Pattern Matching in Large-Scale Graphs

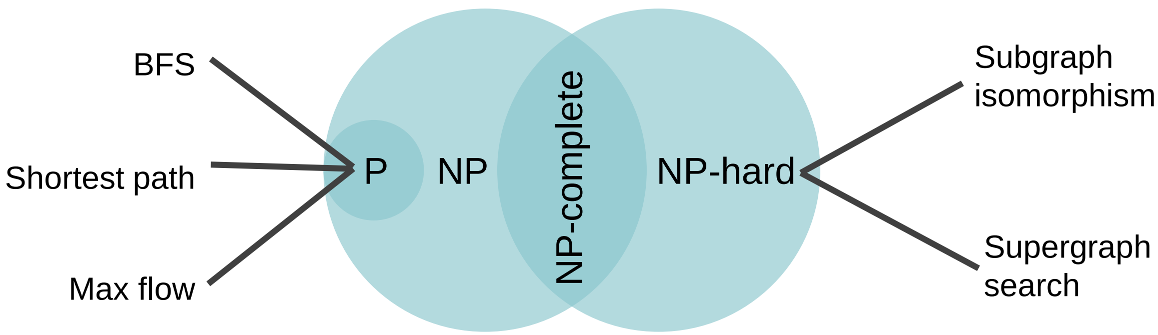
Kunsoo Park

Outline

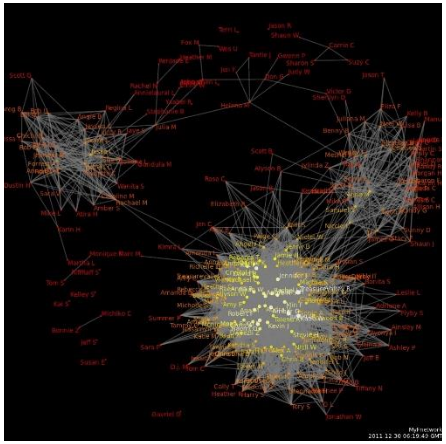
- Problem Definitions
  - Subgraph matching
  - Supergraph search
  - Subgraph search
  - Graph isomorphism
- DAF (subgraph matching)
  - Overview of DAF
  - DAG-graph dynamic programming
  - Adaptive matching order with DAG ordering
  - Performance Evaluation

Big Data Analysis

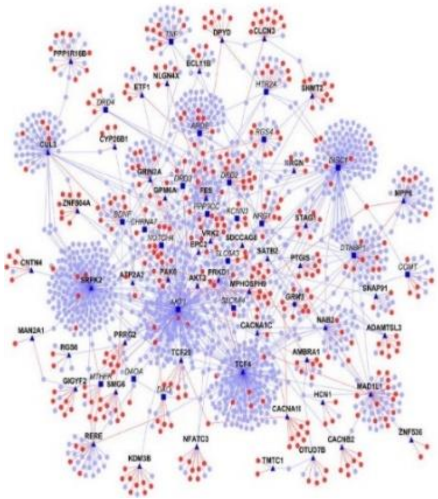
- Research on Big Data analysis has been increasing rapidly.



- Many graph analysis techniques are NP-hard problems.



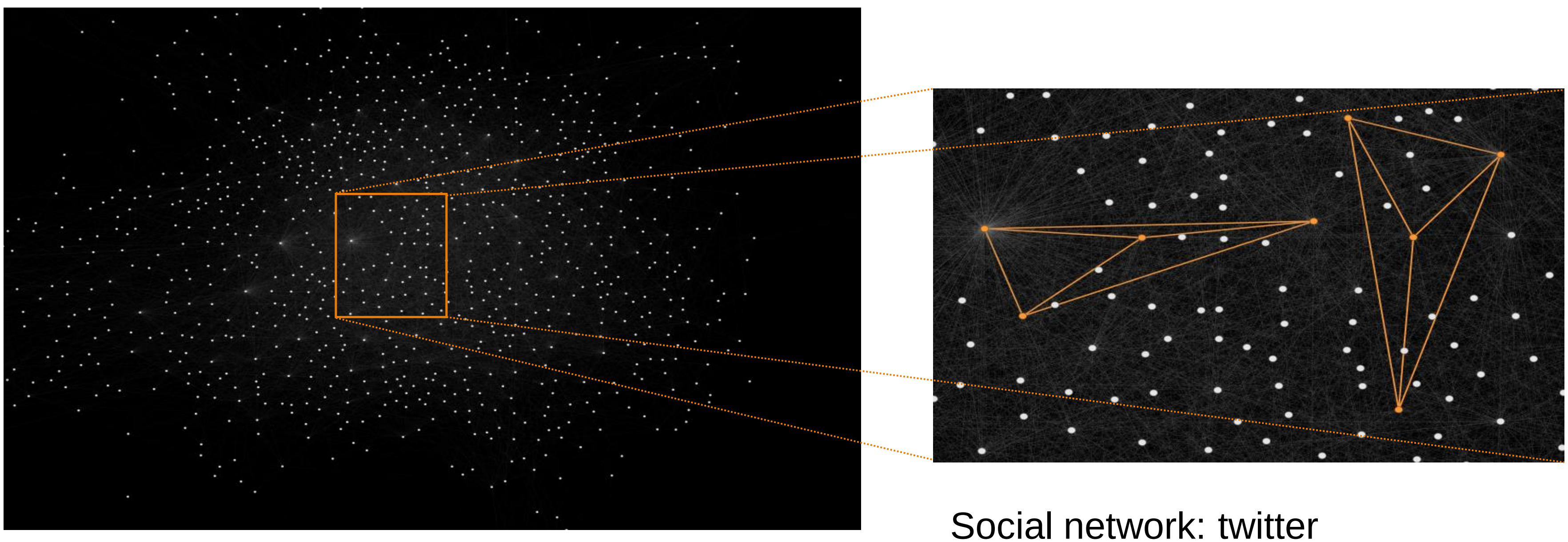
Social network: facebook



Protein-protein interaction network

Subgraph Matching

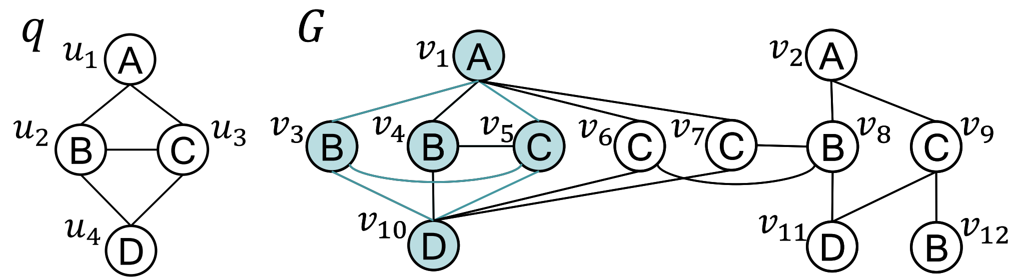
- Subgraph matching (a.k.a. subgraph isomorphism) is the problem of finding patterns in a big graph.



Social network: twitter

Subgraph Matching

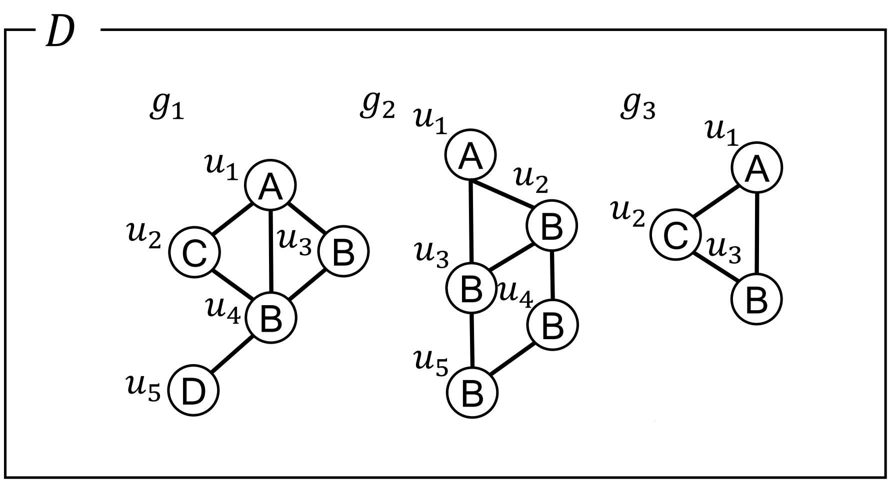
- Embedding

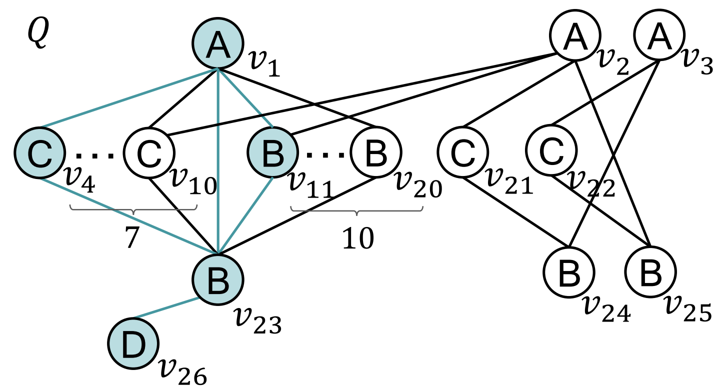


  - Given a query graph $q = (V(q), E(q), L_q)$ and a data graph $G = (V(G), E(G), L_G)$ (undirected, connected, vertex-labeled graphs)
  - An *embedding* of q in G is a mapping $M : V(q) \rightarrow V(G)$ such that
    1. M is injective. (i.e., $M(u) \neq M(u')$ for $u \neq u'$),
    2. $L_q(u) = L_G(M(u))$ for every $u \in V(q)$,
    3. $(M(u), M(u')) \in E(G)$ for every $(u, u') \in E(q)$.
  - e.g., $M = \{(u_1, v_1), (u_2, v_3), (u_3, v_5), (u_4, v_{10})\}$
  - A mapping that satisfies 2 and 3 is called a homomorphism.
- Subgraph matching
  - Find all distinct embeddings of q in G (NP-hard)
  - Fundamental problem in graph analysis: social networks, protein interaction networks, etc.

Supergraph Search

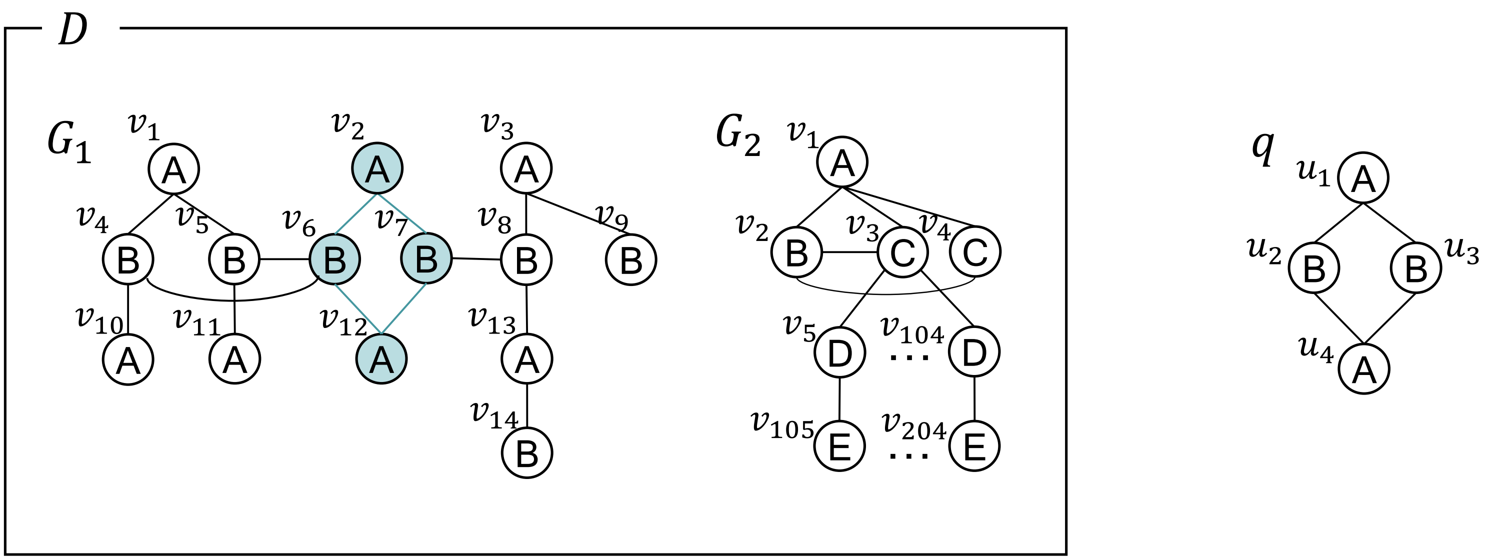
- Supergraph search
  - Given a set of data graphs $D = \{g_1, g_2, \ldots, g_m\}$ and a query graph Q,
  - The problem is to find all the data graphs in D that are contained in Q as subgraphs (NP-hard).
  - e.g., $A_Q = \{g_1, g_3\}$, where $A_Q = \{g_i \in D \mid g_i \subseteq Q\}$





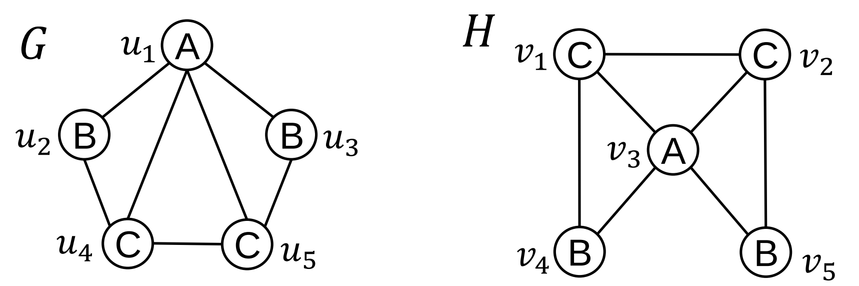
Subgraph Search

- Subgraph search
  - Given a set of data graphs $D = \{G_1, G_2, \ldots, G_m\}$ and a query graph q,
  - The problem is to find all the data graphs in D that contains q as subgraphs (NP-hard).
  - e.g., $A_Q = \{G_1\}$, where $A_Q = \{G \in D \mid q \subseteq G\}$



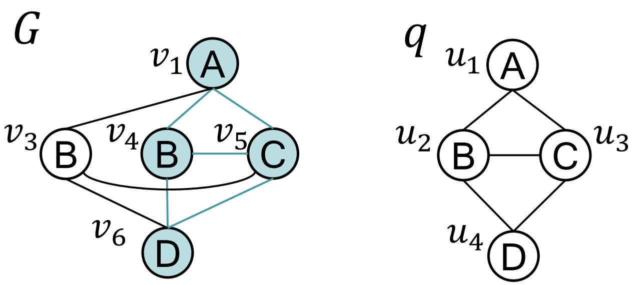
Graph Isomorphism

- Isomorphism

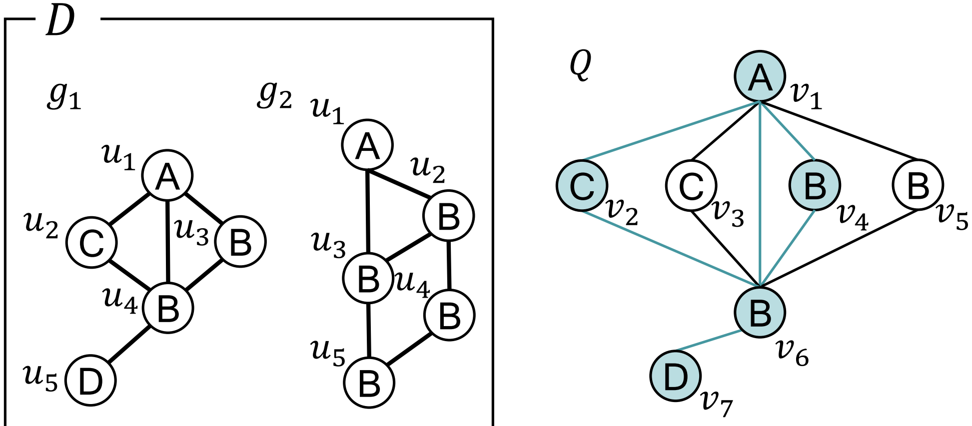


  - Given two graphs $G = (V(G), E(G), L_G)$ and $H = (V(H), E(H), L_H)$,
  - An ***isomorphism*** of G and H is a mapping $M : V(G) \to V(H)$ such that
    1. M is bijective (i.e., one-to-one correspondence),
    2. $L_G(u) = L_H\big(M(u)\big)$ for every $u \in V(G)$,
    3. $\big(M(u), M(u')\big) \in E(H)$ for every $(u, u') \in E(G)$.
  - e.g., $M = \{(u_1, v_3), (u_2, v_4), (u_3, v_5), (u_4, v_1), (u_5, v_2)\}$
- Graph Isomorphism
  - Given two graphs G and H,
  - Determine whether there exists an isomorphism of G and H (not known to be in P or NP-hard)

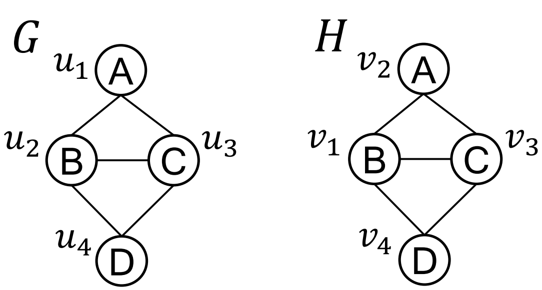
Summary



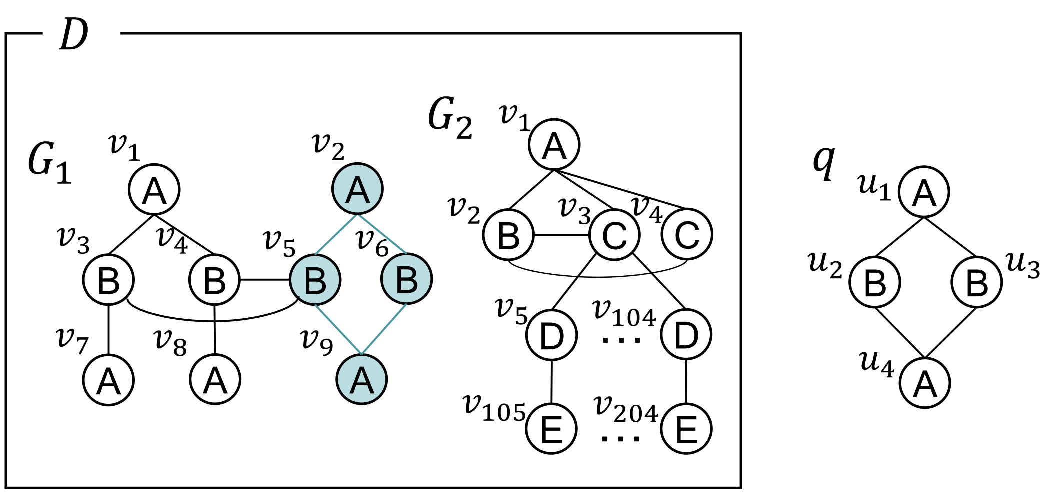
subgraph matching



supergraph search

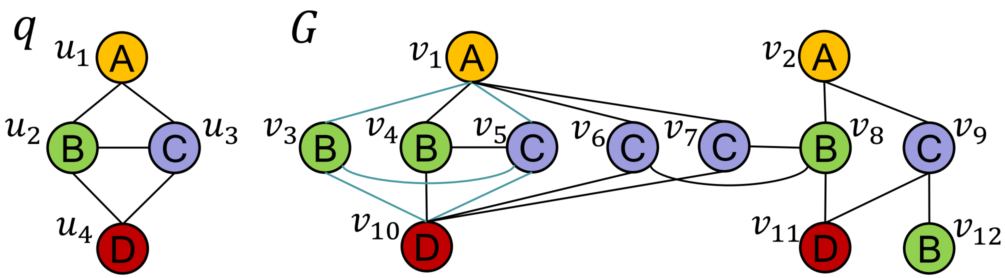


graph isomorphism



subgraph search

General Framework of Subgraph Matching



- Framework
  - Adopt a filtering process to find a <u>candidate set</u> $C(u)$ for each $u \in V(q)$, where $C(u)$ is a subset of $V(G)$ which u can be mapped to (e.g., $C(u_1) = \{v_1, v_2\}$).
  - Choose a linear order of the query vertices, called <u>matching order</u>, and apply backtracking based on the matching order (e.g., (u_1, u_2, u_3, u_4)).
- State-of-the-art algorithms
  - Turbo$_{\text{iso}}$ [Han, Lee & Lee. SIGMOD 2013]
  - CFL-Match [Bi, Chang, Lin, Qin & Zhang. SIGMOD 2016]
- Both use spanning tree q_T of query graph q for a filtering process.
  - Find (potential) embeddings of q_T in G.
  - Candidate sets are stored in an <u>auxiliary data structure</u>.
  - Find all embeddings of q by checking <u>non-tree edges</u> during backtracking.

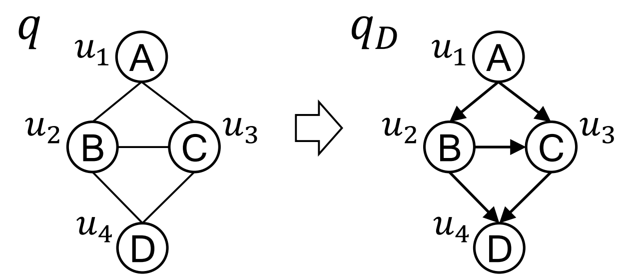
Overview of DAF

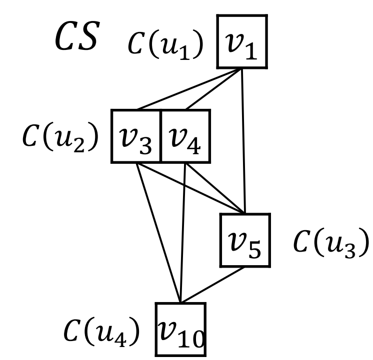
- $BuildDAG$
  - Build a rooted DAG q_D from q.
  - Select root $r \leftarrow \underset{u \in V(q)}{\operatorname{argmin}} \frac{|C_{\text{ini}}(u)|}{\deg_q(u)}$.
    - $v \in C_{\text{ini}}(u)$ if $L_G(v) = L_q(u)$ and $d_G(v) \geq d_q(u)$.
  - Traverse q in BFS order, direct all edges from earlier to later visited vertices.
- $BuildCS$
  - Build candidate space (CS) by using DAG-Graph DP.
- $Backtrack$
  - Find all embeddings of q in CS by applying Adaptive Matching Order and Pruning by Failing Sets

Algorithm 1: DAF

Input: query graph q, data graph G
Output: all embeddings of q in G

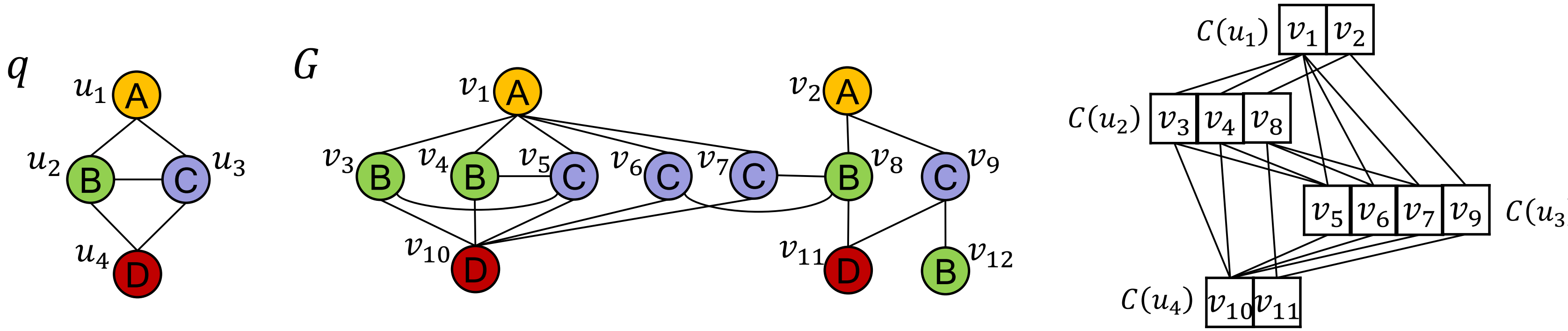
$q_D \leftarrow$ BuildDAG(q, G);
$CS \leftarrow$ BuildCS(q, q_D, G);
$M \leftarrow \emptyset$;
Backtrack(q, q_D, CS, M);





$\{(u_1, v_1), (u_2, v_3), (u_3, v_5), (u_4, v_{10})\}$
$\{(u_1, v_1), (u_2, v_4), (u_3, v_5), (u_4, v_{10})\}$

Candidate Space (CS)



Initial CS

- <u>Candidate space</u> on q and G consists of **candidate set** & **edges**.
  - There is a candidate set $C(u)$ for each $u \in V(q)$, where $C(u) \subseteq C_{\text{ini}}(u)$.
  - There is an edge between $v \in C(u)$ and $v' \in C(u')$ iff $(u, u') \in E(q)$ and $(v, v') \in E(G)$.
- CS is a complete search space for all embeddings of q in G.

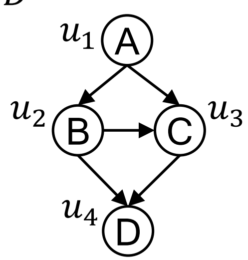
Dynamic Programming

- Dynamic Programming (DP)
  - Algorithm design technique for optimization problems which solves a problem by solving subproblems and combining the solutions to subproblems
- Types of DP
  - DP between string and string: edit distance, longest common subsequence
  - DP between tree and tree: tree edit distance
  - DP between tree and graph: graph problems for trees and series-parallel graphs
  - DP between DAG and graph (new)
  - DP between graph and graph (X)

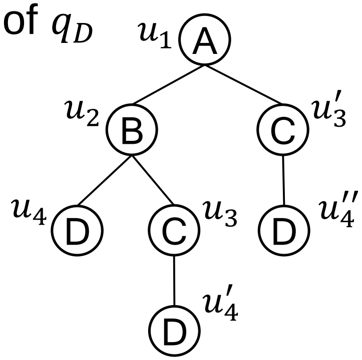
Weak Embedding of DAG

- Given a rooted DAG g
- *Path tree of* g is tree g' such that each root-to-leaf path in g' corresponds to distinct root-to-leaf path in g, and g' shares common prefixes of all root-to-leaf paths.

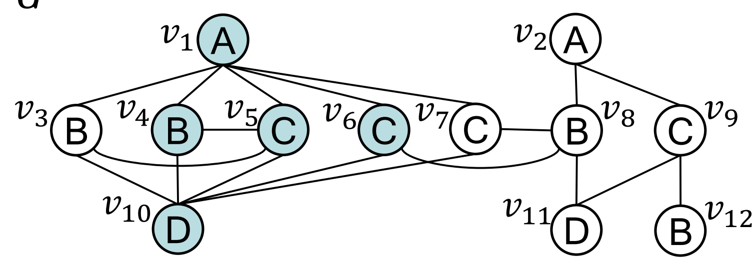
q_D



Path tree of q_D



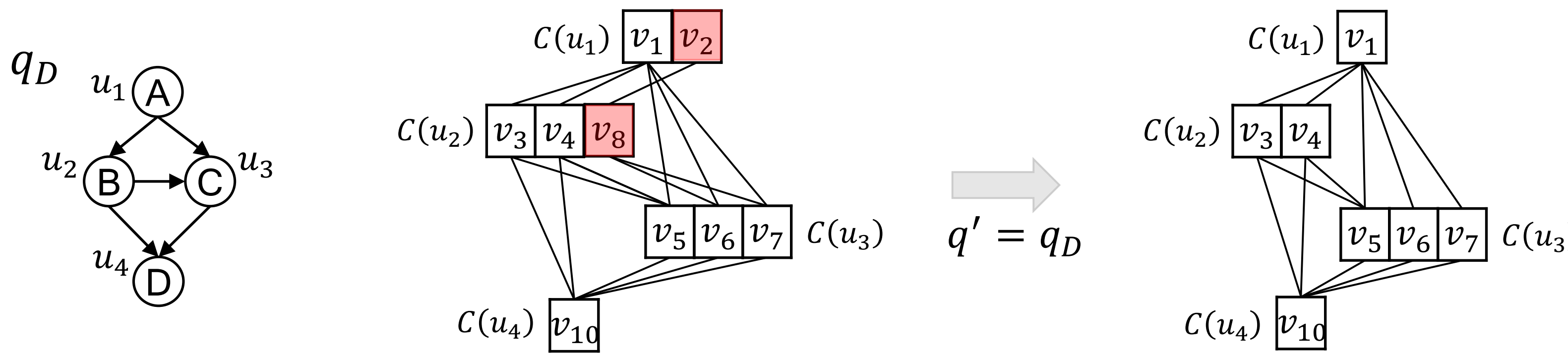
G



- For rooted DAG g with root u, a *weak embedding* of g *at* v is a homomorphism M' of the path tree of g such that $M'(u) = v$.
  - e.g. $\{(u_1, v_1), (u_2, v_4), (u_4, v_{10}), (u_3, v_5), (u_4', v_{10}), (u_3', v_6), (u_4'', v_{10})\}$
- Every embedding is a weak embedding (but, converse is not true). → Weak embedding is a <u>necessary condition</u> for embedding.

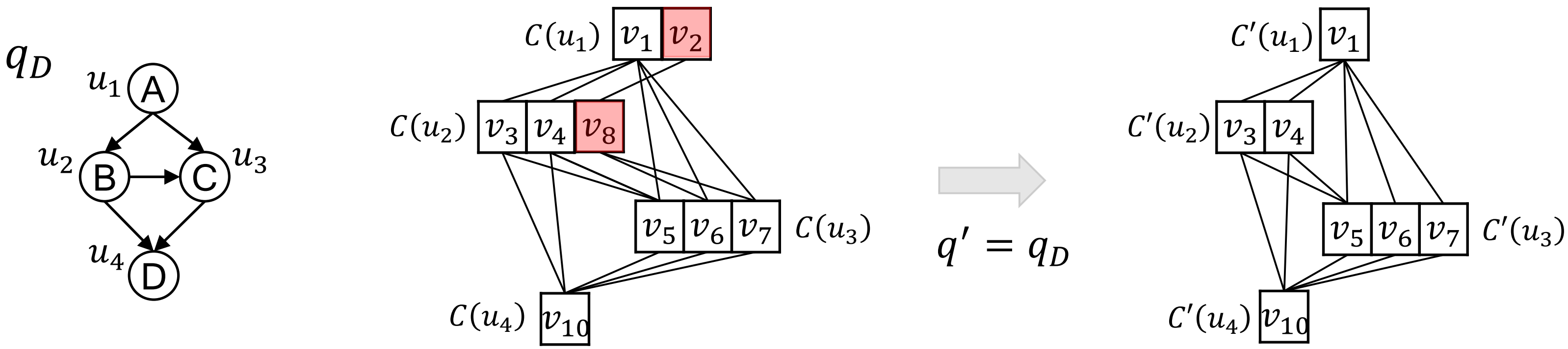
DAG-Graph DP

- Given a CS and a query DAG q'



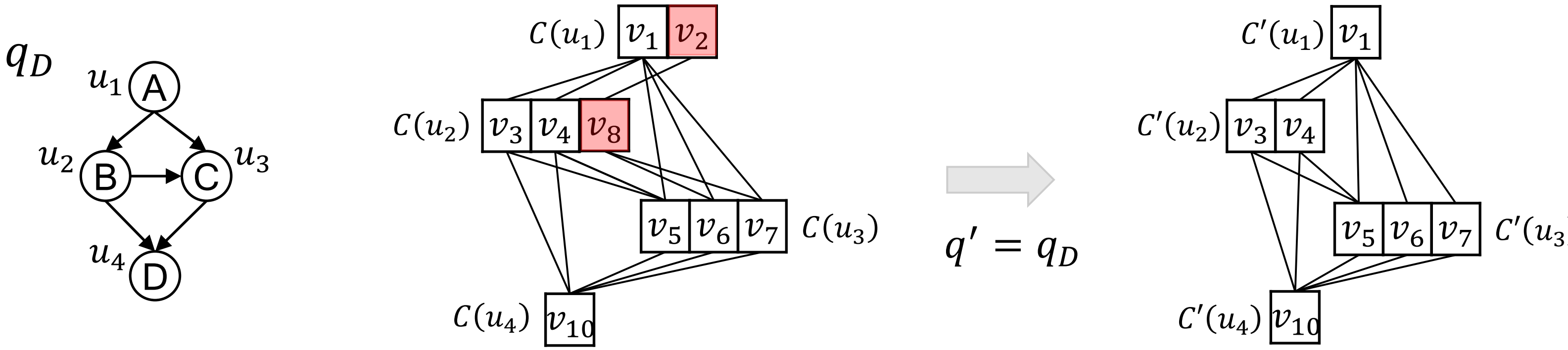
- Define $D[u, v] = 1$ if $v \in C(u)$; 0 otherwise.
- We refine D into D' by dynamic programming
- Definition: $D'[u, v] = 1$ iff $D[u, v] = 1$ and there is a weak embedding of q'_u at v in the CS (q'_u is sub-DAG of q' rooted at u)
- Recurrence: $D'[u, v] = 1$ iff $D[u, v] = 1$ and $\exists v_c$ adjacent to v such that $D'[u_c, v_c] = 1$ for every child u_c of u in q'

DAG-Graph DP



- Definition: $D'[u, v] = 1$ iff $D[u, v] = 1$ and there is a weak embedding of q'_u at v in the CS (q'_u is sub-DAG of q' rooted at u)
- Recurrence: $D'[u, v] = 1$ iff $D[u, v] = 1$ and $\exists v_c$ adjacent to v such that $D'[u_c, v_c] = 1$ for every child u_c of u in q'
- $D'[u_4, v_{10}] = 1$
- $D'[u_3, v_5] = D'[u_3, v_6] = D'[u_3, v_7] = 1$
- $D'[u_2, v_3] = 1$ because $D'[u_4, v_{10}] = D'[u_3, v_5] = 1$.
  $D'[u_2, v_4] = 1$. $D'[u_2, v_8] = 0$
- $D'[u_1, v_1] = 1$ because $D'[u_2, v_3] = D'[u_3, v_5] = 1$.
  $D'[u_1, v_2] = 0$

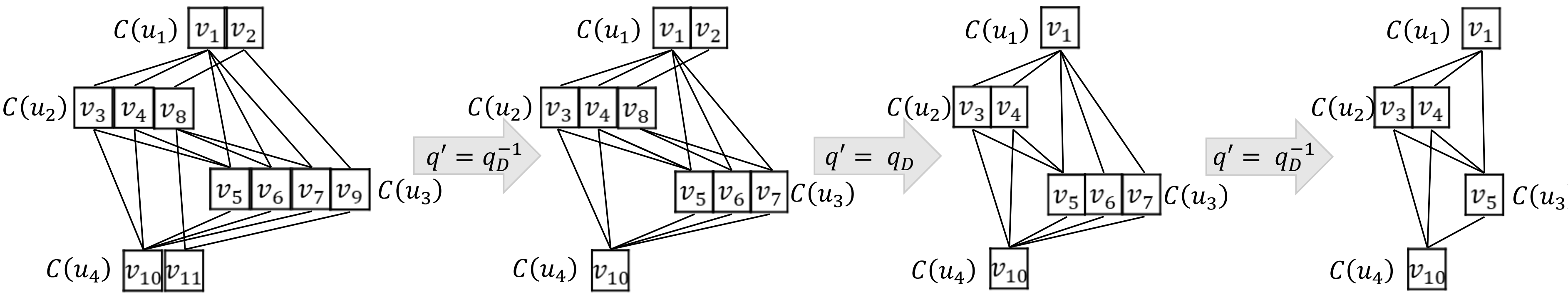
DAG-Graph DP



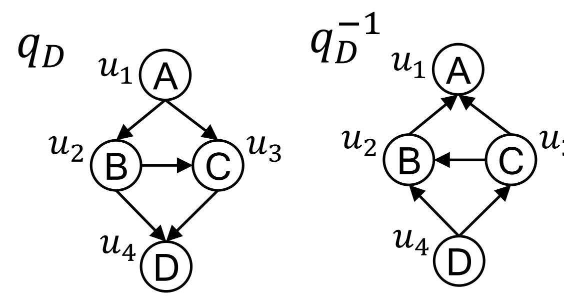
- Given a CS and a query DAG q'
- Refinement of CS
  - $v \in C'(u)$ iff $v \in C(u)$ and there is a weak embedding of q'_u at v in CS

- Compute $C'(u)$ by dynamic programming in bottom-up fashion.
- **Lemma.** Given a CS on q and G, time complexity of DAG-Graph DP is $O(|E(q)| \times |E(G)|)$.

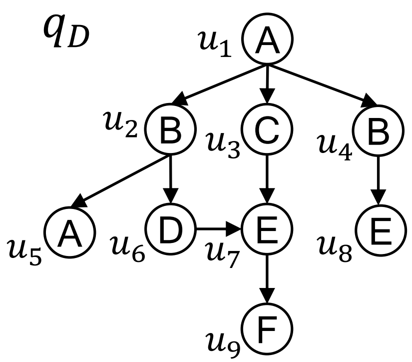
Refinements of CS using DAG-Graph DP



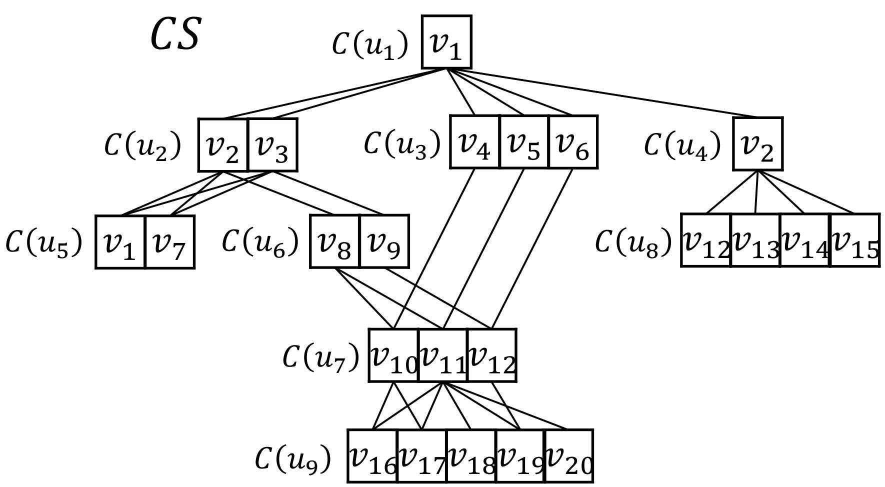
- Build a compact CS
  - Starting from initial CS, repeat DAG-Graph DP with $q' = q_D$ and $q' = q_D^{-1}$ alternately.
  - Ideally, repeat until no changes occur.
  - Empirically, 3 steps are enough for optimization.
    - Filtering rate after 3 steps was < 1%



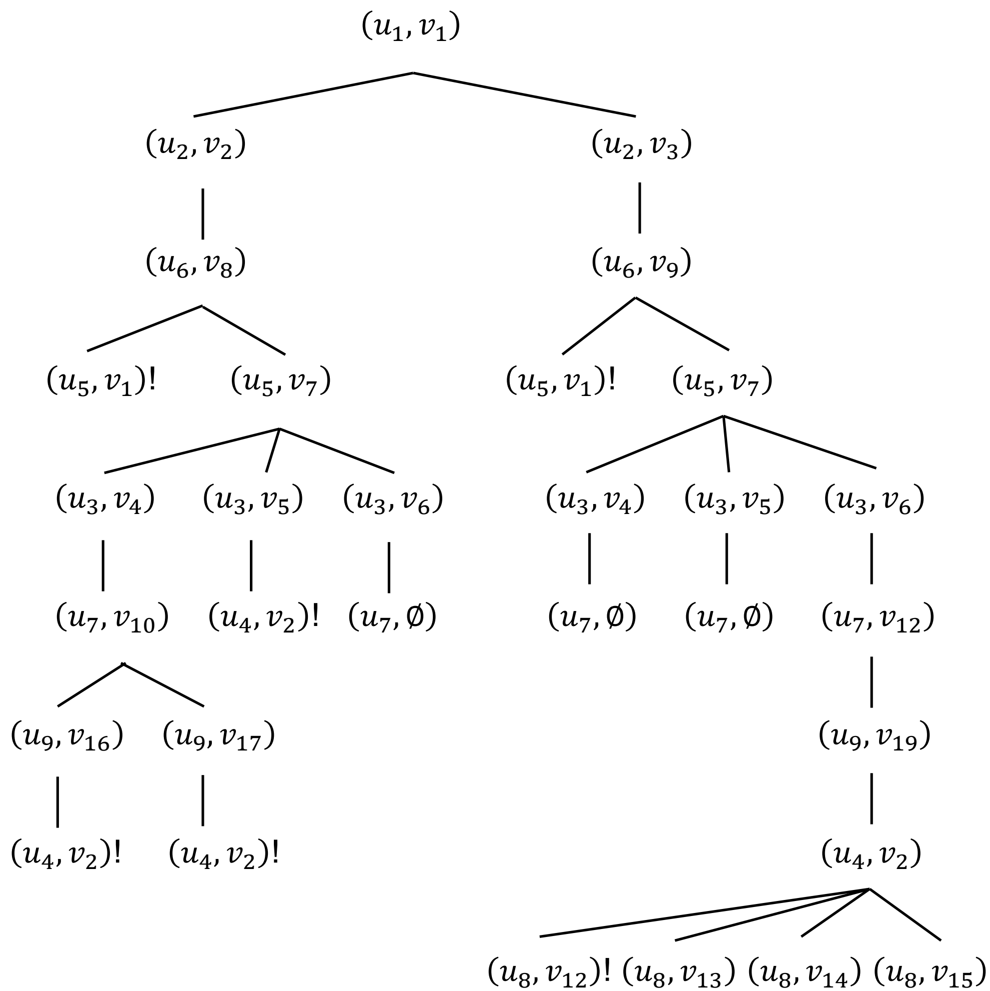
Search Tree



q_D
u_1 A
u_2 B
u_3 C
u_4 B
u_5 A
u_6 D
u_7 E
u_8 E
u_9 F



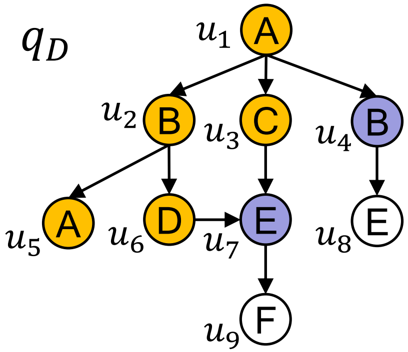
CS
$C(u_1)$ v_1
$C(u_2)$ v_2 v_3
$C(u_3)$ v_4 v_5 v_6
$C(u_4)$ v_2
$C(u_5)$ v_1 v_7
$C(u_6)$ v_8 v_9
$C(u_8)$ v_{12} v_{13} v_{14} v_{15}
$C(u_7)$ v_{10} v_{11} v_{12}
$C(u_9)$ v_{16} v_{17} v_{18} v_{19} v_{20}

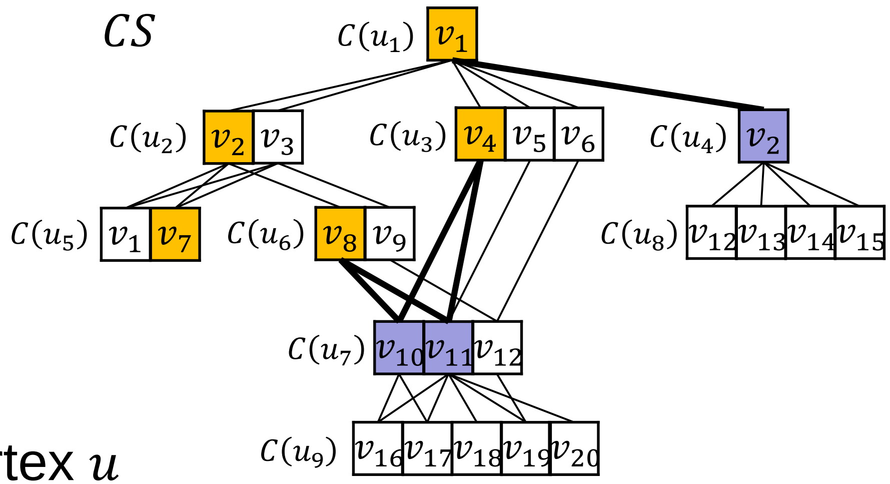


(u_1, v_1)
(u_2, v_2)
(u_2, v_3)
(u_6, v_8)
(u_6, v_9)
(u_5, v_1)!
(u_5, v_7)
(u_5, v_1)!
(u_5, v_7)
(u_3, v_4)
(u_3, v_5)
(u_3, v_6)
(u_3, v_4)
(u_3, v_5)
(u_3, v_6)
(u_7, v_{10})
(u_4, v_2)!
$(u_7, \emptyset)$
$(u_7, \emptyset)$
$(u_7, \emptyset)$
(u_7, v_{12})
(u_9, v_{16})
(u_9, v_{17})
(u_9, v_{19})
(u_4, v_2)!
(u_4, v_2)!
(u_4, v_2)
(u_8, v_{12})! (u_8, v_{13}) (u_8, v_{14}) (u_8, v_{15})

DAG-Ordering

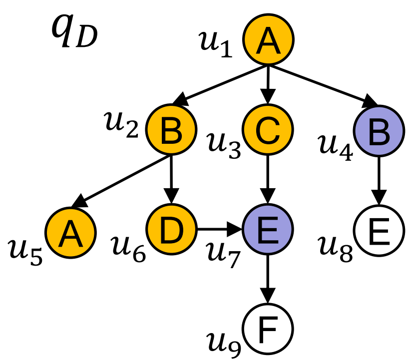
- An unvisited query vertex u is *extendable* regarding partial embedding M if all parents of u are matched in M.
  - e.g., extendable vertices regarding $M = \{(u_1, v_1), (u_2, v_2), (u_3, v_4), (u_5, v_7), (u_6, v_8)\}$ are $\{u_4, u_7\}$.
- Always select **extendable vertex** u as next vertex to map.

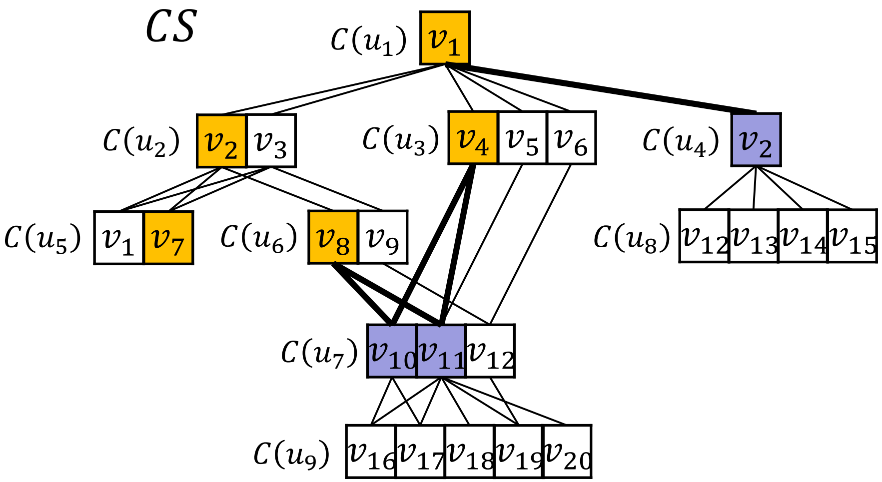




- ***Extendable candidates*** of vertex u
  - $C_M(u) = \bigcap_{u_p \in parent(u)} N_u^{u_p}\left(M(u_p)\right)$, where $N_u^{u_p}(v_p)$ is the list of vertices v adjacent to v_p in G such that $v \in C(u)$
  - $C_M(u_4) = N_{u_4}^{u_1}(v_1) = \{v_2\},\ C_M(u_7) = N_{u_7}^{u_3}(v_4) \cap N_{u_7}^{u_6}(v_8) = \{v_{10}, v_{11}\}$

Backtracking Framework

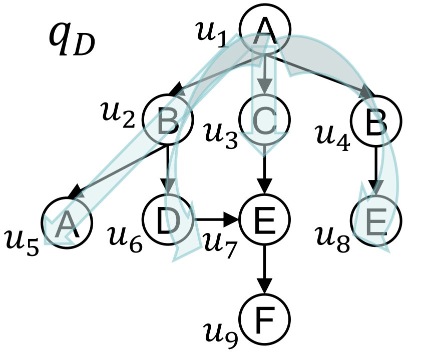




- **Lemma.** Suppose that we are given partial embedding M and extendable vertex u. For every unvisited candidate $v \in C_M(u)$, $M \cup \{(u, v)\}$ is a partial embedding.
  - e.g., $M \cup \{(u_7, v_{10})\}$ is a partial embedding.
- Backtracking framework
  - Select an extendable vertex u regarding current partial embedding M.
  - Extend M by mapping u to each unvisited extendable candidate $v \in C_M(u)$ and recurse.

Adaptive Matching Order

- Suppose we extend a partial embedding M.
- Among all extendable vertices, which one?
- Candidate-size order
  - Select u such that $|C_M(u)|$ is minimum.
  - e.g., select u_4 in previous example
- Path-size order
  - Select u such that $w_M(u)$ is minimum.
  - $w_M(u)$ estimates number of path embeddings.
  - *Infrequent-path-first* strategy
    - Aim to match a path in q_D that is infrequent in CS first



Performance Evaluation

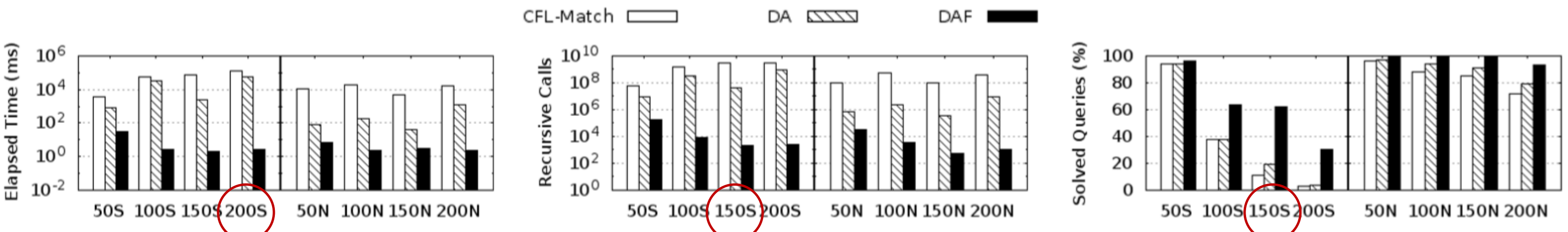
- Following existing algorithms are evaluated
  - **VF2**, **QuickSI**, **GraphQL**, **GADDI**, **SPath**, **Turbo$_{\text{ISO}}$**, **CFL-Match**,
  - **DA** (DAG-graph DP, Adaptive matching order), and **DAF** (DA + Failing set).

- Six real datasets

Data graph (G)	$\|V(G)\|$	$\|E(G)\|$	$\|\Sigma\|$	**Avg degree**
Yeast	3,112	12,519	71	8.04
Human	4,674	86,282	44	36.91
HPRD	9,460	37,081	307	7.83
Email	36,692	183,831	20	10.02
DBLP	317,080	1,049,866	20	6.62
YAGO	4,295,825	11,413,472	49,676	5.31

Comparing with CFL-Match

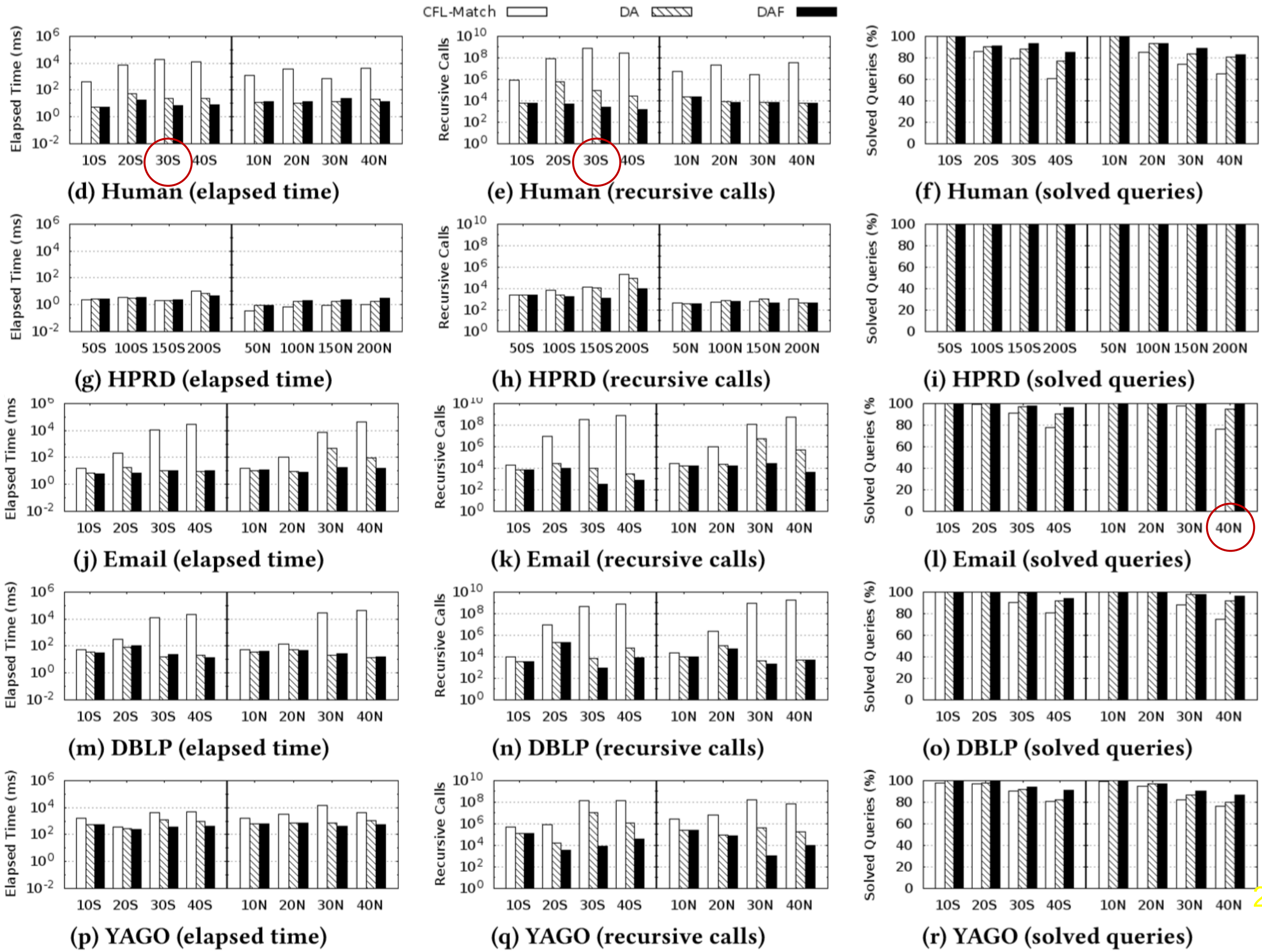
- For each data graph, 8 query sets are generated (100 queries in each set).
  - 50S = 100 sparse (avg-deg ≤ 3) query graphs of 50 vertices
  - 200N = 100 non-sparse (avg-deg > 3) query graphs of 200 vertices
- Each query graph is generated by random walk on G.
- For each query, measure **running time to find first** 10^5 **embeddings**.
- Time limit of **10 min** for each query.
  - Solved/unsolved queries
- n: minimum number of solved queries in compared algorithms.
- Measure **avg. running time, # recursions** for n fastest queries, and **% of solved queries**.



(a) Yeast (elapsed time) (b) Yeast (recursive calls) (c) Yeast (solved queries)

- DAF outperforms CFL-Match by up to 4-orders-of-magnitude in running time and 6-orders-of-magnitude in # recursions (Yeast).

Comparing with CFL-Match

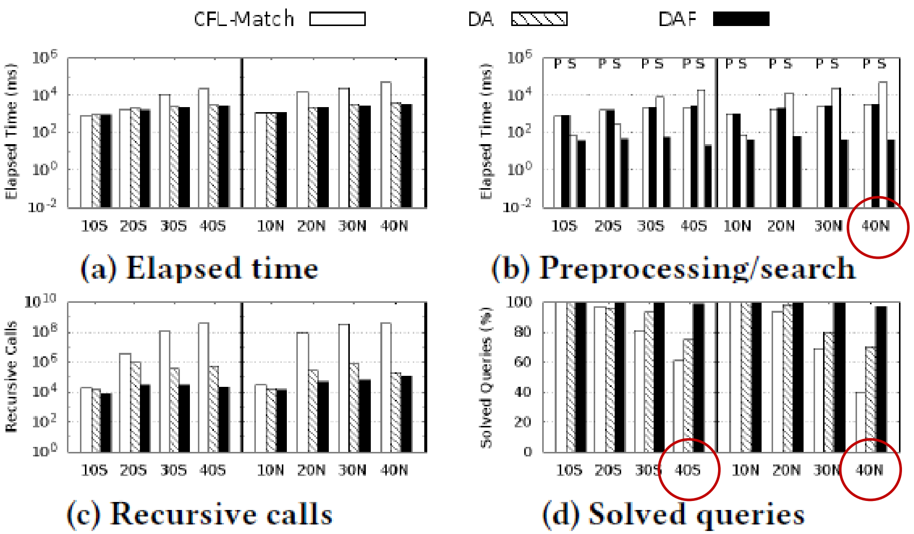


•

Experiment with Billion-Scale Graph

- Twitter graph
  - 41.7 million vertices
  - 1.47 billion edges

- DAF outperforms CFL-Match.
  - As query sizes increase, the gap between them in solved queries increases.
  - DAF is 3-orders-of-magnitude faster in search time (40N in Figure 12b).



(a) Elapsed time (b) Preprocessing/search

(c) Recursive calls (d) Solved queries

Figure 12: Elapsed time, recursive calls, and solved queries of CFL-Match, DA, and DAF on Twitter. For CFL-Match and DAF, elapsed time is divided into preprocessing time and search time.

Conclusion

- DAF
  - DAG-graph dynamic programming
  - Adaptive matching order with DAG ordering
  - Pruning by failing sets
- Future work
  - Extending our work to parallel and distributed platforms
  - Finding applications of our techniques in related problems