 1. Let tmp_file 1' be the resulting file.
Read a line from schema_classes. Let $v_t$ be the node and $class(v_t)$ be the class of the line.
Read a line from tmp_file 1'. Let $u, l, c_s$ be the target, the label, and the source of the line, respectively.
**while** tmp_file 1' does not reach EOF **do**
  **if** $u =$ "leaf" **then** ▷ $u$ is a text leaf node
    Add a triple $(c_s, l, LEAF)$ to schema_edges.
    Read a line from tmp_file 1'. Let $u, l, c_s$ be the target, the label, and the source of the line, respectively.
  **else if** $u < v_t$ **then** ▷ $u$ is a non-text leaf node
    Add a triple $(c_s, l, LEAF2)$ to schema_edges.
    Read a line from tmp_file 1'. Let $u, l, c_s$ be the target, the label, and the source of the line, respectively.
  **else if** $u > v_t$ **then**
    Read a line from schema_classes. Let $v_t$ be the node and $class(v_t)$ be the class of the line.
  **else if** $u = v_t$ **then**
    Add a triple $(c_s, l, class(v_t))$ to schema_edges.
    Read a line from tmp_file 1'. Let $u, l, c_s$ be the target, the label, and the source of the line, respectively.
  **end if**
**end while**

(d) Writing edges to schema_edges: $\mathcal{O}(|E|/B)$

Thus, the I/O cost of our algorithm is as follows.

$$\mathcal{O}\left(\frac{|E|}{B} + \frac{|V|}{B} + sort(|E|)\right) = \mathcal{O}\left(\frac{|V|}{B} + sort(|E|)\right)$$

The external R-way merge sort algorithm is an efficient algorithm for sorting large files externally, and we have a number of implementations of the algorithm, e.g., UNIX sort. Therefore, the above estimation suggests that our algorithm extracts a schema from a large graph efficiently, if only such commands are available.

## 4. Evaluation Experiment

In this section, we present experimental results on our algorithm. The algorithm was implemented in Ruby 2.4.2. The parallelized class extraction was implemented by Ruby Gem parallel (version 1.12.0)(注1). All the evaluation experiments were executed on a machine with Intel Xeon E5-2623 v3 3.0GHz CPU, 16GB RAM, 2TB SATA HDD, and Linux CentOS 7 64bit. We used the sort command (GNU coreutils 8.22) in order to sort files externally in the preprocessing and the edge extraction, and we limited the maximum memory usage of the sort command to 1GB by using option "-S".

In our experiments, we use the following two datasets.

(注1)：https://github.com/grosser/parallel

Table 1 Graphs generated by SP$^2$Bench

| $\|E\|$ | $\|V^*\|$ | $\|L\|$ | size (GB) |
|---|---|---|---|
| 100,073 | 19,369 | 24 | 0.01 |
| 1,000,009 | 187,066 | 24 | 0.10 |
| 10,000,457 | 1,730,250 | 26 | 1.04 |
| 100,000,380 | 17,823,525 | 26 | 10.35 |

Table 2 Graphs from DBPedia

| $\|E\|$ | $\|V^*\|$ | $\|L\|$ | size (GB) |
|---|---|---|---|
| 50,000 | 1,077 | 2,772 | 0.01 |
| 15,373,833 | 313,036 | 14,130 | 2.72 |
| 76,868,920 | 1,177,165 | 22,147 | 12.80 |
| 153,737,783 | 1,457,983 | 23,343 | 25.11 |

**SP$^2$Bench** [12] (SP2B, for short) is a benchmark tool and generates RDF (N-Triples) files based on DBLP, a computer science bibliography. We generate four graphs of different sizes in Table 1. In the following, by $V^*$, we mean the non-leaf nodes in a set $V$ of nodes. Thus classes of nodes in $V^*$ are extracted.

The total number of unique RDF types is 12. Nodes whose RDF type is "Article" are the largest number of nodes. Note that we regard every edge label *rdf:_i* as the same regardless of the value of $i$, since the number $i$ is not important.

The reason why we use this tool is that (1) it has its explicit schema and thus we can compare the schema and the schema extracted by our algorithm and (2) the tool generates graphs of various sizes, which is useful to investigating the performance of our algorithm.

**DBPedia** project extracts structured data from Wikipedia. Among the real-world graphs, it is one of the datasets with the largest number of unique edge labels. We downloaded three benchmark dataset graphs (注2) and created another graph with $|E| = 50,000$, which is the first 50,000 lines of the smallest graph of the three. Thus we use the four RDF (N-Triples) graphs in Table 2. The total number of unique RDF types in the graph with $|E| = 15,373,833$ is 54,736.

In the following, first we give the evaluation of the class extraction since this is the most complex and time-consuming process. Then, we give the evaluation of the preprocessing and the edge extraction. To evaluate the class extraction quality, we introduce two scores *Score1* and *Score2*, based on RDF types assigned to each node. SP2B is an RDF benchmark tool and each non-leaf node in graphs generated by SP2B has one RDF type. On the other hand, each non-leaf node in DBPedia has one or more RDF types. In the following definition, two particular classes LEAF and LEAF2, which leaf nodes belong to, are omitted.

*Score1* becomes larger as extracted classes contain smaller numbers of different types. By $types(v)$ we mean the set of types assigned to $v$. The set of nodes having type $t$ in class $c$ is denoted $nodes(t, c)$. Then $Score1$ is defined as follows.

$$Score1 = \frac{1}{|V^*|} \sum_{v \in V^*} \frac{1}{|types(v)|} \sum_{t \in types(v)} \frac{|nodes(t, class(v))|}{|class(v)|}.$$

*Score2* becomes larger as a type is distributed to smaller numbers of different classes. Let $total(t)$ be the total number of nodes having type $t$, and let $max(t) = \max_c nodes(t, c)$. Then $Score2$ is the mean of ratio of the two, that is,

$$Score2 = \frac{1}{|T|} \sum_{t \in T} \frac{max(t)}{total(t)}.$$

Thus, the more nodes having type $t$ are grouped into the same class, the higher $Score2$ is.

Firstly, we give the evaluation of the parallelization. We measured the execution time and the memory usage of the class extraction algorithm (non-parallel version and the parallelized version), and calculated the class extraction scores. In this experiment, we set the parameter $\alpha$ to 1 in light utility function and the parallel number $k$ to 4.

Tables 3 and 4 show the results. Each row whose parallel number is 1 represents a result by the non-parallel class

Table 3 Execution time of the class extraction (non-parallel and parallelized versions) for SP2B graphs

| SP2B | $\|E\|$ | | |
|---|---|---|---|
| parallel number $k$ | 1,000,009 | 10,000,457 | 100,000,380 |
| Light $k = 1$ (non-parallel) | 6.70 | 64.23 | 651.07 |
| Light $k = 4$ (parallelized) | 209.31 | 1,669.85 | 19,886.80 |
| Original $k = 1$ (non-parallel) | 13.26 | 111.43 | 1065.99 |

Table 4 Execution time of the class extraction (non-parallel and parallelized versions) for DBPedia graphs

| DBPedia | $\|E\|$ | | |
|---|---|---|---|
| parallel number $k$ | 15,373,833 | 76,868,920 | 153,737,783 |
| Light $k = 1$ (non-parallel) | 11,242.80 | 110,555.53 | 150,143.69 |
| Light $k = 4$ (parallelized) | 4,803.85 | 42,071.79 | 58,026.58 |
| Original $k = 1$ (non-parallel) | - | - | - |

Table 5 Class extraction scores for the parallelized and non-parallel version ($\alpha = 1$) for the SP2B graph with $|E| = 10,000,457$

| parallel number | $\|C\|$ | Score 1 | Score 2 | Mean |
|---|---|---|---|---|
| 1 (non-parallel) | 3 | 72.53 | 100.00 | 86.26 |
| 4 (parallelized) | 3 | 72.53 | 100.00 | 86.26 |

Table 6 Class extraction scores for the parallelized and non-parallel version ($\alpha = 1$) for the DBPedia graph with $|E| = 15,373,833$

| parallel number | $\|C\|$ | Score 1 | Score 2 | Mean |
|---|---|---|---|---|
| 1 (non-parallel) | 1,327 | 70.06 | 76.97 | 73.51 |
| 4 (parallelized) | 1,309 | 70.60 | 76.42 | 73.51 |

(注2) : http://benchmark.dbpedia.org/

Table 7 Memory usage of the class extraction algorithm

| Dataset | non-parallel | parallelized |
|---|---|---|
| SP2B ($\|E\| = 100,000,380$) | 11.1 MB | 7.5 MB |
| DBPedia ($\|E\| = 153,737,783$) | 116.6MB | 89.6 MB |

extraction algorithm. Each execution time is in seconds. Table 3 shows the execution time of the class extraction (non-parallel and parallelized versions) for SP2B graphs of different sizes. This result can be described as follows. In SP2B, (1) the execution time is almost linear to the number of edges $|E|$. The execution time is also almost linear to the number of non-leaf nodes $|V^*|$ since $|V^*|$ is proportional to $|E|$ in SP2B. (2) The parallelization makes the execution time significantly slow. Since the number of classes extracted for SP2B is significantly smaller than DBPedia (details are presented below) and the cost of calculating the light utility function for each node is considerably small, the overhead of parallelization is relatively large. Therefore, the execution time of the parallelized version increased due to the parallelization cost.

Table 4 shows the execution time of the class extraction (non-parallel and parallelized versions) for DBPedia graphs of different sizes. This result can be described as follows. (1) The execution time of DBPedia is much longer than that of SP2B since $|L|$ is relatively large in DBPedia and thus the number of extracted classes $|C|$ greatly increases. (2) At first, the growth rate of execution time of DBPedia rapidly grows compared to that of $|V^*|$. As the size of graph grows, the growth rate of execution time is getting closer to that of $|V^*|$. (1) and (2) suggest that $|L|$ and $|V^*|$ mostly affect the execution time of our algorithm. (3) The parallelized version is more than two times faster than the non-parallel version in DBPedia. Since the calculation cost of the utility function is considerably large, the parallelization is effective to make the execution time shortened.

Tables 5 and 6 show $|C|$ and the scores for both versions. This result shows that parallelization does not affect $|C|$ and the scores, thus we have almost no difference in the class extraction quality.

We also measured the memory usage of the class extraction algorithm. Table 7 shows the result. For the SP2B graph with $|E| = 100,000,380$, maximum memory usage of the parallelized version is about 7.5MB. On the other hand, that of the non-parallel version is 11.1MB. We also measured the memory usage for the DBPedia graph with $|E| = 153,737,783$. Maximum memory usage of the parallelized version is about 89.6MB. On the other hand, that of the non-parallel version is about 116.6MB. These results show that the parallelized version was about 10-20 percent less memory usage than non-parallel version. Thus, our class extraction algorithm is completed with sufficiently small memory usage to the input graph file.

Table 8 Class extraction scores for the SP2B graph with $|E| = 10,000,457$

| utility function | $\|C\|$ | Score 1 | Score 2 | Mean |
|---|---|---|---|---|
| Light($\alpha = 1$) | 3 | 72.53 | 100.00 | 86.26 |
| Light($\alpha = 10$) | 22 | 99.45 | 88.83 | 94.14 |
| Original | 6 | 97.02 | 95.32 | 96.17 |

Table 9 Class extraction scores for DBPedia graph with $|E| = 50,000$

| utility function | $\|C\|$ | Score 1 | Score 2 | Mean |
|---|---|---|---|---|
| Light($\alpha = 1$) | 179 | 67.30 | 73.87 | 70.58 |
| Light($\alpha = 10$) | 687 | 89.74 | 27.77 | 58.76 |
| Original | 81 | 43.07 | 81.69 | 62.38 |

Table 10 Class extraction scores for DBPedia graph with $|E| = 15,373,833$

| utility function | $\|C\|$ | Score 1 | Score 2 | Mean |
|---|---|---|---|---|
| Light($\alpha = 1$) | 1,327 | 70.06 | 76.97 | 73.51 |
| Light($\alpha = 10$) | 48,631 | 85.12 | 3.77 | 44.44 |
| Original | - | - | - | - |

Secondly, we compare our light utility function and the original utility function. We also examined how parameter $\alpha$ in our light utility function affects the class extraction scores. Let us make a comparison of the following three cases: extracting classes with (1) the original utility function [5], (2) our light utility function with $\alpha = 1$, and (3) $\alpha = 10$. We ignore incoming edge labels in all the cases. In this experiment, we use parallel number $k = 1$. Tables 8, 9, and 10 show the results. Table 8 shows the class extraction scores and the number of classes $|C|$ for the SP2B graph with $|E| = 10,000,457$. The result shows that both of utility functions achieve high scores. The reason why such high scores are obtained is that the graphs were generated the benchmark tool so $|L|$ is small and nodes having the same RDF type have a similar set of labels.

Tables 9 and 10 show the class extraction scores and the number of classes $|C|$ for the DBPedia graph with $|E| = 50,000$ and $|E| = 15,373,833$, respectively. The maximum mean of score 70.58 is obtained at $\alpha = 1$, which is higher than the value 62.38 obtained by the original. We could not extract schemas from DBPedia graph with $|E| = 15,373,833$ with the original utility function within reasonable time (24h) because the computation cost of the function is too high.

Overall, the above results suggest that the parameter $\alpha$ introduced in our light utility function works effectively for extracting classes from graphs having fewer unique edge labels such as SP2B. As shown in the tables, our light utility

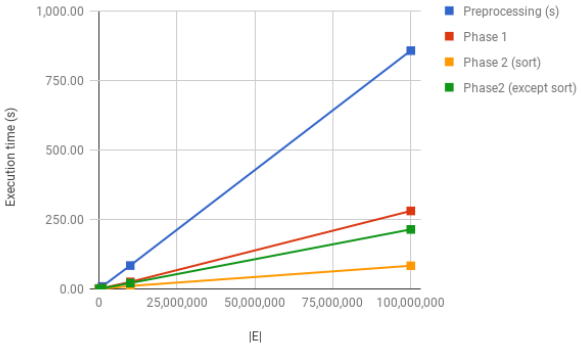


Figure 2 Execution time of the preprocessing and the edge extraction (SP2B)

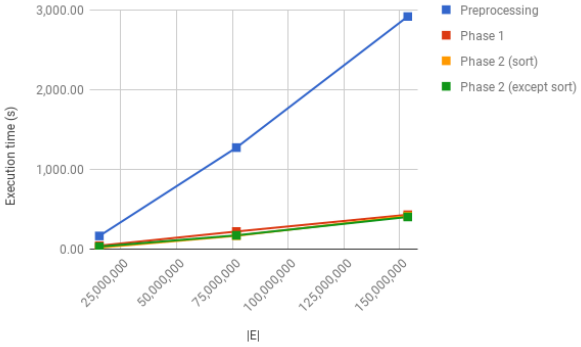


Figure 3 Execution time of the preprocessing and the edge extraction (DBPedia)

function can extract appropriate schemas efficiently.

Finally, we give the evaluation of the preprocessing and the edge extraction. In this experiment, we used the input files of the edge extraction algorithm obtained by the class extraction algorithm with the parallel number $k = 1$ for SP2B and $k = 4$ for DBPedia. We measured the execution time of the preprocessing and the edge extraction. Figs. 2 and 3 plot the execution time of each step. This result means that the execution time of the preprocessing and the edge extraction algorithm are almost linear to $|E|$.

We also measured the memory usage of each step. We observed that each edge extraction step except sorting was executed under 10MB memory usage. External sorting in the preprocessing and the edge extraction step is the most memory consuming step, and its maximum memory usage is about 1.1GB since we limit the maximum memory usage of the sort command to 1GB. Thus, the memory usage of the schema extraction mostly depends on external sorting.

## 5. Conclusion

In this paper, we proposed an external memory algorithm for extracting a schema from a graph using parallel processing and our novel utility function. Experimental results suggest that our algorithm can extract schemas more efficiently and appropriately than the previous utility function, that the parallelization of the class extraction makes the execution time more than two times faster for DBPedia, and that the memory usage of the schema extraction mostly depends on external sorting.

## Acknowledgment

This work was supported by JSPS KAKENHI Grant Number JP17K00150.

## References

[1] M.F. Fernandez and D. Suciu, “Optimizing regular path expressions using graph schemas,” Proceedings of the Fourteenth International Conference on Data Engineering (ICDE 1998), pp.14–23, 1998.

[2] R. Goldman and J. Widom, “Dataguides: Enabling query formulation and optimization in semistructured databases,” Technical Report 1997-50, Stanford InfoLab, 1997.

[3] R. Goldman and J. Widom, “Approximate Dataguides,” Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, pp.436–445, 1999.

[4] S. Nestorov, S. Abiteboul, and R. Motwani, “Extracting schema from semistructured data,” Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1998), pp.295–306, 1998.

[5] Q.Y. Wang, J.X. Yu, and K.F. Wong, “Approximate graph schema extraction for semi-structured data,” in Proceedings of EDBT 2000, pp.302–316, Springer, 2000.

[6] S. Navlakha, R. Rastogi, and N. Shrivastava, “Graph summarization with bounded error,” Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD 2008), pp.419–432, 2008.

[7] Y. Luo, G.H. Fletcher, J. Hidders, Y. Wu, and P. De Bra, “External memory k-bisimulation reduction of big graphs,” Proc. CIKM 2013, pp.919–928, 2013.

[8] X. Hu, Y. Tao, and C.W. Chung, “Massive graph triangulation,” Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2013), pp.325–336, 2013.

[9] Z. Zhang, J.X. Yu, L. Qin, L. Chang, and X. Lin, “I/O efficient: Computing sccs in massive graphs,” Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2013), pp.181–192, 2013.

[10] Z. Zhang, J.X. Yu, L. Qin, Q. Zhu, and X. Zhou, “I/O cost minimization: Reachability queries processing over massive graphs,” Proceedings of the International Conference on Extending Database Technology (EDBT 2012), pp.468–479, 2012.

[11] N. Suzuki, K. Ikeda, and Y. Kwon, “An algorithm for all-pairs regular path problem on external memory graphs,” IEICE Transactions, vol.99-D, no.4, pp.944–958, 2016.

[12] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, “SP$^2$Bench: a SPARQL Performance Benchmark,” Proceedings of the 25th International Conference on Data Engineering (ICDE 2009), pp.222–233, 2009.