Study of Subgraph Isomorphism and Java Implementation of QuickSI Algorithm

Ajinkya A. Kale (aak9225@rit.edu)

Prof. Carlos R. Rivero

05/16/2016
Contents

I. ABSTRACT
II. MOTIVATION
III. INTRODUCTION
IV. SUBGRAPH ISOMORPHISM
V. MINIMUM SPANNING TREE
VI. QI-SEQUENCE
   1) Pseudo Entries
   2) QI-Sequence Algorithm
VII. EFFECTIVE QI-SEQUENCE
   1) Preprocessing
   2) Finding Minimum Spanning Tree
      a) Selectfirstedge Algorithm
      b) Selectspanningedge Algorithm
VIII. QuickSI ALGORITHM
   1) Example of Quick SI with Effective QI-Sequence
      a) Simulation of QuickSI algorithm
IX. RESULTS AND EXPERIMENTS
X. CONCLUSION AND FUTURE WORK
XI. REFERENCES
Figures

1) Fig 1 – Example graph for minimum spanning tree
2) Fig 2 – solution for minimum spanning tree
3) Fig 3 - Mesh graph
4) Fig 4 – Query graph
5) Fig 5 – Data graph 1
6) Fig 6 – Data graph 2
7) Fig 7 – Recursion tree for QuickSI

Tables

1) Table 1 minimum spanning tree solution
2) Table 2 QI-Sequence for mesh graph Fig.3
3) Table 3 QI-Sequence for query graph Fig 4
4) Table 4 Effective QI-Sequence for query graph 4
I. ABSTRACT:

Graph is a data structure that stores the data and its relationships. It models any type of complex objects, and maintains its topology. Molecular structures of a chemical compound are the most sophisticated example of graph, where the structure and its topological information can be modeled and its relationship can be represented with graph. Other examples include social networks, biological structures, software engineering, chemistry etc. To perform operations on graph it requires efficient and meticulous algorithms. In this capstone project I aimed to solve the problem of exact subgraph Isomorphism (matching) of a query graph on graph database, and developed a Java implementation of QuickSI algorithm.

II. MOTIVATION:

In many real world applications, where graphs are used to model the data and its problem, there is a requirement for an efficient and effective algorithm to solve the graph containment problem. For example, in field of biochemistry, given a molecular structure of a chemical compound ‘X’, how many other compounds contains the exact molecular structure of ‘X’.

A naive solution to this problem is to explore all the nodes of data graph following the random sequence. The computation cost for this approach is very expensive and so the problem is categorized as NP-hard problem. To overcome this computation cost, the authors of paper [1] has presented a novel technique and an algorithm to find exact matching of query graph.

III. INTRODUCTION:

The algorithm to find subgraph Isomorphism are based on filter and verification approach. The filtering phase collects all the possible matchings relevant to the query and verification phase finds the exact matching for the query. The traditional algorithms explore in all the directions of the data graph in search of mapping which in turn increases the search space. As the graphs grows in size, the computation cost increase significantly. Moreover, it doesn’t use any heuristics or structural, topological information of a query graph. The motivation is to improvise on this approach and develop a technique which guides the matching algorithm in the correct direction and reduce the search space, rather than exploring all the possibilities. Based on this research and analysis, authors have proposed an QuickSI algorithm and an optimal search sequence technique for matching.
IV. SUBGRAPH ISOMORPHISM:

It is a graph containment problem, which extracts the graphs from a graph database, on a constraint that extracted graphs contain the query graph i.e. query graph is subgraph of the extracted graphs. To assert that a data graph contains the subgraph, it must have vertices greater than or equal to query graph vertices and exhibits the common properties, such as bipartiteness and bijection mapping as invariant.

V. MINIMUM SPANNING TREE:

Spanning tree of graph covers all the vertex and edges such that no cycle exists and total cost of considered edges is minimum. This concept is used in generating the sequence. Following is an example that generates the Minimum spanning tree for graph in Fig 1

```
Fig. 1
```

Following are the edges considered while generating minimum spanning tree and their weights.
Fig. 2 Spanning Tree for Fig. 1

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC</td>
<td>2</td>
</tr>
<tr>
<td>CF</td>
<td>3</td>
</tr>
<tr>
<td>CA</td>
<td>4</td>
</tr>
<tr>
<td>AB</td>
<td>1</td>
</tr>
<tr>
<td>ED</td>
<td>5</td>
</tr>
<tr>
<td>GH</td>
<td>3</td>
</tr>
<tr>
<td>FI</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 1
VI. QI-SEQUENCE:

QI Sequence represents a search order and topological information about the query graph, this sequence is a randomized sequence. There could be $n$ possible QI-Sequence, where $n$ is the number of vertices, and each vertex can be starting vertex in the algorithm. To generate this sequence, a spanning tree of the query graph is calculated by using depth first search algorithm starting at a random vertex. At each iteration, algorithm maintains the parent of the visited vertex.

1) Pseudo Entries:
This entry record the information missed by the algorithm. Since spanning trees does not contain cycles, it is guaranteed to miss an edge. To incorporate this missing edge we record a pseudo edge [ R: Edge]. The sequence also contains a degree of the visited vertex [R: Deg] only if it has more than 2 edges. The rationale behind this is, one of the edges will be the edge from its parent and other edge would lead to next vertex, which is the only available path. But if the vertex has more than 2 edges, than we have multiple paths to explore. This information is maintained in order to indicate the QuickSI algorithm to find the candidate vertex that forms an edge equivalent to pseudo edge and degree. In other words, this information represents the properties of the particular vertex and edges it forms.

2) QI-Sequence Algorithm:
Generate_QI_Sequence (G (V, E)):
   S: Stack
   E: Hashset of Edges
   V: Hashset of Vertices
   PrevN: previous visited node

   While S is not empty:
      N = pop S
      If N is not seen:
         add edge between PrevN and N to the E
         Foreach vertex v belongs to and neighbors of N:
            If not visited:
               Add v to V
            Else:
               Add edge between v and N to E
      PrevN = N
Following is the graph “mesh” created in order to test the algorithm

![Fig. 3 mesh graph](image)

<table>
<thead>
<tr>
<th>Type</th>
<th>T.i.p</th>
<th>T.i.1</th>
<th>T.i.v</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T0</td>
<td>A</td>
<td>V1</td>
</tr>
<tr>
<td>T2</td>
<td>T1</td>
<td>B</td>
<td>V2</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td>Deg : 3</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>T2</td>
<td>E</td>
<td>V5</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td>Deg:4</td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>T3</td>
<td>H</td>
<td>V8</td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td>Deg:3</td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>T4</td>
<td>I</td>
<td>V9</td>
</tr>
<tr>
<td>T6</td>
<td>T5</td>
<td>F</td>
<td>V6</td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td>Deg:3</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td>Edge: V6 – V5</td>
<td></td>
</tr>
<tr>
<td>T7</td>
<td>T6</td>
<td>C</td>
<td>V3</td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td>Edge: V2 - V3</td>
<td></td>
</tr>
<tr>
<td>T8</td>
<td>T4</td>
<td>G</td>
<td>V7</td>
</tr>
<tr>
<td>T9</td>
<td>T8</td>
<td>D</td>
<td>V4</td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td>Edge:V4 – V5, Edge V4 – V1</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td>Deg: 3</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Q1-Sequence for Fig. 3
VII. EFFECTIVE QI-SEQUENCE:

This is an optimal ordering of the vertex in order to perform subgraph matching efficiently. Effective QI-Sequence is generated by finding the spanning tree over the weighted query graph. In this project we have focused on Effective QI-Sequence technique to perform matching with QuickSI algorithm.

1) Preprocessing:
Before finding the spanning tree, we need to convert the graph into weighted graph by adding the average inner supports for vertex and edges. Inner support is defined as, number of isomorphic mappings of query graph to graphs in graph database. To calculate the average inner support of vertex of query graph, we count the total number of mapping of all the query graph vertices with the vertices of graphs in graph databases.

Followings formula is implemented as algorithm that calculates the average inner support for vertex [1]

\[
\text{Inner Support for vertex} = \frac{\text{count of vertex mapped to graph } \in D}{\text{total number of graph } g \text{ that contains vertex } v \text{ and } g \in D}
\]

To calculate the average inner support for the edges of query. Total number of mapped edges are counted. And this count is normalized by counting the number of graphs in graph database that contains that edge.

Following is the formula for counting the inner support for edges [1]

\[
\text{Inner support for edges} = \frac{\text{count of edge e i.e mapped to graph } g \in D}{\text{total number of graph } g \text{ that contains edge } e \text{ and } g \in D}
\]
2) Finding Minimum Spanning Tree:
In order to find the minimum spanning tree, a modified version of Prims algorithm is used. Instead of starting the algorithm from a random vertex, we chose the vertex that has smallest average inner support. If average inner supports are equal, then one with the lesser degree is chosen. After filtering, if there are still more than 1 vertex, then first vertex is randomly chosen.

a) Following is selectfirstedge algorithm [1]

selectfirstedge(P, q^w):
P = set of edges in query graph
q^w = weighted query graph
p' = stores filtered edges

if p > 1:
    p' = \{ e | e \in P \land \exists (e' \in P) (\deg(e.u) + \deg(e.v)) \leq (\deg(e'.u) + \deg(e'.v)) \}

if p' > 1:
    return random edge e from p'

Before the prims algorithms starts, selectfirstedge algorithm is used to get the starting edge. At each iteration of the prims algorithm, an optimal edge is to be chosen from the available search space of edge. The paper presents an algorithm called selectspanningedge, which finds the next optimal edge to construct spanning tree.

The algorithm works in three filtering steps,

1. In first step it considers all the edges in EdgeSet as a candidate edge and filters only those edges whose weights are less than or equal to its own weight.

2. In further step of filtering, a count of edges in induced subgraph formed with each filtered edge is considered. The edge that has largest induced subgraph count is then passed down for further filtering.

3. In third step, degrees of each filtered edge is considered as a filtering parameter. The smallest degree of destination vertex of each edge is
returned. However, if there are multiple edges with same destination vertex degree, then random edge is returned.

b) Following is \textit{selectspanningedge} algorithm

\textit{selectspanningedge}(P,q^w):
\begin{itemize}
  \item \textit{P}: edge set of query graph
  \item \textit{q}^w: weighted query graph
  \item \textit{s} = selected edge
\end{itemize}

\begin{align*}
\alpha &= \text{set of edges filtered in step 1} \\
\beta &= \text{set of edges filtered in step 2} \\
\gamma &= \text{set of edges filtered in step 3} \\
\alpha &= \{e \mid e \in P \land \forall (e' \in P) \exists (w(e) \leq w(e'))\} \\
\text{if } \alpha > 1: & \quad \beta = \{e \mid e \in P \land \forall (e' \in \alpha) \exists |\text{IndG}(V_t \cup e.v)| \geq |\text{IndG}(V_t \cup e')|\} \\
\text{if } \beta > 1: & \quad \gamma = \{e \mid e \in P \land \forall (e' \in \beta) \exists (\text{deg}(e.v) \leq \text{deg}(e'.v))\} \\
\text{if } \gamma > 1: & \quad s = \text{random edge from } \gamma \\text{else:} \\
& \quad s = \text{edge from } \alpha \text{ OR } \beta \\
\text{return } s
\end{align*}
VIII. QuickSI:

QuickSI works on preprocessed search sequence, generated by QI-Sequences or Effective QI-Sequence. The algorithm processes each spanning tree entry “T” one at a time sequentially. It tries to find a bijection mapping of the T entry vertex. In order to get accurate mappings, it maintains two arrays H [ T1, T2, … Td] and F [0,1…n]. Each index of array H stores the mapping for T.i.v, this information crucial and is required to find the parent of T.i, where i is range of 0 to d and d is the length of sequence. F array is used to indicated the visited vertex of data graph g.

This algorithm works in two phases.

i) It generates all the possible candidate vertices of data graph g

ii) It considers each candidate vertex as a potential mapping to T.v and proceeds ahead with next entry in the sequence.

Phase 1:
It considers the vertex set of the data graph, the vertex is a candidate vertex if and only if it has exact same label as that of T.i.v and has an edge with mapped data graph vertex of parent T.p, with an additional constraint that it has exact same edge label as that of edge between T.i and T.p.i, i.e edge between current vertex in sequence and its parent vertex.

Phase 2:
It chooses the vertex from candidate vertex set, and maps it to T.v by storing its value at index of T.v in H array. and proceeds ahead to find the mapping for the next entry in the sequence.
If the sequence in exhausted and partial matching is found, algorithm than backtrack to the previous level in the sequence and consider the next vertex from candidate vertex set. If it finds complete mapping than algorithm stops and concludes that data graph g contains the query graph i.e query graph is subgraph of data graph g.
Following is an example of how QuickSI works.

Fig. 4 query graph
Fig. 5 data graph 1

Fig. 6 data graph 2
Initial step of finding the subgraph isomorphism is to generate the sequence. We first generate the QI-Sequence for the query graph mentioned in fig 4. However, for this project we have considered working with Effective QI-Sequence.

The nodes of the query graph have color code associated with it, in order to visually understand their mappings in data graph.

<table>
<thead>
<tr>
<th>Type</th>
<th>T.i.p</th>
<th>T.i.l</th>
<th>T.v</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T0</td>
<td>A</td>
<td>V0</td>
</tr>
<tr>
<td>T2</td>
<td>T1</td>
<td>B</td>
<td>V1</td>
</tr>
<tr>
<td>T3</td>
<td>T2</td>
<td>C</td>
<td>V2</td>
</tr>
<tr>
<td>T4</td>
<td>T3</td>
<td>E</td>
<td>V3</td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>T4</td>
<td>E</td>
<td>V6</td>
</tr>
<tr>
<td>T6</td>
<td>T4</td>
<td>E</td>
<td>V5</td>
</tr>
<tr>
<td>T7</td>
<td>T4</td>
<td>A</td>
<td>V4</td>
</tr>
<tr>
<td>T8</td>
<td>T1</td>
<td>A</td>
<td>V7</td>
</tr>
<tr>
<td>T9</td>
<td>T8</td>
<td>B</td>
<td>V8</td>
</tr>
<tr>
<td>T10</td>
<td>T9</td>
<td>D</td>
<td>V9</td>
</tr>
<tr>
<td>R10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3 QI sequence for query graph in Fig. 4

Here we present the Effective QI-Sequence that is considered by QuickSI algorithm for the example.

<table>
<thead>
<tr>
<th>Type</th>
<th>T.i.p</th>
<th>T.i.l</th>
<th>T.v</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T0</td>
<td>A</td>
<td>V0</td>
</tr>
<tr>
<td>T2</td>
<td>T1</td>
<td>B</td>
<td>V1</td>
</tr>
<tr>
<td>T3</td>
<td>T2</td>
<td>C</td>
<td>V2</td>
</tr>
<tr>
<td>T4</td>
<td>T3</td>
<td>A</td>
<td>V3</td>
</tr>
<tr>
<td>T5</td>
<td>T4</td>
<td>A</td>
<td>V4</td>
</tr>
<tr>
<td>T6</td>
<td>T4</td>
<td>E</td>
<td>V5</td>
</tr>
<tr>
<td>T7</td>
<td>T6</td>
<td>D</td>
<td>V9</td>
</tr>
<tr>
<td>T8</td>
<td>T7</td>
<td>B</td>
<td>V8</td>
</tr>
<tr>
<td>T9</td>
<td>T1</td>
<td>A</td>
<td>V7</td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T10</td>
<td>T4</td>
<td>E</td>
<td>V6</td>
</tr>
</tbody>
</table>

Table 4 Effective QI Sequence for query graph in Fig. 4
a) Simulation of QuickSI:
The algorithm starts with the vertex V0 of the sequence, and finds the candidate vertices of data graph as u1, u3, u8, u12.
It then considers u1 as potential mapping of V0, and proceeds ahead in the sequence to find the mapping for V1. Possible candidate vertices for V1 mapping are u4, u14.
Algorithm maps V1 to u4 and moves ahead in search of mapping for V2. Similarly, it repeats the process for rest of the entries in sequence.
If algorithm finds all the mapping for the entries in the sequence it concludes that a successful mapping is found. As shown in the fig. 7, the leftmost branch of the recursion tree leads to successful mapping.
If the algorithm fails to find the successful mapping at any given point, it backtracks and chose another vertex from candidate vertex set, to look for mapping in another path. In the fig 7. recursion tree, the algorithm fails to find the mapping of V2 from u14, as a result the algorithm backtracks to level 0 and chose u8 as mapping for V0 and proceeds down the tree.
To investigate matching on data graph 2, algorithm finds 2 possible mapping for V0 and one mapping for V1, but fails to find any mapping for V2. As a result, data graph 2 does not contain query graph.
Following are the results and exacting mapping for the query graph over data graphs.

Complete mapping for data graph 1 = \{0=1, 1=4, 2=7, 3=8, 4=12, 5=9, 9=13, 8=6, 7=3, 6=11\}

Partial mappings for data graph 2 = \{0=0, 1=1\}, \{0=6, 1=1\}

**IX. RESULTS AND EXPERIMENTS:**

The Java implementation of QuickSI was tested on AIDS dataset consisting of 1000 query graph of 24 edges and 1000 data graphs. Parser in Java was written in order to create graph objects from igraph file using JgraphT library. Each query graph was tested on all the data graphs. Every mappings of query graph in data graph was recorded. The total mapping results were 10,050. There are incorrect results for few query graphs because of their complex, trivial structure.

**X. CONCLUSION AND FUTURE WORK:**

A successful implementation of the QuickSI, Effective QI-sequence and QI-sequence algorithms was devised in Java and tested on real dataset. The source code has some minor bugs. I am currently working on the project and fixing the source code bugs. Current implementation is considering the incorrect additional mapping. For example, for query graph 19 testing on data graph 7762, implementation results 2 mappings, however there are 0 mappings, i.e query graph 19 in not a subgraph of data graph 7762.

The future work for this algorithm would be to focus on improvising the efficiency. The algorithm to find the candidate vertex set can be improved. Instead of iterating over all the data graph vertex set, a pre computation can be performed to store the vertices based on their labels. Other area that needs to be focused on is the study of randomized QI-Sequence and its performance with Effective QI-Sequence.
XI. REFERENCES


2. Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, Jeong-Hoon Lee: An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. PVLDB 6(2): 133-144 (2012)