Improving Subgraph Isomorphism by Postponing Cartesian Product

Author: Ishan D Gulhane
Advisor: Dr. Carlos Rivero

December 2016
Contents

1 Introduction 2
   1.1 Key Words .............................................. 2

2 Algorithms 4
   2.1 Brute Force Algorithm ................................. 4
   2.2 Refine Search Space Algorithm .......................... 6
      2.2.1 Pseudo subgraph isomorphism using Bipartite matching ... 6
      2.2.2 Search Order Generation ............................. 8
   2.3 Subgraph Matching by postponing Cartesian product ........... 10
      2.3.1 Core-Forest-Leaf (CFL) Decomposition Framework ........ 11
      2.3.2 Compact path-index(CPI) Construction .................. 13
      2.3.3 Matching Order Generation ........................... 18

3 Example 20
   3.1 Preprocessing for Refine Search Space ..................... 21
   3.2 Preprocessing for Postpone Cartesian Product .............. 23
   3.3 Embedding’s for query Graph ............................. 26

4 Results 27

5 Future Work 31

6 Acknowledgment 33
1 Introduction

A graph database is an alternate way to find the solution for problems that are difficult to solve using relational databases. Graphs are widely used in social networks and proximity algorithms, i.e. they are used to find the relationship between the two entities rather than the entities themselves. Pattern matching in graphs is challenging due to the NP-completeness of subgraph isomorphism.

Graph database takes a graph query as an input and retrieves a collection of graphs from the databases which matches the given pattern graph. For e.g. finding all the heterocyclic compounds that contain the given aromatic ring or matching the protein pattern of a species in another species. Graph database can be classified into two types. First category is a large collection of small graphs. The main challenge in graph pattern matching is to reduce the number of pairwise pattern matching. The second category consists of very few large graphs. The challenge here is to improve the graph pattern matching.

1.1 Key Words

1. Subgraph
   - It is a graph that exists in another graph.
   - A graph F is a subgraph of graph G if
     - \( V(F) \rightarrow V(G) \)
     - \( E(F) \rightarrow E(G) \)

2. Bijection
   - A bijection is mapping from set A to set B that is one to one and onto.

Figure 1: Bijection [1]
3. Graph Isomorphism

- Two graphs $G$ and $H$ are said to be isomorphic if there is a bijection which preserves the adjacency and non-adjacency.

Figure 2 preserves the adjacency and non-adjacency mappings of query nodes in the graph.

- $1 \rightarrow 1$
- $2 \rightarrow 2$
- $3 \rightarrow 4$
- $4 \rightarrow 3$

![Figure 2: Graph Isomorphism](image-url)
2 Algorithms

2.1 Brute Force Algorithm

The algorithm consists of two steps:

- Find all the feasible candidates for each node in the subgraph.
- The resulting product of feasible candidate forms the search space. The algorithm performs the search for the subgraph embeddings in this search space.

Algorithm: Brute Force Implementation

Input: Graph pattern P and Graph G
Output: Feasible mappings of P in Graph G

// Search the Mappings for all the nodes in P that correspond to nodes in Graph G
// A node in G is mapping of a node in P if the label of node P is a subset of label of node G
for each node \( u \in V(P) \) do
    \( \phi(u) \leftarrow \{ v \mid v \in V(G), F_u(v) = \text{true} \} \)
end

// Search the matchings in a depth first manner between P and G
// It iterates on the \( i^{th} \) node of P to find the feasible mappings for the node
Search(1)
void Search(i)
begin
    for each \( v \in \phi(u_i), v \text{ is free} \) do
        //Check if \( u_i \) can be mapped to \( v \) by considering their edges
        if not Check(\( u_i, v \)) then
            continue;
        //Update the mapping and search the next node
        \( \phi(u_i) \leftarrow v; \)
        // Check for complete pattern match
        if \( i < |V(P)| \) then
            Search(i + 1);
        else if \( F_\phi(G) \) then
            Report \( \phi; \)
    end
end
end
// Checks if node $u_i$ can be mapped to node $v$ by considering their edges
boolean Check($u_i$, $v$)
begin
    for each edge $e(u_i, u_j) \in E(P)$, do
        contains ← false;
        for each edge $e(v,v') \in G(P)$, do
            if node $u_j$ is a mapping of node $v'$ then
                contains ← true;
                break;
            end
        end
    if contains ← false then
        return false;
    end
end
return true;
end

The worst case time complexity is exponential. In practice, the complexity depends on the size of the search space. The size of the search space can be reduced by leveraging the following information:

- Using local pruning and global structural information to remove infeasible mappings of query node.
- Leveraging vertex information to generate adapted subgraph for query graph to improve the speed of matching.
- Order the matching query nodes to reduce the size of Cartesian product, thus reducing the size of search space.
2.2 Refine Search Space Algorithm

The basic operating unit of graph structure is a graph motif. It can be a simple graph or composed of other graph motifs by means of concatenation, disjunction, and repetition. Selection operator $\sigma$ takes a graph pattern as an input and produces one or more mappings from the pattern as an output. The Cartesian product operator $\times$ takes two collections of the graph and performs join either using a predicate on attributes of graph or by concatenation by edges. The composition operator generates new graphs by combining information from the matched graphs [3].

The core of GraphQL is the graph algebra where selection and composition operators are used to generalize and rewrite the matched graphs. The author [3] proposes to improve the speed by pruning the search space locally and retrieving the feasible matches using the global structural information. The author [3] further optimizes the search using cost model designed for graphs.

2.2.1 Pseudo subgraph isomorphism using Bipartite matching

In this algorithm, the author [3] proposes to reduce the overall search space with the help of pseudo subgraph isomorphism. For each node $u$ in the pattern graph and its sub mapping $v$ in the graph, we verify whether the adjacent subgraph of node $u$ is sub isomorphic with the adjacent subgraph of node $v$. The author [3] proposes to check sub-isomorphism of the adjacency graph with the help of bipartite matching, i.e. to check the presence of semi-perfect matching between neighbors of nodes $u$ and $v$. If semi-perfect matching is not present, we can safely remove the node from the search space. This check is done iteratively until all the infeasible mappings are removed. In this way, the algorithm prunes the search space globally.
Algorithm : Pseudo subgraph isomorphism[3]

Input: Graph pattern P, Graph G, Search Space $\phi(\mu_1) \times \phi(\mu_2) \times \ldots \times \phi(\mu_k)$
Output: Reduced Search Space $\phi'(\mu_1) \times \ldots \times \phi'(\mu_k)$

begin
  check $\leftarrow$ true;
  while check
    check$\leftarrow$ false;
    for each node $u \in V(P)$ do
      // Iterate all the mappings of node $u$ in graph $G$
      remark $\leftarrow$ false;
      for each $v \in V(G), u \in V(P)$ do
        // Construct bipartite graph $B_{u,v}$
        $B_{u,v} \leftarrow$ Bipartite matching between $N_p(u)$ and $N_g(v)$
        if $B_{u,v} == \text{null}$
          remark $\leftarrow$ true;
        end
      end
      check$\leftarrow$ true;
    end
    if remark$==$true
      remove $v$ from $\phi(\mu)$
      Remark all adjacent nodes of $v$
    else
      unmark $\langle u,v \rangle$
    end
  end
end
2.2.2 Search Order Generation

In simple terms, subgraph matching is equivalent to multiple joins being performed on mappings of query node. The efficiency of the algorithms depends upon the order in which the search is performed. A search order is represented as a binary tree where joint operation is the internal node and nodes of the pattern graph are the leaf nodes.

The above image shows 2 different search ordering techniques for the query graph with nodes A, B and C.

The result size of join i is estimated by $\text{Size}(i) = \text{Size}(i.\text{left}) \times \text{Size}(i.\text{right}) \times \gamma(i)$ where

- $i.\text{left}$ and $i.\text{right}$ are the left and right child nodes of $i$
- $\gamma(i)$ is the reduction factor
  - $\gamma(i)$ is $0.5$ (number of edges with already visited nodes of pattern graph)

The traditional dynamic programming approach takes exponential time for finding an order with minimum joint operations in a pattern graph of size k. In order to optimize, the author[3] uses the greedy approach of selecting the leaf node that minimizes the number of joint operations.
Algorithm : Search Order Generation[3]

Input: Graph pattern P, Mappings of P and G \langle u, v \rangle , \gamma - 0.5
Output: Updated Search Order

priority queue ← Sorts Vertex based on computed cost
List visited; // Contains list of processed vertices
// Mark all the vertex of P as unvisited
unvisited ← Add node u ∈ V(P)
while size of unvisited > 0
    for each node u ∈ unvisited
        if u ∉ visited
            count ← 0
            total ← 0
            for each node v ∈ visited
                if edge \langle u, v \rangle == true
                    count++
                    total ← size of mappings of v
                end
            end
            if count≠0
                value ← size of mappings of u * total * (\gamma^count)
                add \langle u, value \rangle to priority queue
            end
        end
        node u ← remove head of priority queue
        add node u to visited
        remove node u from unvisited
        clear priority queue
    end
return visited
2.3 Subgraph Matching by postponing Cartesian product

The author[2] tries to address two problems to improve the speed of subgraph matching:

- To reduce the number of unpromising intermediate results caused by the product of infeasible node mappings of query graph nodes to data graph nodes.
- To generate an effective matching order to reduce the number intermediate results.

The author[2] proposes following solutions for the above-mentioned problems:

- Develop a compact auxiliary path based data structure, i.e. Compact Path Index (CPI) for accurately estimating the number of embeddings of query path which helps to generate the mappings of the query graph in the data graph. The size of CPI is $O(|E(G)| \times |V(G)|)$, where $E(G)$ and $V(G)$ are the edges and vertices of the graph $G$.

- Develop a new framework for generating effective matching order. This framework decomposes query graph into the core, forest, and leaf subgraph. To generate all the embedding’s of query graph, in the following order: core, forest, and leaf nodes.
2.3.1 Core-Forest-Leaf (CFL) Decomposition Framework

In order to postpone the Cartesian product and generate an effective matching order, the query graph is decomposed into three substructures namely core, forest, and leaf substructure. The core structure is a minimal connected subgraph of the query graph. Leaf structure consists of the all the vertices of a graph with degree one. Forest structure consists of all the connecting nodes between the core and leaf structures.

There are two types of edges in the graph, one that forms a part of any spanning tree of a graph called as the tree edges while the edges that are not part of the spanning tree are called as the non-tree edges of the graph.

Initially, the query graph is decomposed into the core forest structure which consists of a small, dense subgraph consisting of all non-tree edges of any spanning tree of a graph. This process is performed iteratively to remove all the nodes of degree one from the query graph to get the final set of vertices in core structure, i.e. core set ($V_C$). The set of removed nodes forms the part of forest structure. The forest structure is a set of connected trees having one vertex in common with the core structure. The forest structure is further subdivided to get the forest set ($V_T$) and leaf set ($V_L$). The following figure shows decomposition of query graph into core, forest, and leaf sets.

![Figure 4: CFL Decomposition[2]](image)

Removal of an initial set of vertices of degree one ($u_7, u_8, u_9, u_{10}$) gives a new set
of vertices of degree one \((u_3, u_4, u_5, u_6)\). After removing the new set of vertices of degree one, we get the core set \((u_0, u_1, u_2)\).

Algorithm: CFL decomposition[2]

Input: Graph pattern \(P\)
Output: List of edges in Core, forest-leaf substructure

```
list results; // Contains list of edges for core, forest-leaf substructure respectively.
list forest_leaf; //Contains list of edges for forest leaf substructure.
change ← true
while change
  change ← false // Contains all removed vertices of degree 1
  for each vertex \(u \in P\)
    if number of edges of \(u == 1\)
        remove_vertices ← remove_vertices ∪ u
        change ← true
  if change == true
    List removed_edges;// contains list of edges to be removed
    for each vertex \(v \in remove_vertices\)
      removed_edges ← removed_edges ∪ all edges of \(v\)
      remove \(v\) and all its edges from \(P\)
    forest_leaf ← forest_leaf ∪ removed_edges
  end
  list core_leaf ← all remaining edges of \(P\)
  results ← results ∪ core_leaf
  reverse forest_leaf
  results ← results ∪ forest_leaf
  return results
```
2.3.2 Compact path-index(CPI) Construction

Compact path-index(CPI) is used to generate the mappings of all the vertices where all the false positive query vertices are removed using the tree and the non-tree edges of the query graph. For constructing the CPI, the vertices of query graph are partitioned into the different list according to their levels in a breadth-first traversal (BFS) of the query graph. The root node is the node with a maximum number of edges and minimum initial mappings. The non-tree edges, i.e. edge that is not present in BFS traversal are classified into two types, namely: same-level non-tree edges (S-NTE) i.e. edges between the nodes present at the same level and cross-level non-tree edges (C-NTE) i.e. edges between the nodes present in different levels.

CPI is constructed in two stages as follows:

- Top Down Construction
- Bottom Up Refinement

Top Down Construction

The top down CPI is constructed by visiting the query nodes in level-order. The false positive node mappings are pruned by using the non-tree edges and edges with the previous level i.e. parent nodes.

The algorithm constructs the CPI in following steps:

- Forward Candidate Generation: In this step, we process the vertices at each according to their order present in the level list. For each processed vertex v, we have 2 set of vertices: one set of neighbors that are already visited and other is the set of unvisited vertices present at the same level. We prune the nodes whose neighbor set size of visited nodes is less than the corresponding node in query graph.

- Backward Candidate Pruning: This step uses the set of unvisited nodes at the same level in step 1 to further filter candidates that do not have nontree edges at the same level. Backward pruning processes the vertices in reverse level order.
Algorithm: Top Down Construction[2]

Input: A query q, root vertex r, initial mappings of query vertices and a data graph G
Output: CPI Graph of q over G

for each v in G with label \( l_q(r) \) and degree at least \( d_q(r) \) do
  \( r.C \leftarrow r.C \cup v \)
List visited \( \leftarrow r \)
Set v.cnt \( \leftarrow 0 \) for all v in G
for each level lev from 2 to max_level do
  for each query vertex u at level lev do
    Cnt \( \leftarrow 0 \)
    for each query vertex \( u' \in N_q(u) \) do
      if \( u' \) is visited then
        for each vertex \( v' \in u'.C \) do
          for each vertex \( v \in NG(v') \) with \( l_q(u) \) and \( d_q(u) \) do
            if v.cnt = Cnt then
              v.cnt \( \leftarrow v.cnt + 1 \)
            Cnt \( \leftarrow Cnt + 1 \)
          else if \( (u',u) \) is a S-NTE then
            u.UN \( \leftarrow u.UN \cup u' \)
        for each vertex \( v \) in mappings(u) with v.cnt \( \neq \) Cnt do
          mappings(u) \( \leftarrow \) mappings(u) - v
          v.cnt \( \leftarrow 0 \)
        visited \( \leftarrow u \)
      for each query vertex u at level lev in reverse order do
        Cnt \( \leftarrow 0 \)
        for each vertex \( u' \in u'.UN \) do
          for each vertex \( v' \in u'.C \) do
            for each vertex \( v \in NG(v') \) with \( l_q(u) \) and \( d_q(u) \) do
              if v.cnt = Cnt then
                v.cnt \( \leftarrow v.cnt + 1 \)
              Cnt \( \leftarrow Cnt + 1 \)
          for each vertex \( v \) in mappings(u) with v.cnt \( \neq \) Cnt do
            mappings(u) \( \leftarrow \) mappings(u) - v
            v.cnt \( \leftarrow 0 \)
        for each query vertex u at level lev do
          up \( \leftarrow u.p \) // Parent of node u in bfs order
for each vertex \( v_p \in u.C \) do
  for each vertex \( v \in NG(v_p) \) with \( \text{lq}(u) \) do
    if \( v \in u.C \) then
      CPI Graph \( \leftarrow \) addedge\((v,v_p)\)
  return CPI Graph

Example

Initially, we add the mappings of the root node in the list of visited nodes. The mappings of \( v_1 \) are \( v_1, v_2 \). For node \( v_2 \), the \( u.N \) is \( v_1 \) and \( u.UN \) is \( v_2 \). The mappings generated for \( v_2 \) in Forward Candidate Generation step are \( v_3, v_5, v_7, v_9 \).
For node v3, the u.N is v1, v2, the u.UN is and the mappings generated is v4, v6, v8. While performing backward candidate pruning, v9 is removed from the mapping of v2 because it does not have the required nontree edge. Similarly, we process the remaining levels of the tree and generate the candidate set of all the nodes. The mappings for the query nodes is as follows:

- v1 → v1, v2
- v2 → v3, v5, v7
- v3 → v4, v6, v8
- v4 → v11, v12

**Bottom Up Refinement**

Top down construction removes all the false positive mappings for vertices which do not have the required number of edges with parents and same level nodes in BFS of query graph i.e. we consider the ancestor information. Bottom-up refinement prunes the mappings based on the presence of edges with the child nodes, i.e. nodes present in the next level of BFS of query graph.
Algorithm : Bottom Up Refinement[2]

Input: A query q, root vertex r, initial mappings of query vertices and CPI Graph
Output: updated mappings of vertices for query graph

for each query vertex u in bottom up order do
   Cnt ← 0
   for each lower level neighbor vertex u’ of u in query q do
      for each vertex v’ ∈ u’.C do
         for each vertex v ∈ NG(v’) with l_q(u) and d_q(u) do
            if v.cnt = Cnt then
               v.cnt ← v.cnt + 1
            Cnt ← Cnt + 1
   for each vertex v in mappings(u) with v.cnt != Cnt do
      mappings(u) ← mappings(u) - v
      v.cnt ← 0
   visited ← u

In this step, we refine the mappings removing the infeasible mappings such as v7, v8, and v2. The following figure shows the final mappings of nodes.

Figure 8: Graph
2.3.3 Matching Order Generation

The author[2] proposes a path based ordering of core vertices to further reduce the size of the search space for finding the embedding’s of query q in data graph G. We first get the list of all root to leaf paths of query q. The first path is the path with a minimum number of embeddings. We continue selecting paths with minimum number of vertices until all the paths are visited.

Algorithm : Matching Order Generation[2]

Input: query q, root, mappings
Output: matching order query vertices

\[ P \leftarrow \text{root-to-leaf paths in q} \]
\[ \text{sequence} \leftarrow \text{add all the vertices for path with minimum number of mappings} \]
while all paths are visited
select the next path minimum number of mappings
add all the vertices to sequence list
return sequence

Algorithm : CFL-Matching[2]

Input: query q and data graph g
Output: list of embedding’s of query q

\[(v_c, v_{fl}) \leftarrow \text{decompose query vertices} \]
\[ \text{mappings} \leftarrow \text{generate mappings for query vertices} \]
\[ \text{CPI} \leftarrow \text{Construct CPI Graph (q, G, mappings, } v_c, v_{fl} \) \]
for each core embedding in Core-Match do
matching \leftarrow matching \times \text{forest-leaf-match}
return matching
The algorithm for core matching and forest matching is as follows:

Algorithm : Core & Forest Matching[2]

Input: query q, data graph g, matching_order, mappings,index, visited_list
Output: list of core embedding’s of query q

if index = size of matching_order +1
    core_embeddings ← core_embeddings ∪ visited_list
else
    for each vertex v in mappings(matching_order(index)) do
        if v /∈ visited & isValidInsertion(v)
            visited_list ← visited_list ∪ v
            core_match(q, g, matching_order,mappings,index+1,visited_list)
            visited_list ← visited_list - v
3 Example

Figure 9: BaseGraph

Figure 10: Query Graph
3.1 Preprocessing for Refine Search Space

The following output shows the initial using and refined search order which reduces the size of joins, thus reducing the search space.

The initial search order using BFS is as follows:

1. Vertex Id : v1 Label : [b]
2. Vertex Id : v3 Label : [d]
3. Vertex Id : v2 Label : [c]

The refined search order generated using search order generation algorithm proposed in section 2.2.2. The result size of join i is estimated by $\text{Size}(i) = \text{Size}(i.\text{left}) \times \text{Size}(i.\text{right}) \times \gamma(i)$ where

- $i.\text{left}$ and $i.\text{right}$ are the left and right child nodes of $i$
- $\gamma(i)$ is the reduction factor
  - $\gamma(i)$ is 0.5 (number of edges with already visited nodes of pattern graph)

1. Vertex Id : v2 Label : [c]
2. Vertex Id : v3 Label : [d]
3. Vertex Id : v1 Label : [b]
The following table shows initial mappings of each query node in base graph:

<table>
<thead>
<tr>
<th>Pattern Graph Node</th>
<th>Base Graph Node</th>
</tr>
</thead>
</table>
| Vertex Id : v2 Label : [c] | Vertex Id : v3 Label : [b, c, e],  
Vertex Id : v7 Label : [c],  
Vertex Id : v11 Label : [c, e],  
Vertex Id : v13 Label : [b, c, d],  
Vertex Id : v15 Label : [b, c],  
Vertex Id : v17 Label : [c] |
| Vertex Id : v1 Label : [b] | Vertex Id : v1 Label : [a, b],  
Vertex Id : v3 Label : [b, c, e],  
Vertex Id : v4 Label : [b],  
Vertex Id : v5 Label : [b, e],  
Vertex Id : v6 Label : [b, d],  
Vertex Id : v9 Label : [b],  
Vertex Id : v12 Label : [a, b],  
Vertex Id : v13 Label : [b, c, d],  
Vertex Id : v15 Label : [b, c],  
Vertex Id : v18 Label : [b] |
| Vertex Id : v3 Label : [d] | Vertex Id : v2 Label : [d, e],  
Vertex Id : v6 Label : [b, d],  
Vertex Id : v13 Label : [b, c, d],  
Vertex Id : v14 Label : [d, e],  
Vertex Id : v16 Label : [d, e],  
Vertex Id : v19 Label : [d] |

**Search space after pruning**

Node v7, v13, v15 are pruned from the mappings of node v2 of query because the nodes along with their adjacent neighbors does not have pseudo subgraph isomorphism with node v2 and its neighbors.  
For example: node v7 does have a neighbor with label d. Hence it is pruned. Similarly, we prune the mappings for node v1 and v3 of query graph.
The following table refined search space using the neighbor information to remove the infeasible mappings of query node.

<table>
<thead>
<tr>
<th>Pattern Graph Node</th>
<th>Base Graph Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Id : v2 Label : [c]</td>
<td>Vertex Id : v3 Label : [b, c, e], Vertex Id : v7 Label : [c], Vertex Id : v11 Label : [c, e]</td>
</tr>
<tr>
<td>Vertex Id : v1 Label : [b]</td>
<td>Vertex Id : v5 Label : [b, e], Vertex Id : v6 Label : [b, d], Vertex Id : v13 Label : [b, c, d], Vertex Id : v18 Label : [b]</td>
</tr>
<tr>
<td>Vertex Id : v3 Label : [d]</td>
<td>Vertex Id : v2 Label : [d, e], Vertex Id : v6 Label : [b, d], Vertex Id : v14 Label : [d, e], Vertex Id : v19 Label : [d]</td>
</tr>
</tbody>
</table>

### 3.2 Preprocessing for Postpone Cartesian Product

Initially, the node v2 is selected as the root node for the query graph. We generate the BFS list of query nodes required for generating the CPI.

BFS List for query Tree:
- Vertex Id : v2 Label : [c]
- Vertex Id : v1 Label : [b], Vertex Id : v3 Label : [d]

The following table shows initial mappings of each query node in base graph:
### Table 3: Initial Mappings

<table>
<thead>
<tr>
<th>Pattern Graph Node</th>
<th>Base Graph Node</th>
</tr>
</thead>
</table>
| Vertex Id : v2 Label : [c] | Vertex Id : v3 Label : [b, c, e],  
                           | Vertex Id : v7 Label : [c],  
                           | Vertex Id : v11 Label : [c, e],  
                           | Vertex Id : v13 Label : [b, c, d],  
                           | Vertex Id : v15 Label : [b, c],  
                           | Vertex Id : v17 Label : [c] |
|Vertex Id : v1 Label : [b] | Vertex Id : v1 Label : [a, b],  
                           | Vertex Id : v3 Label : [b, c, e],  
                           | Vertex Id : v4 Label : [b],  
                           | Vertex Id : v5 Label : [b, e],  
                           | Vertex Id : v6 Label : [b, d],  
                           | Vertex Id : v9 Label : [b],  
                           | Vertex Id : v12 Label : [a, b],  
                           | Vertex Id : v13 Label : [b, c, d],  
                           | Vertex Id : v15 Label : [b, c],  
                           | Vertex Id : v18 Label : [b] |
|Vertex Id : v3 Label : [d] | Vertex Id : v2 Label : [d, e],  
                           | Vertex Id : v6 Label : [b, d],  
                           | Vertex Id : v13 Label : [b, c, d],  
                           | Vertex Id : v14 Label : [d, e],  
                           | Vertex Id : v16 Label : [d, e],  
                           | Vertex Id : v19 Label : [d] |

### Refined Mappings after Top Down Construction

Node v15 is pruned from the mappings of root node because its degree is less than the degree of the root node.  
Node v1, v4, v12, v15 are pruned from the mappings of node v1 because their degree is less than the degree of the node v1.
Table 4: Top Down Construction

<table>
<thead>
<tr>
<th>Pattern Graph Node</th>
<th>Base Graph Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Id : v2 Label : [c]</td>
<td>Vertex Id : v3 Label : [b, c, e], Vertex Id : v7 Label : [c], Vertex Id : v11 Label : [c, e], Vertex Id : v13 Label : [b, c, d], Vertex Id : v17 Label : [c]</td>
</tr>
<tr>
<td>Vertex Id : v1 Label : [b]</td>
<td>Vertex Id : v3 Label : [b, c, e], Vertex Id : v5 Label : [b, e], Vertex Id : v6 Label : [b, d], Vertex Id : v13 Label : [b, c, d], Vertex Id : v18 Label : [b]</td>
</tr>
<tr>
<td>Vertex Id : v3 Label : [d]</td>
<td>Vertex Id : v2 Label : [d, e], Vertex Id : v6 Label : [b, d], Vertex Id : v13 Label : [b, c, d], Vertex Id : v14 Label : [d, e], Vertex Id : v16 Label : [d, e], Vertex Id : v19 Label : [d]</td>
</tr>
</tbody>
</table>

Refined Mappings after Bottom Up Refinement

Node v7, v13 are pruned from the mappings of node v2 because they don’t have the required number of edges with child nodes of the BFS list.
The following table shows final mappings of each query node in base graph:
Table 5: Bottom Up Refinement

<table>
<thead>
<tr>
<th>Pattern Graph Node</th>
<th>Base Graph Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Id : v2 Label : [c]</td>
<td>Vertex Id : v3 Label : [b, c, e], Vertex Id : v11 Label : [c, e], Vertex Id : v17 Label : [c]</td>
</tr>
<tr>
<td>Vertex Id : v1 Label : [b]</td>
<td>Vertex Id : v3 Label : [b, c, e], Vertex Id : v5 Label : [b, e], Vertex Id : v6 Label : [b, d], Vertex Id : v13 Label : [b, c, d], Vertex Id : v18 Label : [b]</td>
</tr>
<tr>
<td>Vertex Id : v3 Label : [d]</td>
<td>Vertex Id : v2 Label : [d, e], Vertex Id : v6 Label : [b, d], Vertex Id : v13 Label : [b, c, d], Vertex Id : v14 Label : [d, e], Vertex Id : v16 Label : [d, e], Vertex Id : v19 Label : [d]</td>
</tr>
</tbody>
</table>

3.3 Embedding’s for query Graph

Table 6: Embedding 1

<table>
<thead>
<tr>
<th>Pattern Graph Node</th>
<th>Base Graph Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Id : v2 Label : [c]</td>
<td>Vertex Id : v17 Label : [c]</td>
</tr>
<tr>
<td>Vertex Id : v1 Label : [b]</td>
<td>Vertex Id : v18 Label : [b]</td>
</tr>
<tr>
<td>Vertex Id : v3 Label : [d]</td>
<td>Vertex Id : v19 Label : [d]</td>
</tr>
</tbody>
</table>

Table 7: Embedding 2

<table>
<thead>
<tr>
<th>Pattern Graph Node</th>
<th>Base Graph Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Id : v2 Label : [c]</td>
<td>Vertex Id : v3 Label : [b, c, e]</td>
</tr>
<tr>
<td>Vertex Id : v1 Label : [b]</td>
<td>Vertex Id : v6 Label : [b, d]</td>
</tr>
<tr>
<td>Vertex Id : v3 Label : [d]</td>
<td>Vertex Id : v2 Label : [d, e]</td>
</tr>
</tbody>
</table>

Table 8: Embedding 3

<table>
<thead>
<tr>
<th>Pattern Graph Node</th>
<th>Base Graph Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Id : v2 Label : [c]</td>
<td>Vertex Id : v11 Label : [c, e]</td>
</tr>
<tr>
<td>Vertex Id : v1 Label : [b]</td>
<td>Vertex Id : v13 Label : [b, c, d]</td>
</tr>
<tr>
<td>Vertex Id : v3 Label : [d]</td>
<td>Vertex Id : v14 Label : [d, e]</td>
</tr>
</tbody>
</table>
4 Results

The yeast database is used for performing the regression testing of algorithms. Results of 50 queries of size 2,3,4,5,6 and 7 are analyzed to measure the performance of algorithms. It is a highly connected graph consisting of 3112 multi-labeled vertices and 12519 edges.

Following are the observations from the result generated:

- Brute force algorithm works better for query graphs with less number of mappings.
- Algorithms using bipartite matching and postponing Cartesian product algorithms reduce the size of search space significantly.
- Subgraph Matching by postponing Cartesian product works well for graph queries having same level edges and with queries where nodes are present in the form of paths.
- Refine search space algorithms work better than Cartesian product in queries which have a large number of mappings and have few false positive mappings, i.e. there are fewer nodes to be pruned.
- For complex queries, i.e. highly connected queries subgraph matching algorithms using bipartite matching and Cartesian product improves the speed by 3-4 times.
For the query in figure 11, the size of the search space is reduced significantly by removing all the incorrect mappings. The speed of subgraph is significantly improved for the bipartite matching and postponing Cartesian product algorithms. Table 9 shows the time taken by the algorithms to get all the embeddings of query graph.

Table 9: Result

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Taken(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force Implementation</td>
<td>1398502</td>
</tr>
<tr>
<td>Refine search space using Bipartite Matching</td>
<td>891547</td>
</tr>
<tr>
<td>Subgraph Matching by postponing Cartesian product</td>
<td>421006</td>
</tr>
</tbody>
</table>
Figure 12: Query Graph

For the query in figure 12, the size of search space the size of mappings and the number incorrect mappings is small. Brute force is faster for performing subgraph isomorphism. Table 10 shows the time taken by the algorithms to get all the embedding’s of query graph.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Taken (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force Implementation</td>
<td>4986</td>
</tr>
<tr>
<td>Refine search space using Bipartite Matching</td>
<td>5888</td>
</tr>
<tr>
<td>Subgraph Matching by postponing Cartesian product</td>
<td>8149</td>
</tr>
</tbody>
</table>
Figure 13: Clique Results

Figure 14: Query Graph Results
5 Future Work

The future work involves developing new algorithms which leverage the use of already computed CPI of queries that are part of other queries. This helps to improve the speed CPI computation of new query graph.

For example:

Suppose we have computed the CPI for the query graph in figure 15 and store it database along with the root node for computing the CPI.

Figure 15: Old Query Graph

Figure 16: New Query Graph
Since the old query is a subgraph of a new query graph, we can compute the CPI for new query graph with the help CPI computed for the old query graph. The query in figure 16 can be compressed with node v123 formed by combining nodes v1, v2, v3. Now we can compute the CPI for the compressed query graph as shown in figure 17 with the help of CPI and check for the additional edges between new node and v4, i.e. check for edges v2-v4 and v3-v4. The compressed query graph will help to avoid unnecessary computation involved in finding the mappings for the node v1, v2, v3. We can further refine the mappings of v1, v2, v4 by checking the edges between v2-v4 and v3-v4.

Figure 17: Compressed Query Graph

While developing the new algorithms we need to consider the following two cases for the presence of old query graph in new query graph:

- The presence of old query graph is more restrictive, i.e. contains newly added edges between the nodes of the old query graph.
- The presence of old query graph is more generic, i.e. nodes of old query graph has edges with other nodes of the new query graph.
6 Acknowledgment

I would like to sincerely thank Dr. Carlos Rivero for his guidance and feedback that helped me to complete my project. I would also like to thank Dr. Joe Geigel for his guidance throughout the colloquium.

References

