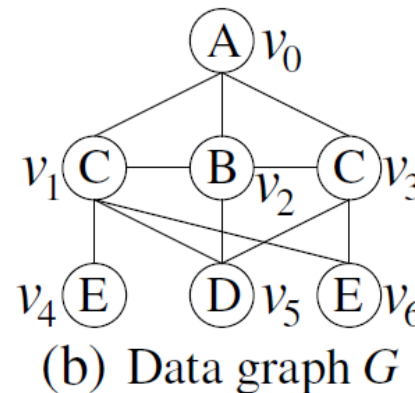
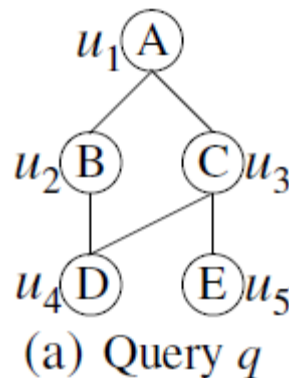


All-Matching

➤ Subgraph Matching

Given a query q and a large data graph G , the problem is to extract all subgraph isomorphic embeddings of q in G .

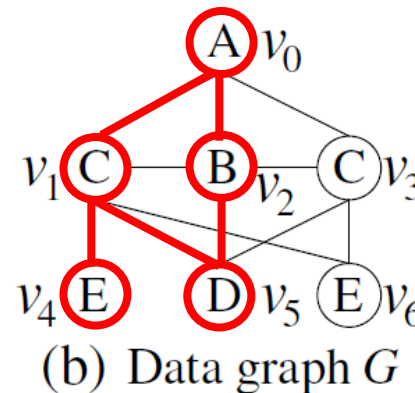
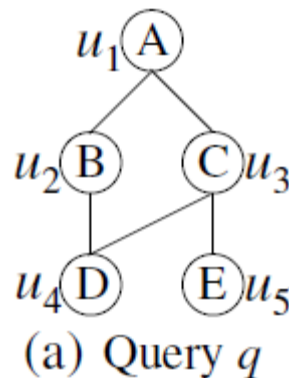


There are **three** subgraph isomorphic embeddings of q in G .

All-Matching

➤ Subgraph Matching

Given a query q and a large data graph G , the problem is to extract all subgraph isomorphic embeddings of q in G .



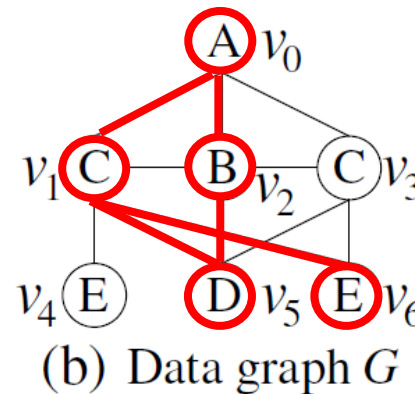
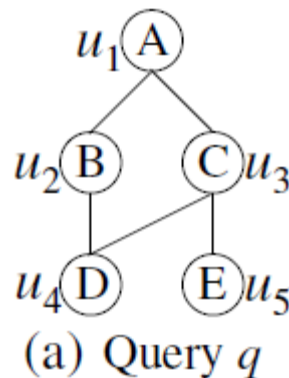
Embedding One

u_1	u_2	u_3	u_4	u_5
v_0	v_2	v_1	v_5	v_4

All-Matching

➤ Subgraph Matching

Given a query q and a large data graph G , the problem is to extract all subgraph isomorphic embeddings of q in G .



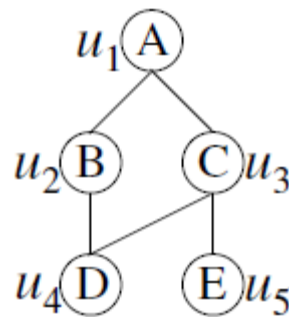
Embedding Two

u_1	u_2	u_3	u_4	u_5
v_0	v_2	v_1	v_5	v_6

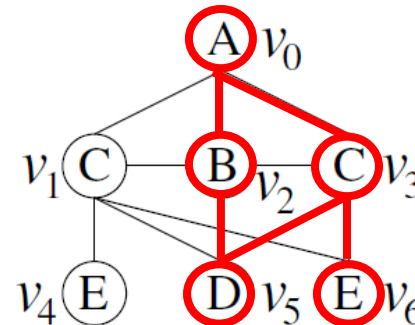
All-Matching

➤ Subgraph Matching

Given a query q and a large data graph G , the problem is to extract all subgraph isomorphic embeddings of q in G .



(a) Query q



(b) Data graph G

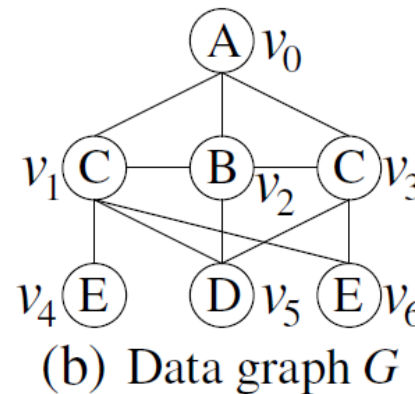
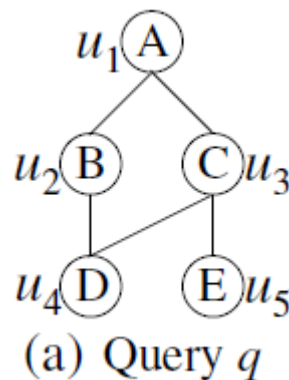
Embedding Three

u_1	u_2	u_3	u_4	u_5
v_0	v_2	v_3	v_5	v_6

Existing Work

➤ Ullmann's algorithm [J.ACM'76]

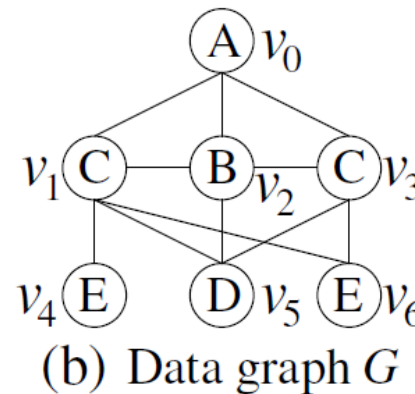
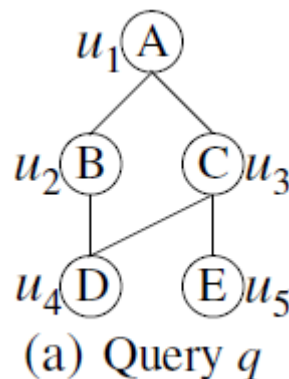
- First algorithm to enumerate all subgraph isomorphic embeddings
- A **backtracking** algorithm that maps query vertices one by one, following a **random** order.
- Example: A random order for query q could be $(u_1, u_4, u_2, u_3, u_5)$



Existing Work

➤ Ullmann's algorithm [J.ACM'76]

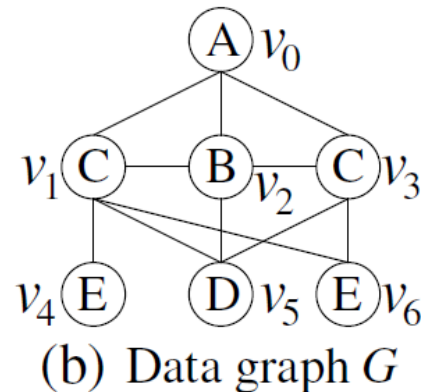
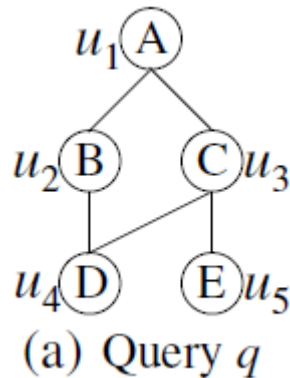
- First algorithm to enumerate all subgraph isomorphic embeddings
- A **backtracking** algorithm that maps query vertices one by one, following a **random** order.
- Example: A random order for query q could be $(u_1, u_4, u_2, u_3, u_5)$



Existing Work

➤ VF2 [IEEE Trans'04] and QuickSI [VLDB'08]

- Independently propose to enforce **connectivity** of the matching order
- Example : A connected order for query q could be $(u_1, u_2, u_4, u_3, u_5)$



- QuickSI further removes false-positive intermediate results by first processing infrequent query vertices and edges.

Subgraph Matching

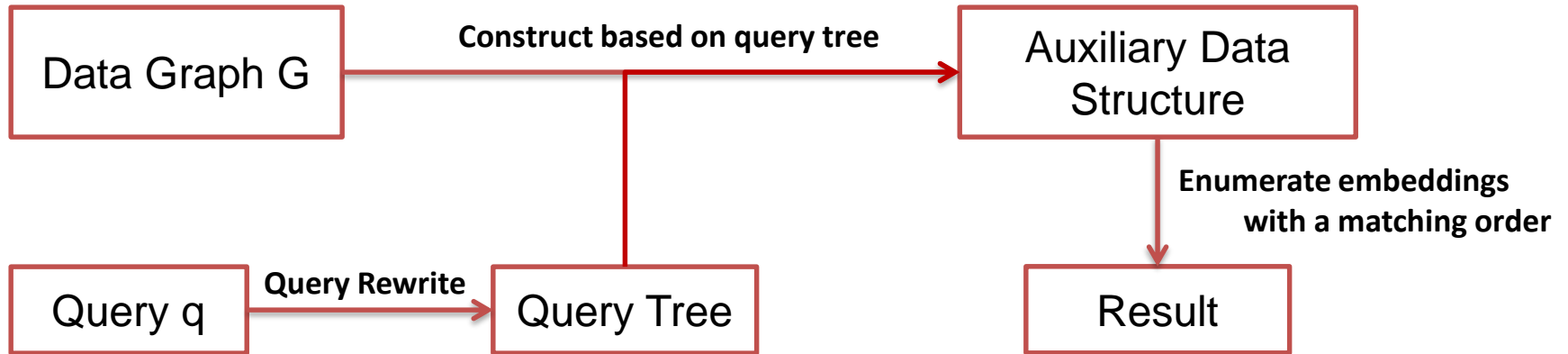
- Nevertheless, it is still very difficult to enumerate subgraph matchings.

Due to 1) Data graph could be very large and dense.

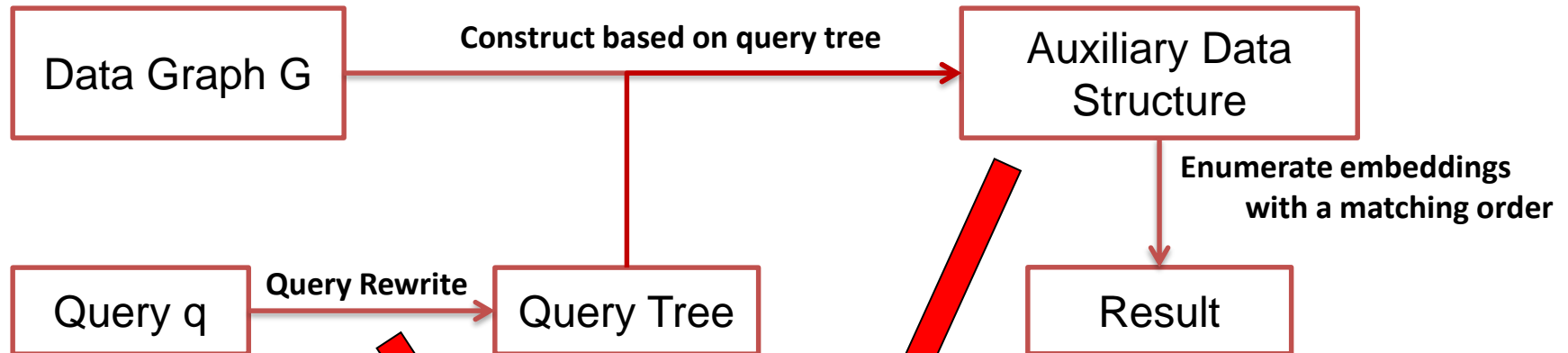
2) There could be exponential number of embeddings of q in G .

- Two Advanced Approaches.
 - TurboISO
 - CFL-Match

TurboISO Overview



TurboISO Overview



- Key Techniques

- Neighborhood Equivalence Classes (NEC)

- Merging vertices with same neighbors to reduce query size
 - Combine / Permute strategy to enumerate results

- Candidate Region Exploration

- Constructed on-the-fly based on q
 - A path-based data structure containing all embeddings of q in G

Will be explained with example in detail.

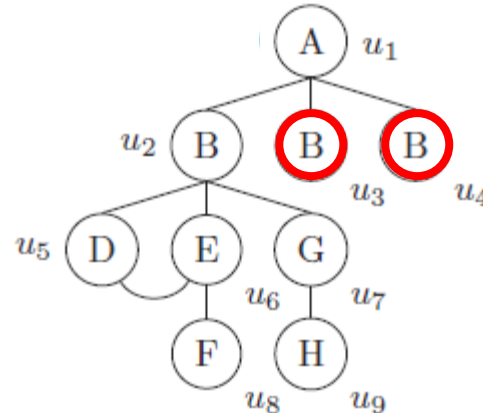
Neighborhood Equivalence Class(NEC)

- Definition

Let \simeq be an *equivalence relation* over all query vertices in q such that, $u_i(\in V(q)) \simeq u_j(\in V(q))$ if for every embedding m that contains (u_i, v_x) and (u_j, v_y) ($v_x, v_y \in V(g)$), there exists an embedding m' such that

$$m' = m - \{(u_i, v_x), (u_j, v_y)\} \cup \{(u_i, v_y), (u_j, v_x)\}$$

- Example



u_3 and u_4 are equivalent.

The Neighborhood Equivalence Class(NEC) of a query vertex u is a set of query vertices, which are equivalent to u .

Query Rewrite

- Given a query q , we rewrite it into a NEC Tree in following steps:

- 1) Root Node Selection

Ranking function $Rank(u) = \frac{freq(g, L(u))}{deg(u)}$

$freq(g, l)$: number of **data** vertices in g that have label l

$deg(u)$: the degree of u

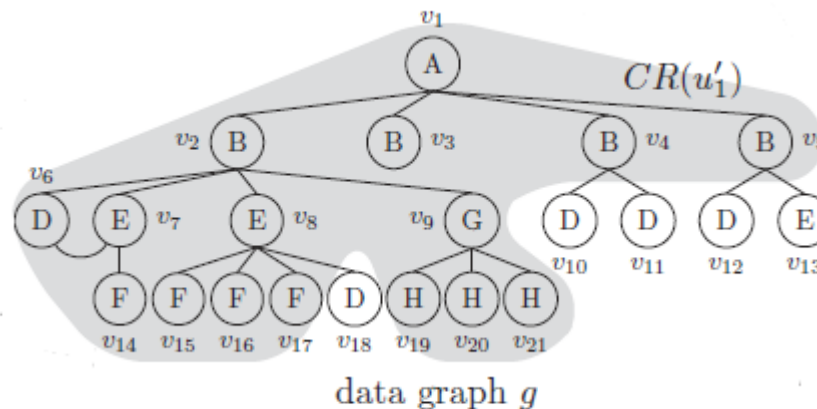
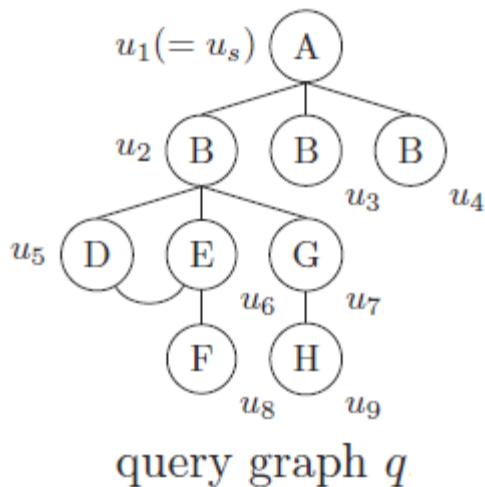
The vertex with **smallest** ranking value will be selected as the root node.

Query Rewrite

- Given a query q , we rewrite it into a NEC Tree in following steps:

1) Root Node Selection

Ranking function $Rank(u) = \frac{freq(g, L(u))}{deg(u)}$



$$Rank(u_1) = 1 / 3$$

$$Rank(u_2) = 4 / 4$$

$$Rank(u_3) = 4 / 1$$

$$Rank(u_4) = 4 / 1$$

$$Rank(u_5) = 5 / 2$$

$$Rank(u_6) = 3 / 3$$

$$Rank(u_7) = 1 / 2$$

$$Rank(u_8) = 4 / 1$$

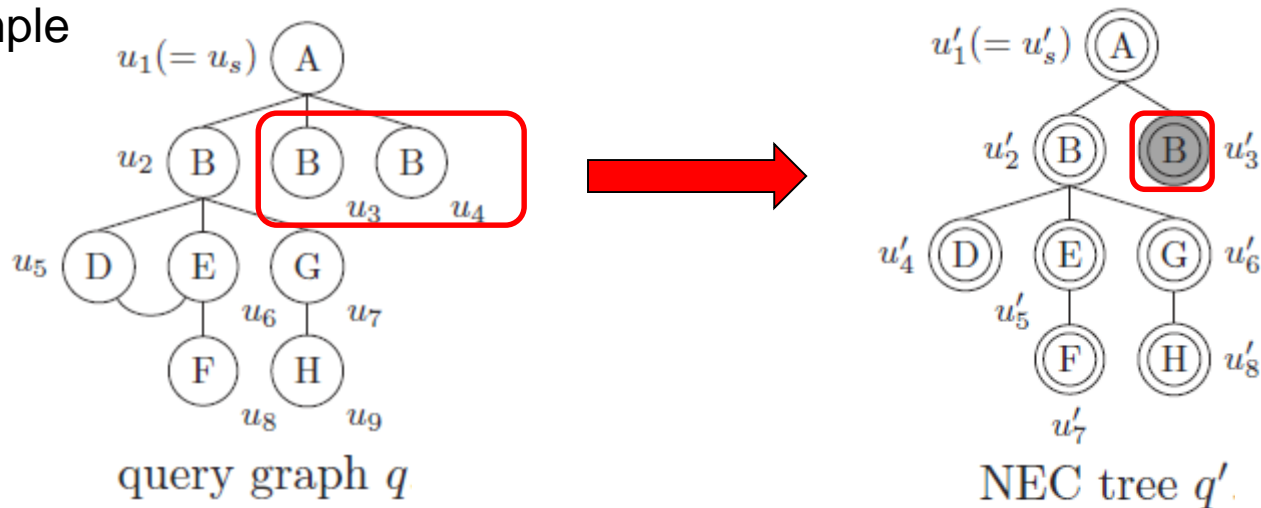
$$Rank(u_9) = 3 / 1$$

Hence, u_1 is selected as the root node.

Query Rewrite

- Given a query q , we rewrite it into a NEC Tree in following steps:
 - 2) Rewrite to NEC Tree by
 - I. Performing BFS from the root node
 - II. Merging vertices from same NEC into a single vertex

Example

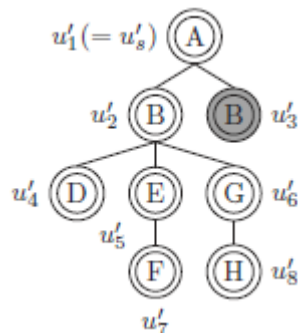


u_1 has been selected as the root node in the previous step.

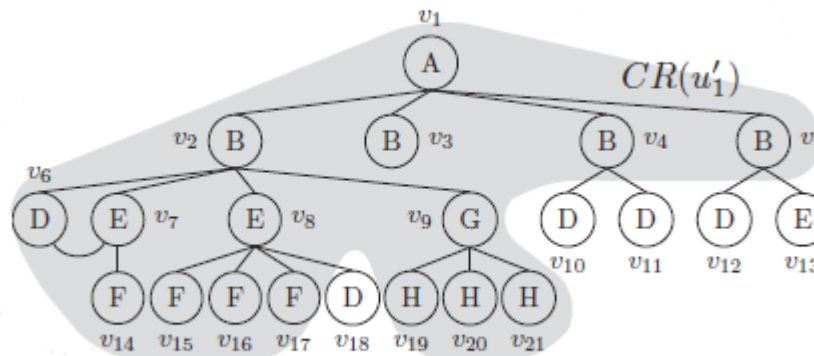
Candidate Region Exploration

- Candidate Subregion

Candidate subregion denoted as $CR(u, v)$, contains data vertices such that they match u and are child vertices of v in the DFS tree, where u is an NEC vertex, and v is a data vertex.



NEC tree q'



data graph g

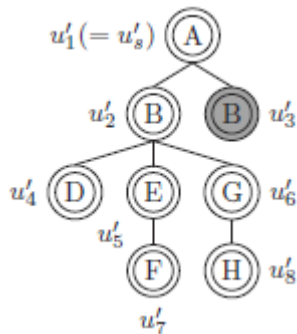
CR	candidate vertices
$CR(u'_1, v_s^*)$	$\{v_1\}$
$CR(u'_2, v_1)$	$\{v_2\}$
$CR(u'_3, v_1)$	$\{v_2, v_3, v_4, v_5\}$
$CR(u'_4, v_2)$	$\{v_6\}$
$CR(u'_5, v_2)$	$\{v_7, v_8\}$
$CR(u'_6, v_2)$	$\{v_9\}$
$CR(u'_7, v_7)$	$\{v_{14}\}$
$CR(u'_7, v_8)$	$\{v_{15}, v_{16}, v_{17}\}$
$CR(u'_8, v_9)$	$\{v_{19}, v_{20}, v_{21}\}$

candidate subregions.

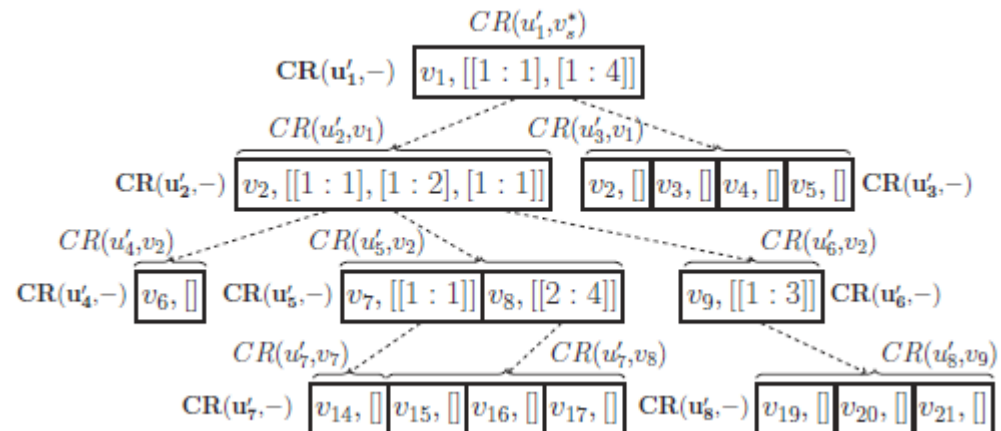
$CR(u'_2, v_1)$ represents the candidates for u'_2 of NEC tree q' in g , with u'_1 (the parent of u'_2 in q') mapped to v_1 .

Candidate Region Exploration

- Candidate Region
 - Organize all candidate subregions together
 - It has the same structure as the NEC tree
 - To facilitate the embedding enumeration.
- The candidate region of the previous example



NEC tree q'

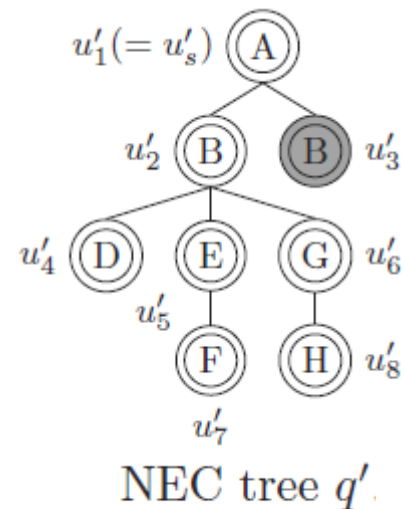
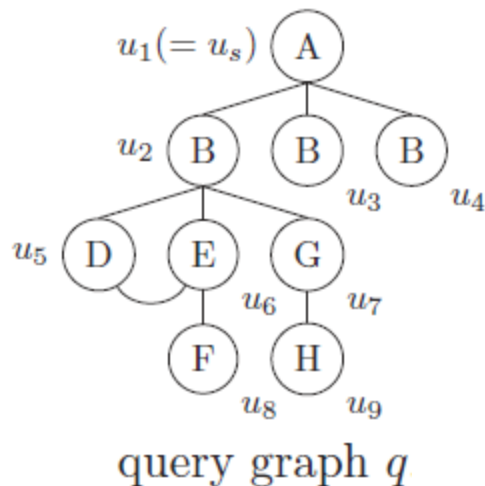


Candidate Region Exploration

To enumerate embeddings of query q in data graph G ,

- Only traverse the candidate region
- Probe G only for non-tree edges validation

Non-tree edges are the edges in query graph q but not in NEC Tree q' .

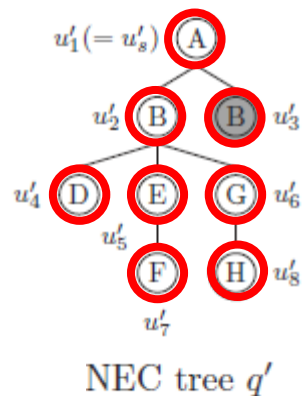


(u_5, u_6) is a non-tree edge.

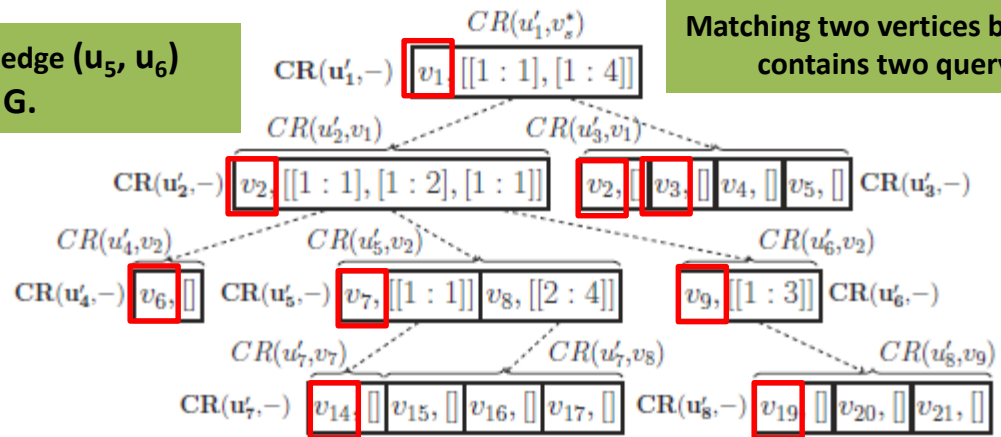
Candidate Region Exploration

- To enumerate embeddings of q in G ,
 - Only traverse the candidate region
 - Probe G only for non-tree edges validation

Example



Validate non-tree edge (u_5, u_6) here using G .



Combine / Permute to get all embeddings based on each NEC node.
In this example, as NEC node u_3' contains two query nodes, this embedding corresponds to 2 embeddings by permuting v_2 and v_3 .

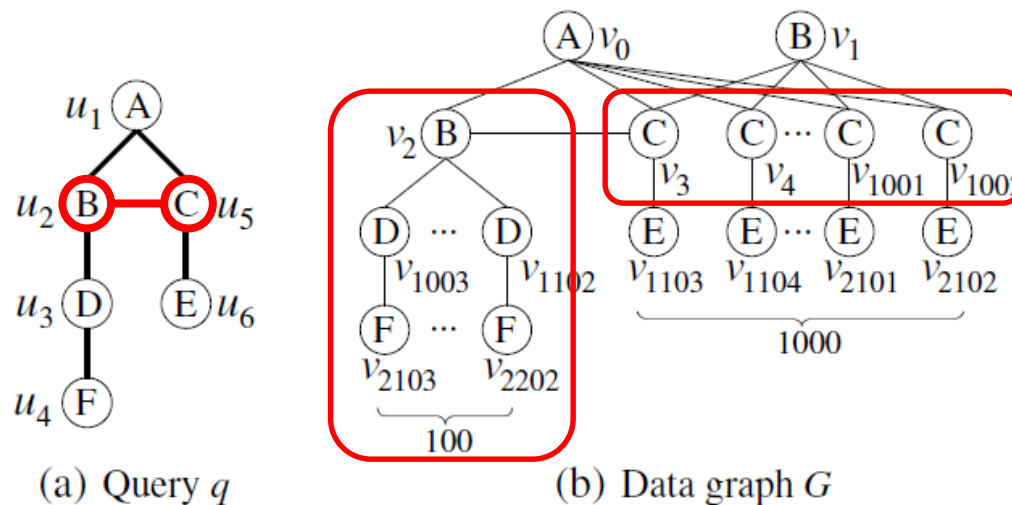
With matching order
($u_1', u_2', u_4', u_5', u_7', u_3', u_6', u_8'$)

CFL-Match

- Overview of CFL-Match
 - A New Search Framework
 - Core-Forest-Leaf based query decomposition
 - Postpone Cartesian product with search order of core, forest and leaf
 - Compact Auxiliary Data Structure
 - Linear to the size of the data graph G
 - In contrast to the exponential sized data structure in TurboISO

Challenges of Subgraph Matching

Challenge I: Redundant Cartesian Products by Dissimilar Vertices.



10^5 - 100 partial mappings are redundant.

Matching order of QuickSI and Turbo_{ISO} : $(u_1, u_2, u_3, u_4, u_5, u_6)$.
 $(u_1, u_2, u_5, u_3, u_4, u_6)$

Cartesian products:

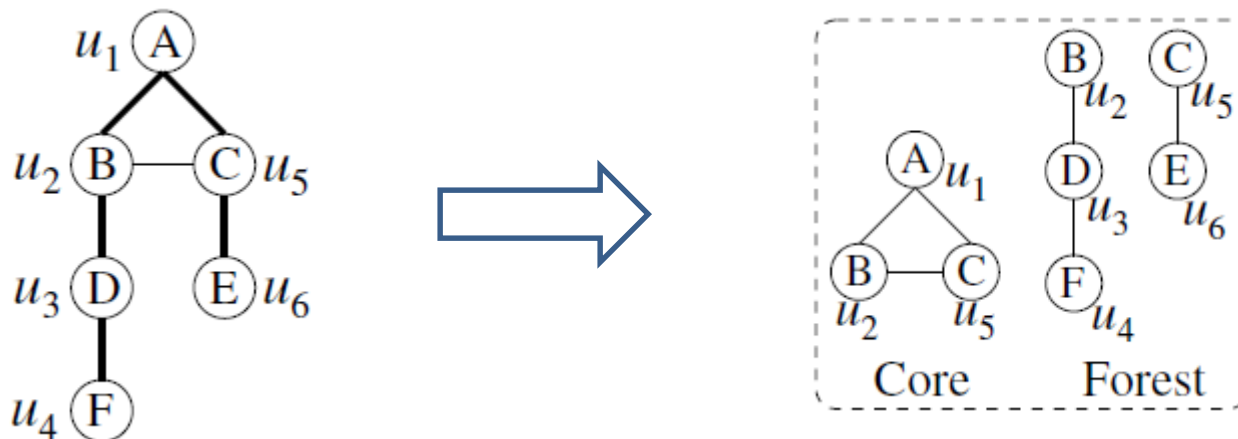
- 100 mappings $(v_0, v_2, v_{1000+i}, v_{2100+i})$ ($3 \leq i \leq 102$) of (u_1, u_2, u_3, u_4)
- 1000 mappings (v_0, v_j) ($3 \leq j \leq 1002$) of (u_1, u_5)

Challenges of Subgraph Matching

Challenge I: Redundant Cartesian Products by Dissimilar Vertices.

Our Solution : Postpone Cartesian products.

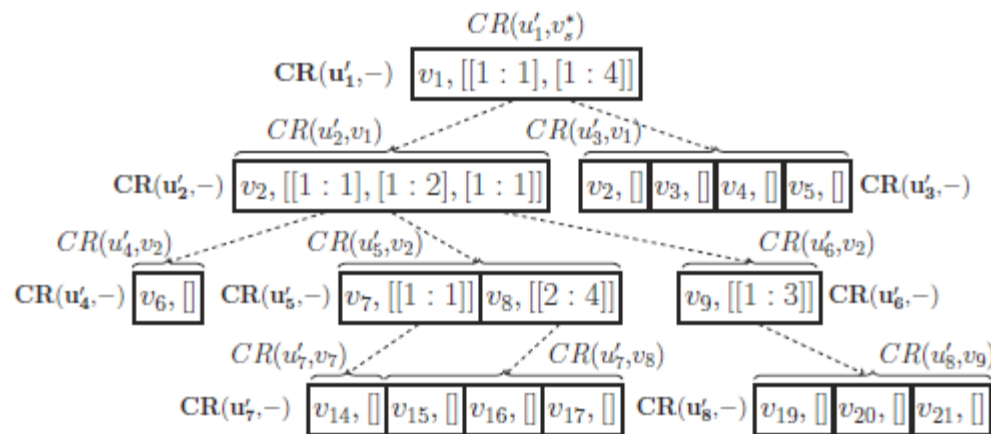
- Decompose q into a **dense subgraph** and a **forest**, and process the dense subgraph first.



Challenges of Subgraph Matching

Challenge II: Exponential size of the path-based data structure in TurboISO.

- TurboISO builds a data structure that materializes all embeddings of query paths in a data graph
- Worst-case space complexity: $O(|V(G)|^{|\mathcal{V}(q)-1|})$.



Challenges of Subgraph Matching

Challenge II: Exponential size of the path-based data structure in TurboISO.

Our Solution:

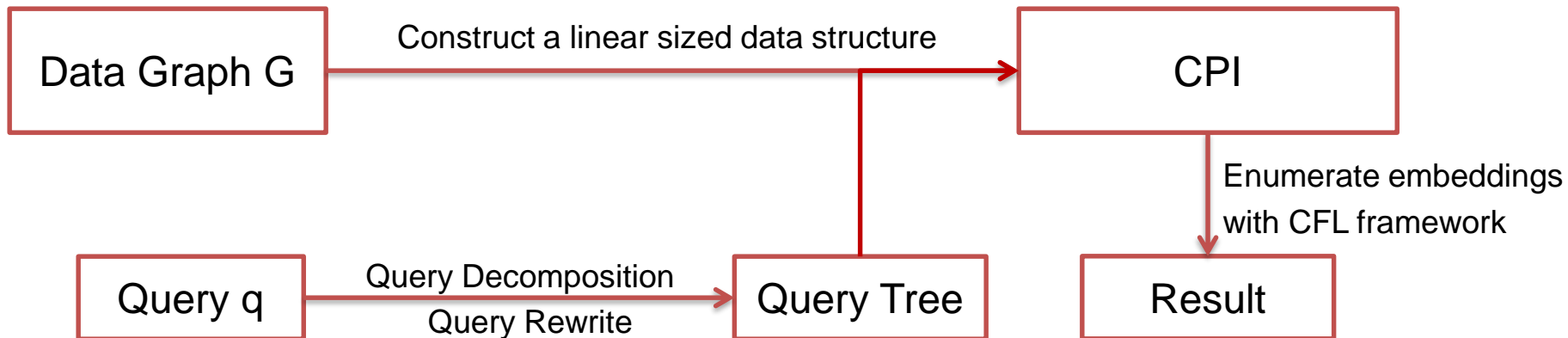
Polynomial-size data structure, compact path-index (CPI) .

Our Approach

➤ CFL-Match

- ❖ A Core-Forest-Leaf decomposition based Framework

- ❖ Compact Path-Index (CPI) based Matching

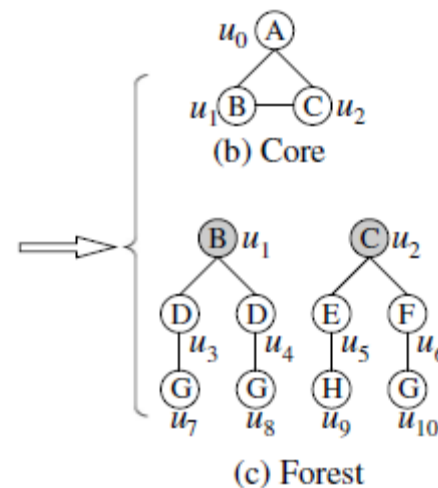
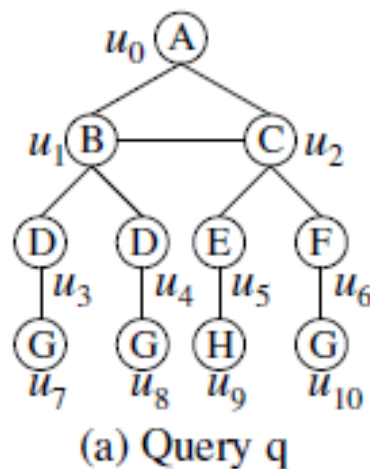


Core-Forest-Leaf Decomposition

➤ Core-Forest Decomposition

Compute the **minimal connected** subgraph **c** containing **all non-tree edges** of **q** regarding any spanning tree.

- The subgraph **c** is the core-structure of **q**, denoted as V_C .
- The subgraph of **q** consisting of all other edges not in the **c**, is called the forest-structure of **q**, denoted as V_T .

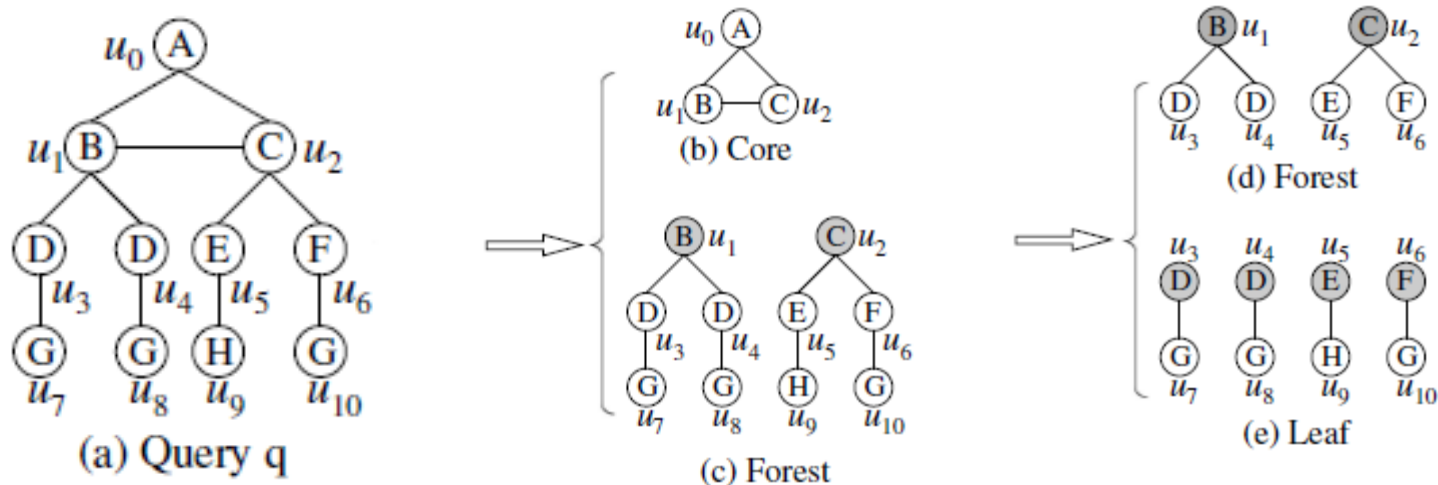


Core-Forest-Leaf Decomposition

➤ Forest-Leaf Decomposition

Compute the set V_l of leaf vertices (degree-one vertices) by rooting each tree at its connection vertex.

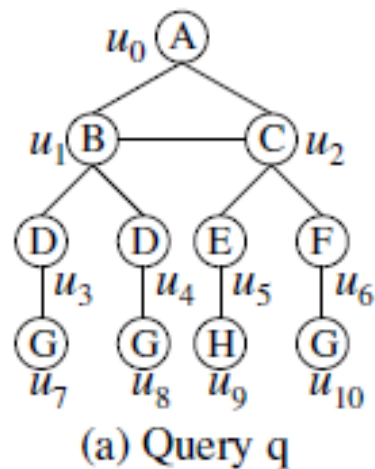
- The set V_l is called the leaf set.
- The set of vertices not in $V_c \cup V_l$ is called the forest set.



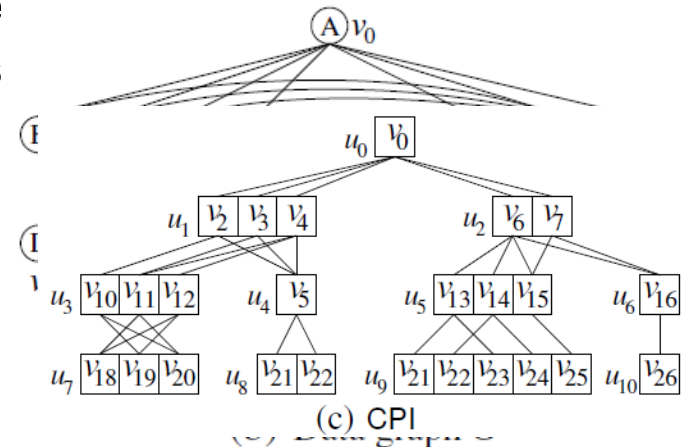
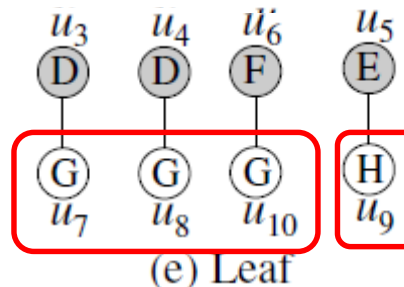
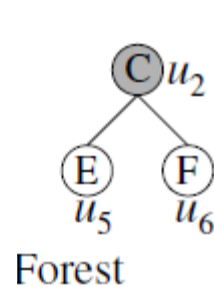
CFL-Match

➤ A Core-Forest Leaf based Framework

- 1) Core-Forest-Leaf Decomposition
- 2) Construct CPI (will be explained later)
- 3) Mapping Extraction
 - i. Core-Match
 - ii. Forest-Match
 - iii. Leaf-Match



group leaf nodes according to label
 or combination instead of enumerating



Auxiliary Data Structure

➤ Compact Path-Index (CPI)

- Compactly store all candidate embeddings of the query tree.

➤ CPI Structure

- **Candidate sets**

Each query node u has a candidate set $u.C$.

- **Edge sets**

This is an edge between $v \in u.C$ and $v' \in u'.C$ for adjacent query nodes u and u' in CPI if and only if (v, v') exists in G .

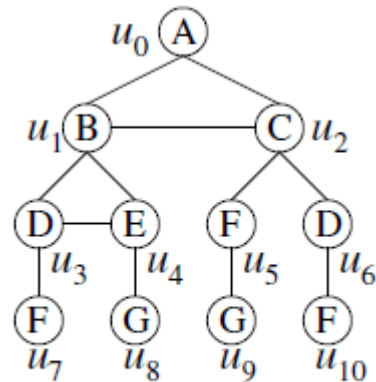
Auxiliary Data Structure

➤ Compact Path-Index (CPI)

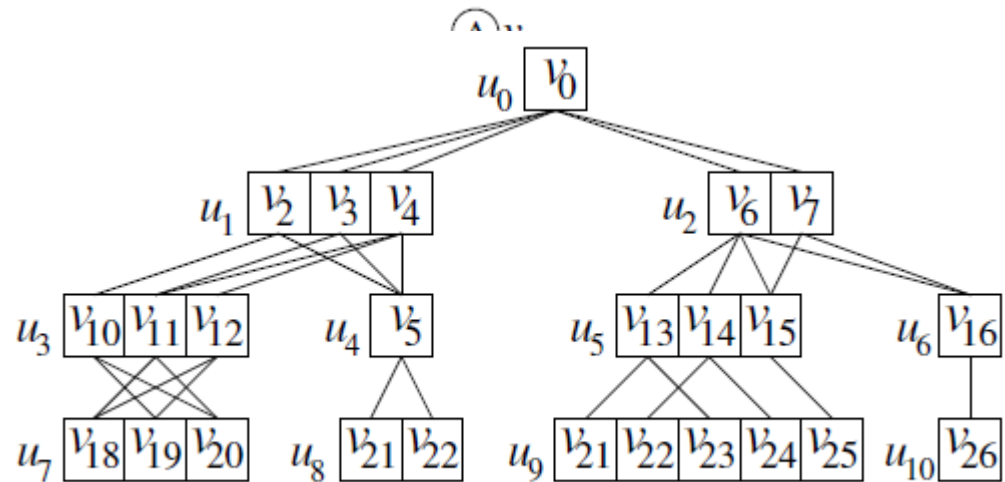
- Compactly store all candidate embeddings of the query tree.

➤ CPI Structure

- Example



(a) Query q



(c) CPI
(d) Data graph G

Auxiliary Data Structure

➤ Soundness of CPI

For every query node u in CPI, if there is an embedding of q in G that maps u to v , then v must be in $u.C$.

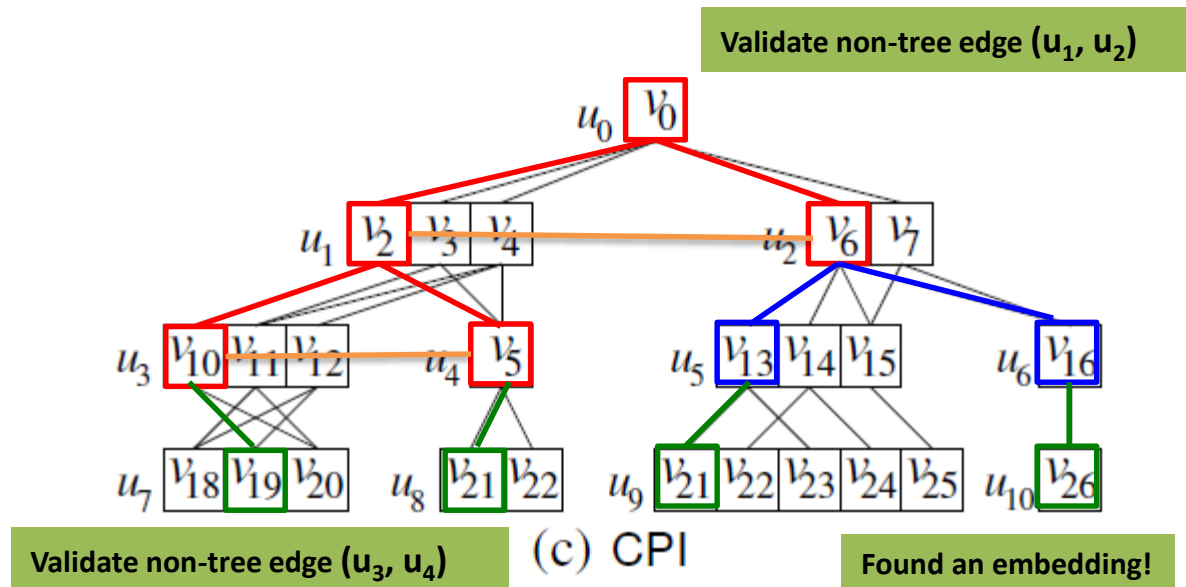
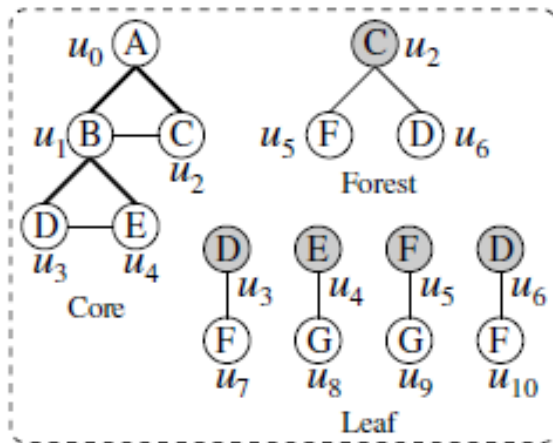
Theorem

Given a sound CPI, all embeddings of q in G can be computed by **traversing only the CPI** while G is only probed for non-tree edge checkings.

➤ The CPI in the previous example is a sound CPI.

CPI-based Match

- Traverse CPI to find mappings for query vertices
 - Only probe **G** for non-tree edge validation



Matching order (u_0 , u_1 , u_4 , u_3 , u_2 , u_5 , u_6 , u_7 , u_8 , u_9 , u_{10})

Tree

Forest

Leaf

Summary

- Graph Pattern Matching in Graph Database
 - Three index-based methods: G-Index, FG-Index, Swift-Index
 - Advanced subgraph isomorphism testing: QuickSI
- Graph Pattern Matching in Single Large Data Graph
 - TurboISO
 - Reduce query size using NEC
 - Candidate Region Exploration
 - CFL-Match
 - CFL-Framework based on query decomposition
 - Compact data structure CPI

Thank you!

Questions?

