Algorithm used in the chapter "RDF^M: An Alternative Approach for Representing, Storing, and Maintaining Meta-knowledge"

May 7, 2021

1 Algorithms for complete execution of program

The total algorithm comprises of 5 separated algorithms divided into *3* steps. The first step i.e., Algorithm 1, converts RDF^M data into JSON data. The second step i.e., Algorithm 2, creates *Compress_JSON and Decompress_JSON* files which compress and decompress the given dataset to reduce memory usage required to process it. The Algorithm 2 also splits the dataset as per predicate. The additional dictionaries for parameters such as subject, object, statement_id, fuzzy or trust value, timestamp, the time interval (start time), time interval (end time), NMK, and graph_id are created to have faster query-processing. In the algorithm the above mentioned parameters are represented as *sub, obj, uid1, param1, param2, param31, param32, param4, uid2*, respectively. The final step, i.e., Algorithms 3, 4, and 5, are used to process *user_queries*. Algorithm 3 asks the user to enter a query to process the dataset. It reads a given predicates from predicate_dictionary files along with compression-index and decompression-index files and displays the query result. It contains codes to get user queries from the console and passes them to Algorithm 4 for processing, it also stores the result in comma-separated format. Algorithm 5 shows the matching procedure for the queries where a) subject and object both variable, b) subject given and object variable (the whole Algorithm is given in pdf [1]). Two sets of dictionaries are maintained i.e., result and current. Current dictionaries set is initialized for each user query and its intersection or union is carried out with result dictionaries set. The terms are matched upon their location w.r.t. the previous encounters in user queries. For example to process the query

Select * where $\{P1[,,(,),](?s,?o,?i,?i1), P2[?c,,(,),](?s,?o1,?i2,?i3)\}$, we need to go through the following steps.

First The user query is compressed using the compression-index file.

Second Sort the queries based on the number of subjects and objects connected with the predicates *P1 and P2*. Suppose for this query predicates are sorted in [*P1, P2*] order.

Third Dictionary files, for each predicate, *P2 and P1* are loaded into the memory.

Forth These predicate dictionaries are used to form the result as per a user query. Suppose, after processing *P2*, we retrieve { S1:(O1, U1, 0.8,...), (O2, U2, 0.7,...), S2:(O3, U3, 0,...)}. And after processing *P1*, we retrieve { S1:(O4, U4, 0,...), S3:(O5, U5, 0,...)}.

[1] https://github.com/sangeetadas123/Algorithm_for_matching_function

Fifth After matching process, result will contain { S1:(O1, U1, 0.8,...), (O2, U2, 0.7,...), (O4, U4, 0,...)} —— i.e. the key S2 is removed and (O4, U4, 0,...) is added under S1 to form the new result. }

Sixth Result is decompressed using decompression-index file and displayed.

Complexity of Algorithms For Algorithms 1 and 2, suppose there are **n** numbers of statements in the dataset, and for each statement, it took constant time to execute. So, the complexity of both algorithms is $\mathcal{O}(\mathbf{n})$. For Algorithms 3 and 4, suppose the query length of a particular query is **n** and for each query pattern it took constant time to process. So, the complexity of both algorithms is $\mathcal{O}(\mathbf{n})$. For Algorithm 5, suppose there is **n** number of query patterns present in a query and each query pattern takes **m** number of iterations to execute the query pattern. So, the complexity of the algorithm is $\mathcal{O}(n * m)$, where **m** is the length of predicate dictionary.

Algorithm 1 Step 1: Conversion of RDF^M format file into json file format

Input: Input file in RDF^M format.
Output: Json file containing dataset in predicate dictionary of the format.
pred:[sub, obj, uid1, param1, param2, param31, param32, param4, uid2, NMK]

procedure CONVERT RDF^M FORMAT TO *JSON* FILE FORMAT
  Read the input file and store the contents in *input_data*.
  Initialize *pred* as empty dictionary.
  for each *line* in *input_data* **do**
    Split the *line* into various terms.
    Identify the various terms as *subject, predicate, object, uid, certainty, timestamp,time interval, NMK value.*
    Append *pred*[*predicate*] with [*subject, object, uid,...*]
  end for
  Write the *pred* dictionary to json file.
end procedure

Algorithm 2 Step 2: Creation of Compressed and Decompressed files , and Dictionaries

Input: Input file containing dataset in predicate dictionary of the format.
pred:[sub, obj, uid1, param1, param2, param31, param32, param4, uid2]
Output: Compressed and Decompressed dictionary files, one for each predicate along with compressed-index and decompressed-index files.

procedure CREATE *Compress_JSON and Decompress_JSON* AND *Create_Dicts* FILES
  Initialization: a counter variable (ctr) to 40000. Create two dictionaries; One for compression-index file, another for decompression-index file
  Read input as DATA
  for each triple in DATA **do**
    Check
    if subject is new **then**
      Replace subject with ctr value and add subject:ctr in compression-index dict.
      Add ctr:subject in decompression-index dict.
    end if
    if predicate is new **then**
      predicate:predicate in compression-index dict
      Add predicate:predicate in decompression-index dict.
    end if
    if object is new **then**
      Replace object with ctr value and add object:ctr in compression-index dict.
      Add ctr:object in decompression-index dict.
    end if
    if uid1 is new **then**
      Replace uid1 with ctr value and add uid1:ctr in compression-index dict.
      Add ctr:uid1 in decompression-index dict.
    end if
    if uid2 is new **then**
      Replace uid2 with ctr value and add uid2:ctr in compression-index dict.
      Add ctr:uid2 in decompression-index dict.
    end if
    Append to sub, obj, uid1, param1, param2, param31, param32, param4 and uid2 dictionaries.
  end for
  Write compression-index_file
  Write decompression-index_file.
  Write the predicate dictionaries to a unique file dictionaries_¡predicate¿.
end procedure

Algorithm 3 Step 3: Process user_queries

Input: compression-index-file, decompression-index file and predicate-dictionaries file
Output: Result as per user query, stored in csv format.

```
1: procedure PROCESS user_queries
2:     Display choice selection MENU:
3:     for Insertion do
4:         Read triple entered by user.
5:         Create predicate dictionaries for given triple.
6:         Open dictionary file corresponding to the entered predicate.
7:         if File exist then
8:             Append triple into file
9:         else Create new dictionary file for predicate and append into it.
10:        end if
11:    end for
12:    for Search do
13:        Read user query from console.
14:        Compress user query by loading compression-index_file.
15:        Replace user given sub, obj with their indexes.
16:        Sort user queries:
17:        for each predicate in user query do
18:            Read dictionary_ < predicate >
19:            Assign weight to predicate by length of dictionary_¡predicate¿
20:        end for
21:        Call bubble_sort() with weights and user query as function parameters.
22:        Call Process_query()
23:    end for
24:    for ASK query do
25:        All the steps in search as same.
26:        Except no need to decompress result table.
27:        Check result table
28:        if non-empty - Display YES then
29:        else empty - Display NO
30:        end if
31:    end for
32:    for CONSTRUCT query do
33:        Extract user queries from WHERE part of CONSTRUCT query.
34:        Pass the user queries as search to Process_query()
35:        All the steps in search as same.
36:        Convert the result received from Process_query() to CONSTRUCT format.
37:        Display the result.
38:    end for
39: end procedure
```

Algorithm 4 Step 3.1: Process_query

Input: Dataset and user-query read by Algorithm 3
Output: Returns the result of query on the given dataset.

```
1: procedure EXECUTE user_queries
2:     Display choice selection MENU:
3:     for each line in user_query do
4:         call matching_function(line)
5:     end for
6:     Preparing display.
7:     for each variable in SELECT line do
8:         if variable is in result table then
9:             Check yes or no
10:            if yes then
11:                copy result table triple.
12:            end if
13:            if no then
14:                skip.
15:            end if
16:        end if
17:    end for
18:    Return the result to search() function.
19:    Read decompression-index file for decompressing result table.
20:    for each entry in result table do
21:        Replace entry with its decompression-index.
22:        Display result to user.
23:    end for
24: end procedure
```

Algorithm 5 Step 3.2: matching_function

Input: line send by Algorithm 4.
Output: Result send back to Algorithm 4.

```
1: procedure EXECUTE line containing user_query
2:     for each line do
3:         if sub is variable, obj is variable then.
4:             if subject variable is new then
5:                 Append the contents of sub-dict of result-table with that of sub-dict of current-table. And Add variable to list in vars['sub']
6:             end if
7:             if sub is variable and is previously encountered then
8:                 Find the position of previous encounter and select the associated dictionary.
9:                 for each key in associated-dict do
10:                    if key exists in current table's sub-dict then
11:                        Make union of current-table's sub-dict[key] and result table's associated-dict[key].
12:                        Store it as value of associated-dict[key].
13:                    end if
14:                    if key not found in sub-dict of current-table then
15:                        Delete the key from associated-dict of result-table.
16:                    end if
17:                end for
18:            end if
19:            if object variable is new then
20:                Append the contents of obj-dict of result-table with that of obj-dict of current-table.
21:                Add variable to list in vars['obj']
22:            end if
23:            if obj is variable and is previously encountered then
24:                Find the position of previous encounter and select the associated dictionary.
25:                for each key in associated-dict do
26:                    if key exists in current table's obj-dict then
27:                        Make union of current-table's obj-dict[key] and result table's associated-dict[key].
28:                        Store it as value of associated-dict[key].
29:                    end if
30:                    if key not found in obj-dict of current-table then
31:                        Delete the key from associated-dict of result-table.
32:                    end if
33:                end for
34:            end if
35:        end if
36:        if sub is given, obj is variable then
37:            Keep the matching given sub in result-table's sub-dict.
38:            for each non-matching key in sub-dict do
39:                Remove subject from obj-dict of result-table.
40:                Remove object from uid-dict of result-table.
41:            end for
42:            Remove the non-matching keys.
43:            if object variable is new then
44:                Append the contents of obj-dict of result-table with that of obj-dict of current-table and add variable to list in vars['obj']
45:            end if
46:            if obj is variable and is previously encountered then
47:                Find the position of previous encounter and select the associated dictionary.
48:                for each key in associated-dict do
49:                    if key exists in current table's obj-dict then
50:                        Make union of current-table's obj-dict[key] and result table's associated-dict[key].
51:                        Store it as value of associated-dict[key].
52:                    end if
53:                    if key not found in obj-dict of current-table then
54:                        Delete the key from associated-dict of result-table.
55:                    end if
56:                end for
57:            end if
58:        end if
59:    end for
60: end procedure
```