Continuous Subgraph Matching

20230721

Keynote

- Continuous Subgraph Matching
- Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction
- TurboFlux, SymBi, etc.

Definitions

- Streaming Graph
- Subgraph Isomorphism
- Subgraph Matching
- Multi-way Join

Streaming Graph

- A graph G is called a **streaming graph** if it changes dynamically following a sequence of update operations ΔG including 4 kinds of updates, i.e., vertex addition/deletion and edge addition/deletion.

- We just consider edge addition and deletion by convention.

Label

- Undirected vertex-labeled graphs
- L is the function mapping a label, i.e., $L(v)$, to a vertex $v \in V(G)$.

- Edge-labeled: for each edge, add a vertex with degree 2

Subgraph Isomorphism

- Given a query graph $Q = \{ V(Q), E(Q), L \}$ and a data graph $G = \{ V(G), E(G), L \}$, Q is subgraph isomorphic to G if there exists an **injective function** $f : V(Q) \rightarrow V(G)$, such that
- (1) $\forall\, u \in V(Q)$, we have $L(u) = L(f(u))$ where $f(u) \in V(G)$;
- (2) $\forall\, e(u_1, u_2) \in E(Q)$, we have $e(f(u_1), f(u_2)) \in E(G)$.

- Label
- Adjacency

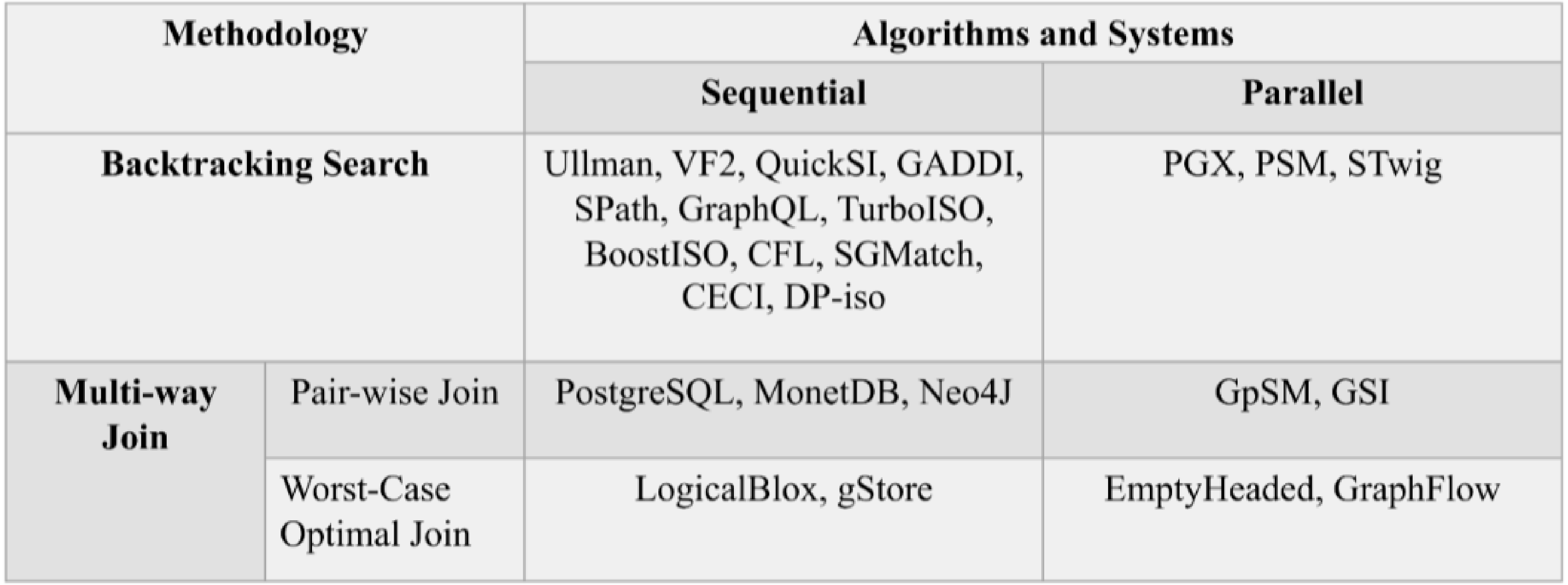
- NP-hard

Subgraph Matching

- Subgraph matching returns **all subgraphs** of G that are **isomorphic** to Q. Each match can be expressed as a set of one-to-one matching pairs $\{(u \leftrightarrow f(u))\}$.

Subgraph Matching

1. exploration-based



Methodology		Algorithms and Systems	
		Sequential	Parallel
Backtracking Search		Ullman, VF2, QuickSI, GADDI, SPath, GraphQL, TurboISO, BoostISO, CFL, SGMatch, CECI, DP-iso	PGX, PSM, STwig
Multi-way Join	Pair-wise Join	PostgreSQL, MonetDB, Neo4J	GpSM, GSI
	Worst-Case Optimal Join	LogicalBlox, gStore	EmptyHeaded, GraphFlow

2. join-based, regard the graph as a database, conduct muti-way joins to find all matches

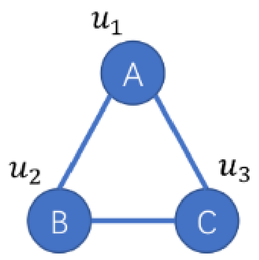
Multi-way Join

- Given Q and G, we can model the SM problem as a **multi-way join**

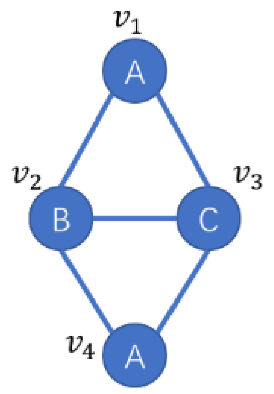
$$Q = \bowtie_{e(u_x,u_y)\in E(Q)} R(u_x, u_y)$$

- where each vertex in $V(Q)$ corresponds to an attribute,
- each edge $e(u_x, u_y) \in E(Q)$ corresponds to a relation $R(u_x, u_y)$ and

$$R(u_x, u_y) = \left\{ e(v_x, v_y) \in E(G) \middle| L(u_x) = L(v_x) \wedge L(u_y) = L(v_y) \wedge L\left(e(u_x, u_y)\right) = L\left(e(v_x, v_y)\right) \right\}$$



Q



G

T_1

u_1	u_2
v_1	v_2
v_4	v_2
…	…

T_2

u_1	u_3
v_1	v_2
v_4	v_2
…	…

T_3

u_2	u_3
v_2	v_3

$$R = T_1 \bowtie T_2 \bowtie T_3$$

Continuous Subgraph Matching (CSM)

- Given a query graph Q, a data graph G, and a graph update stream ΔG, the continuous subgraph matching task is to find the incremental matches (i.e., newly added or decreased subgraph matches) of Q for each update operation in ΔG.

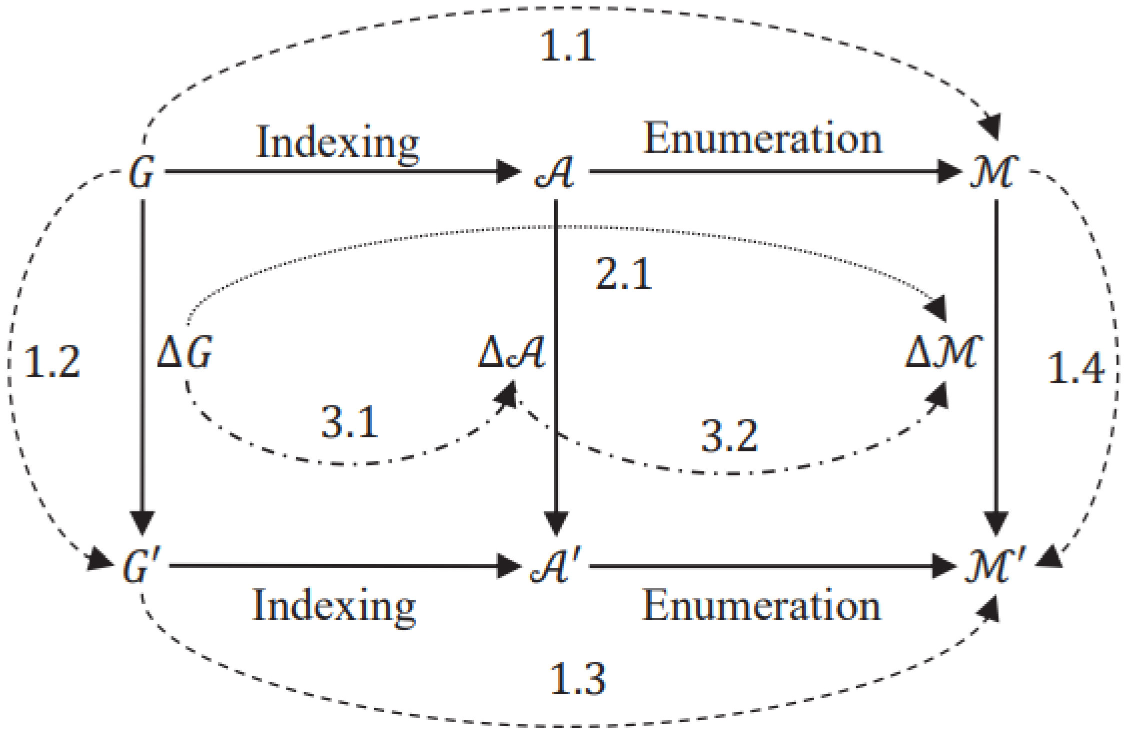
Continuous Subgraph Matching (CSM)

- Incremental matches: the increased (positive) or decreased (negative) subgraph matches of Q owing to the graph updates.

- Finding incremental matches involves 2 major steps, namely
- (1) **candidate maintenance** (including candidate generation and index update) there is a trade-off between update efficiency and candidate accuracy
- (2) **incremental match generation**.

- Due to the NP-hardness of subgraph isomorphism, **incremental matching generation** dominates the overall cost.

Incremental view maintenance



source: An In-Depth Study of Continuous Subgraph Matching

Continuous Subgraph Matching (CSM)

- w/ auxiliary data structure
    - IncIsoMatch, Graphflow, …
- w/o auxiliary data structure
    - SJ-Tree, TurboFlux, SymBi, CaLiG,…

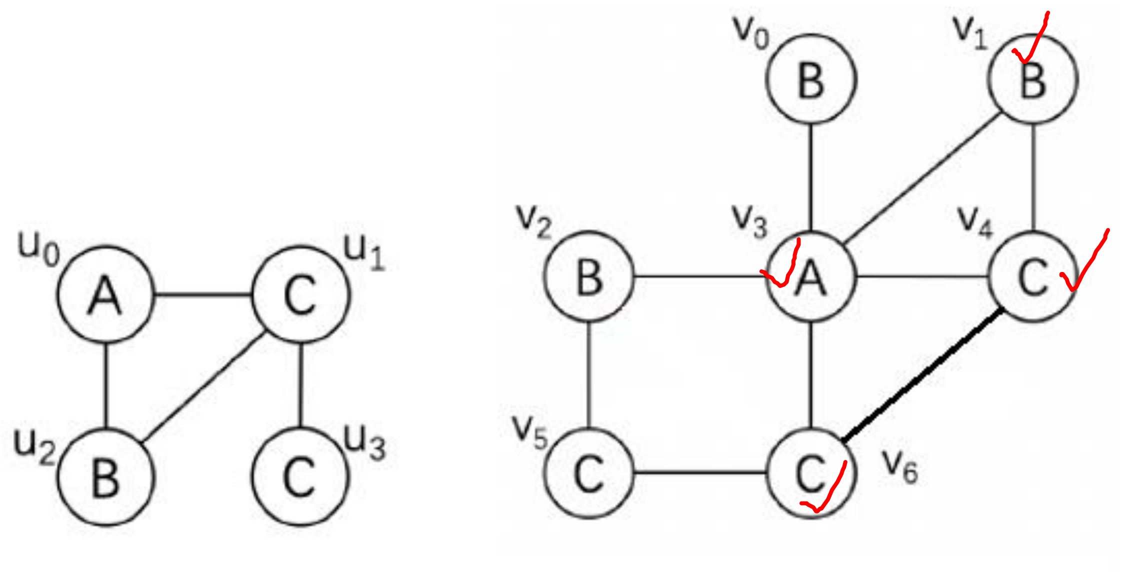
CaLiG (Candidate Lighting Graph)

- CaLiG organizes **candidate matching pairs** in a vertex-pair graph, where each node represents a pair (u, v) of **query vertex $\boldsymbol{u}$** and **data vertex $\boldsymbol{v}$**.
- A pair of vertices u and v form a matching pair, shorted as (u, v)-MP, if $L(u) = L(v)$.
- Each (u, v)-MP has a lighting state ON or OFF, indicating whether v can **match** u or not, i.e., whether v is a candidate of u.
- (u, v)-MP.$state \in \{ON,\ OFF\}$

source: Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction

CaLiG Index (Directed Graph)

- The CaLiG index for Q and G is a directed graph where each **node** represents an MP.
- There is a **directed edge** from (u_i, v_j)-MP to (u_k, v_l)-MP if it holds that
- (1) $e(u_i, u_k) \in E(Q)$ and $e\left(v_j, v_l\right) \in E(G)$, and
- (2) (u_i, v_j)-MP is ON or (u_i, v_j)-MP is turned OFF **after** (u_k, v_l)-MP.



(a) Query Graph (b) Streaming Graph

$$f = \{ u_1 \leftrightarrow v_4, u_3 \leftrightarrow v_6, u_0 \leftrightarrow v_3, u_2 \leftrightarrow v_1 \}$$

$CaLiG$ contains $1 \times 1 + 1 \times 3 + 2 \times 3 = 10$ nodes.

```
for each $(u, v)$-MP $\in CaLiG$ do
    for each $(u', v') \in (N_Q(u), N_G(v))$ do
        if $(u', v')$-MP $\in CaLiG$ then
            add an edge from $(u', v')$-MP to $(u, v)$-MP;
```

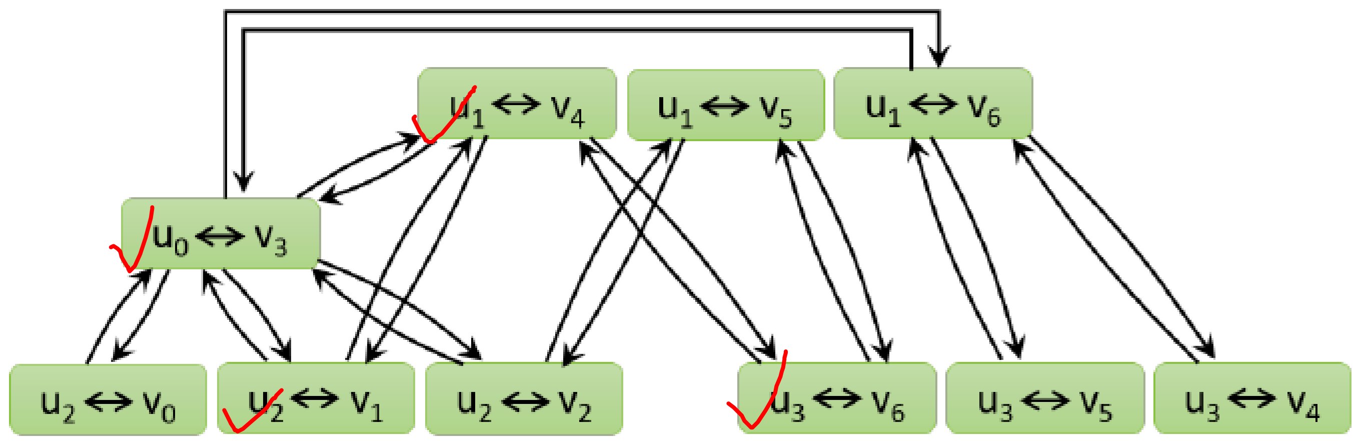
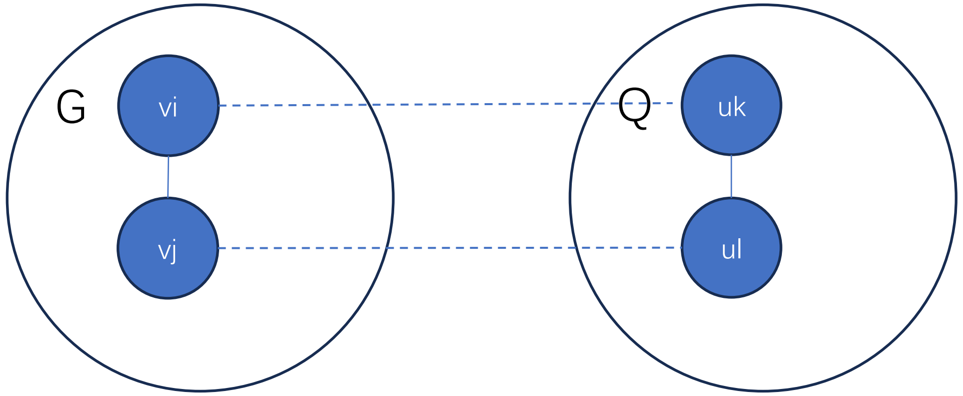


Fig. 4. CaLiG for the query graph and data graph in Figure 1, where the lighting states of all MPs are "ON"

Bigraph for (u, v)-MP

- v matches u only if v's neighbors match u's neighbors as well.
- Although each neighbor of u can find a candidate in v's neighbors, v's neighbors may still fail to match u's neighbors when one neighbor of v is taken as candidates of multiple neighbors of u, violating the **injective** requirement.

- Bipartite graph

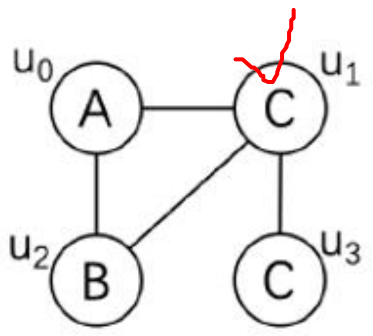


Bigraph for (u, v)-MP

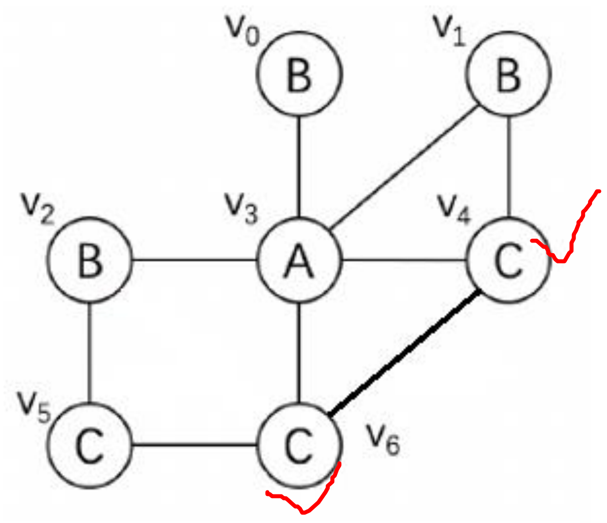
- A bigraph for (u, v)-MP, denoted by $BI(u, v)$, is a bipartite graph with 2 disjoint sets $N_Q(u)$ and $N_G(v)$, where there is an edge between $u_i \in N_Q(u)$ and $v_j \in N_G(v)$ iff. (u_i, v_j)-MP is an **in-neighbor** of (u, v)-MP in the CaLiG.

- For a matching pair (u, v), to determine whether v matches u, we can just consider the in-neighbors of (u, v)-MP in CaLiG, rather than examine all the vertex pairs

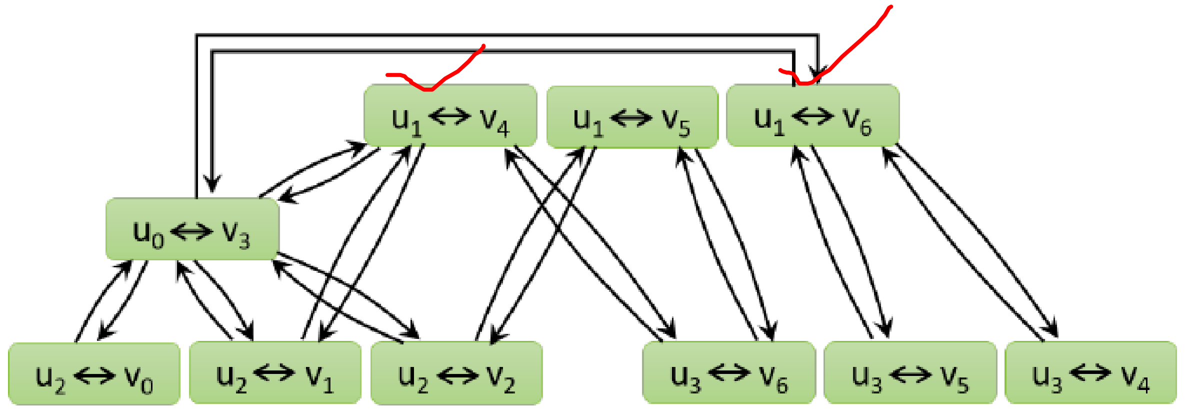
$$\{(u_i, v_j) | u_i \in N_Q(u) \land v_j \in N_G(v) \land L(u_i) = L(v_j)\}.$$

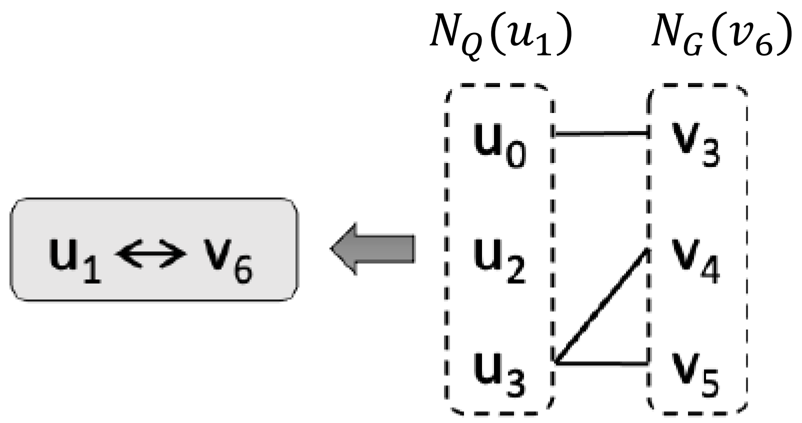


(a) Query Graph

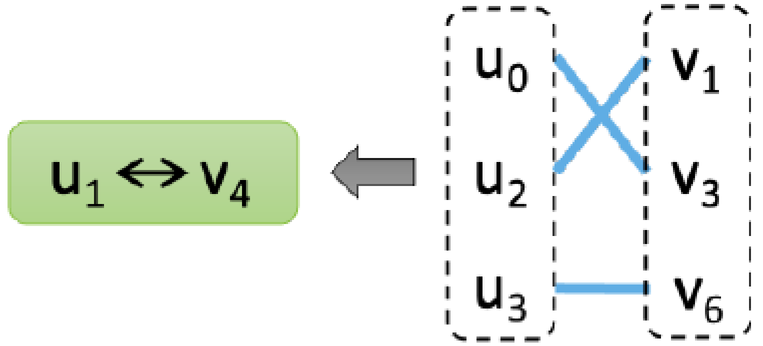


(b) Streaming Graph


$u_1 \leftrightarrow v_4$
$u_1 \leftrightarrow v_5$
$u_1 \leftrightarrow v_6$
$u_0 \leftrightarrow v_3$
$u_2 \leftrightarrow v_0$
$u_2 \leftrightarrow v_1$
$u_2 \leftrightarrow v_2$
$u_3 \leftrightarrow v_6$
$u_3 \leftrightarrow v_5$
$u_3 \leftrightarrow v_4$



(a) No injective matching in the bigraph $BI(u_1, v_6)$ of (u_1, v_6)-MP.



(b) Having an injective matching in the bigragh $BI(u_1, v_4)$ of (u_1, v_4)-MP.

u_2 does not have any candidates, which means when we take $(v_6 \leftrightarrow u_1)$ as a partial match,
no candidates will be available for u_2. Thus, v_6 should not be a candidate of u_1. (u_1, v_6)-MP.state should be OFF.

Bigraph for (u, v)-MP

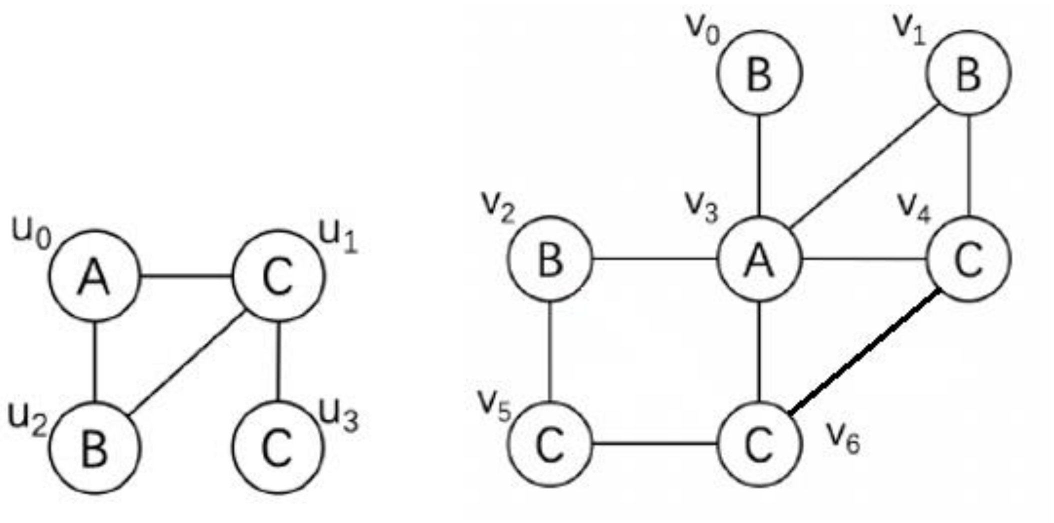
- Given a (u, v)-MP, its lighting state is OFF iff. there is no injective matching for $N_Q(u)$ in $BI(u, v)$.
- An OFF-state node means it does not belong to any match.

- Compute matching for a bigraph
- Hungarian algorithm: $O(|V||E|)$
- Hopcroft–Karp algorithm: $O\left(\sqrt{|V|}|E|\right)$

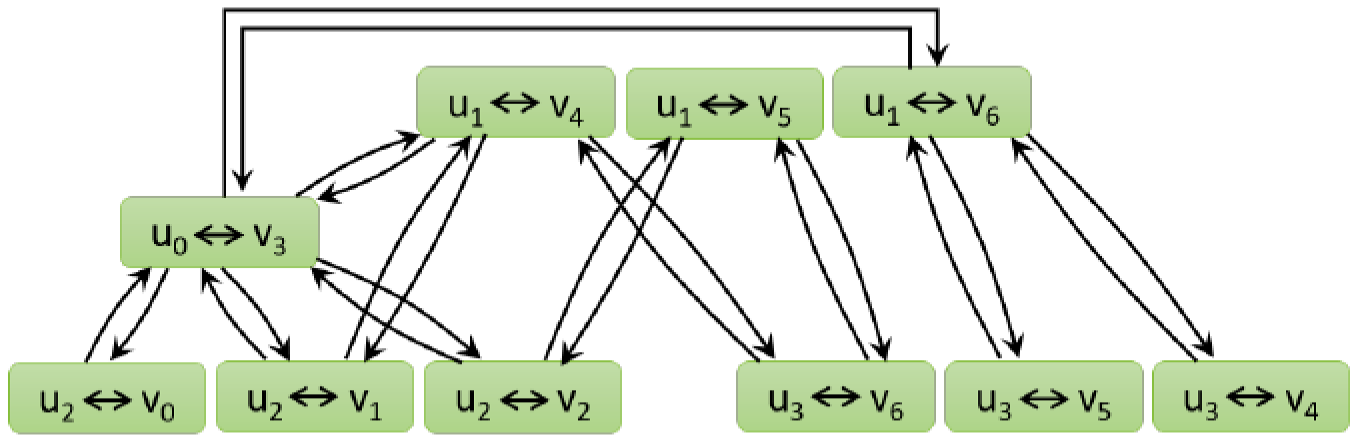
- Hall's Theorem

CaLiG Initialization

Algorithm 1: ConstructCaLiG(G, Q)
Add all nodes in CaLiG w/ initial state ON.
Add all edges according to adjacency.



(a) Query Graph (b) Streaming Graph



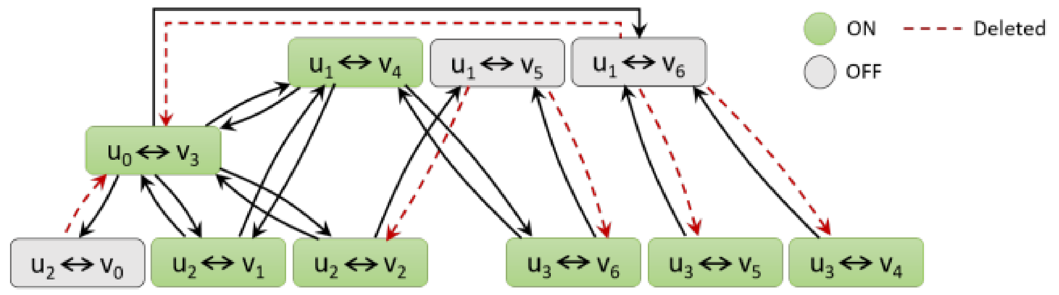
$u_1 \leftrightarrow v_4$
$u_1 \leftrightarrow v_5$
$u_1 \leftrightarrow v_6$
$u_0 \leftrightarrow v_3$
$u_2 \leftrightarrow v_0$
$u_2 \leftrightarrow v_1$
$u_2 \leftrightarrow v_2$
$u_3 \leftrightarrow v_6$
$u_3 \leftrightarrow v_5$
$u_3 \leftrightarrow v_4$

CaLiG Initialization

$BI(u_2, v_0), BI(u_1, v_5), BI(u_1, v_6)$ do not have any injective matching. We remove their out-going edges.

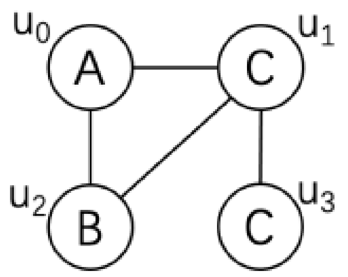
Algorithm 2: IndexInitialization(*CaLiG*)

```
Input: A candidate lighting graph $CaLiG$
1 for each $(u, v)$-MP $\in$ $CaLiG$ do
2     build a bigraph $BI(u, v)$ for $(u, v)$-MP;
3 for each $(u, v)$-MP $\in$ $CaLiG$ do
4     if $(u, v)$-MP.state = ON and $BI(u, v)$ has no injective matching then
5         $(u, v)$-MP.state $\leftarrow$ OFF;
6         OFF-Propagation($CaLiG$, $(u, v)$-MP);
```

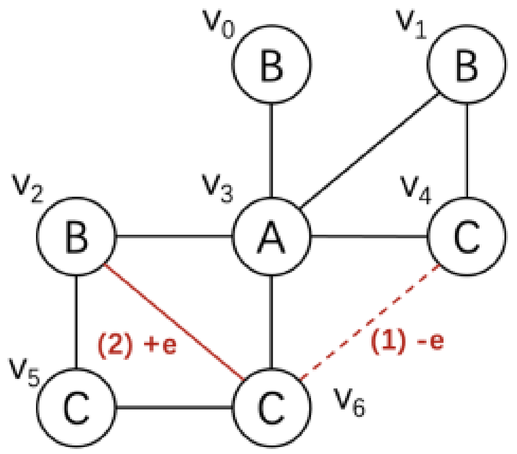


ON
OFF
Deleted
$u_1 \leftrightarrow v_4$
$u_1 \leftrightarrow v_5$
$u_1 \leftrightarrow v_6$
$u_0 \leftrightarrow v_3$
$u_2 \leftrightarrow v_0$
$u_2 \leftrightarrow v_1$
$u_2 \leftrightarrow v_2$
$u_3 \leftrightarrow v_6$
$u_3 \leftrightarrow v_5$
$u_3 \leftrightarrow v_4$

(a) Step 1. Turn off all nodes that do not have any injective matchings.



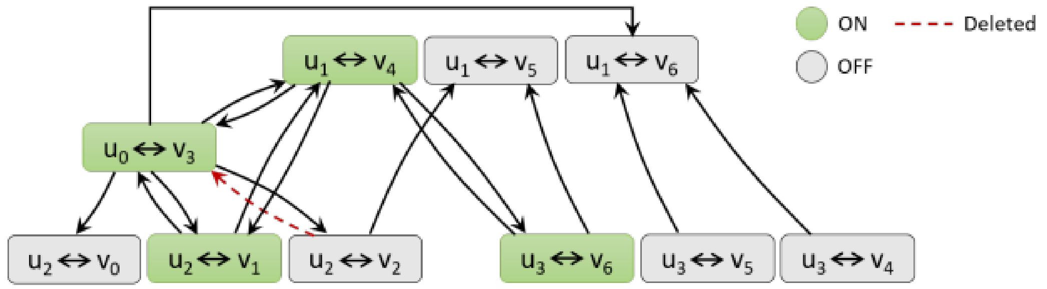
(a) Query Graph



(b) Streaming Graph

CaLiG Initialization

(u_2, v_2)-*MP*, (u_3, v_5)-*MP and* (u_3, v_4)-*MP are turned off.*



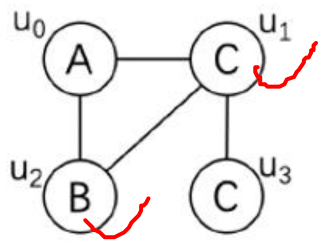
ON
Deleted
OFF
$u_1 \leftrightarrow v_4$
$u_1 \leftrightarrow v_5$
$u_1 \leftrightarrow v_6$
$u_0 \leftrightarrow v_3$
$u_2 \leftrightarrow v_0$
$u_2 \leftrightarrow v_1$
$u_2 \leftrightarrow v_2$
$u_3 \leftrightarrow v_6$
$u_3 \leftrightarrow v_5$
$u_3 \leftrightarrow v_4$

(b) Step 2. OFF-Propagation.

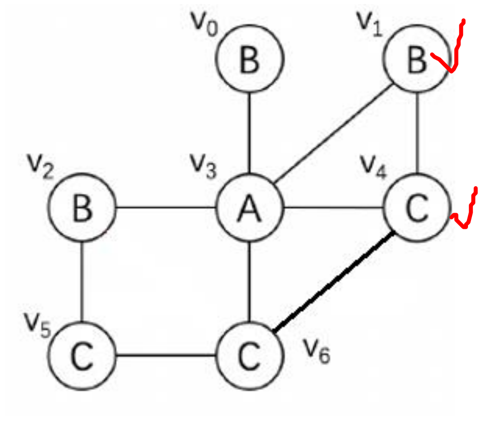
```
Algorithm 2: IndexInitialization($CaLiG$)
  Input: A candidate lighting graph $CaLiG$
1 for each $(u,v)$-MP $\in CaLiG$ do
2     build a bigraph $BI(u,v)$ for $(u,v)$-MP;
3 for each $(u,v)$-MP $\in CaLiG$ do
4     if $(u,v)$-MP.state = ON and $BI(u,v)$ has no injective matching then
5         $(u,v)$-MP.state $\leftarrow$ OFF;
6         OFF-Propagation($CaLiG$, $(u,v)$-MP);
```

```
Algorithm 3: OFF-Propagation($CaLiG$, $(u,v)$-MP)
  Input: $CaLiG$ and a matching pair $(u,v)$-MP that was turned off in the previous round
1 for each $(u',v')$-MP $\in Out_{CaLiG}(u,v)$ do
2     if $(u',v')$-MP.state = ON then
3         delete the edge from $(u,v)$-MP to $(u',v')$-MP;
4         update $BI(u',v')$;
5         if $BI(u',v')$ has no injective matching then
6             $(u',v')$-MP.state $\leftarrow$ OFF;
7             OFF-Propagation($CaLiG$, $(u',v')$-MP);
```

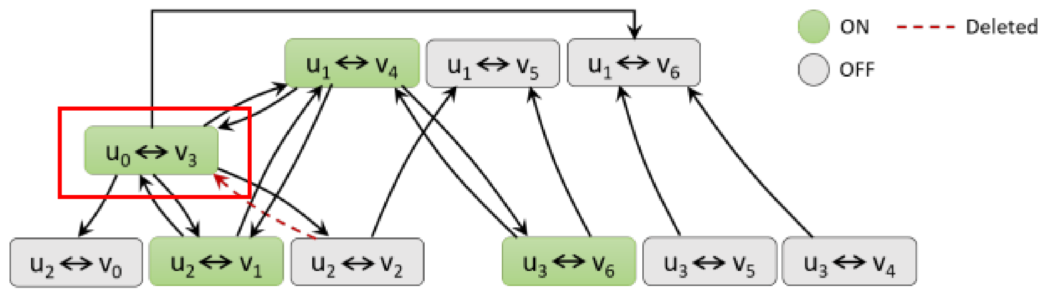
CaLiG Initialization



(a) Query Graph



(b) Streaming Graph



(b) Step 2. OFF-Propagation.

Algorithm 3: OFF-Propagation(*CaLiG*, (u, v)-MP)

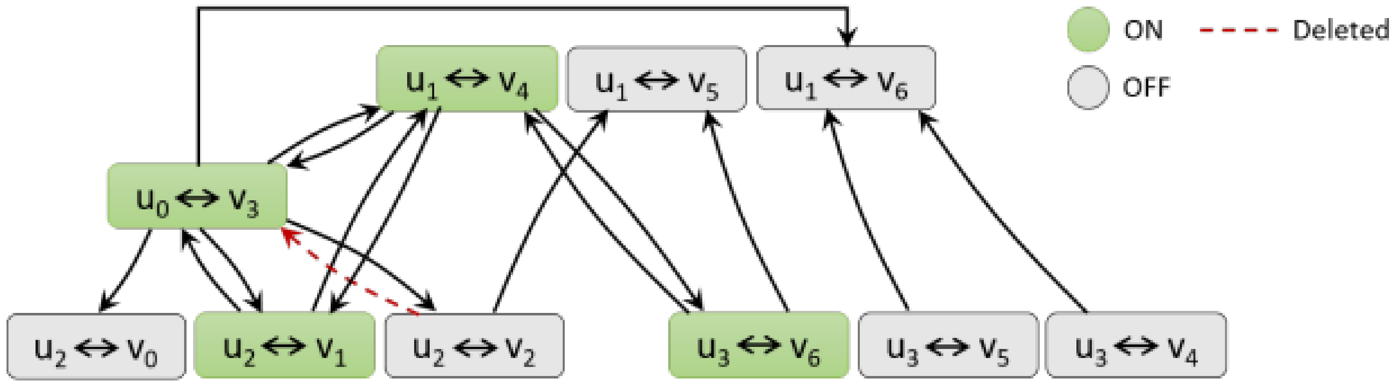
```
Input: CaLiG and a matching pair $(u, v)$-MP that was turned off in the previous round
1 for each $(u', v')$-MP $\in Out_{CaLiG}(u, v)$ do
2     if $(u', v')$-MP.state = ON then
3         delete the edge from $(u, v)$-MP to $(u', v')$-MP;
4         update $BI(u', v')$;
5         if $BI(u', v')$ has no injective matching then
6             $(u', v')$-MP.state $\leftarrow$ OFF;
7             OFF-Propagation(CaLiG, $(u', v')$-MP);
```

CaLiG Initialization

- CaLiG has a **directed edge** from (u_i, v_j)-MP to (u_k, v_l)-MP iff.
- (1) $e(u_i, u_k) \in E(Q)$ and $e(v_j, v_l) \in E(G)$, and
- (2) (u_i, v_j)-MP is ON or (u_i, v_j)-MP is turned OFF **after** (u_k, v_l)-MP.

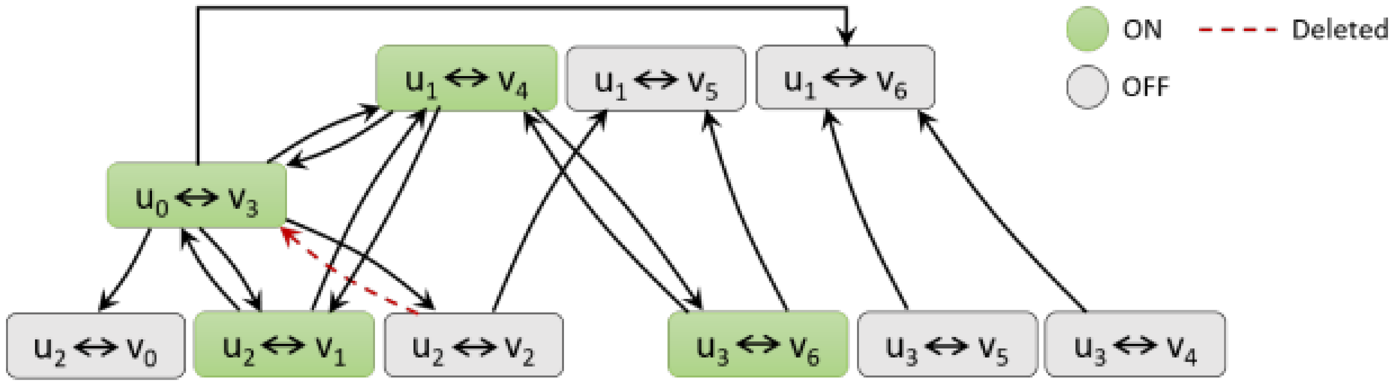
(2) Is assured by OFF-Propagation.

u2 ↔ v2 is turned OFF after u1 ↔ v5


ON
Deleted
OFF
$u_1 \Leftrightarrow v_4$
$u_1 \Leftrightarrow v_5$
$u_1 \Leftrightarrow v_6$
$u_0 \Leftrightarrow v_3$
$u_2 \Leftrightarrow v_0$
$u_2 \Leftrightarrow v_1$
$u_2 \Leftrightarrow v_2$
$u_3 \Leftrightarrow v_6$
$u_3 \Leftrightarrow v_5$
$u_3 \Leftrightarrow v_4$

CaLiG Initialization

- For each ON node (u, v)-MP, its OFF in-neighbors have been deleted when the in-neighbors are turned OFF in the previous round, so the remaining MPs are all ON nodes.

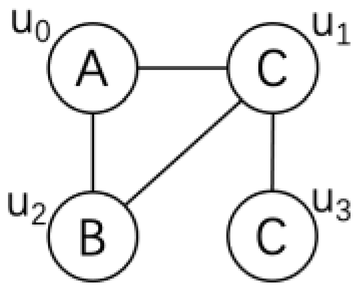


ON
Deleted
OFF
$u_1 \leftrightarrow v_4$
$u_1 \leftrightarrow v_5$
$u_1 \leftrightarrow v_6$
$u_0 \leftrightarrow v_3$
$u_2 \leftrightarrow v_0$
$u_2 \leftrightarrow v_1$
$u_2 \leftrightarrow v_2$
$u_3 \leftrightarrow v_6$
$u_3 \leftrightarrow v_5$
$u_3 \leftrightarrow v_4$

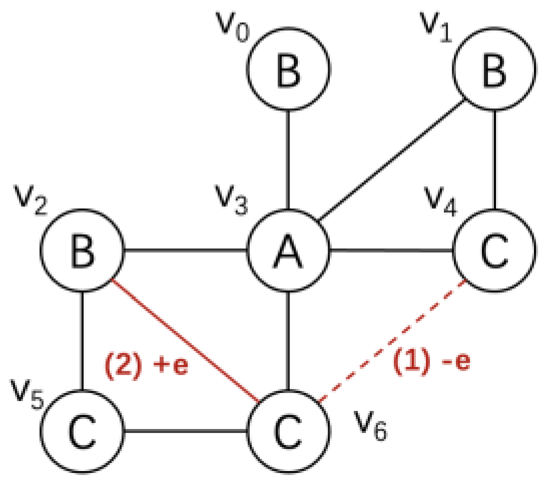
OFF $\rightarrow$ OFF
ON $\rightarrow$ OFF
ON $\rightarrow$ ON
OFF $\rightarrow$ ON

CaLiG Initialization

Candidate sets in common framework			
C(u0)	v3		
C(u1)	v4		
C(u2)	v1	v2	
C(u3)	v4	v5	v6

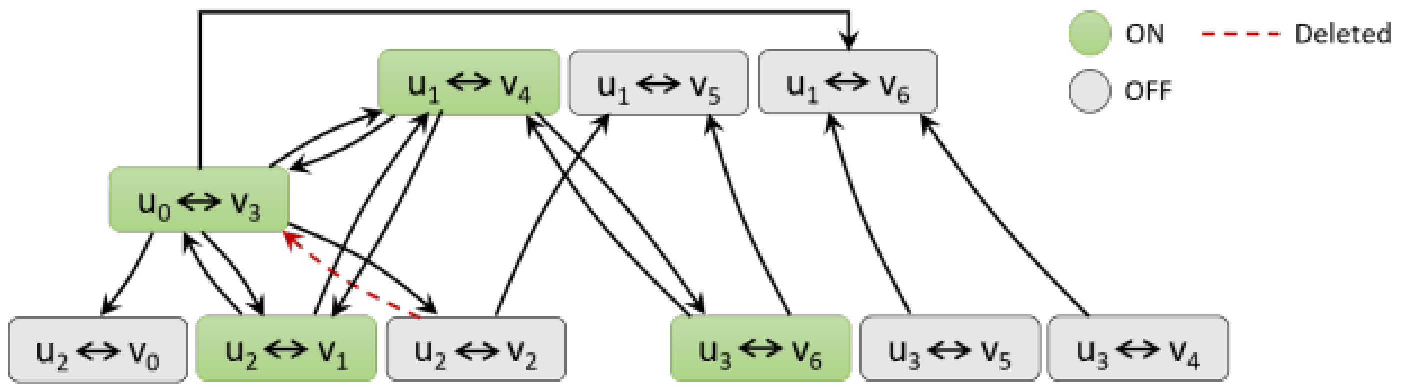


(a) Query Graph



(b) Streaming Graph

但在这里使用类似的BI(u,v)能做到和CaLiG一样的filtering


ON
Deleted
OFF
$u_1 \Leftrightarrow v_4$
$u_1 \Leftrightarrow v_5$
$u_1 \Leftrightarrow v_6$
$u_0 \Leftrightarrow v_3$
$u_2 \Leftrightarrow v_0$
$u_2 \Leftrightarrow v_1$
$u_2 \Leftrightarrow v_2$
$u_3 \Leftrightarrow v_6$
$u_3 \Leftrightarrow v_5$
$u_3 \Leftrightarrow v_4$

Complexity Analysis

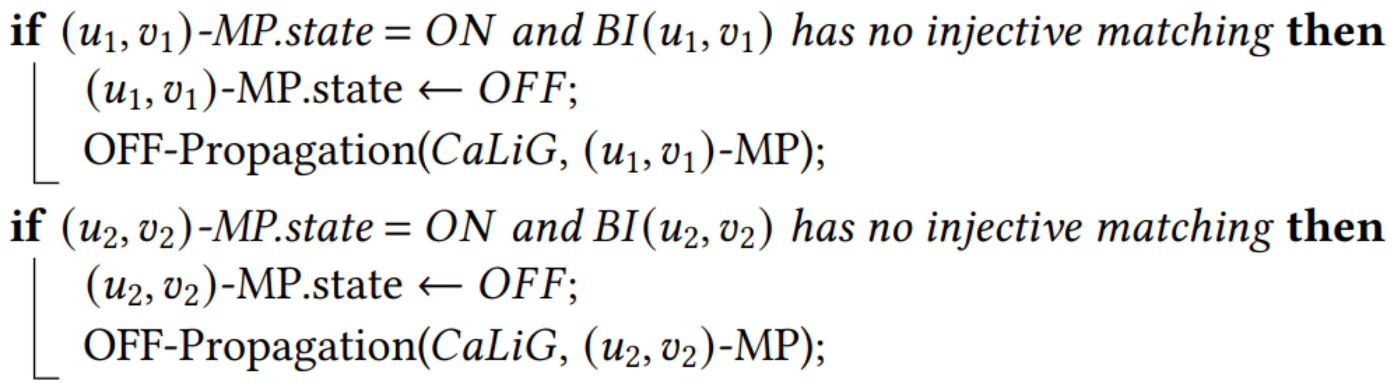
- The number of nodes in CaLiG is $O(|V(G)| \times |V(Q)|)$
- The number of edges is $O(|E(G)| \times |E(Q)|)$
- The storage cost of $BI(u, v)$ is $O(|N_Q(u)| \times |N_G(v)|)$
- The overall space cost is $O(|E(Q)| \times |E(G)|)$

- Each edge in CaLiG is visited at most once.
- Injective matching is computed in time $O(\Delta(Q)^{2.5})$
- The overall time cost is $O(|E(Q)| \times |E(G)| \times \Delta(Q)^{2.5})$

Edge Deletion

- Delete $e(v_1, v_2)$ from data graph G
- $e(v_1, v_2)$ maybe a candidate of a query edge $e(u_1, u_2)$ *s.t.* $L(u_1) = L(v_1)$ and $L(u_2) = L(v_2)$.
- This will cause edge deletion in CaLiG.

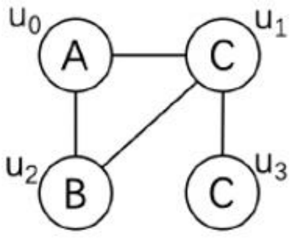
- Delete the edges between (u_1, v_1)-MP and (u_2, v_2)-MP from CaLiG



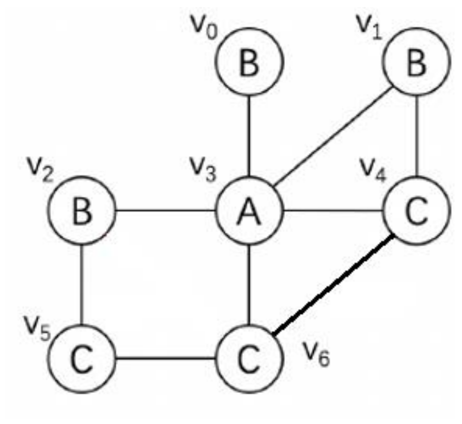
```
if $(u_1, v_1)$-MP.state = ON and BI$(u_1, v_1)$ has no injective matching then
    $(u_1, v_1)$-MP.state $\leftarrow$ OFF;
    OFF-Propagation(CaLiG, $(u_1, v_1)$-MP);
if $(u_2, v_2)$-MP.state = ON and BI$(u_2, v_2)$ has no injective matching then
    $(u_2, v_2)$-MP.state $\leftarrow$ OFF;
    OFF-Propagation(CaLiG, $(u_2, v_2)$-MP);
```

Edge Deletion

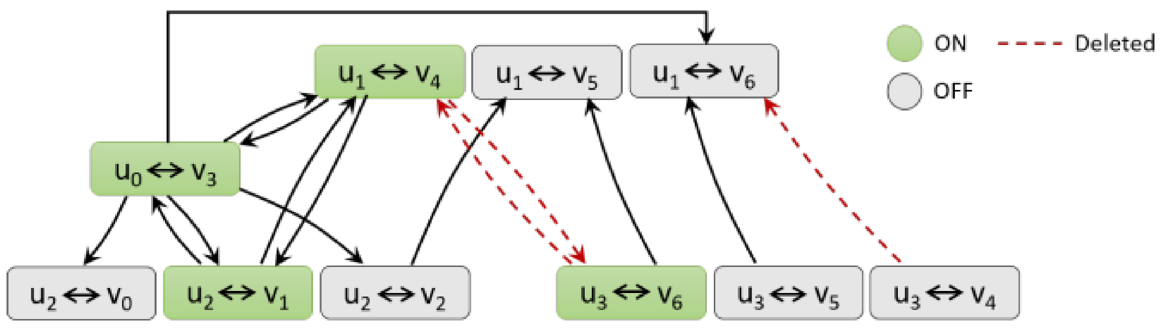
- Delete $e(v_4, v_6)$
- Check $BI(u_1, v_4)$ and $BI(u_3, v_6)$
- Check $BI(u_0, v_3)$ and $BI(u_2, v_1)$



(a) Query Graph

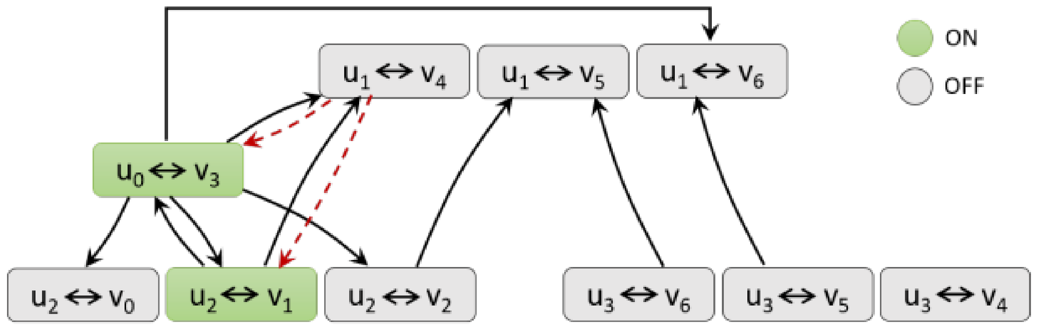


(b) Streaming Graph



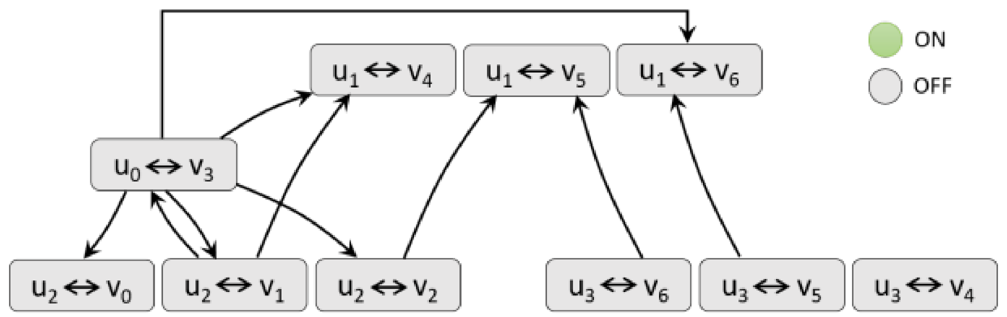
ON
Deleted
OFF
$u_1 \leftrightarrow v_4$
$u_1 \leftrightarrow v_5$
$u_1 \leftrightarrow v_6$
$u_0 \leftrightarrow v_3$
$u_2 \leftrightarrow v_0$
$u_2 \leftrightarrow v_1$
$u_2 \leftrightarrow v_2$
$u_3 \leftrightarrow v_6$
$u_3 \leftrightarrow v_5$
$u_3 \leftrightarrow v_4$

(a) Step 1. Delete all update edges in CaLiG.



ON
OFF
$u_1 \leftrightarrow v_4$
$u_1 \leftrightarrow v_5$
$u_1 \leftrightarrow v_6$
$u_0 \leftrightarrow v_3$
$u_2 \leftrightarrow v_0$
$u_2 \leftrightarrow v_1$
$u_2 \leftrightarrow v_2$
$u_3 \leftrightarrow v_6$
$u_3 \leftrightarrow v_5$
$u_3 \leftrightarrow v_4$

(b) Step 2. Two nodes are turned off.



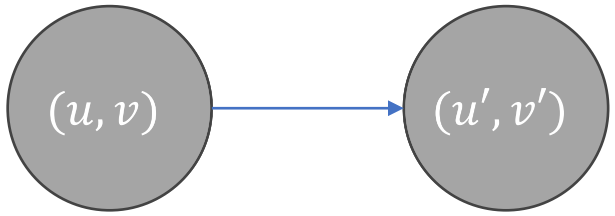
ON
OFF
$u_1 \leftrightarrow v_4$
$u_1 \leftrightarrow v_5$
$u_1 \leftrightarrow v_6$
$u_0 \leftrightarrow v_3$
$u_2 \leftrightarrow v_0$
$u_2 \leftrightarrow v_1$
$u_2 \leftrightarrow v_2$
$u_3 \leftrightarrow v_6$
$u_3 \leftrightarrow v_5$
$u_3 \leftrightarrow v_4$

(c) Step 3. OFF-Propagation.

Edge Addition

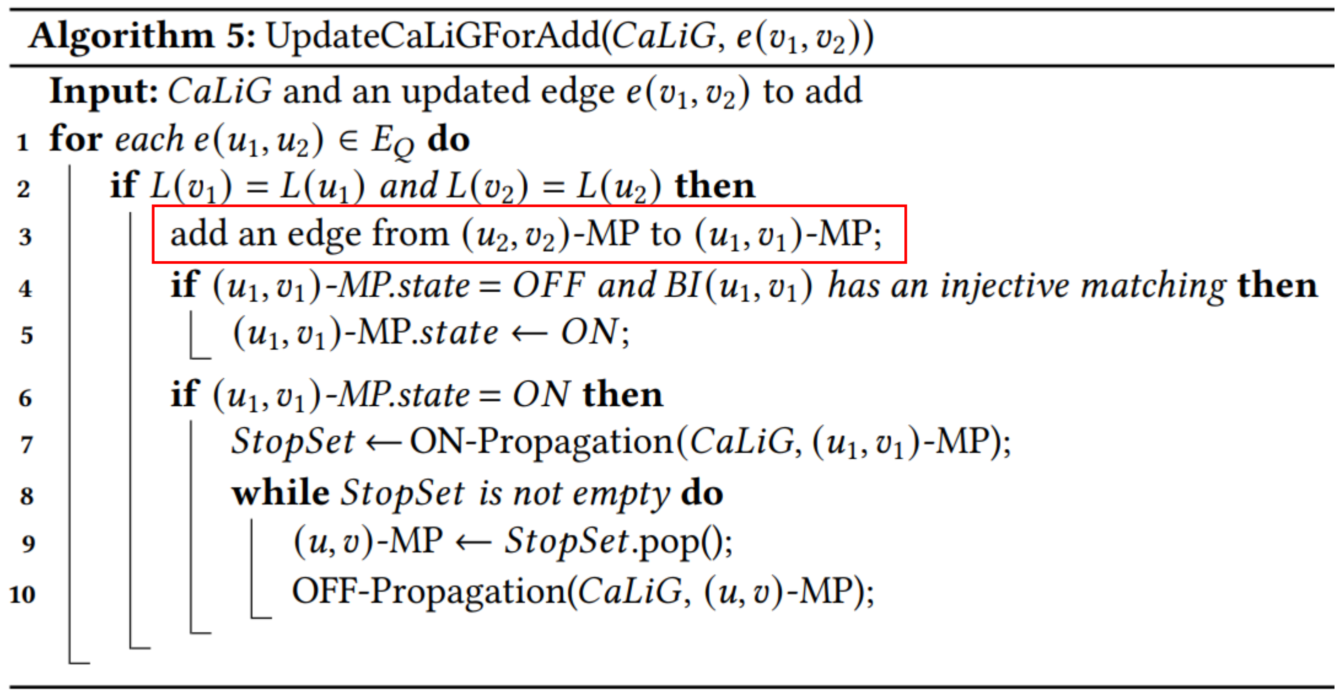
- CaLiG has a **directed edge** from (u_i, v_j)-MP to (u_k, v_l)-MP iff.
- (1) $e(u_i, u_k) \in E(Q)$ and $e\left(v_j, v_l\right) \in E(G)$, and
- (2) (u_i, v_j)-MP is ON or $\left(u_i, v_j\right)$-MP is turned OFF **after** (u_k, v_l)-MP.

- Lemma 4.2. If there is an edge from the OFF-state (u, v)-MP to the OFF-state (u', v')-MP, the turning ON of (u, v)-MP will not cause the turning ON of (u', v')-MP.



Assuming turning on (u, v)-MP causes turning on (u', v')-MP …

Edge Addition

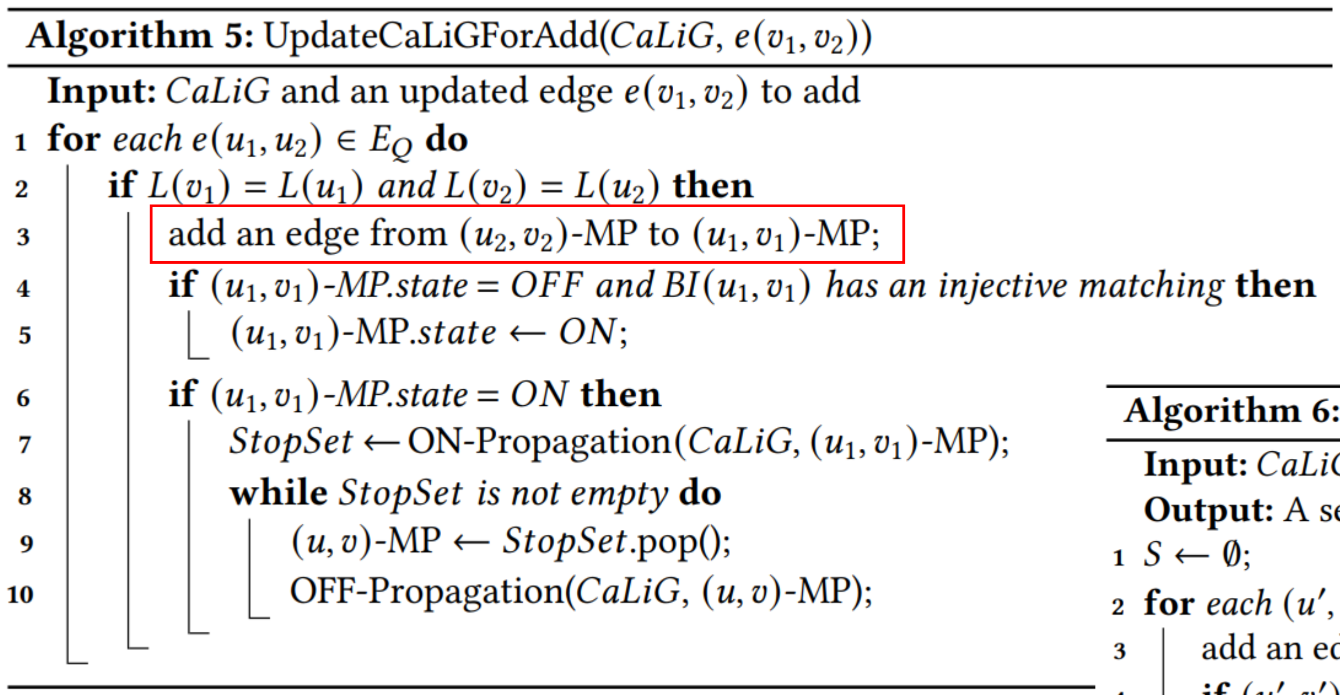


```
Algorithm 5: UpdateCaLiGForAdd($CaLiG$, $e(v_1, v_2)$)
Input: $CaLiG$ and an updated edge $e(v_1, v_2)$ to add
for each $e(u_1, u_2) \in E_Q$ do
    if $L(v_1) = L(u_1)$ and $L(v_2) = L(u_2)$ then
        add an edge from $(u_2, v_2)$-MP to $(u_1, v_1)$-MP;
        if $(u_1, v_1)$-MP.state = OFF and $BI(u_1, v_1)$ has an injective matching then
            $(u_1, v_1)$-MP.state $\leftarrow$ ON;
        if $(u_1, v_1)$-MP.state = ON then
            $StopSet \leftarrow$ ON-Propagation($CaLiG$, $(u_1, v_1)$-MP);
            while $StopSet$ is not empty do
                $(u, v)$-MP $\leftarrow$ $StopSet$.pop();
                OFF-Propagation($CaLiG$, $(u, v)$-MP);
```

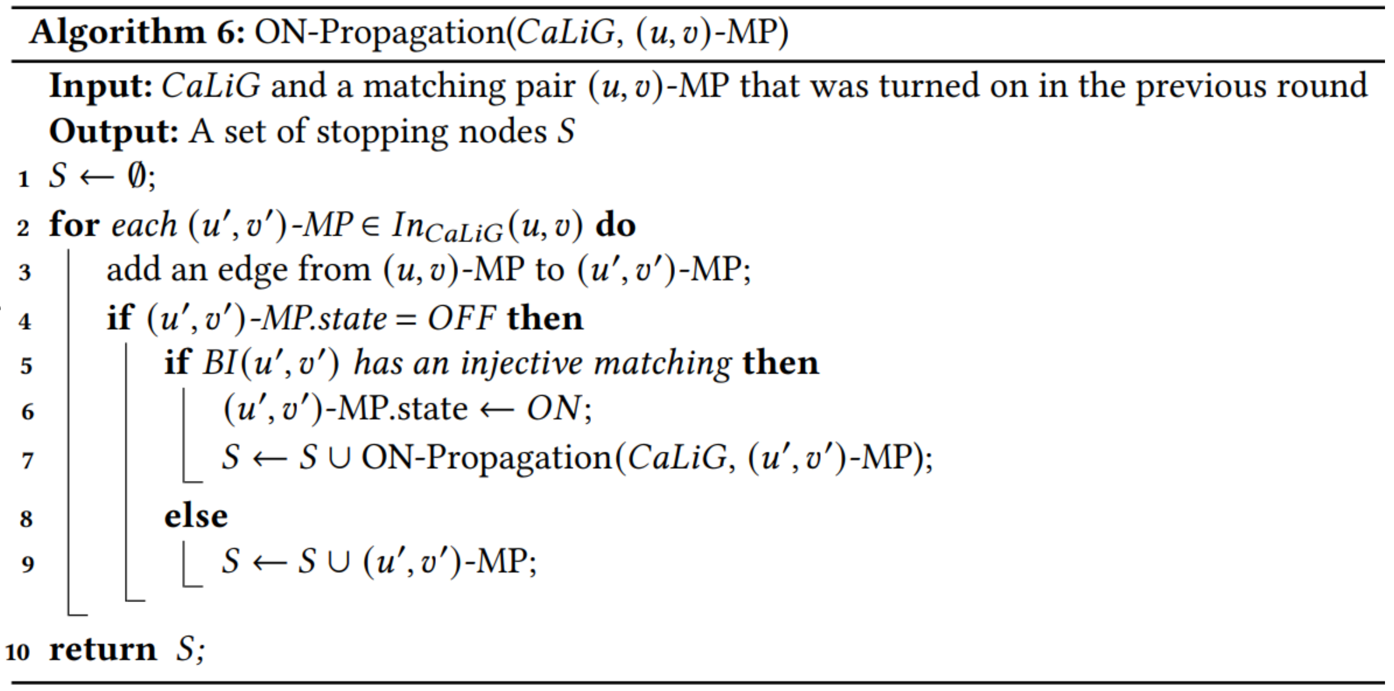
If $e(u_1, u_2)$ doesn't existed in G,
$e((u_2, v_2)$-MP, (u_1, v_1)-MP) doesn't exist in CaLiG.

Edge Addition

When a node is not turned ON by ON-Propagation, it is recorded as a stopping node.



```
Algorithm 5: UpdateCaLiGForAdd($CaLiG$, $e(v_1, v_2)$)
  Input: $CaLiG$ and an updated edge $e(v_1, v_2)$ to add
for each $e(u_1, u_2) \in E_Q$ do
    if $L(v_1) = L(u_1)$ and $L(v_2) = L(u_2)$ then
        add an edge from $(u_2, v_2)$-MP to $(u_1, v_1)$-MP;
        if $(u_1, v_1)$-MP.state = OFF and $BI(u_1, v_1)$ has an injective matching then
            $(u_1, v_1)$-MP.state $\leftarrow$ ON;
        if $(u_1, v_1)$-MP.state = ON then
            StopSet $\leftarrow$ ON-Propagation($CaLiG$, $(u_1, v_1)$-MP);
            while StopSet is not empty do
                $(u, v)$-MP $\leftarrow$ StopSet.pop();
                OFF-Propagation($CaLiG$, $(u, v)$-MP);
```



```
Algorithm 6: ON-Propagation($CaLiG$, $(u, v)$-MP)
  Input: $CaLiG$ and a matching pair $(u, v)$-MP that was turned on in the previous round
  Output: A set of stopping nodes $S$
$S \leftarrow \emptyset$;
for each $(u', v')$-MP $\in In_{CaLiG}(u, v)$ do
    add an edge from $(u, v)$-MP to $(u', v')$-MP;
    if $(u', v')$-MP.state = OFF then
        if $BI(u', v')$ has an injective matching then
            $(u', v')$-MP.state $\leftarrow$ ON;
            $S \leftarrow S \cup$ ON-Propagation($CaLiG$, $(u', v')$-MP);
        else
            $S \leftarrow S \cup (u', v')$-MP;
return $S$;
```

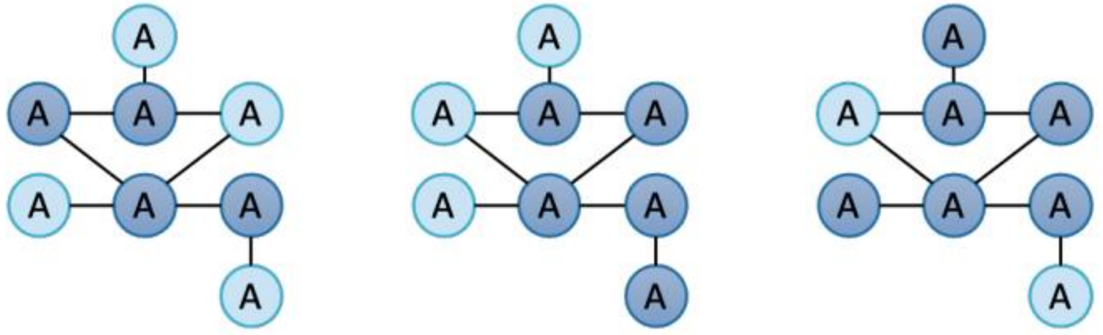
Edge Addition

- Why OFF-Propagation?
- For each ON-state node, the procedure ON-Propagation is invoked to try to turn on more nodes.
- When a node is not turned on by ON-Propagation, it is recorded as a stopping node. The OFF-Propagation is performed for each stopping node to ensure that all the nodes that are newly turned on conforming to subgraph isomorphism.

Kernel-and-Shell Search (KSS)

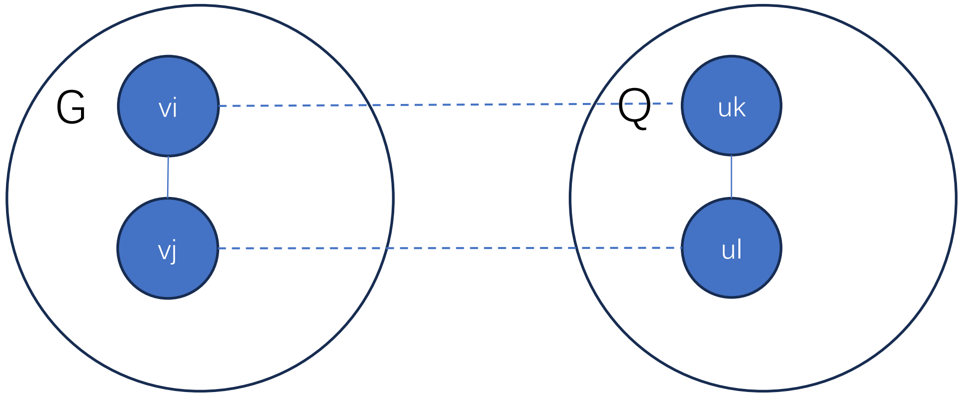
- A Backtracking Search Framework
- Given a query graph Q, its **kernel set**, also known as **connected vertex cover**, is a set of connected vertices *s.t.* each edge in Q has at least one endpoint in the set.

- The **shell set** is the complementary set of the kernel set, where vertices are **independent** of each other.



Kernel-and-Shell Search (KSS)

- Partial Match → Complete Match
- For each **update edge** $e(v_i, v_j)$, we need to find two connected ON-state nodes (u_k, v_i)-MP and (u_l, v_j)-MP in CaLiG that contain v_i and v_j, respectively.
- These 2 nodes will be taken as a partial match which can be extended to a complete match.



Kernel-and-Shell Search (KSS)

- Most existing CSM algorithms accelerate the backtracking by adjusting the **matching order**.
- In contrast, KSS decomposes the query vertices into kernel and shell vertices, making it straightforward to deliver the incremental matches by **joining the partial matches** for kernel vertices and the candidate of shell vertices.

- As there is no edge dependency among shell vertices, the matches can be produced straightforwardly by joining the candidates of shell vertices, w/o any failing backtracks.

Kernel-and-Shell Search (KSS)

- A good decomposition should have as many shell vertices as possible, indicating as few kernel vertices as possible.
- NP-hard
- Approximate algorithm for kernel and shell

Kernel-and-Shell Search (KSS)

Algorithm 7: FindMatches($CaLiG$, $e(v_1, v_2)$, $Kernel$, $Shell$)

Input: $CaLiG$, an updated edge $e(v_1, v_2)$, the conditional kernel set K, and shell set S for $e(v_1, v_2)$

Output: Incremental matches due to $e(v_1, v_2)$

```
1  for each $u_1 \in Q$ do
2      if $L(v_1) = L(u_1)$ and $(u_1, v_1)$-MP.state = ON then
3          $m[u_1] \leftarrow v_1$;
4          for each $u_2 \in N_Q(u_1)$ do
5              if $(u_2, v_2)$-MP $\in In_{CaLiG}(u_1, v_1)$ then
6                  $m[u_2] \leftarrow v_2$;
7                  return KSS($m$, $K$, $S$).
```

The matching pair (u_2, v_2)-MP is then determined from the in-neighbors of (u_1, v_1)-MP (lines 4-6). Thus, we have found partial match $\{(u_1 \leftrightarrow v_1, u_2 \leftrightarrow v_2)\}$, denoted as m.

Algorithm 8: $KSS(m, K, S)$

Input: The partial match m, conditional kernel set K, and shell set S

Output: Incremental matches due to $e(v_1, v_2)$

```
1  if $m.size < |K|$ then
2      $th \leftarrow m.size$;
3      $u \leftarrow K[th]$;
4      $Cand(u) \leftarrow$ generate $u$'s candidates;
5      for each $v \in Cand(u)$ do
6          $m' \leftarrow m$;
7          $m'[u] \leftarrow v$;
8          $KSS(m', K, S)$;
9  else
10     for each $u \in S$ do
11         $Cand(u) \leftarrow$ generate $u$'s candidates;
12     return $m \bowtie_{u \in S} Cand(u)$.
```

Kernel-and-Shell Search (KSS)

```
Algorithm 7: FindMatches($CaLiG$, $e(v_1, v_2)$, $Kernel$, $Shell$)
Input: $CaLiG$, an updated edge $e(v_1, v_2)$, the conditional kernel set $K$, and shell set $S$ for
       $e(v_1, v_2)$
Output: Incremental matches due to $e(v_1, v_2)$
for each $u_1 \in Q$ do
    if $L(v_1) = L(u_1)$ and $(u_1, v_1)$-$MP.state = ON$ then
        $m[u_1] \leftarrow v_1$;
        for each $u_2 \in N_Q(u_1)$ do
            if $(u_2, v_2)$-$MP \in In_{CaLiG}(u_1, v_1)$ then
                $m[u_2] \leftarrow v_2$;
                return $KSS(m, K, S)$.
```

Kernel vertices first, and then shell

```
Algorithm 8: $KSS(m, K, S)$
Input: The partial match $m$, conditional kernel set $K$, and shell set $S$
Output: Incremental matches due to $e(v_1, v_2)$
if $m.size < |K|$ then
    $th \leftarrow m.size$;
    $u \leftarrow K[th]$;
    $Cand(u) \leftarrow$ $\boxed{\text{generate } u\text{'s candidates;}}$
    for each $v \in Cand(u)$ do
        $m' \leftarrow m$;
        $m'[u] \leftarrow v$;
        $KSS(m', K, S)$;
else
    for each $u \in S$ do
        $Cand(u) \leftarrow$ generate $u$'s candidates;
    return $m \bowtie_{u \in S} Cand(u)$.
```

After all the kernel vertices have been matched, we generate candidates for all shell vertices and join the candidates with the partial match to report the incremental.

Backtracking Reduction

For KSS, only the kernel part is matched by backtracking search.

Bottleneck of CSM

- Meaning of CaLiG
- The index-based methods often perform better than the others.
- There is a trade-off between update efficiency and candidate accuracy for index designing.
- The indexes of TurboFlux and SymBi are not tight enough. It is worth spending a little more time to obtain tighter candidates, achieving considerable speedups in the subsequent incremental match generation.

Bottleneck of CSM

- Pruning strategies
- Auxiliary data structure
- Matching order φ

IncIsoMatch

- IncIsoMatch is the 1st CSM algorithm.
- It finds incremental matches by **recomputation** and does not generate the initial **matching order** or **index**.
- On each update, IncIsoMatch extracts a subgraph $G' \subseteq G$ based on the added/deleted edge ($G' = G_{diameter}$) and enumerates results on G' instead of G to reduce the search space.

Graphflow

- Use multi-way joins.
- Graphflow doesn’t generate initial matching order or index.
- Use worst case optimal join (WCOJ) to reduce intermediate results.
- Find ΔM with worst-case optimality

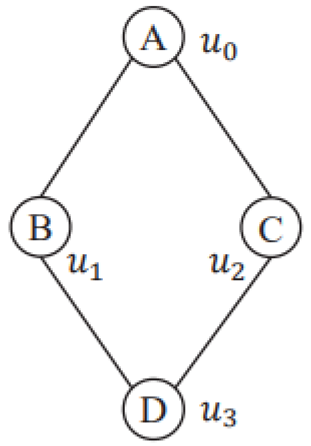
- Considering the fact that **the update edge is contained in the desired matches** if any, the **index-free** algorithms, e.g., IncIsoMatch and Graphflow, find the changed matches by expanding the added/deleted edge w/o generating unnecessary matches.
- Such algorithms may waste too much time in exploring the data graph even if there are not any matches since **they do not employ any information before updating**.

SJ-Tree (Subgraph join tree)

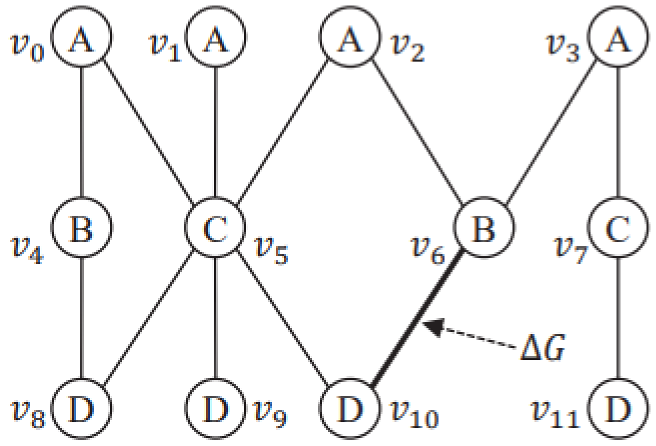
- SJ-Tree defines a **left-deep decomposition tree**, in which each node maintains a set of partial matches. Since the number of matches could be exponential, SJ-tree suffers from an intractable storage overhead.

- φ_0 contains a sequence of query edges.
- The i^{th} leaf node on the left maintains the relation associated with the i^{th} query edge in φ_0, and an internal node records partial results of the query

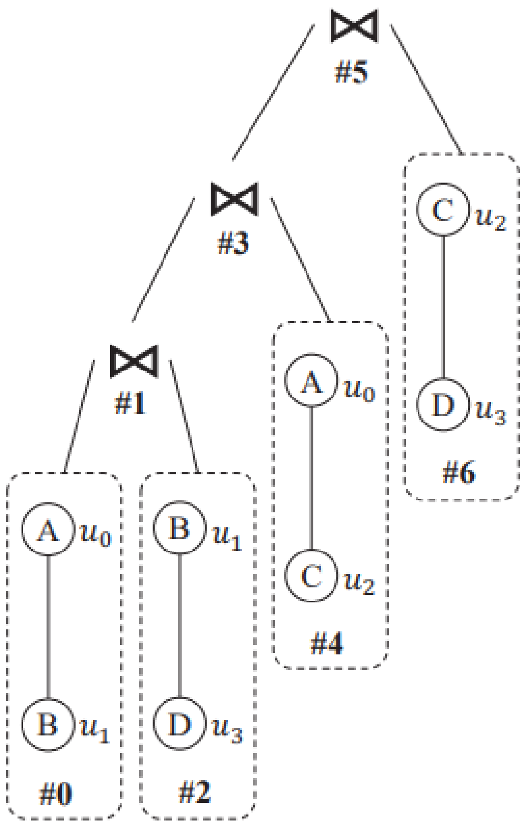
SJ-Tree



(a) Query graph Q.



(c) G': Insert $e(v_6, v_{10})$ into G.



(a) Left-deep tree.

The index of SJ-Tree takes exponential space.

source: An In-Depth Study of Continuous Subgraph Matching

TurboFlux

- Use spanning tree
- Check tree edge first, and then non-tree edges
- Data structure DCG
- For each vertex, record (“subtree all matched”, “path to root”)

SymBi

- TurboFlux spends too much time on **non-tree edges**.
- SymBi uses all query edges.
- Dynamic candidate space (DCS) structure on DAG

- The index constructed by SymBi is **not cost-effective** enough, that is, the candidates could be tightened to archive less total cost.
- **Match Density**
- SymBi doesn’t use bigraph, which brings **reused candidates**.

Summary

Algorithm	Index Space Complexity	Index Time Complexity
IncIsoMatch	N/A	N/A
Graphflow	N/A	N/A
SJ-tree	$O(\|E(G)\|^{\|E(Q)\|})$	$O(\|E(G)\|^{\|E(Q)\|})$
TurboFlux	$O(\|E(G)\|\|V(Q)\|)$	$O(\|E(G)\|\|V(Q)\|)$
SymBi	$O(\|E(G)\|\|E(Q)\|)$	$O(\|E(G)\|\|E(Q)\|)$
CaLiG	$O(\|E(Q)\|\|E(G)\|)$	$O(\|E(Q)\|\|E(G)\|\Delta(Q)^{2.5})$

out of time

out of memory

$$O(\sum_{(u,v)\in D}(|N_{CaLiG}(u,v)| + |N_Q(u)|^{2.5}))$$

Tree queries: TurboFlux
Sparse queries and dense data graph: SymBi
Sparse queries and sparse data graph: Graphflow
Dense queries: Graphflow
Otherwise: SymBi

- 结束