Accelerating In-Memory Subgraph Query Processing on a Single Machine

Shixuan Sun

Senior Research Fellow, NUS

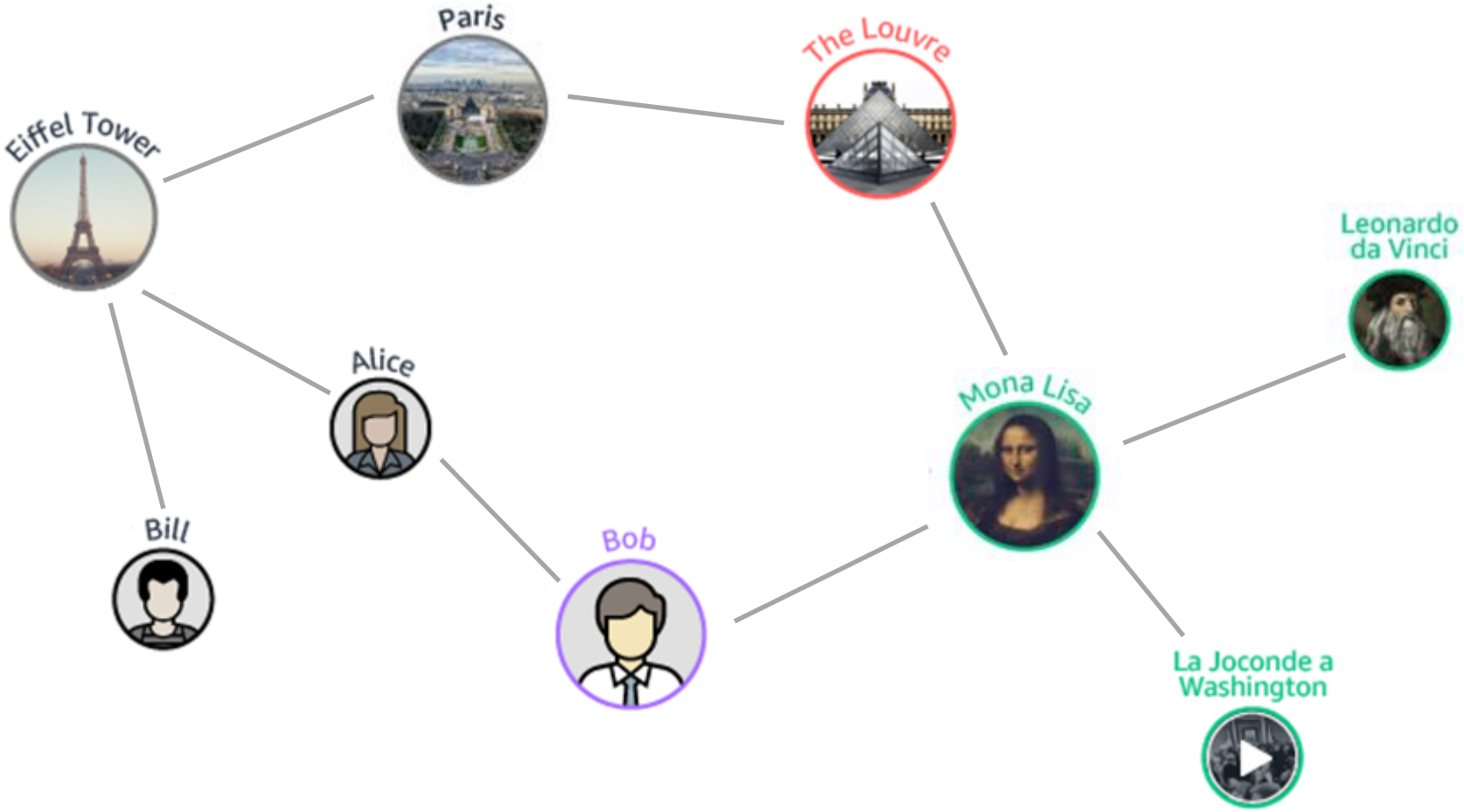
PhD, HKUST, 2020

Outline

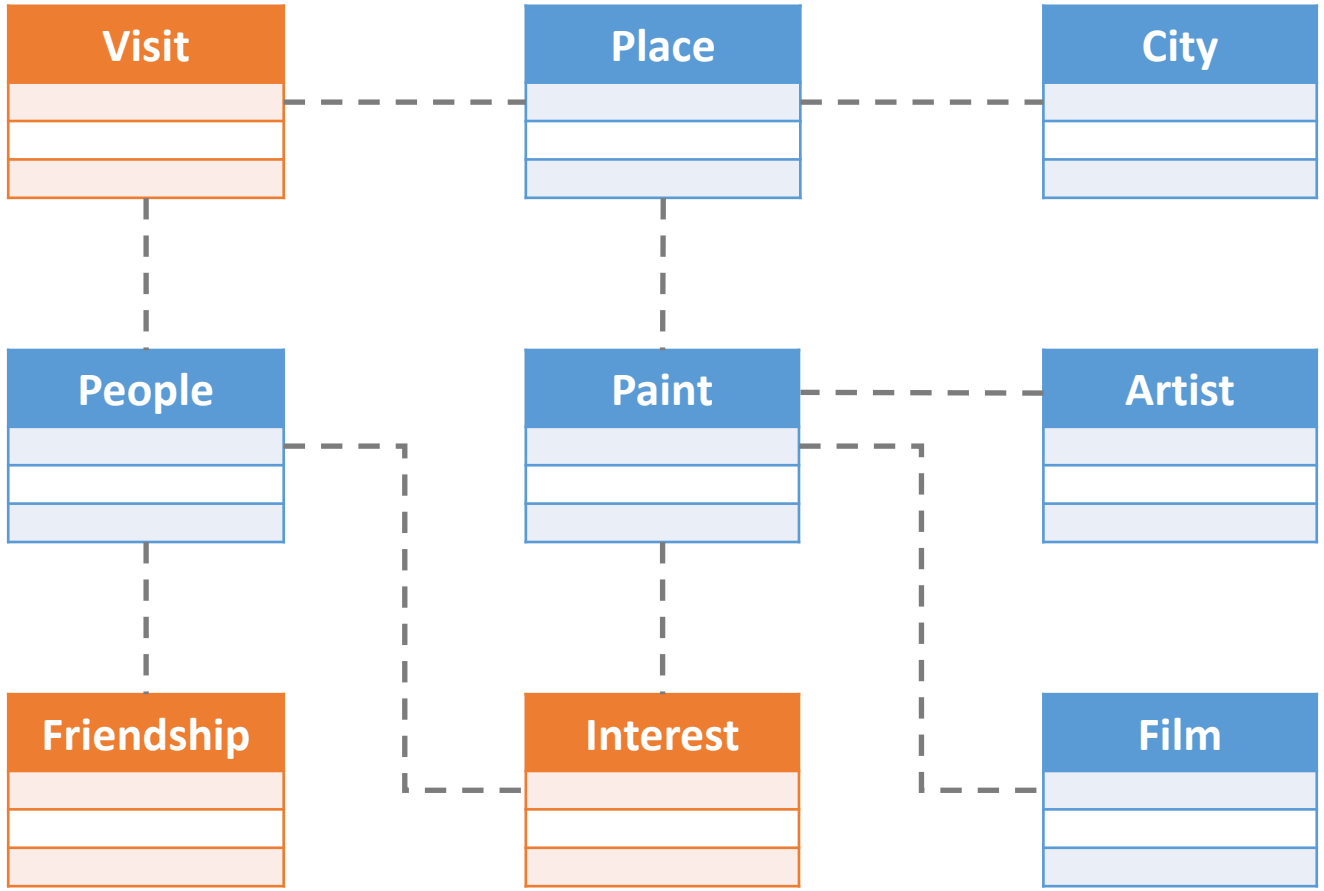
- Benchmark
  - Background
  - In-Memory Subgraph Matching: An In-Depth Study. SIGMOD 2020.
- Algorithms
  - RapidMatch: A Holistic Approach to Subgraph Query Processing. VLDB 2021.
  - PathEnum: Towards Real-Time Hop Constraint s-t Path Enumeration. SIGMOD 2021.
- Parallelization
  - LIGHT: Parallelizing Subgraph Query Processing. ICPADS 2018 & ICDE 2019.
  - ThunderRW: An In-Memory Graph Random Walk Engine. VLDB 2021.

Why graphs?

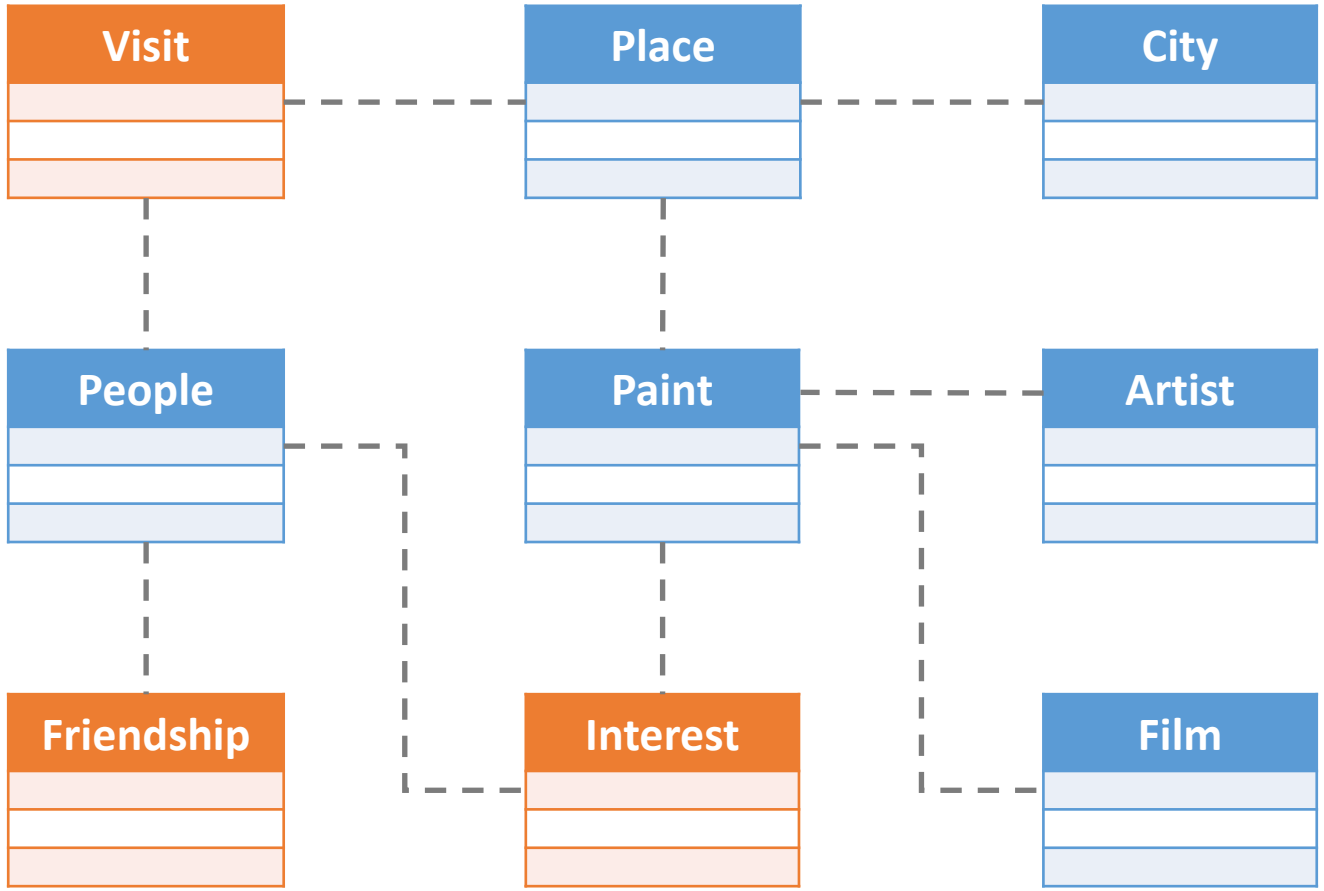
Everything is Naturally Connected



Model Connected World as Structural Data

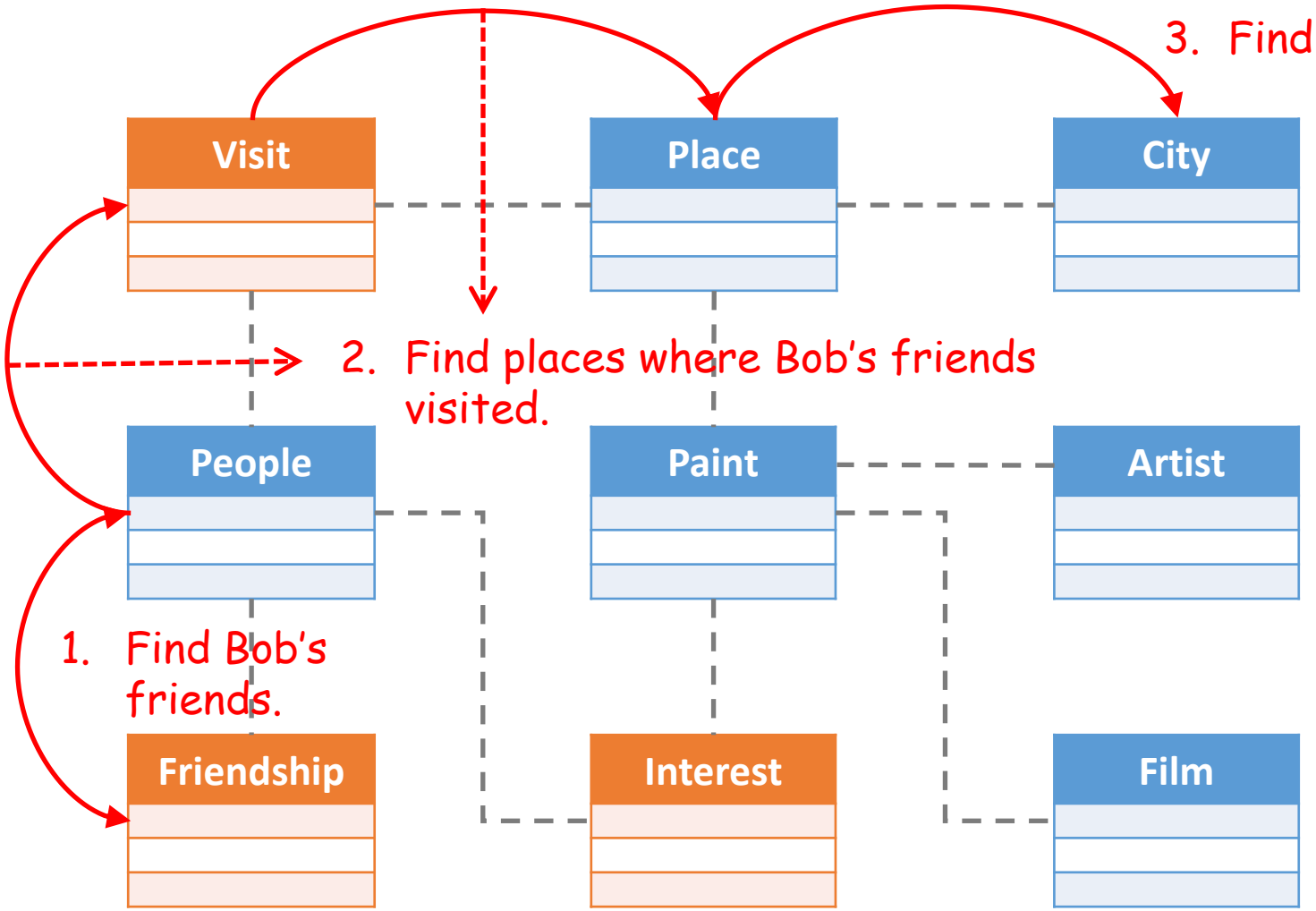


Model Connected World as Structural Data



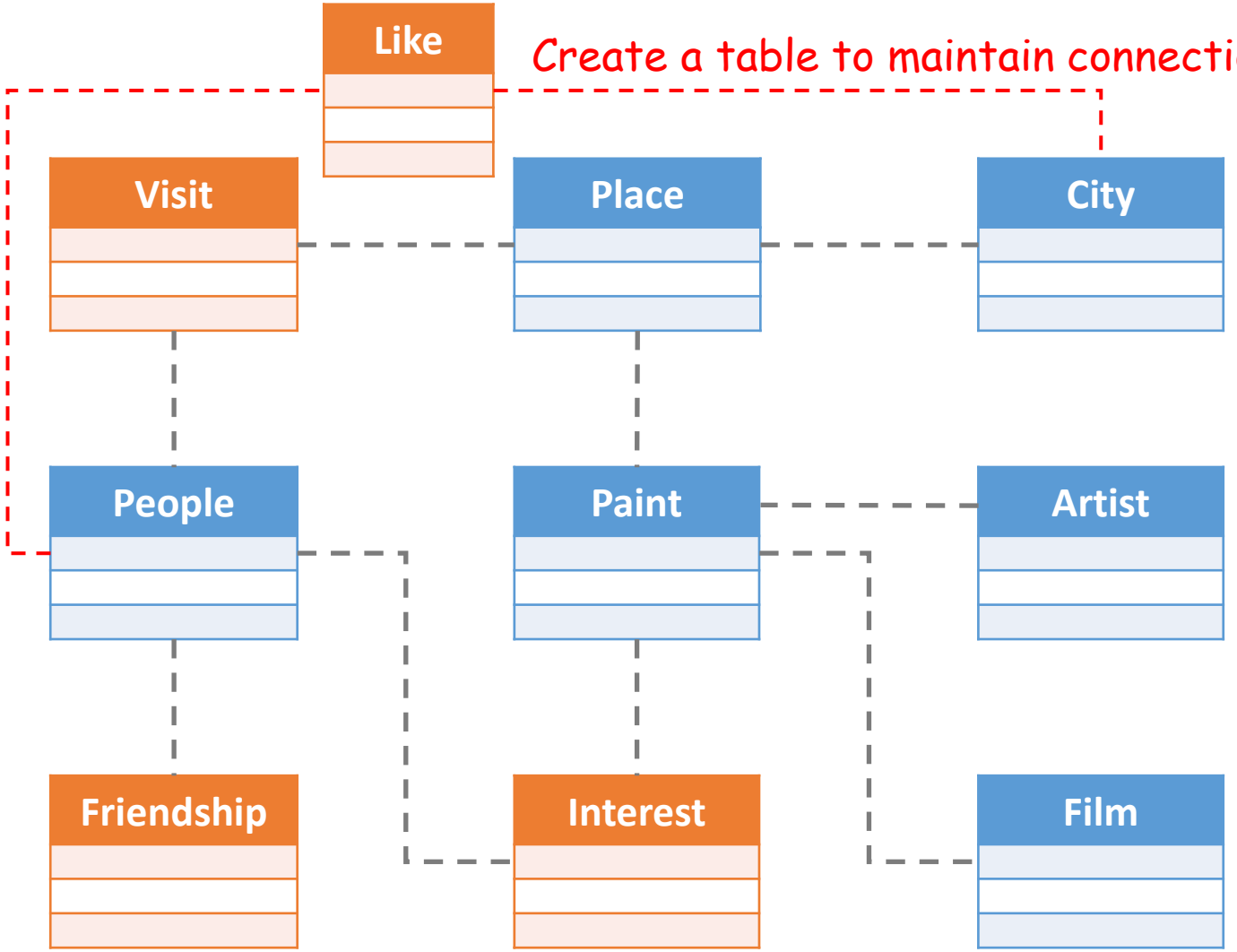
Query 1:
Find cities where Bob's friends visited.

Model Connected World as Structural Data



Query 1:
Find cities where Bob's friends visited.

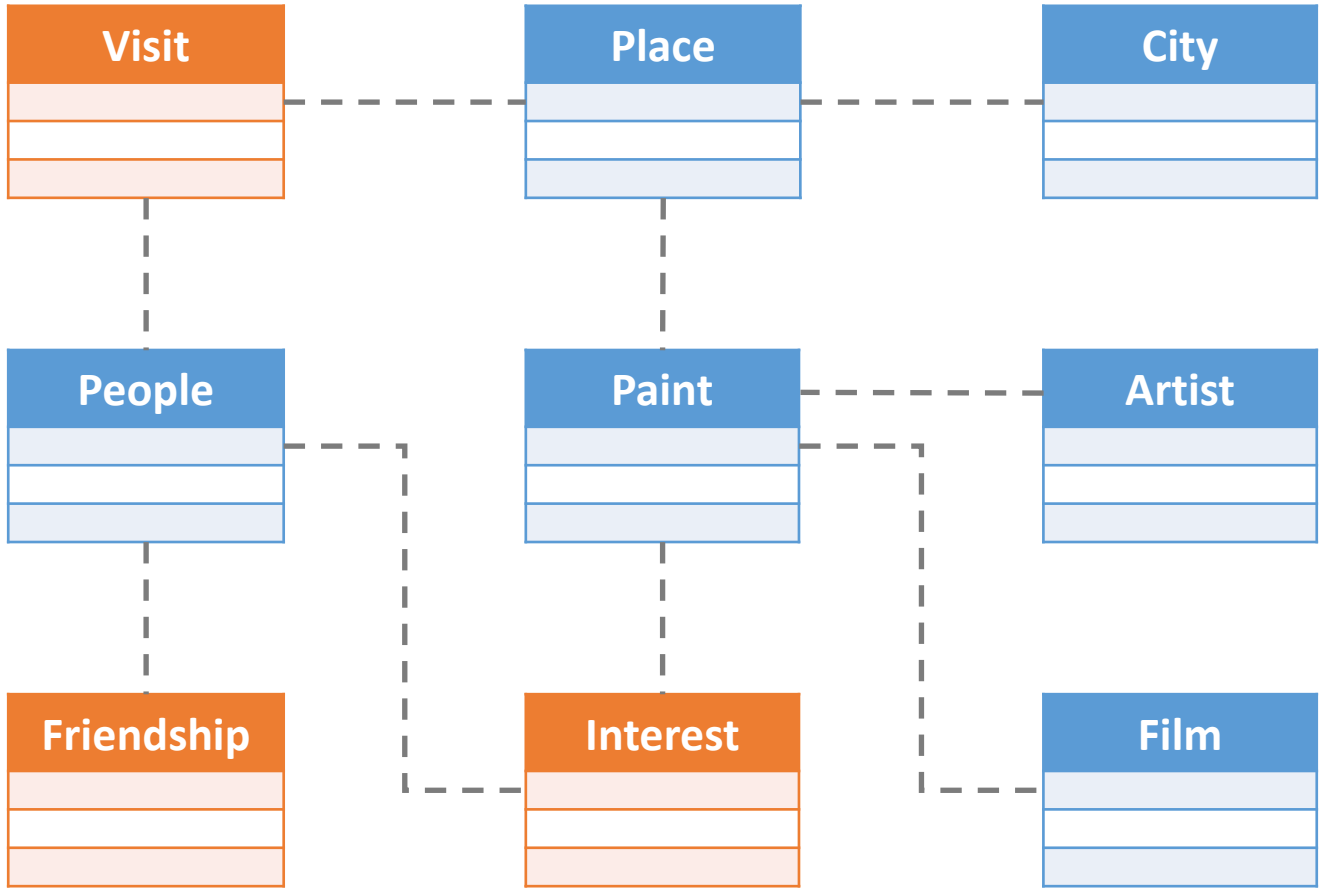
Model Connected World as Structural Data



Query 1:
Find cities where Bob's friends visited.

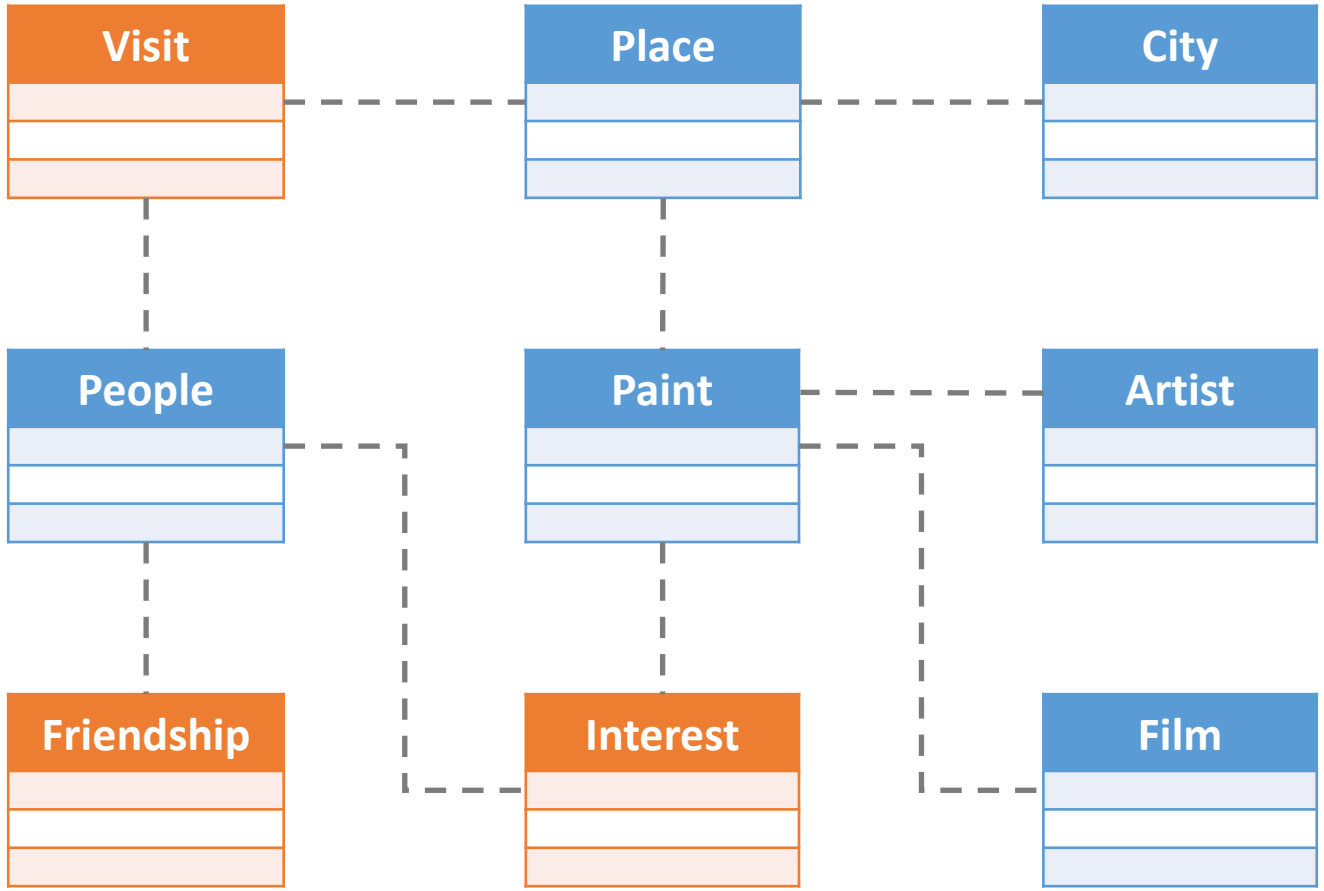
Query 2:
Add a "like" connection between Bob and cities where Bob's friends visited.

Hard to Capture Complex Connections



Difficult to interpret deep connections in data.

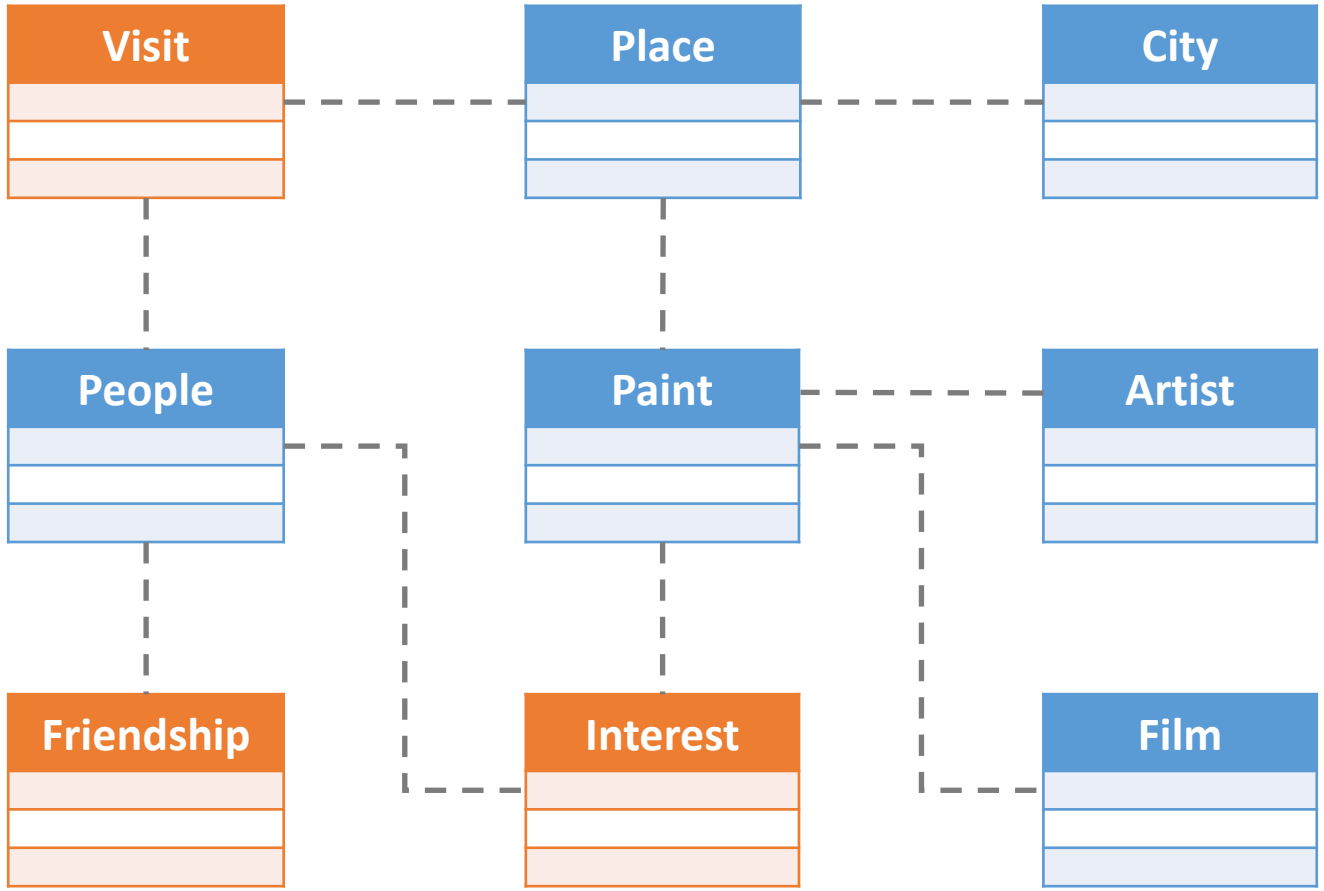
Hard to Capture Complex Connections



Difficult to interpret deep connections in data.

Poor performance for deep connection analysis.

Hard to Capture Complex Connections

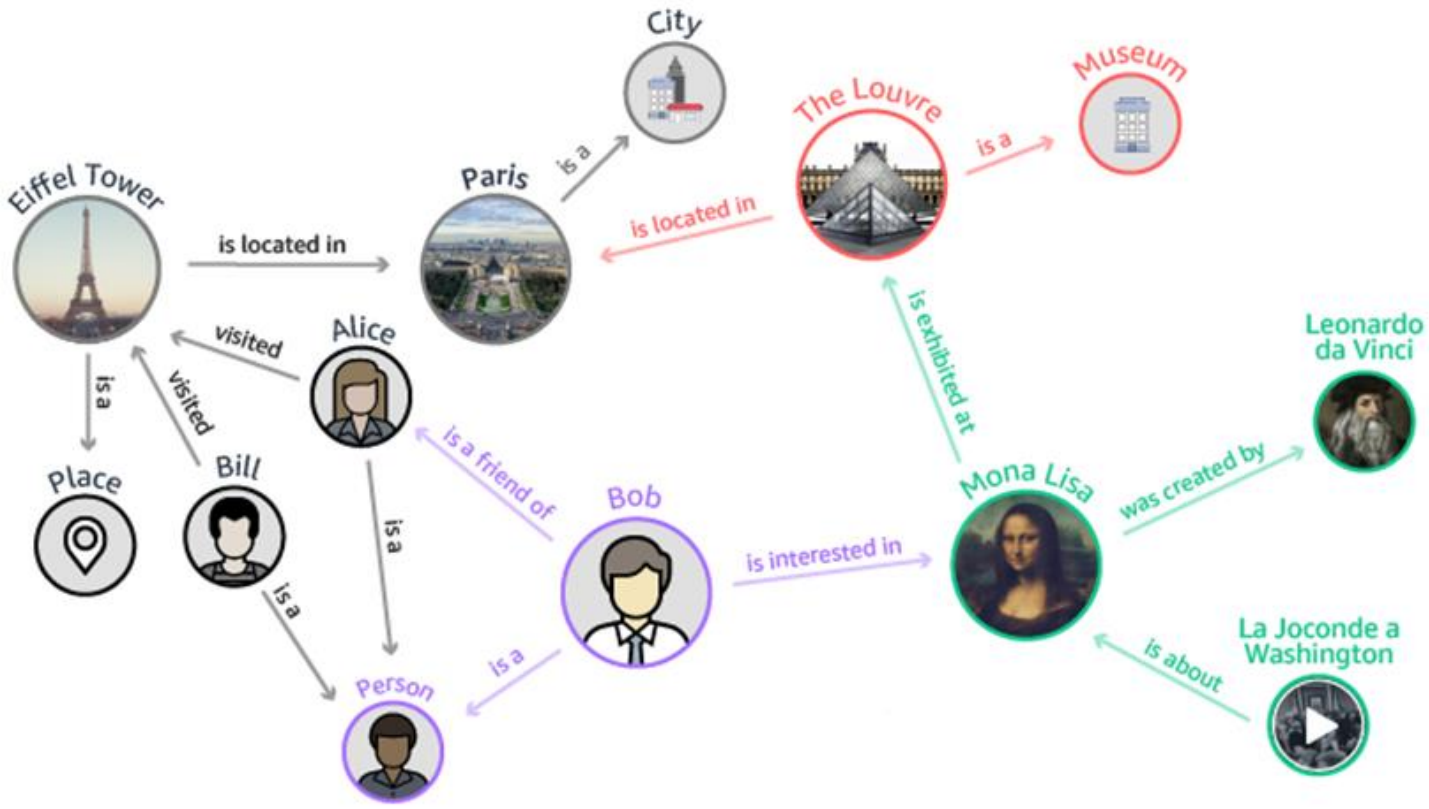


Difficult to interpret deep connections in data.

Poor performance for deep connection analysis.

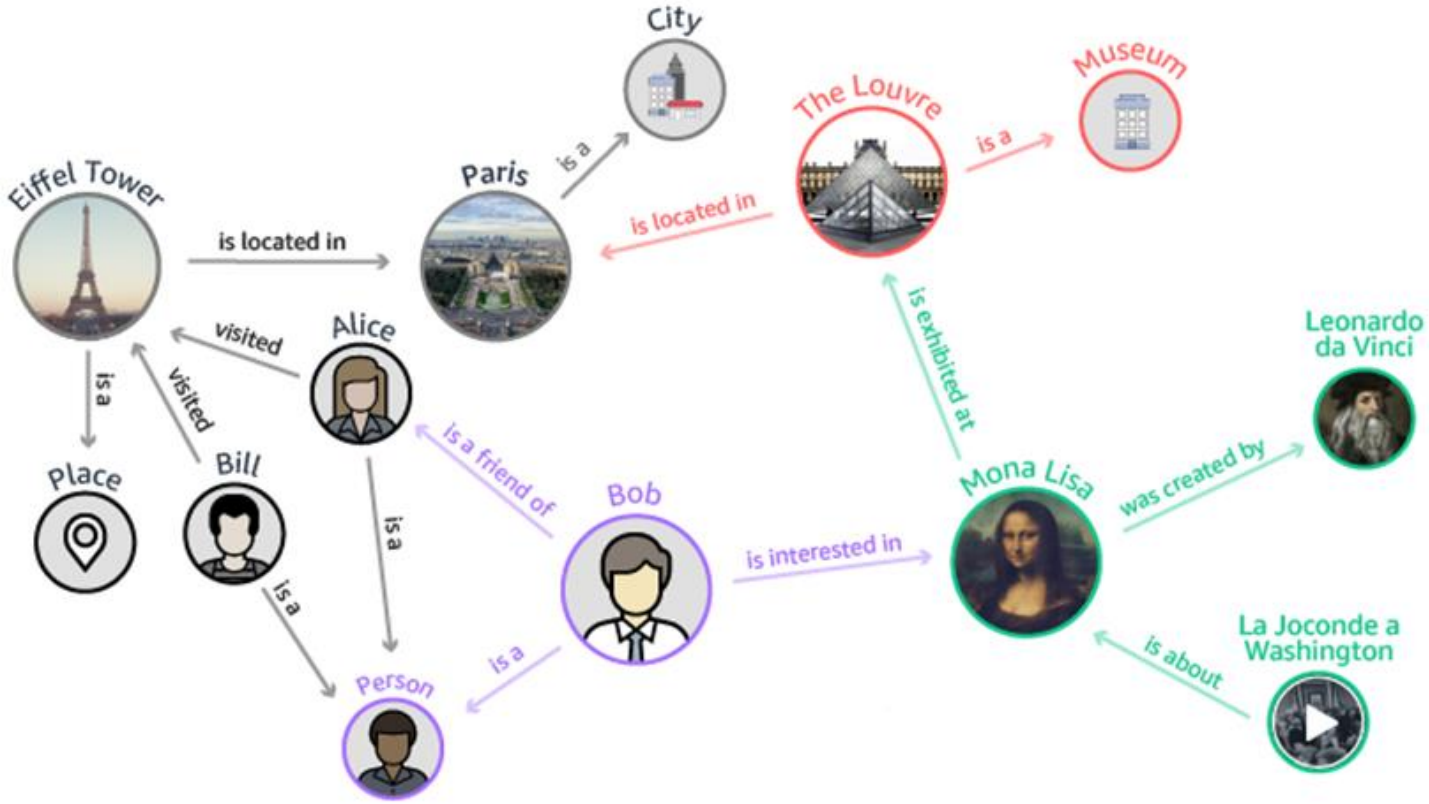
Sophisticated to represent rich connections in data.

Model Connected World as Graphs



Model Connected World as Graphs

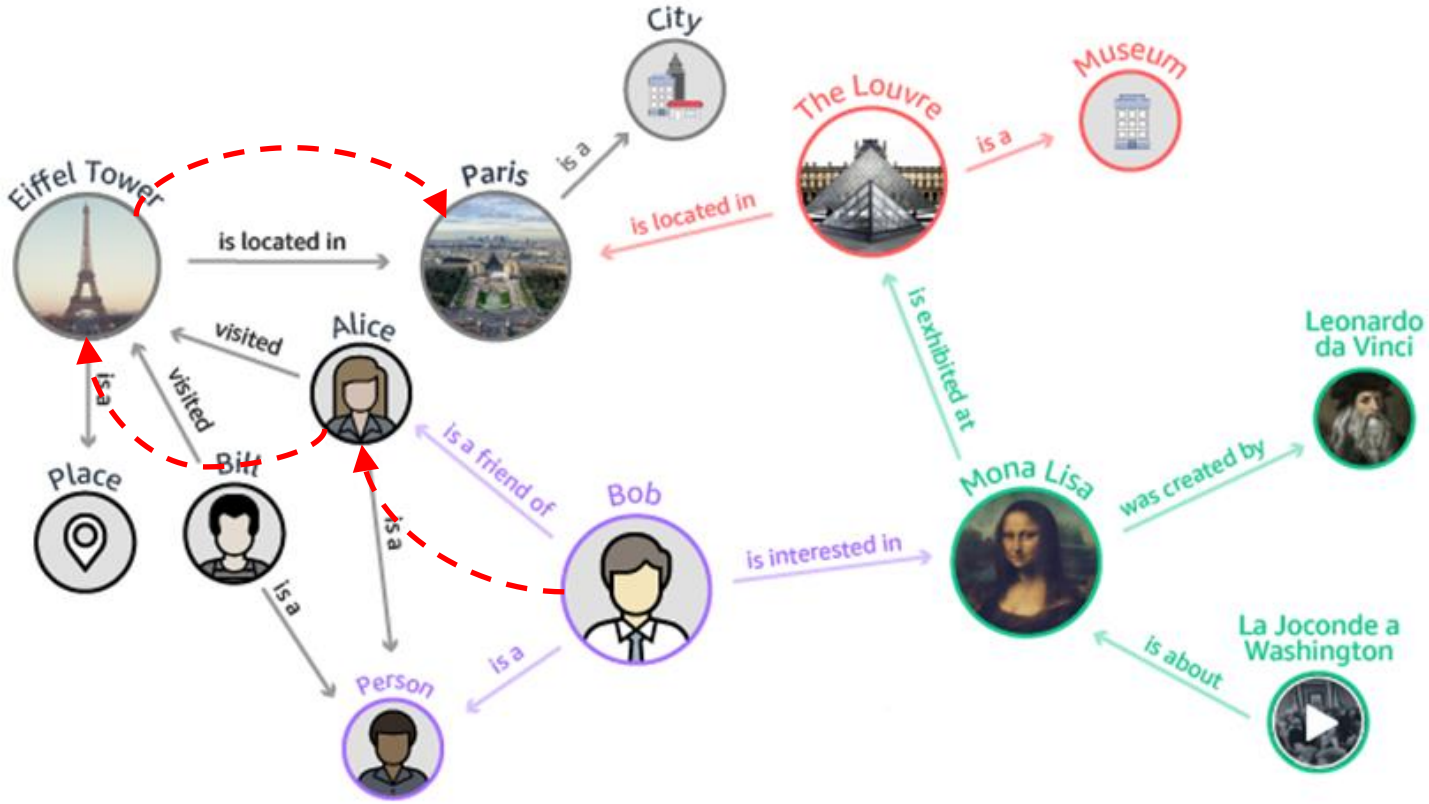
Execute a depth-first search from Bob with the constraint $friend \rightarrow visited \rightarrow located$ on the label sequence.



Query 1:
Find cities where Bob's friends visited.

Model Connected World as Graphs

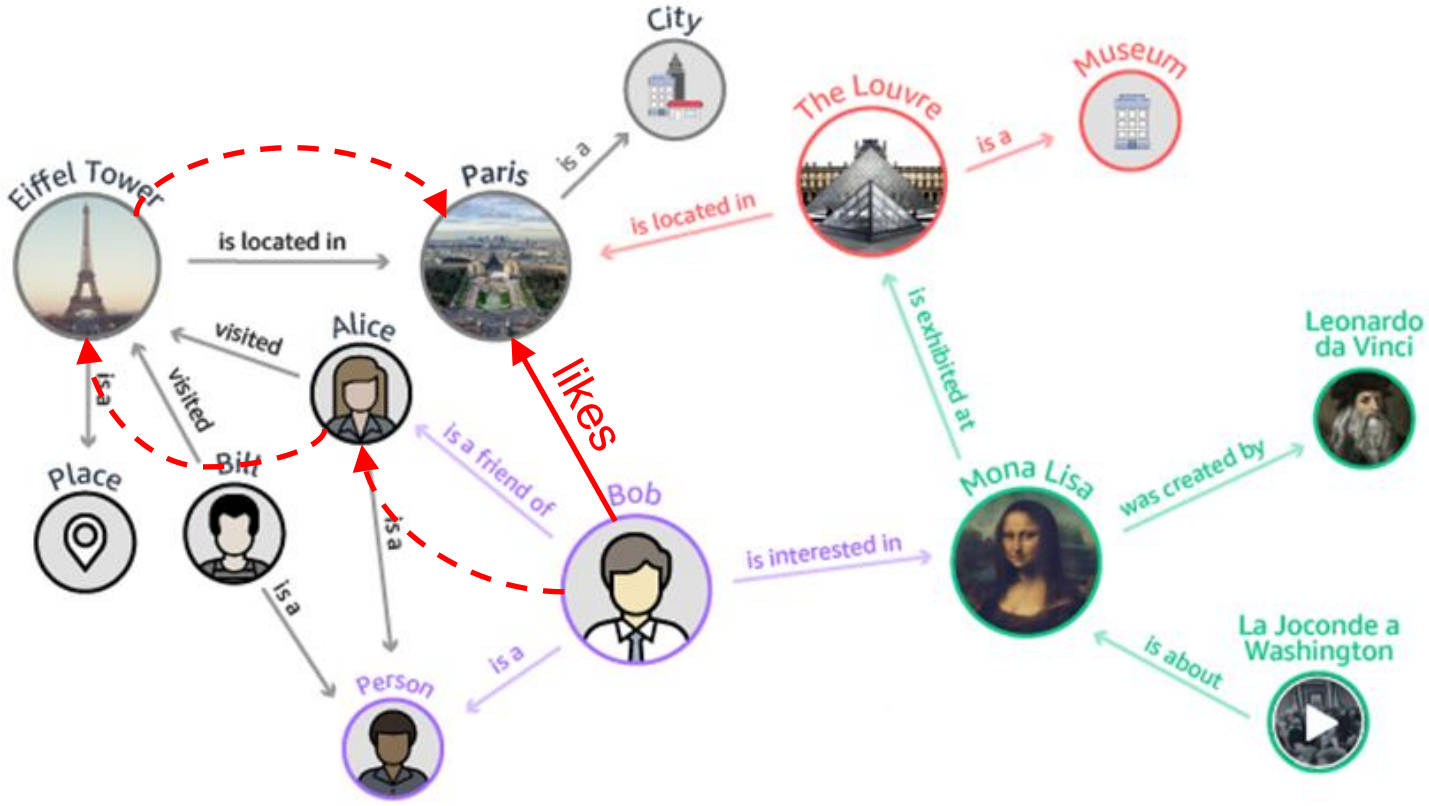
Execute a depth-first search from Bob with the constraint $friend \rightarrow visited \rightarrow located$ on the label sequence.



Query 1:
Find cities where Bob's friends visited.

Model Connected World as Graphs

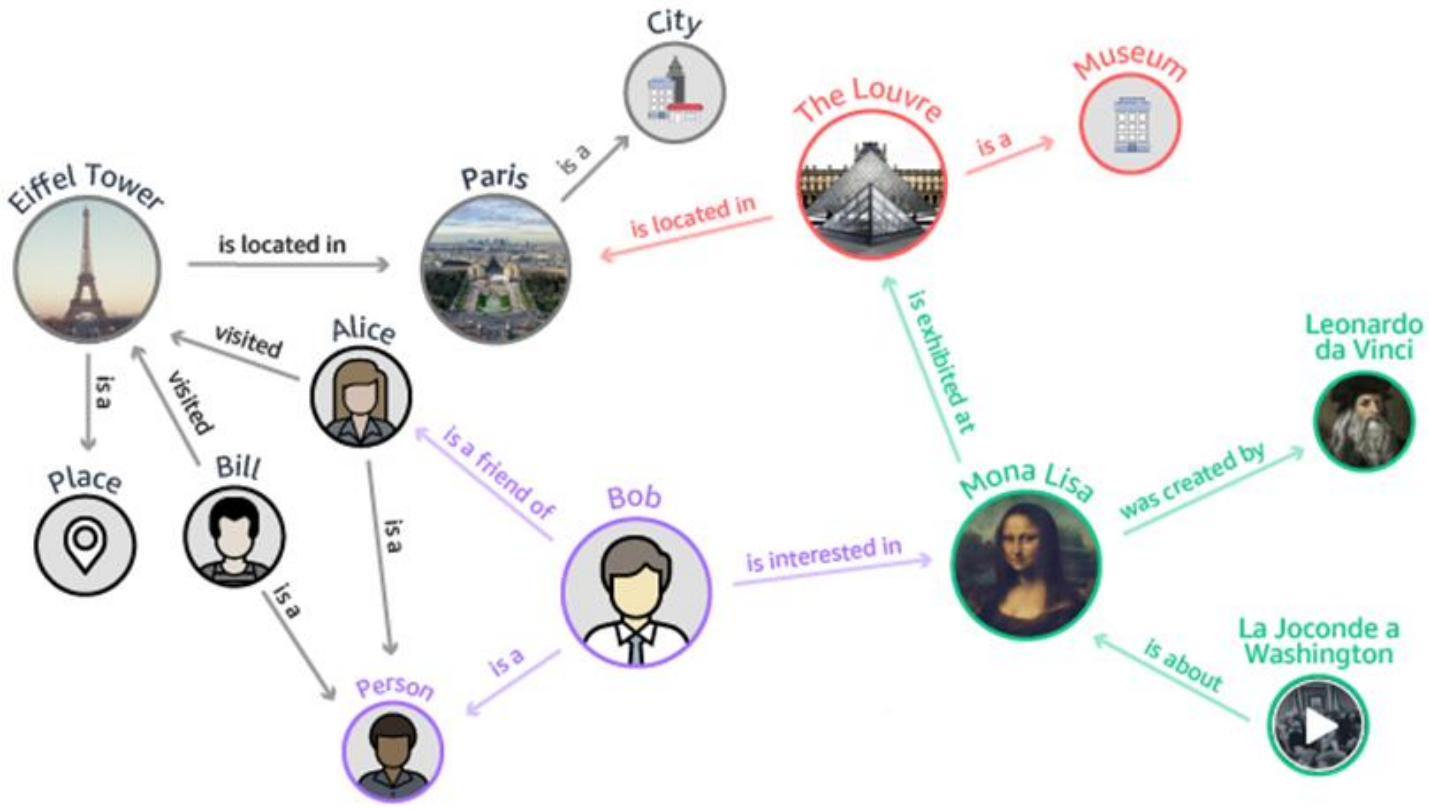
Insert an edge from Bob to Paris in the graph. The edge is labeled as $likes$.



Query 1:
Find cities where Bob's friends visited.

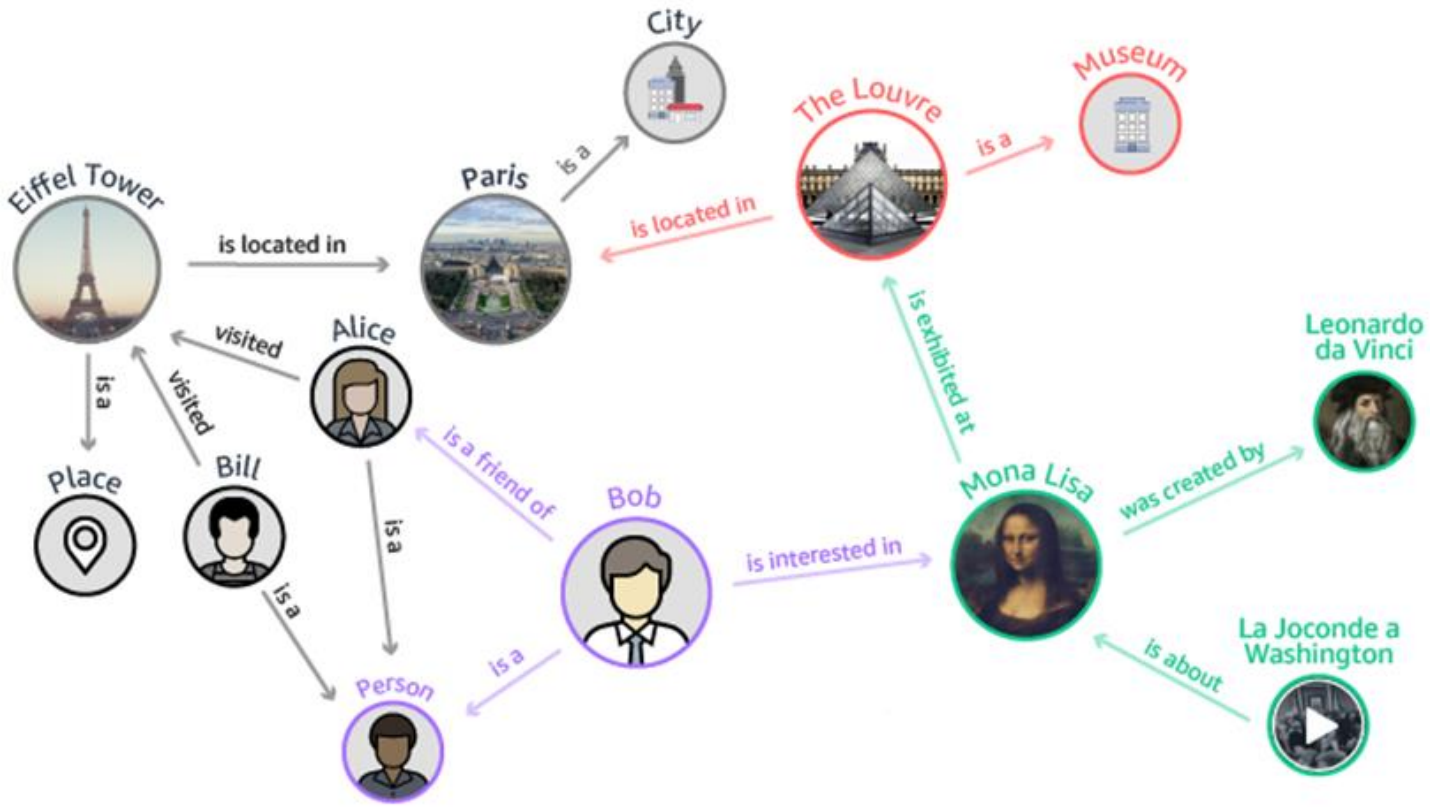
Query 2:
Add a "like" connection between Bob and cities where Bob's friends visited.

Graph is an Effective Way for Us to Understand and Manage Connected Data



High performance for connection queries.

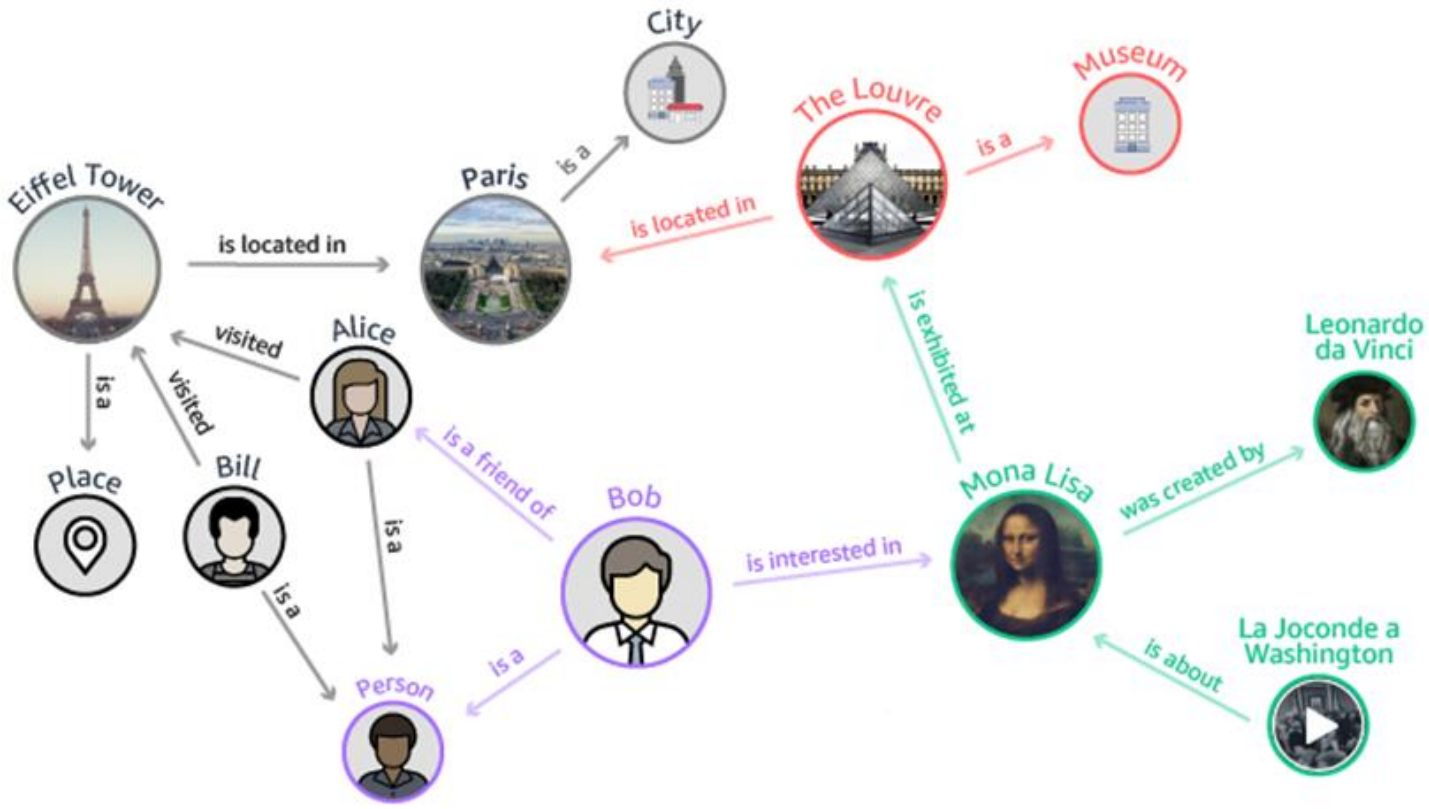
Graph is an Effective Way for Us to Understand and Manage Connected Data



High performance for connection queries.

Flexible to represent rich connections in data.

Graph is an Effective Way for Us to Understand and Manage Connected Data

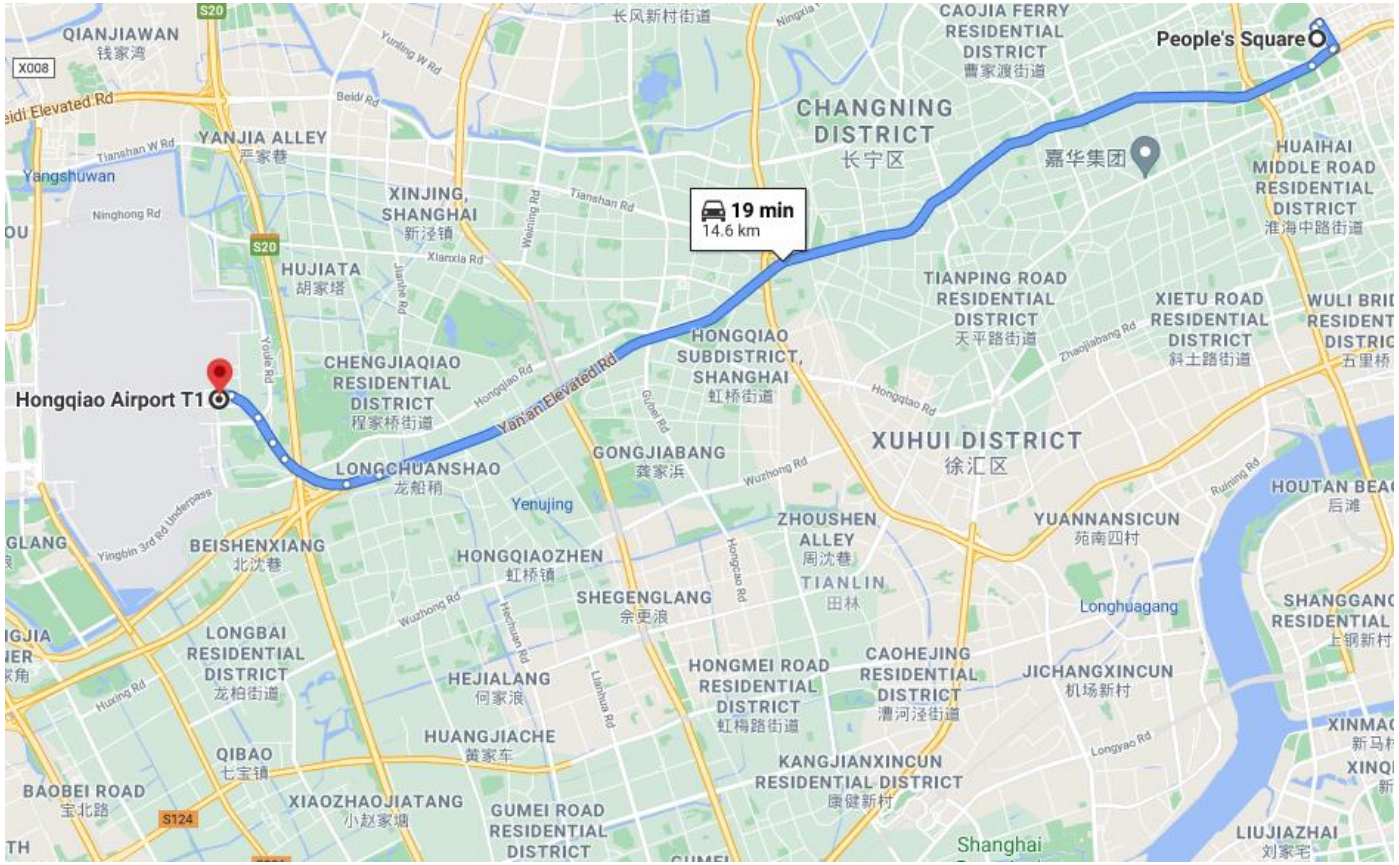


High performance for connection queries.

Flexible to represent rich connections in data.

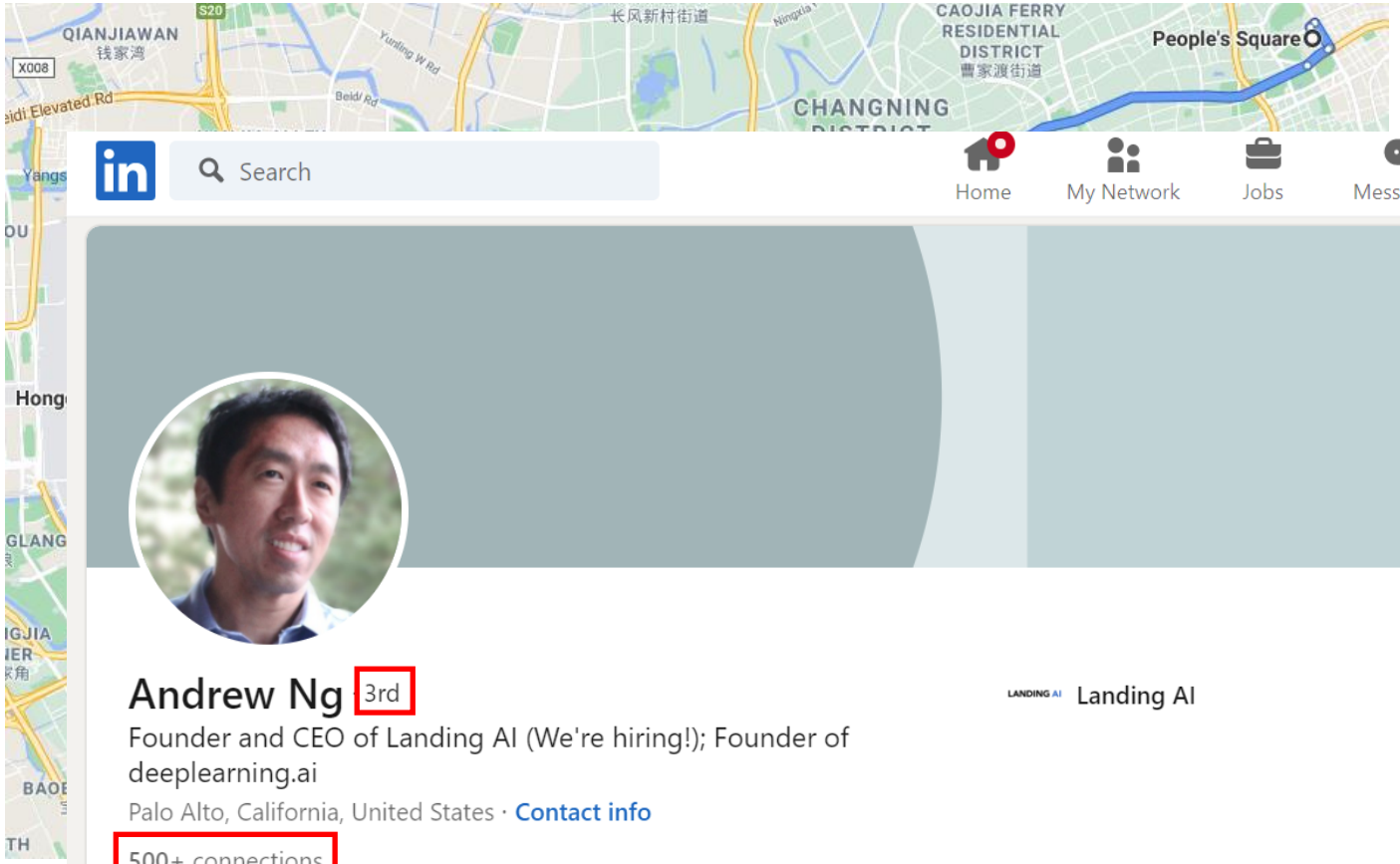
Deep insights into connections among entities.

We Use Graphs Everyday and Everywhere



Road network.

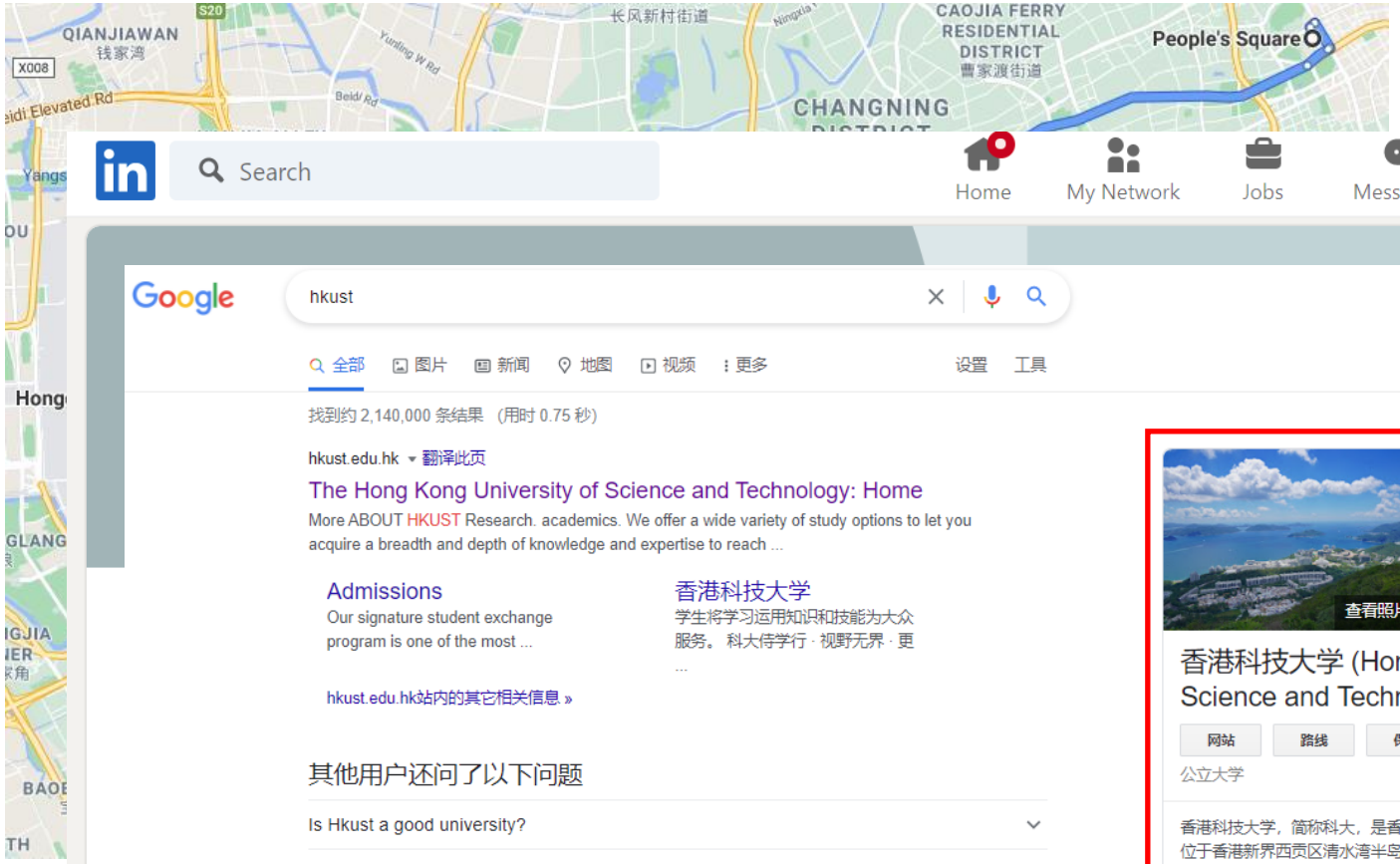
We Use Graphs Everyday and Everywhere



Road network.

Social network.

We Use Graphs Everyday and Everywhere



Road network.

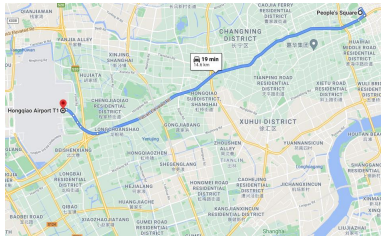
Social network.

Knowledge graph.

What is subgraph query processing? Why is it important?

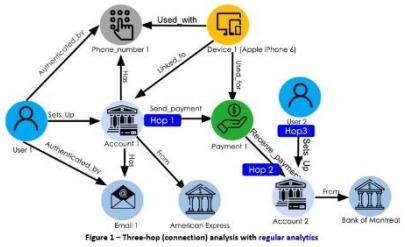
Retrieve Information from Graph Data

Map Navigation | **Social Media** | **Fraud Detection** | **Web Crawler** | **Protein-Interaction**

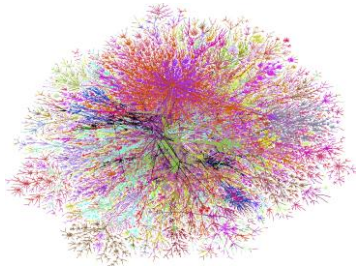


Road Network

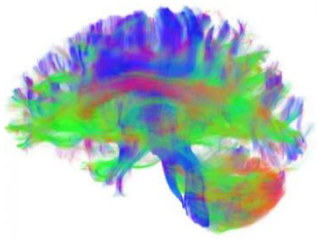
Social Network



Transaction Network

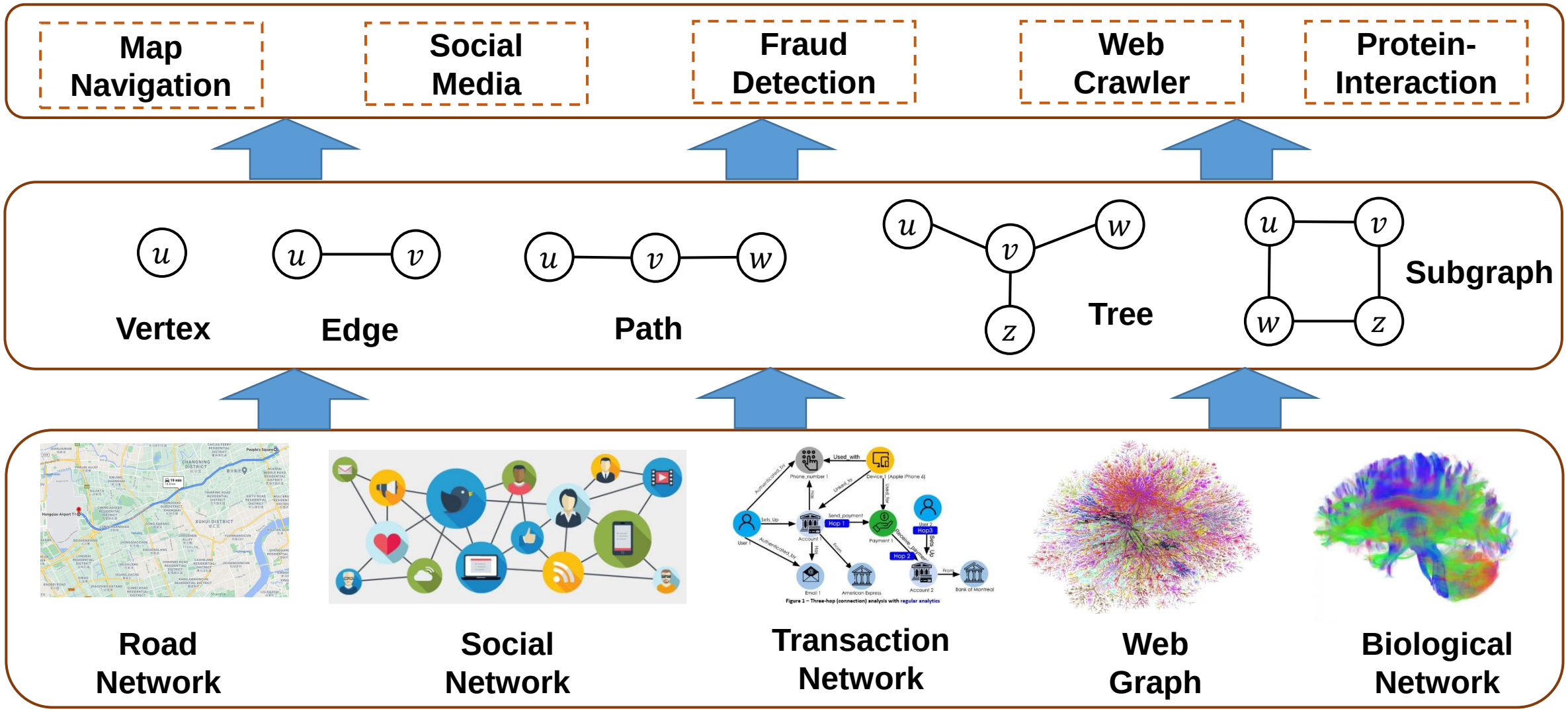


Web Graph



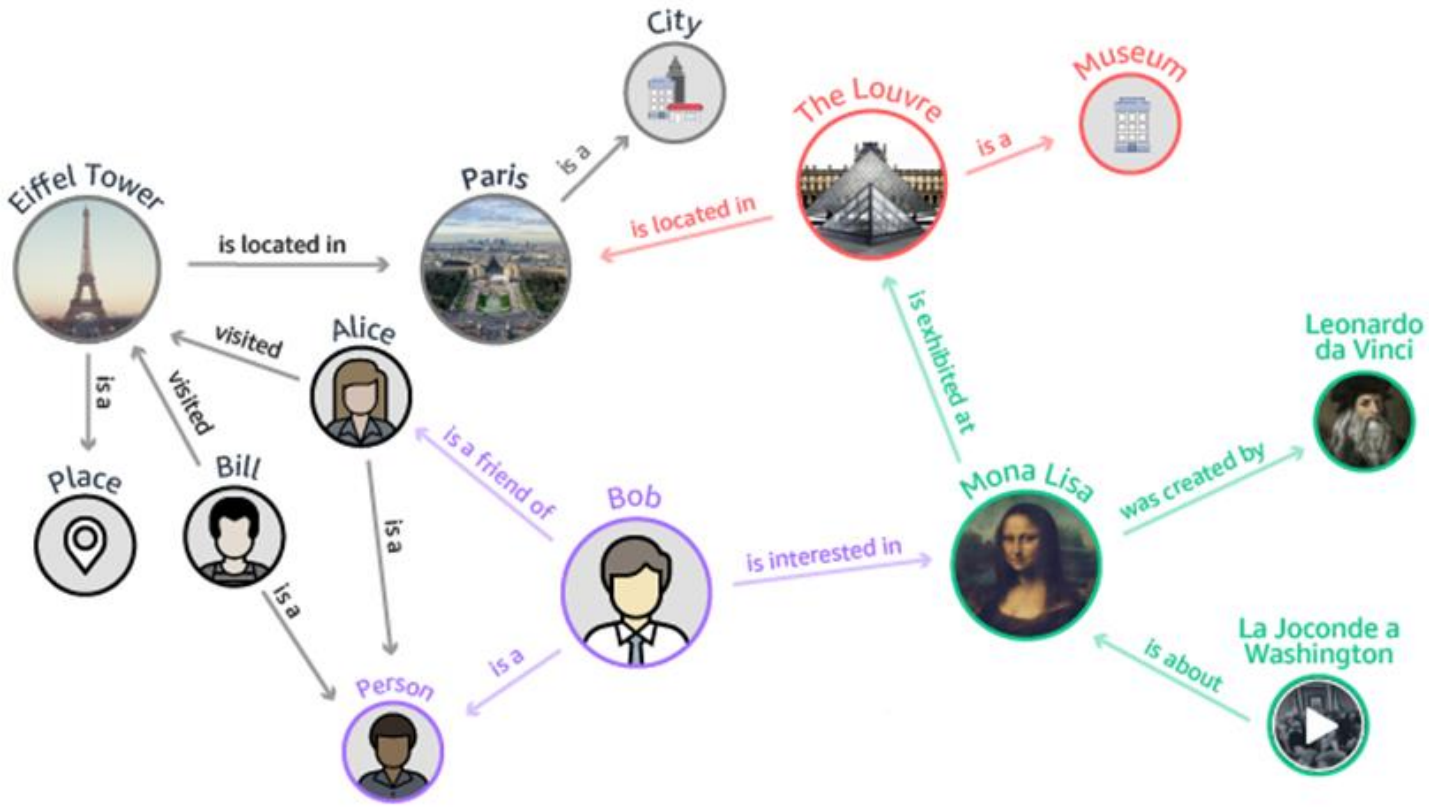
Biological Network

Retrieve Information from Graph Data

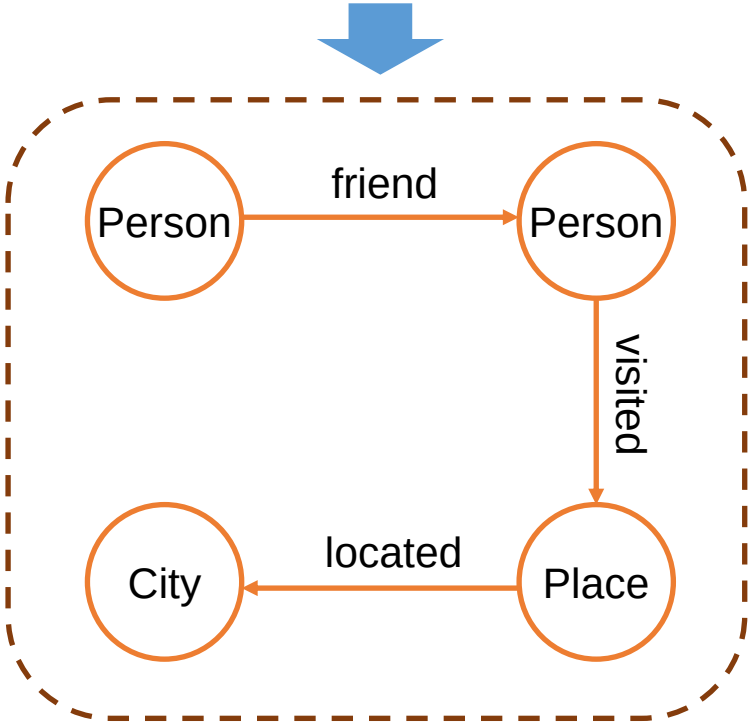


Subgraph Query Example

Result set: {(Bob, Alice, Eiffel Tower, Paris)}.



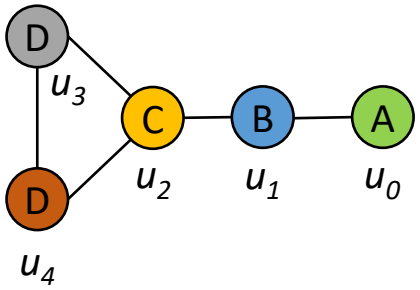
Query:
Enumerate all paths satisfying the pattern $friend \rightarrow visited \rightarrow located$.



Subgraph Matching

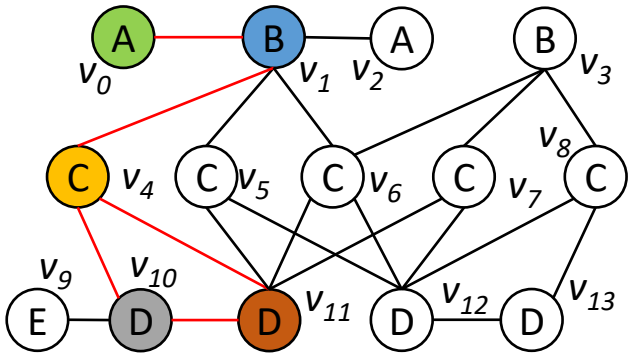
- A subgraph isomorphism (call a match for short) from a graph g to another graph g' is an injective function M from $V(g)$ to $V(g')$ such that:
  - $\forall u \in V(g), L(u) = L(M(u))$ and $\forall e(u, u') \in E(g), e(M(u), M(u')) \in E(g')$.

- Subgraph matching finds all matches from a query graph q to a data graph G.



Query graph q

$$\{(u_0, v_0), (u_1, v_1), (u_2, v_4), (u_3, v_{10}), (u_4, v_{11})\}$$



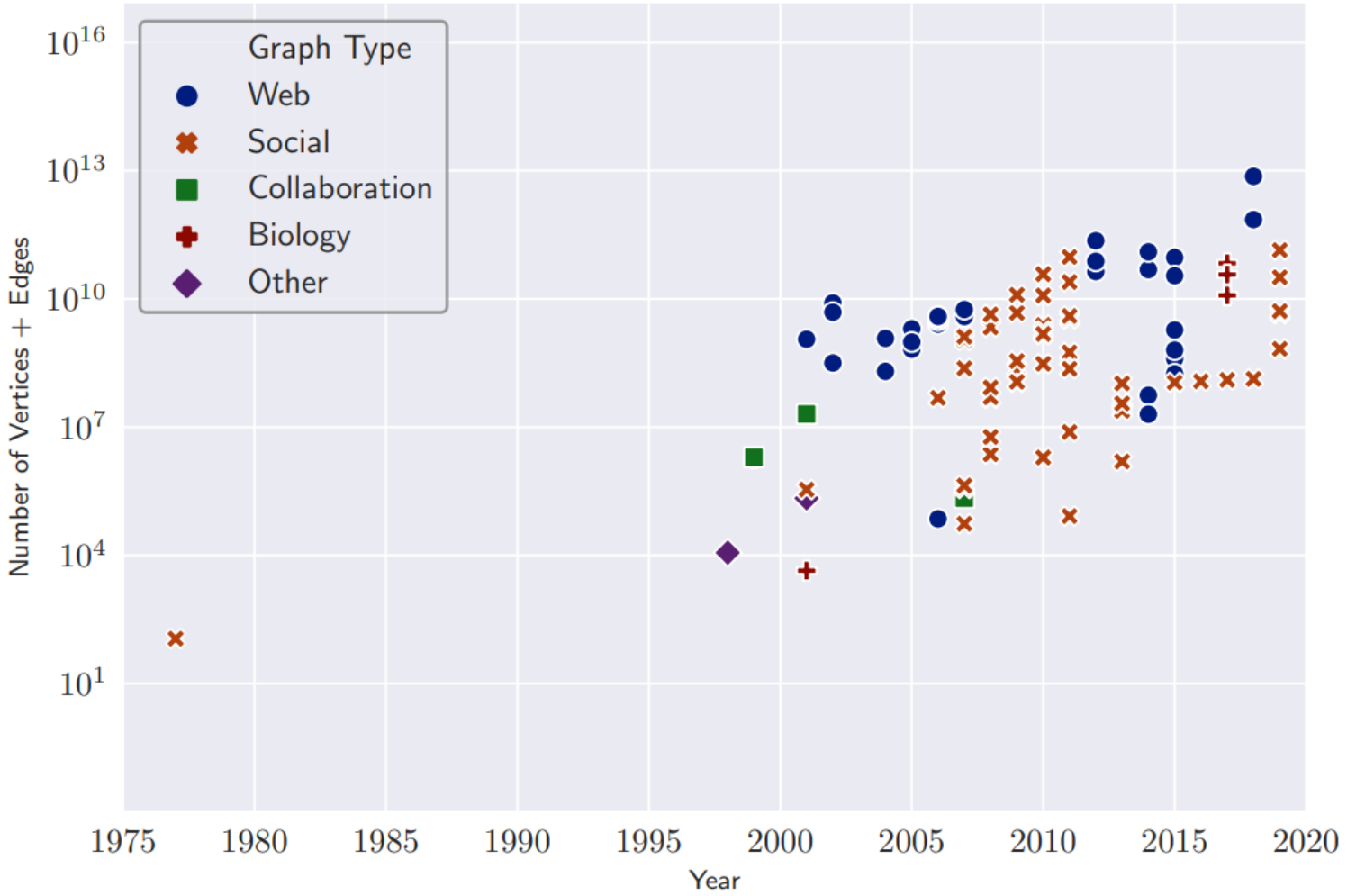
Data graph G

Subgraph Matching

- The ***fundamental operation*** to retrieve information from graph data.
- The ***core functionality*** in graph database management systems.
- The ***primitives*** in many graph analysis operations.

What are the challenges?

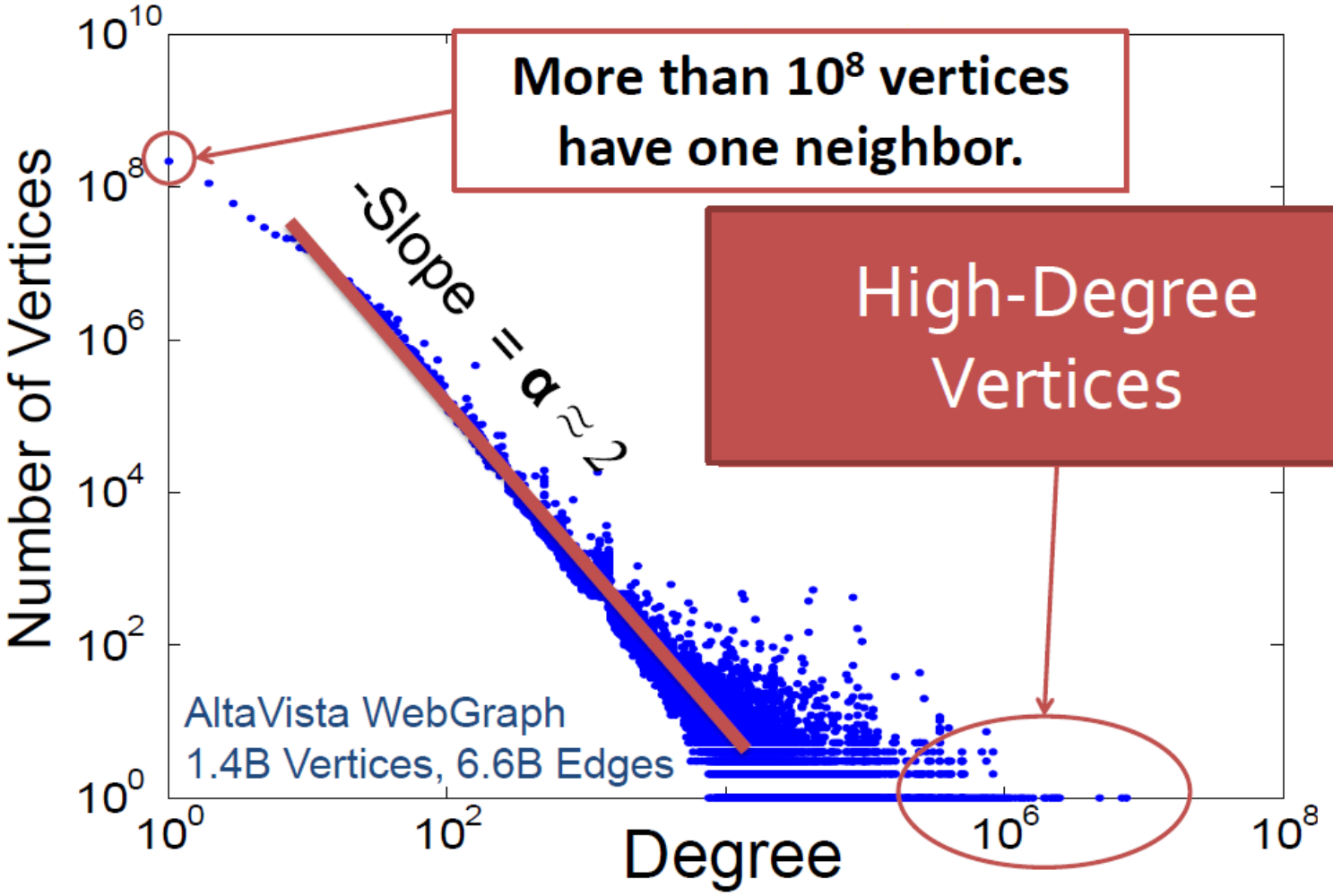
Challenges of Graph Data Processing



Large Volume:
Millions even trillions of edges.

Figure source: Laxman Dhulipala. Provably Efficient and Scalable Shared-Memory Graph Processing. PhD Thesis. 2020.

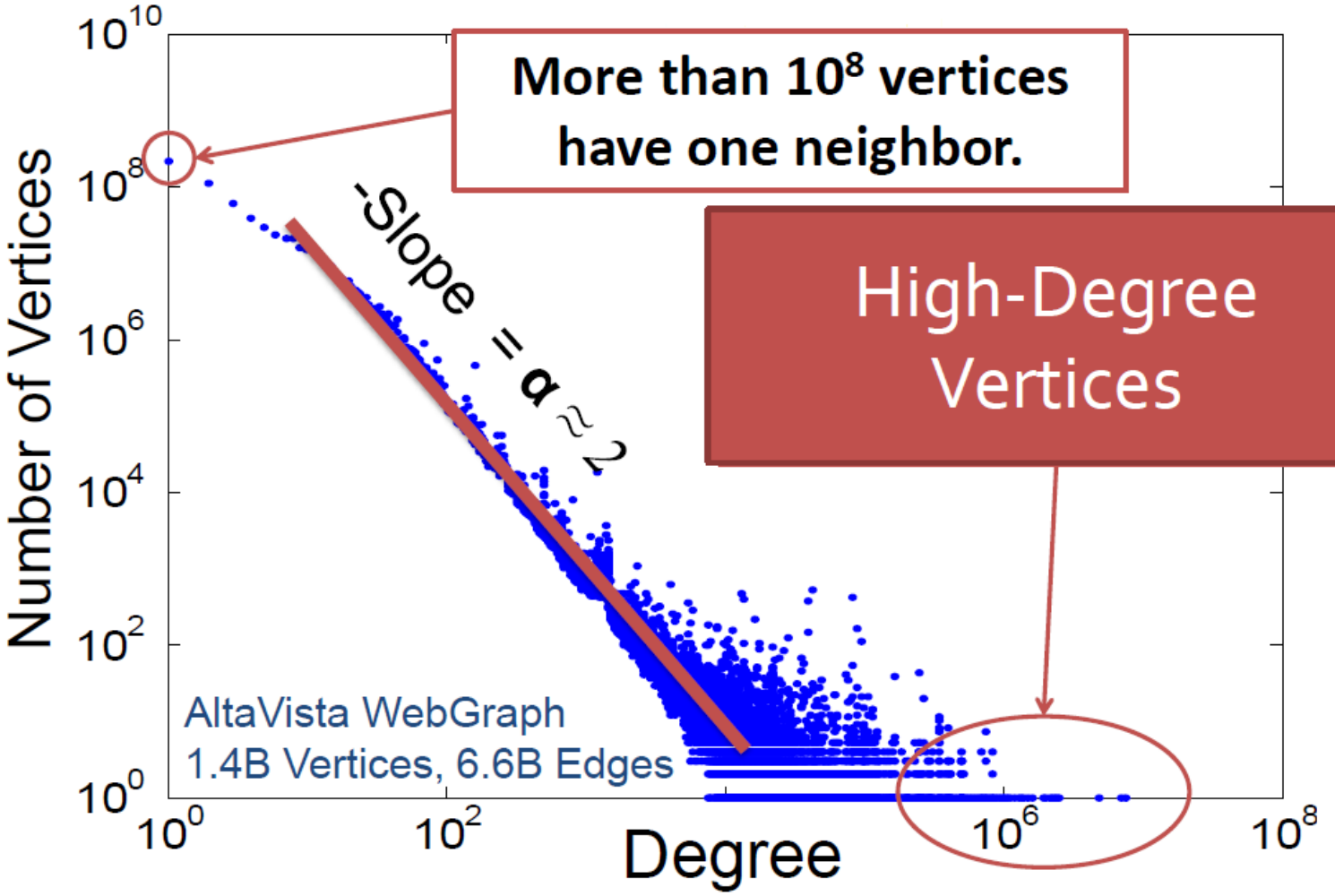
Challenges of Graph Data Processing



Large Volume:
Millions even trillions of edges.

Complex Structure:
Skewness, local dense communities.

Challenges of Graph Data Processing



Large Volume:
Millions even trillions of edges.

Complex Structure:
Skewness, local dense communities.

Poor Hardware Utilization:
Load imbalance, irregular memory access.

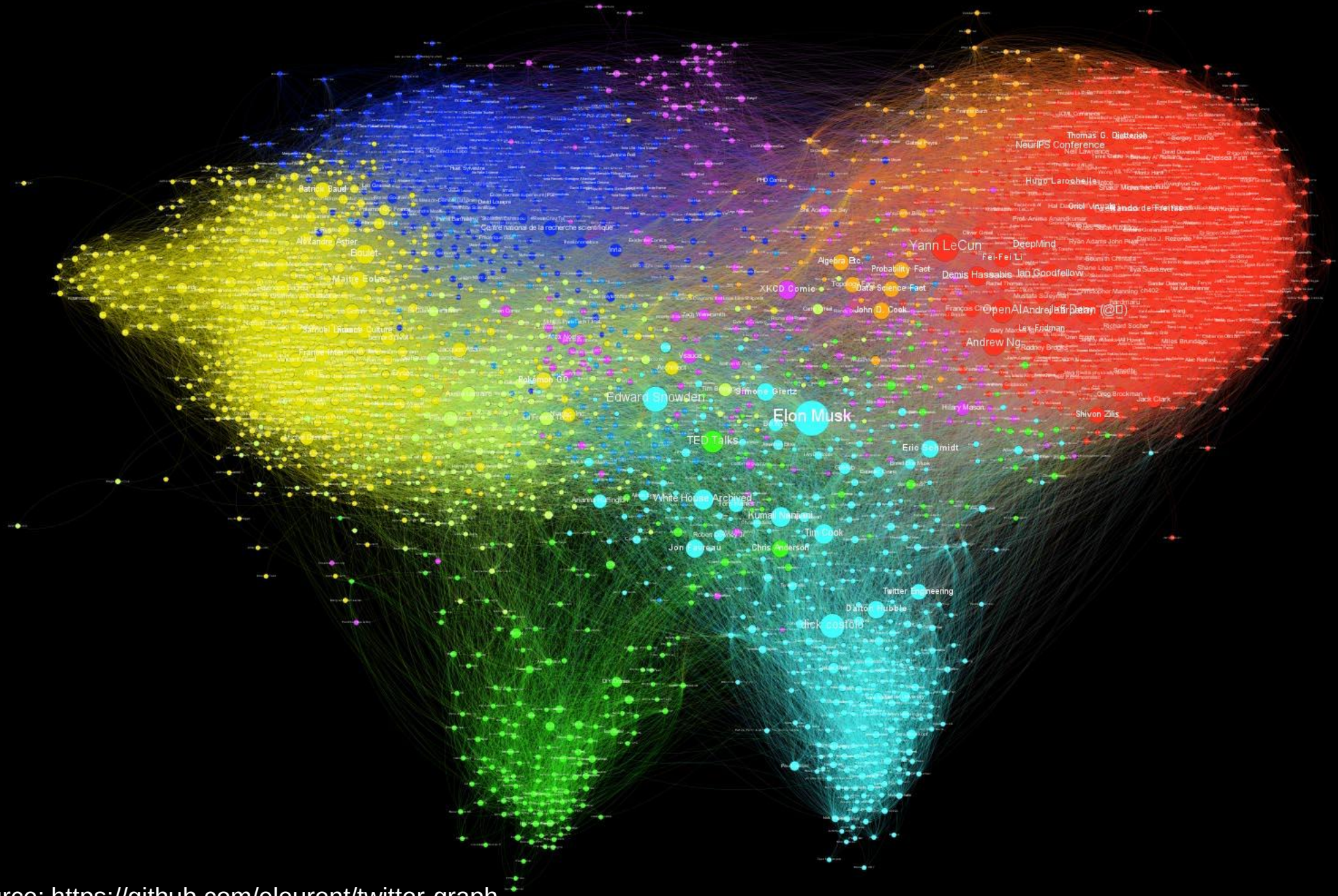
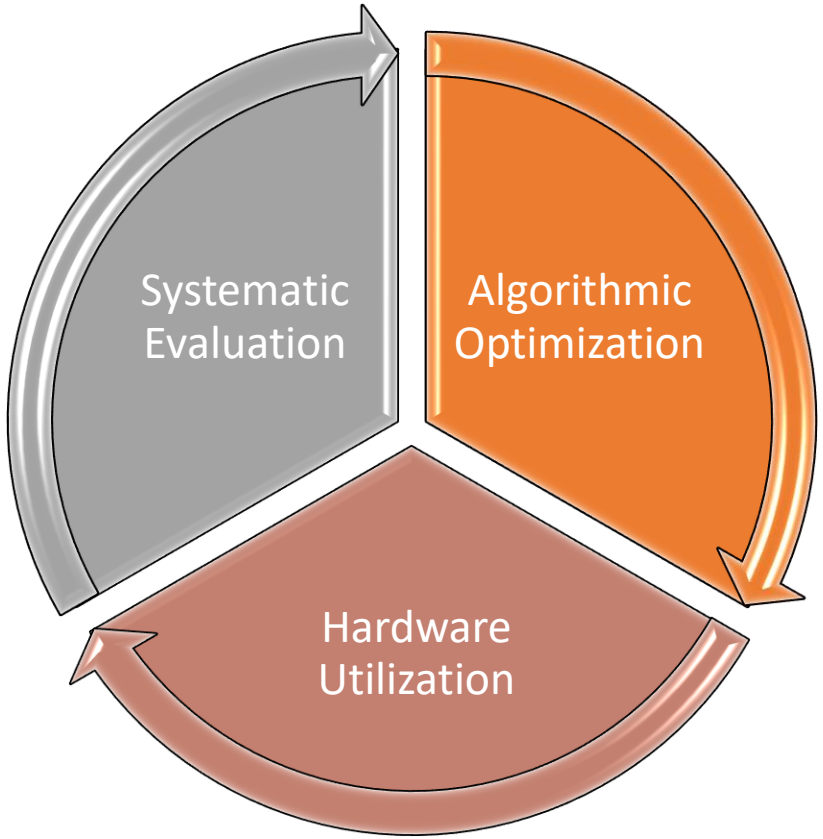


Figure source: https://github.com/eleurent/twitter-graph

My Primary Research

- Develop **efficient and effective** techniques to accelerate subgraph query processing.



In-Memory Subgraph Matching: An In-Depth Study

Shixuan Sun and Qiong Luo

Hong Kong University of Science and Technology

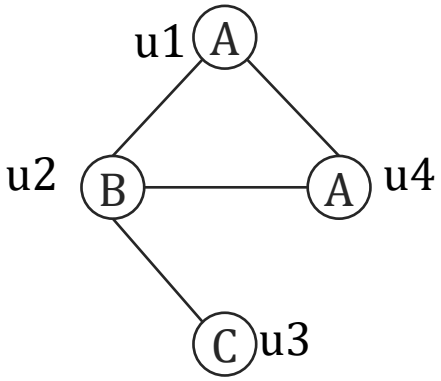
Preliminaries

- Deciding whether g' contains g, i.e., the subgraph isomorphism problem, is NP-complete.

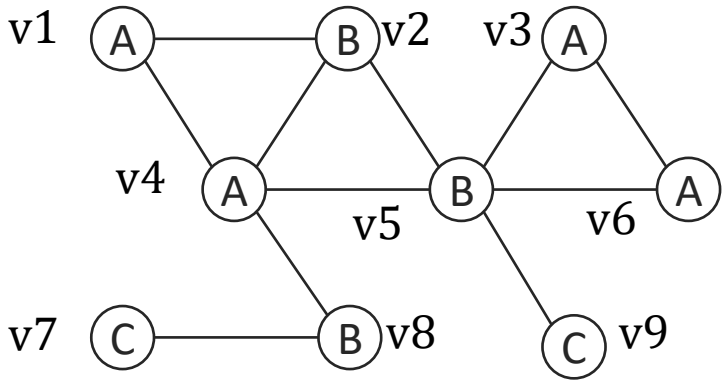
- Deciding whether g' is identical to g, i.e., the graph isomorphism problem, belongs to NP, but not known to be P or NP-complete.

Matching Order

- Given a query graph q, a matching order, denoted as π, is a permutation of vertices in q. $\pi[i]$ is the ith vertex in π, and $\pi[i{:}j]$ is the set of vertices from index i to j in π.
  - Example: π=($u1$,$u2$,$u4$,$u3$), π[1]=$u1$ and π[1:3]={$u1$,$u2$,$u4$}.



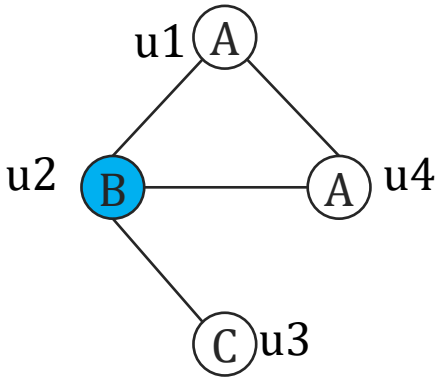
(a). Query graph q



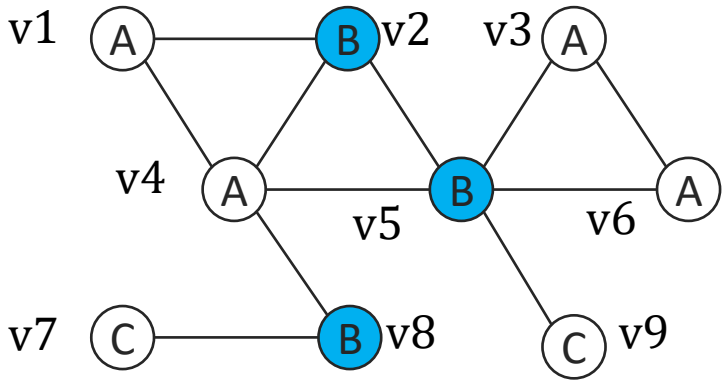
(b). Data graph G

Complete Candidate Set

- Given q and G, a complete candidate set of a query vertex u, denoted as $u.C$, is a set of vertices in G such that if a mapping (u, v) is in any subgraph isomorphism from q to G, then v belongs to $u.C$.
  - Example: $u2.C = \{v | v \in V(G)\ and\ L(v) = L(u2)\} = \{v2, v5, v8\}$.



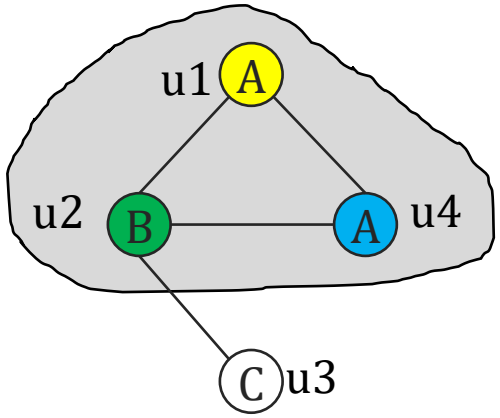
(a). Query graph q



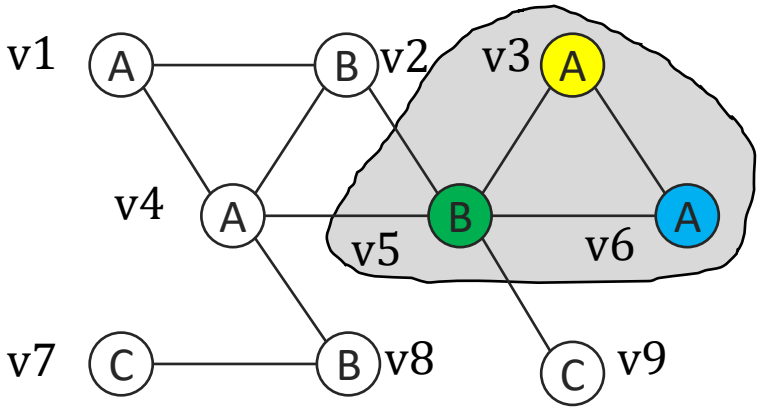
(b). Data graph G

Partial Subgraph Isomorphism (PSI)

- Given q, G and π, a partial subgraph isomorphism f_i is a subgraph isomorphism from $q[\pi[1:i]]$ to G where $1 \leq i \leq |V(q)|$, and $q[\pi[1:i]]$ is a vertex induced subgraph of q given $\pi[1:i]$. Specifically, $f_0 = \{\}$, and $f_{|V(q)|}$ is a subgraph isomorphism from q to G.
  - Example: Suppose $\pi = (u1, u2, u4, u3)$.
  - $f_3 = \{(u1, v3), (u2, v5), (u4, v6)\}$ is a partial subgraph isomorphism.



(a). Query graph q



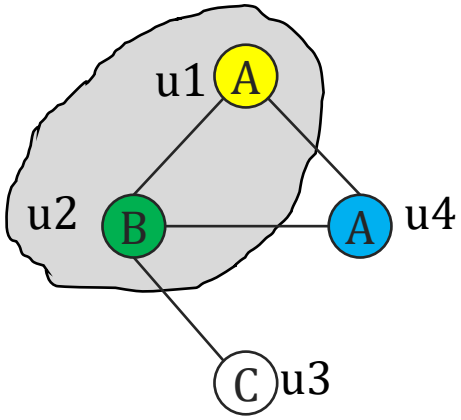
(b). Data graph G

Feasible Mapping

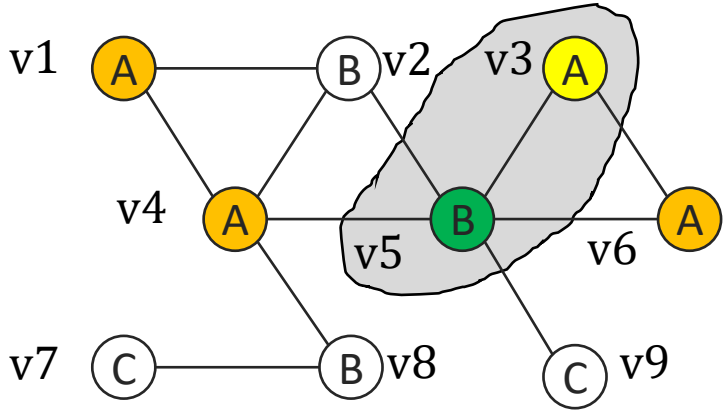
- Given $q,\ G,\ \pi,\ f_i,\ u = \pi[i+1]$ and $v \in u.C$ where $0 \leq i \leq |V(q)| - 1$, a mapping (u, v) is **feasible** if f_i can be extended to f_{i+1} by adding the mapping (u, v) to f_i. Otherwise, the mapping is **infeasible**.
  - Feasible condition:
    1. $L(u) = L(v)$;
    2. $v\ is\ not\ mapped$;
    3. $For\ every\ vertex\ u'\ that\ has\ been\ mapped,\ if\ e(u, u') \in E(q), then\ e\big(v, f(u')\big) \in G$.

Example Feasible Mapping

- Given $\pi = (u1, u2, u4, u3), u4.C = \{v1, v3, v4, v6\}$ and $f_2 = \{(u1, v3), (u2, v5)\}$:
  - the mapping $(u4, v3)$ is infeasible: $v3$ has been mapped;
  - the mapping $(u4, v4)$ is infeasible: $e(v3, v4) \notin E(G)$;
  - the mapping $(u4, v6)$ is feasible;
  - f_2 can be extended to $f_3 = \{(u1, v3), (u2, v5), (u4, v6)\}$ by adding the feasible mapping $(u4, v6)$.



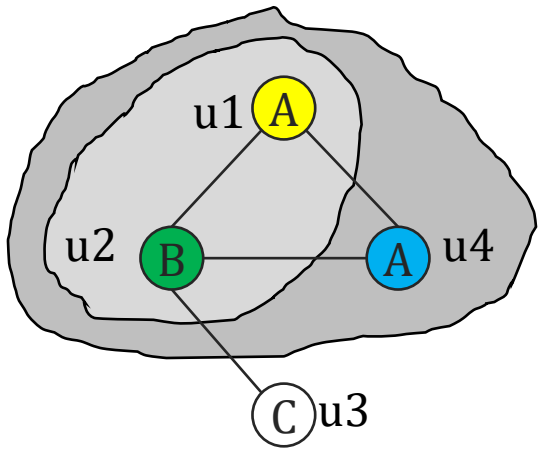
(a). Query graph q



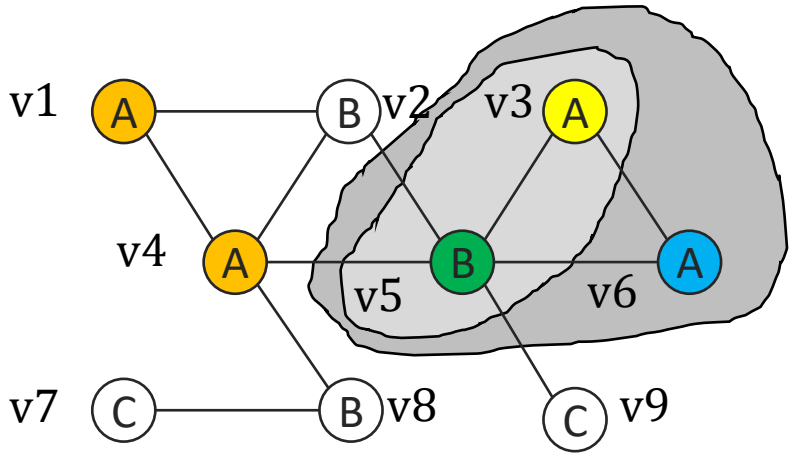
(b). Data graph G

Example Feasible Mapping

- Given $\pi = (u1, u2, u4, u3), u4.C = \{v1, v3, v4, v6\}\ and\ f_2 = \{(u1, v3), (u2, v5)\}$:
  - the mapping $(u4, v3)$ is infeasible: $v3$ has been mapped;
  - the mapping $(u4, v4)$ is infeasible: $e(v3, v4) \notin E(G)$;
  - the mapping $(u4, v6)$ is feasible;
  - f_2 can be extended to $f_3 = \{(u1, v3), (u2, v5), (u4, v6)\}$ by adding the feasible mapping $(u4, v6)$.



(a). Query graph q



(b). Data graph G

Graph Exploration based Approaches

- **General Idea:**

  Input: a query graph q and a data graph G

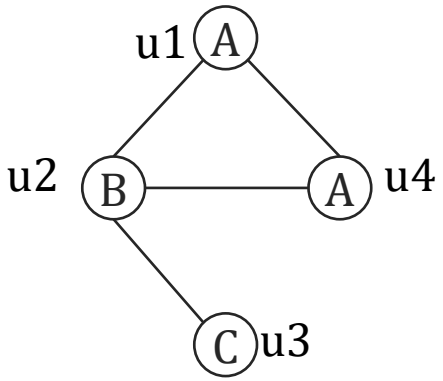
  Output: all subgraph isomorphisms from q to G

  1. Generate a matching order π;
  2. Obtain a complete candidate set $u.C$ for every vertex $u \in V(q)$;
  3. Recursively enumerate all solutions by extending partial subgraph isomorphisms iteratively along π.

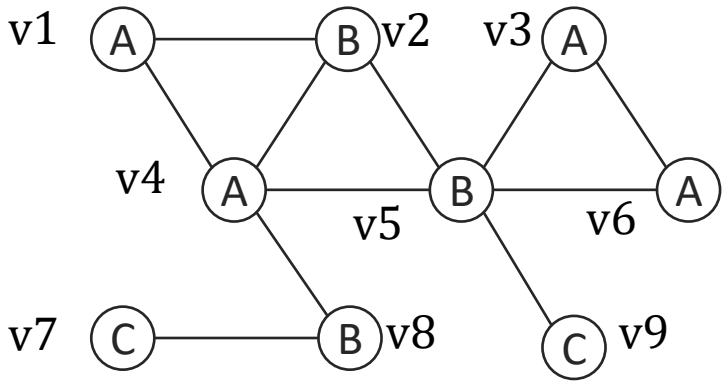
Generate Matching Order and Candidate Sets

- **Generate a matching order:**
  - $\pi =$ the input order of query vertices.
  - Example: $\pi = (u1, u2, u3, u4)$.

- **Generate complete candidate sets:**
  - $u.C = \{v | v \in V(G), L(v) = L(u)\ and\ d(v) \geq d(u)\}$, $for\ every\ u \in V(q)$.
  - Example: $u1.C = \{v1, v3, v4, v6\}, u2.C = \{v2, v5\}, u3.C = \{v7, v9\}, u4.C = \{v1, v3, v4, v6\}$.



(a). Query graph q



(b). Data graph G

Recursive Enumeration Process

$u1.C = \{v1, v3, v4, v6\}$

$u2.C = \{v2, v5\}$

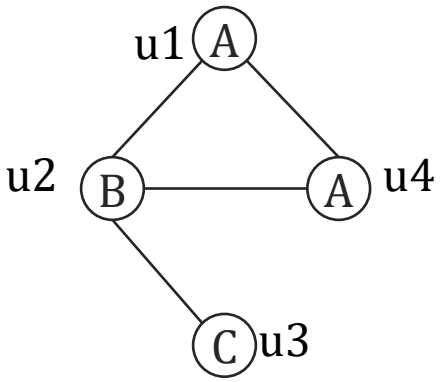
$u3.C = \{v7, v9\}$

$u4.C = \{v1, v3, v4, v6\}$

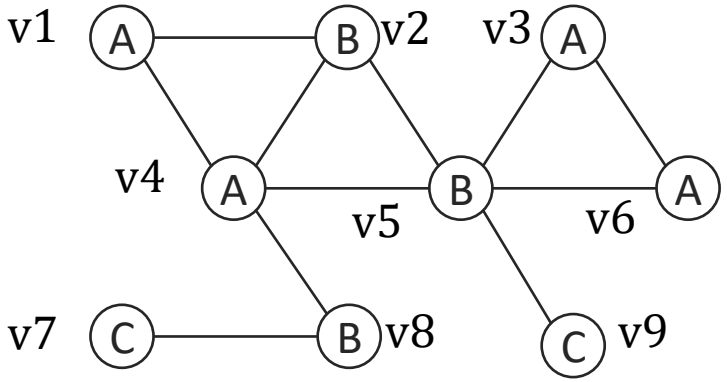
$f_0 = \{\}$

f_0

- Node: a psi
- Edge: a mapping
- Cross: infeasible mapping
- Tick: a solution



(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

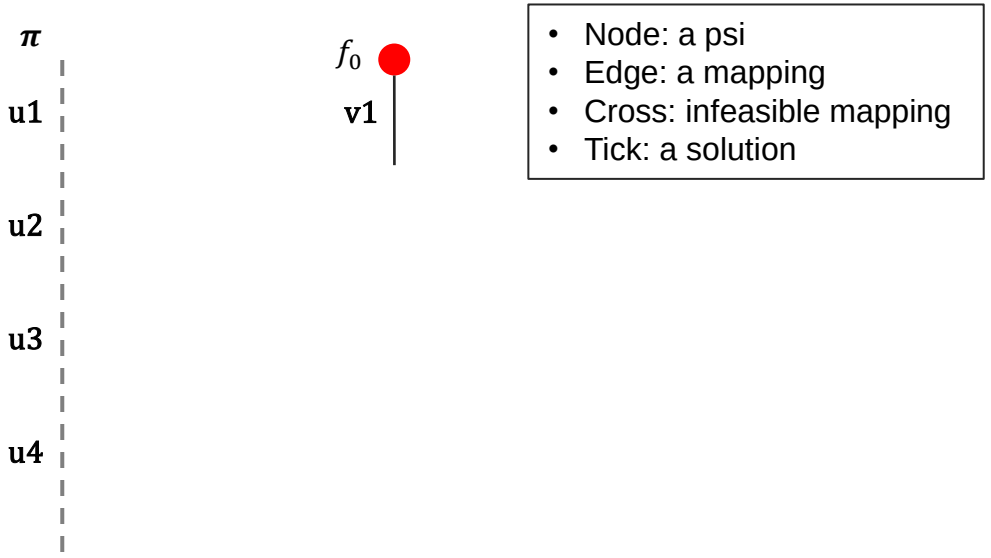
$u1.C = \{v1, v3, v4, v6\}$

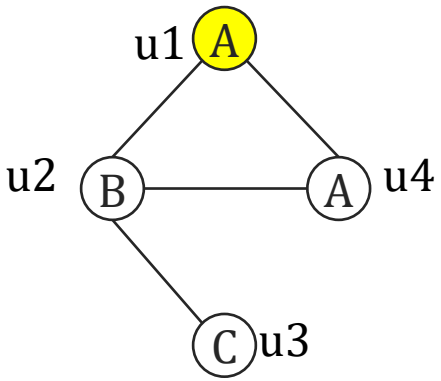
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

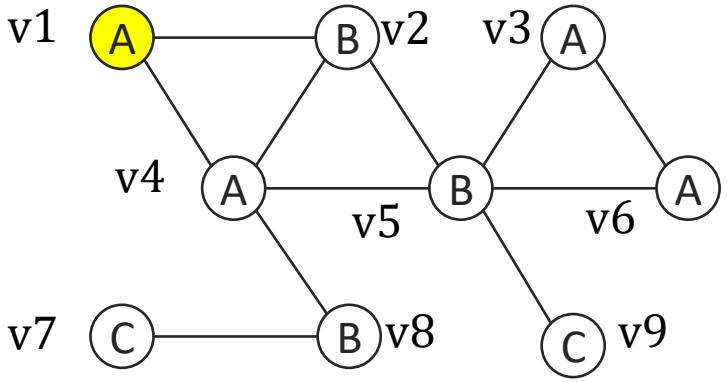
$u4.C = \{v1, v3, v4, v6\}$

$f_0 = \{\}$





(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

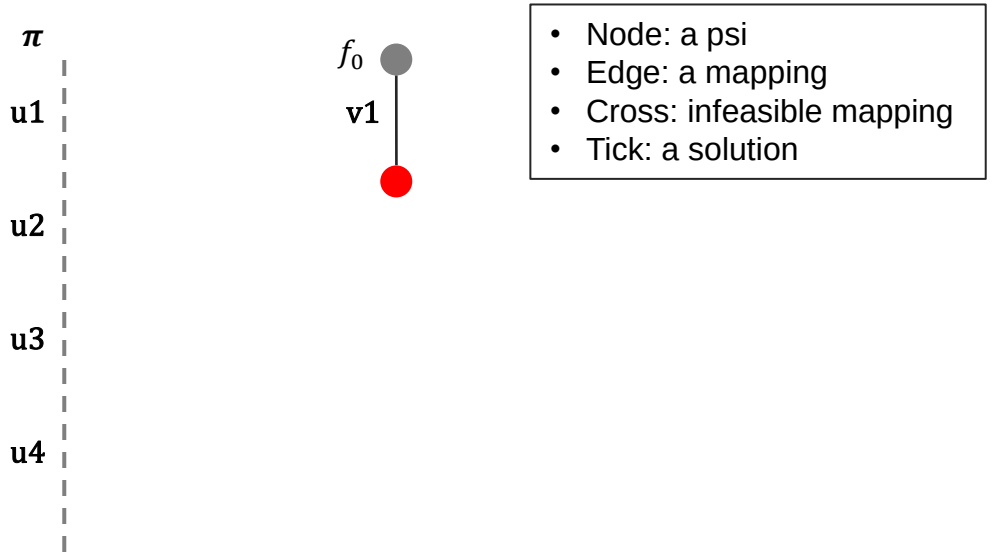
$u1.C = \{v1, v3, v4, v6\}$

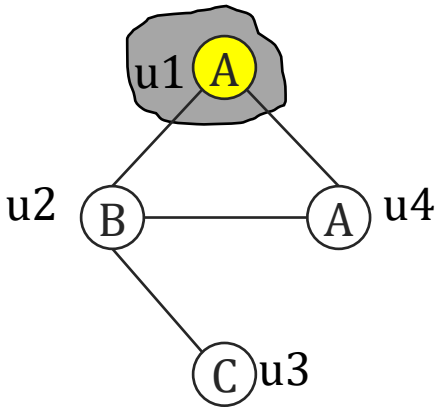
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

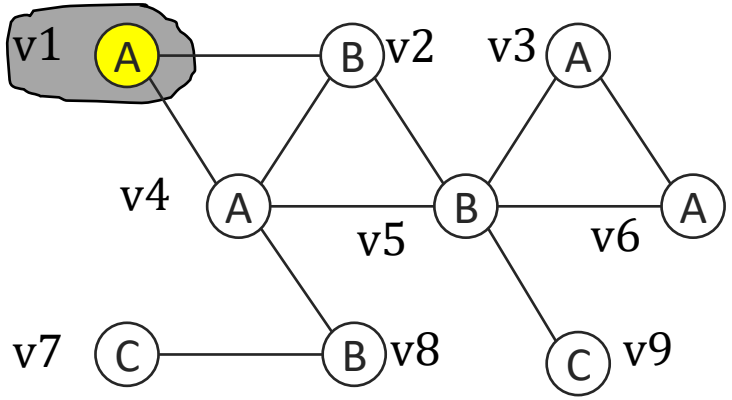
$u4.C = \{v1, v3, v4, v6\}$

$f_1 = \{(u1, v1)\}$





(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

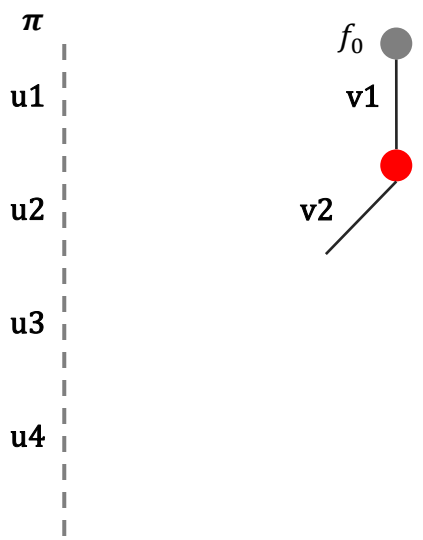
$u1.C = \{v1, v3, v4, v6\}$

$u2.C = \{v2, v5\}$

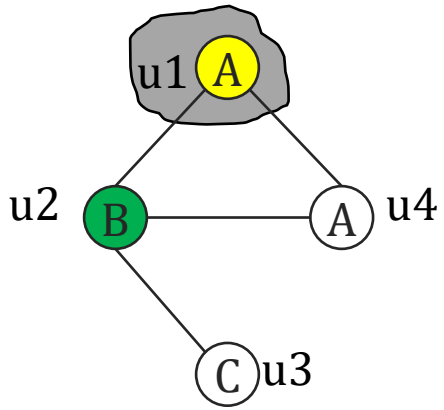
$u3.C = \{v7, v9\}$

$u4.C = \{v1, v3, v4, v6\}$

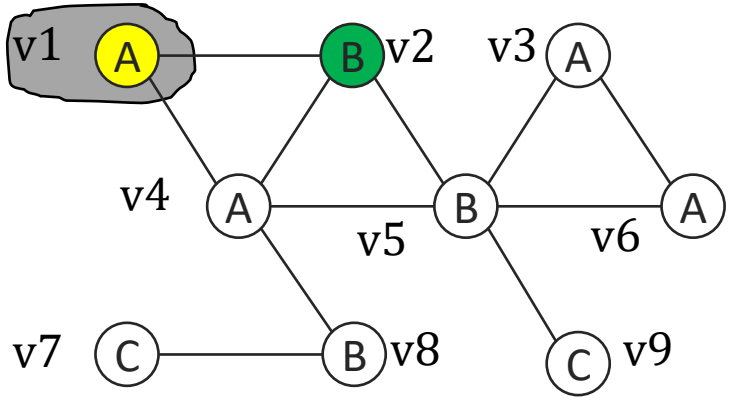
$f_1 = \{(u1, v1)\}$



- Node: a psi
- Edge: a mapping
- Cross: infeasible mapping
- Tick: a solution



(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

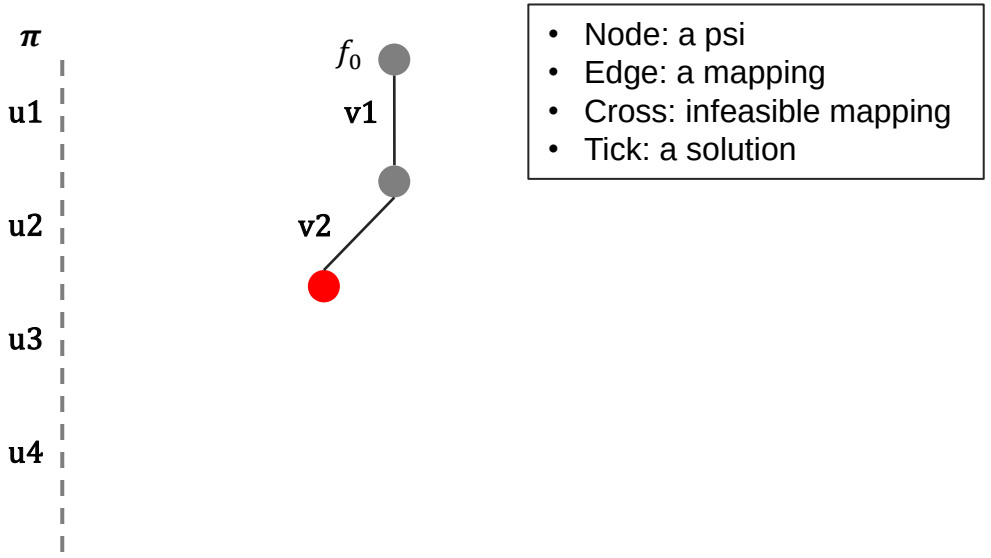
$u1.C = \{v1, v3, v4, v6\}$

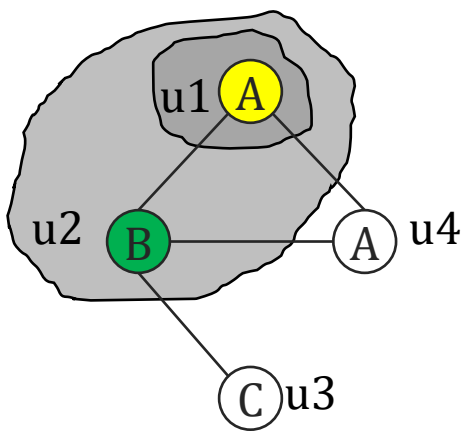
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

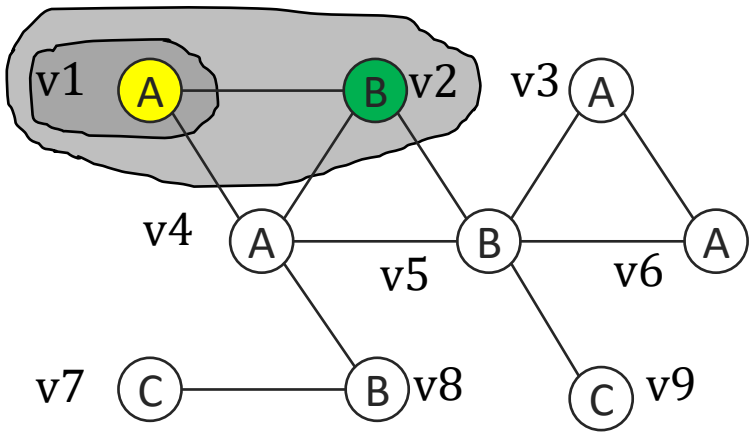
$u4.C = \{v1, v3, v4, v6\}$

$f_2 = \{(u1, v1), (u2, v2)\}$





(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

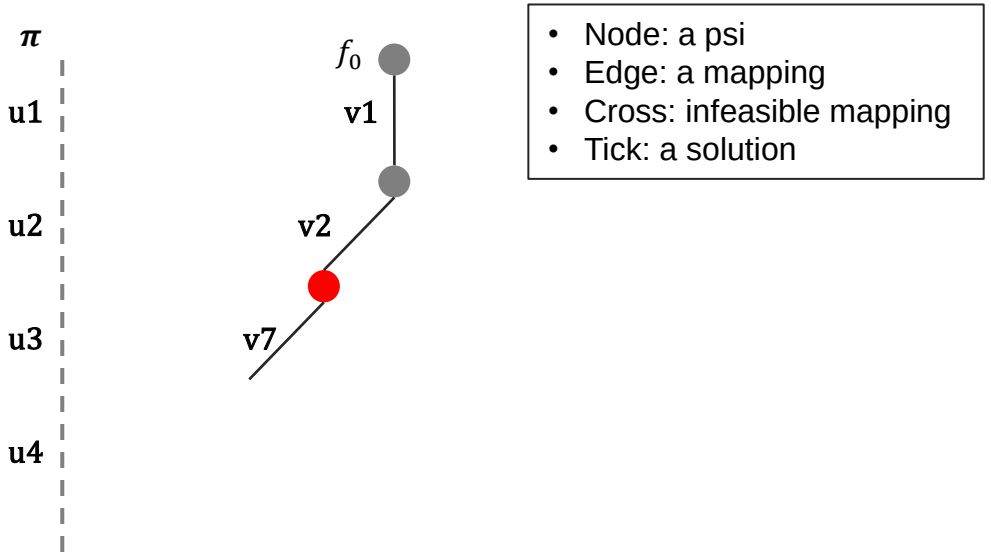
$u1.C = \{v1, v3, v4, v6\}$

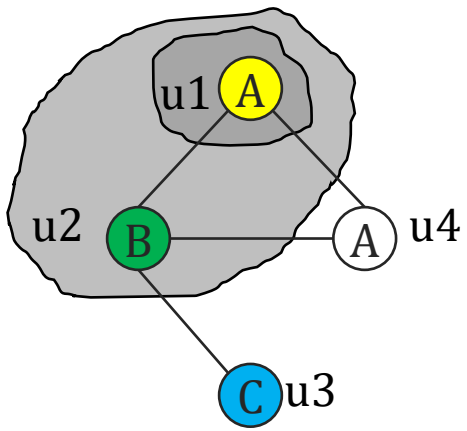
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

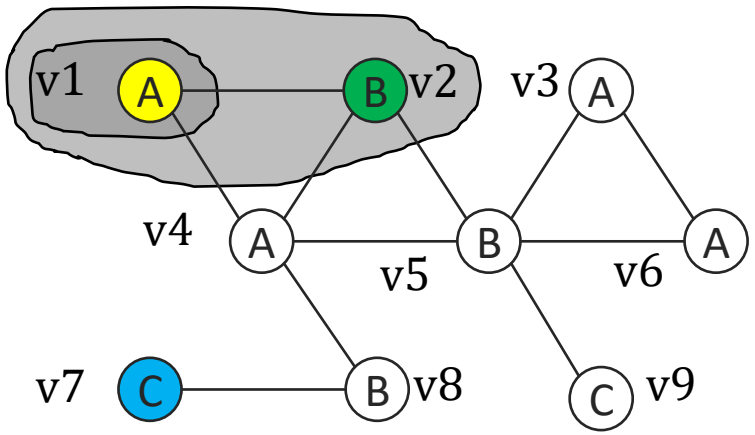
$u4.C = \{v1, v3, v4, v6\}$

$f_2 = \{(u1, v1), (u2, v2)\}$





(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

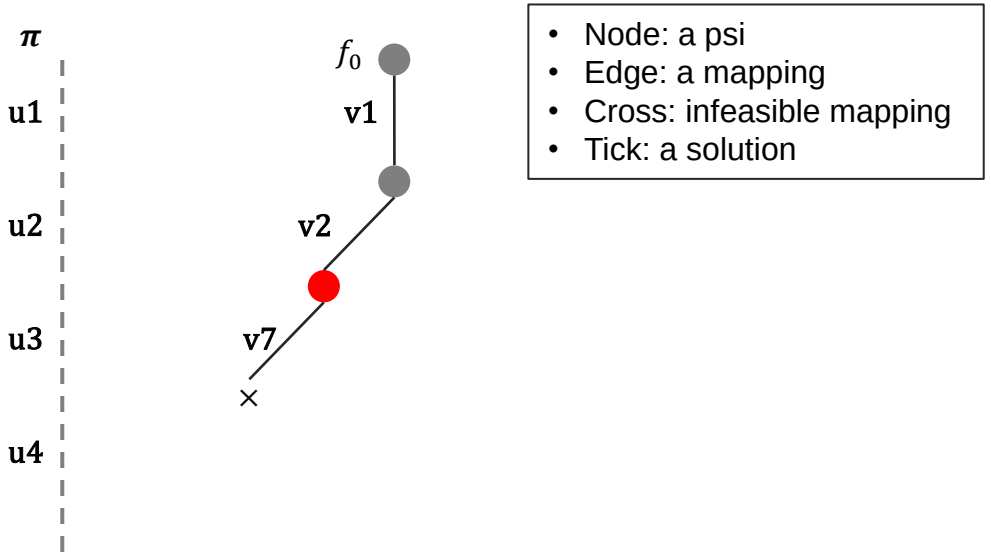
$u1.C = \{v1, v3, v4, v6\}$

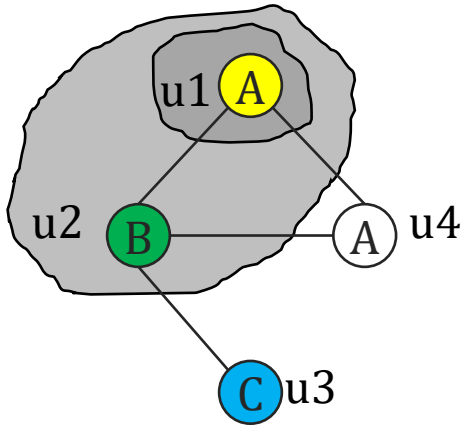
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

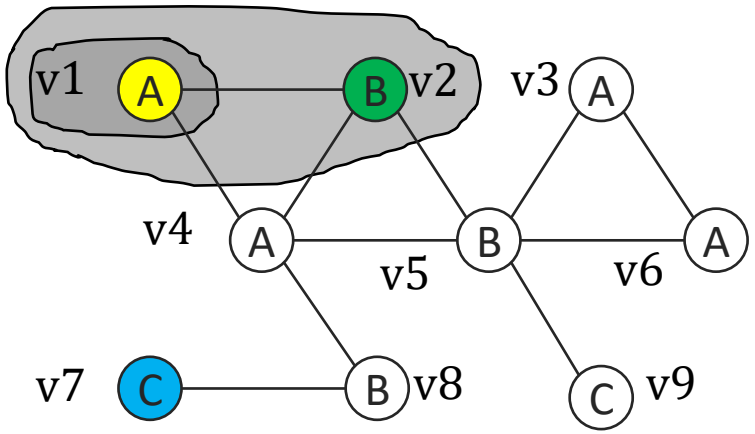
$u4.C = \{v1, v3, v4, v6\}$

$f_2 = \{(u1, v1), (u2, v2)\}$





(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

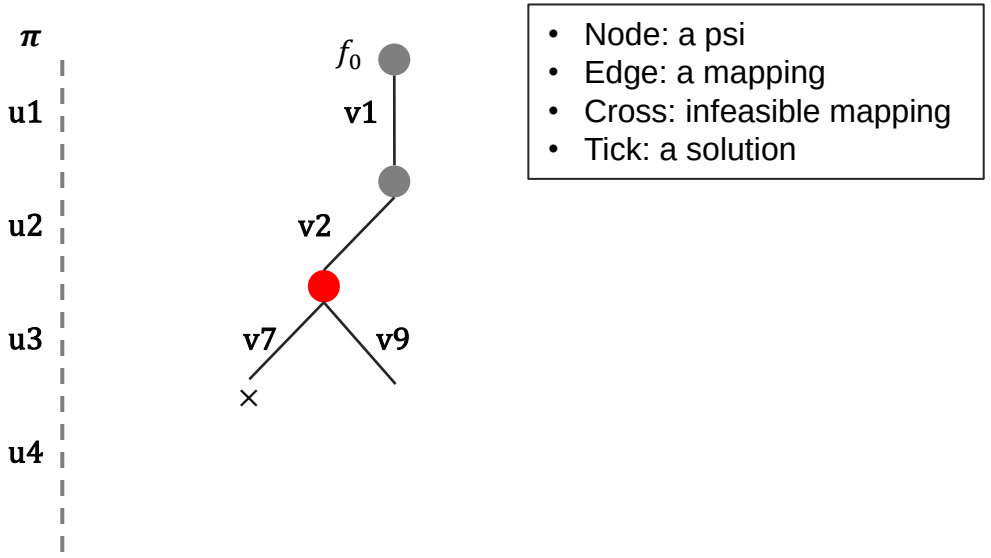
$u1.C = \{v1, v3, v4, v6\}$

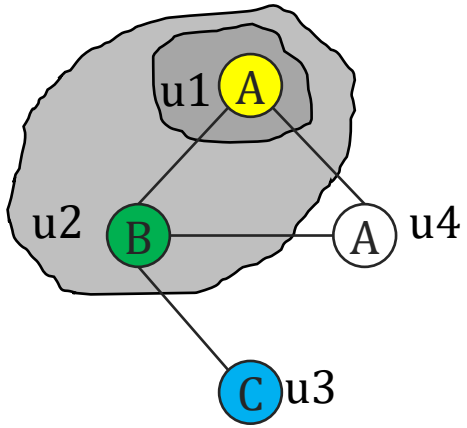
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

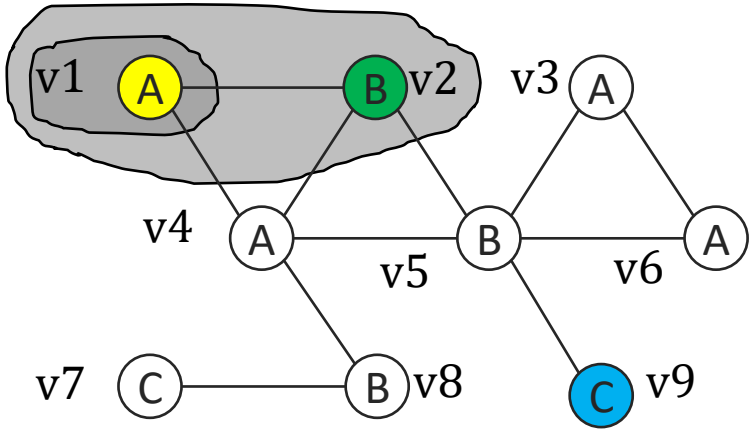
$u4.C = \{v1, v3, v4, v6\}$

$f_2 = \{(u1, v1), (u2, v2)\}$





(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

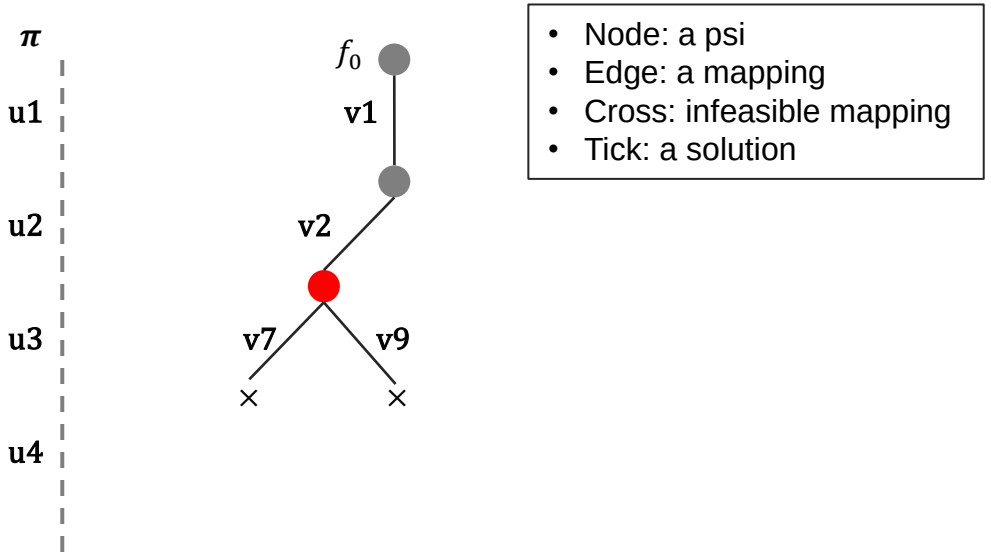
$u1.C = \{v1, v3, v4, v6\}$

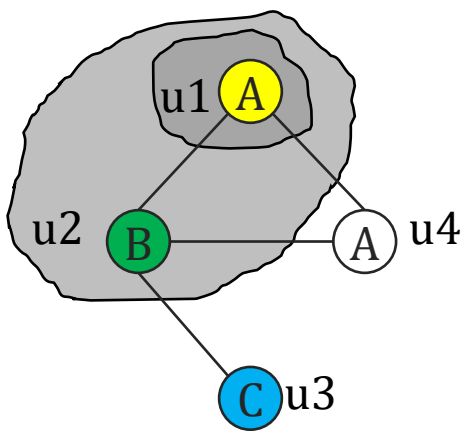
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

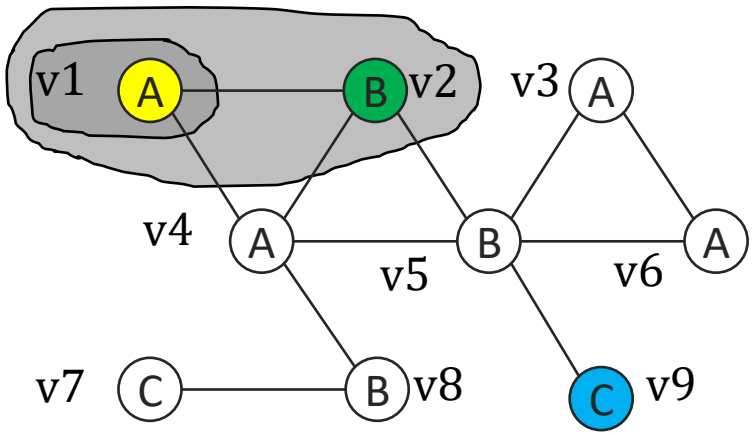
$u4.C = \{v1, v3, v4, v6\}$

$f_2 = \{(u1, v1), (u2, v2)\}$





(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

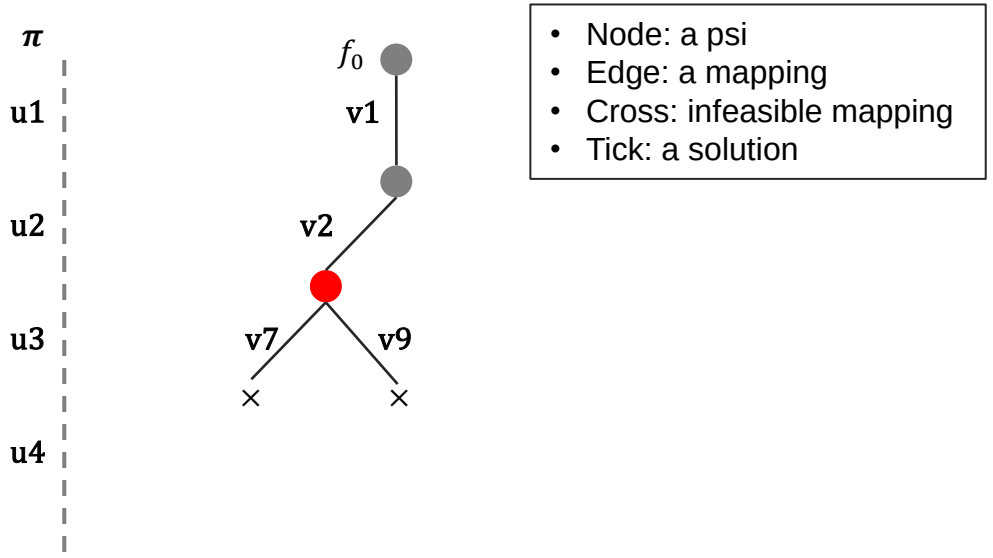
$u1.C = \{v1, v3, v4, v6\}$

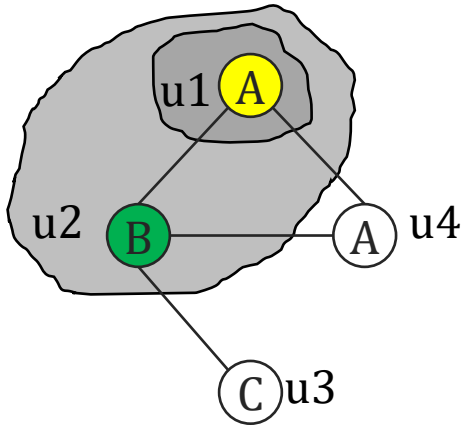
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

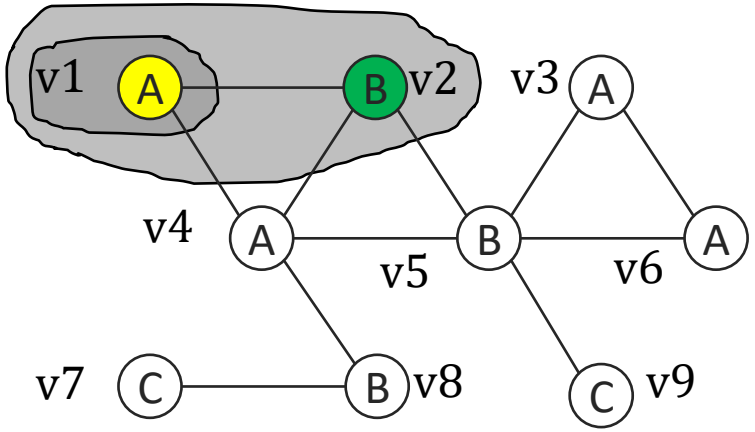
$u4.C = \{v1, v3, v4, v6\}$

$f_2 = \{(u1, v1), (u2, v2)\}$





(a). Query graph q



(b). Data graph G

Backtracking Enumeration Process

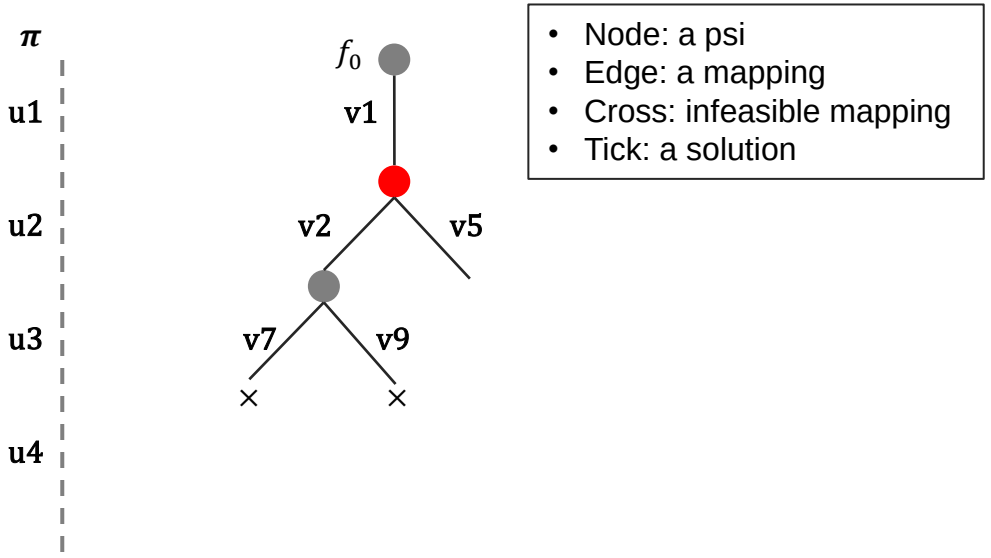
$u1.C = \{v1, v3, v4, v6\}$

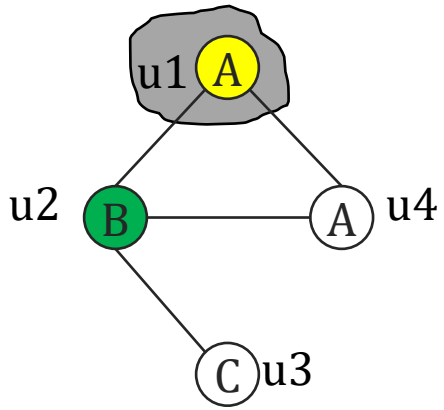
$u2.C = \{v2, v5\}$

$u3.C = \{v7, v9\}$

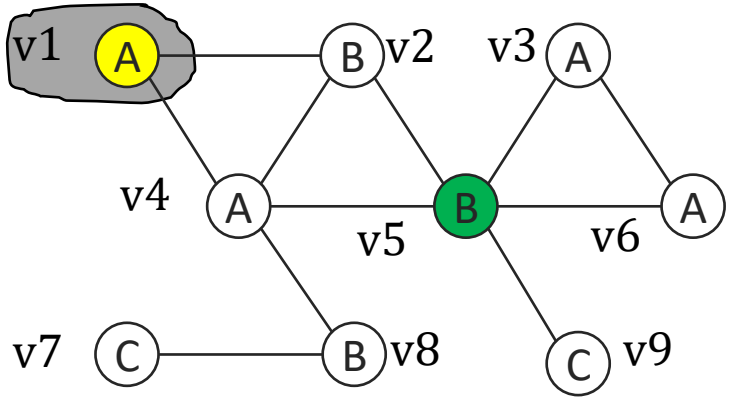
$u4.C = \{v1, v3, v4, v6\}$

$f_1 = \{(u1, v1)\}$



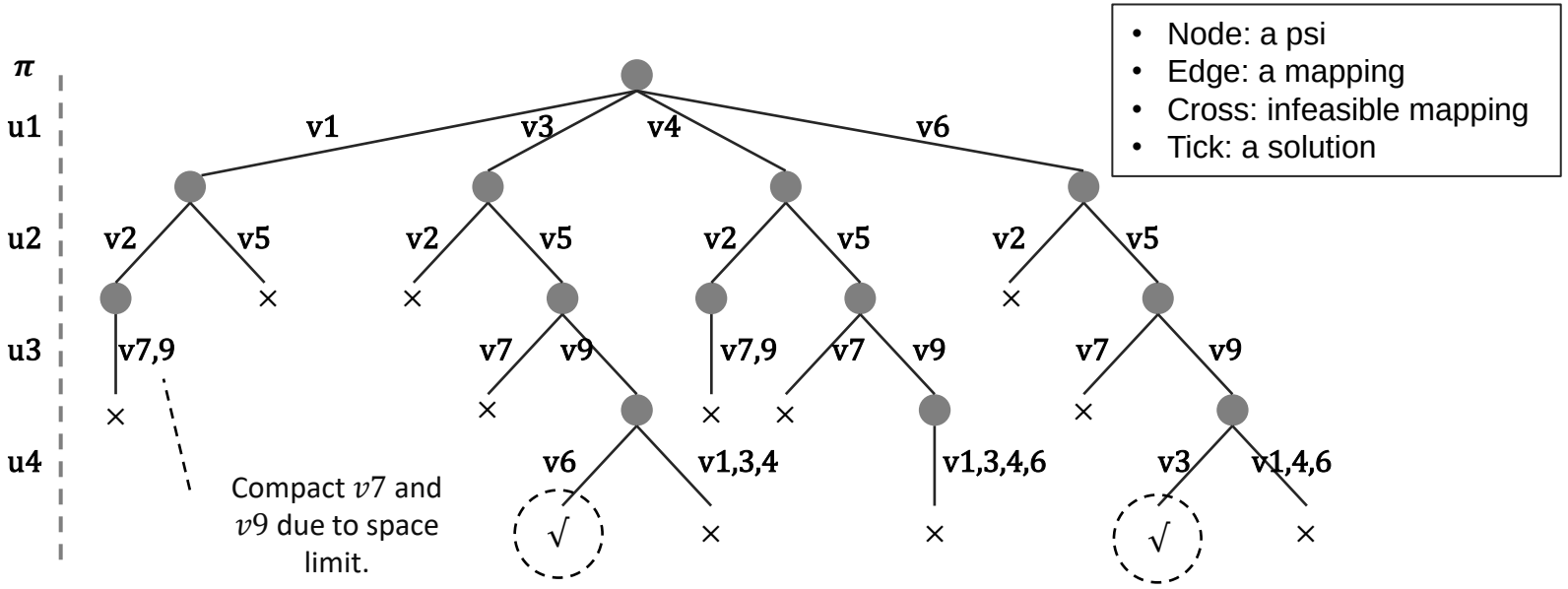


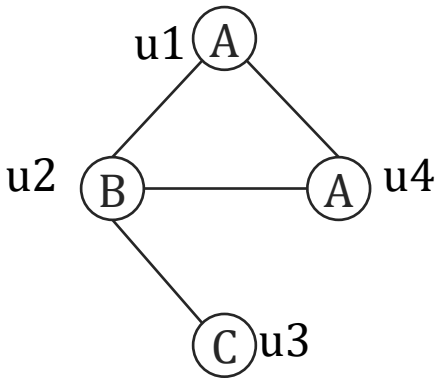
(a). Query graph q



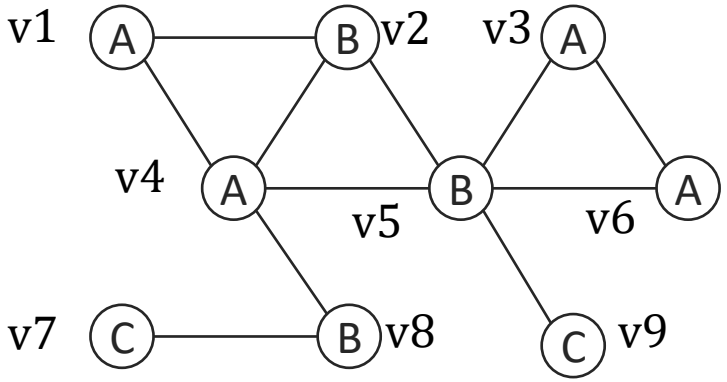
(b). Data graph G

Backtracking Enumeration Process





(a). Query graph q

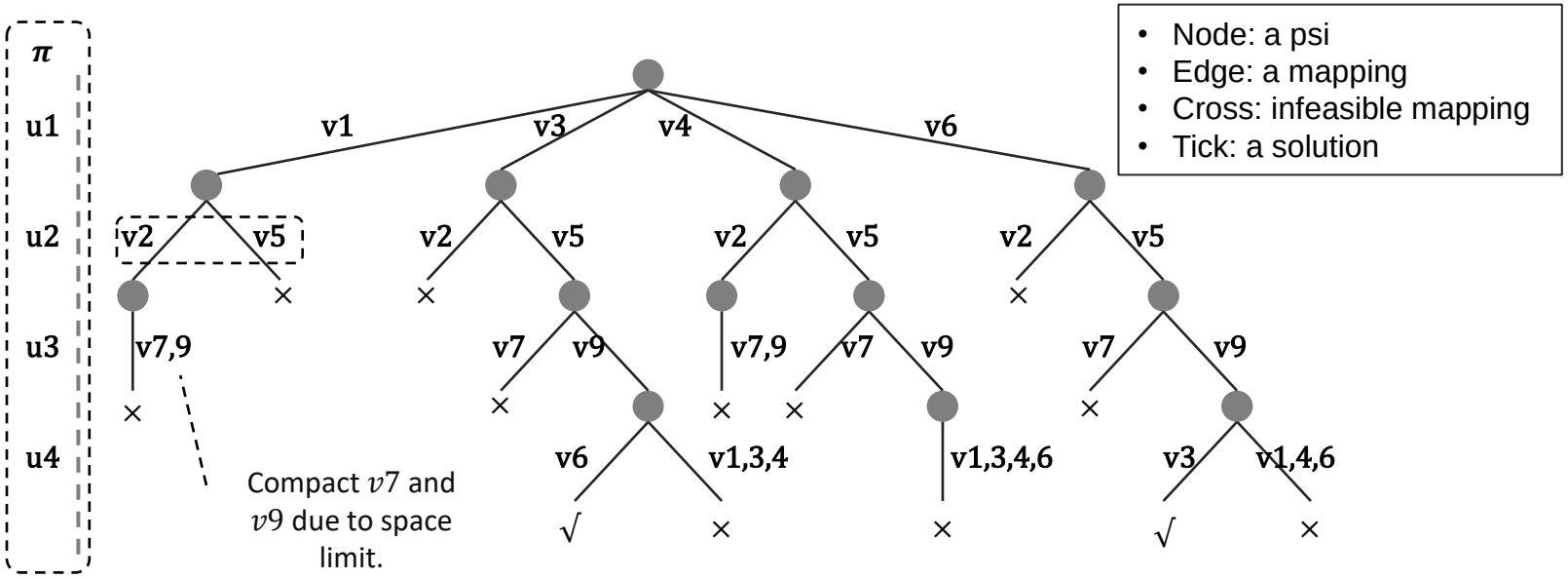


(b). Data graph G

Key Issues of Minimizing Search Space

- **Issue 1:**
  - Optimize the matching order to prune the invalid search paths at an early stage.

- **Issue 2:**
  - Decrease the search breadth of every *psi*.



Representative Algorithms

Communities	Methodologies	Algorithms
Database	Backtracking Search	QuickSI, GADDI, SPath, GraphQL, TurboIso, BoostIso, CFL, SGMatch, CECI, DP-iso, PGX, PSM, STwig
	Multi-way Join	EmptyHeaded, Graphflow, LogicBlox, PostgreSQL, MonetDB, Neo4j, GpSM
Artificial Intelligence	Backtracking Search	Ullmann, VF2, VF2++, VF3, LAD, Glasgow
Bioinformatics	Backtracking Search	RI, VF2+, Grapes

Representative Algorithms

Communities	Methodologies	Algorithms
Database	Backtracking Search	QuickSI, GADDI, SPath, GraphQL, TurboIso, BoostIso, CFL, SGMatch, CECI, DP-iso, PGX, PSM, STwig
	Multi-way Join	EmptyHeaded, Graphflow, LogicBlox, PostgreSQL, MonetDB, Neo4j, GpSM
Artificial Intelligence	Backtracking Search	Ullmann, VF2, VF2++, VF3, LAD, Glasgow
Bioinformatics	Backtracking Search	RI, VF2+, Grapes

Category of Backtracking-Based Algorithms

- Direct-Enumeration: Directly explore G to find all results.
  - Example algorithms: QuickSI, RI and VF2++.

Category of Backtracking-Based Algorithms

- Direct-Enumeration: Directly explore G to find all results.
  - Example algorithms: QuickSI, RI and VF2++.

- Indexing-Enumeration: Construct indexes on G and answer all queries with the assistance of indexes.
  - Example algorithms: GADDI and SGMatch.

Category of Backtracking-Based Algorithms

- Direct-Enumeration: Directly explore G to find all results.
  - Example algorithms: QuickSI, RI and VF2++.

- Indexing-Enumeration: Construct indexes on G and answer all queries with the assistance of indexes.
  - Example algorithms: GADDI and SGMatch.

- Preprocessing-Enumeration: Generate candidate vertex sets per query at runtime and evaluate the query based on candidate vertex sets.
  - Widely used in the latest algorithms proposed in the database community.
  - Example algorithms: GraphQL, TurboISO, CFL, DP-iso and CECI.

Observation

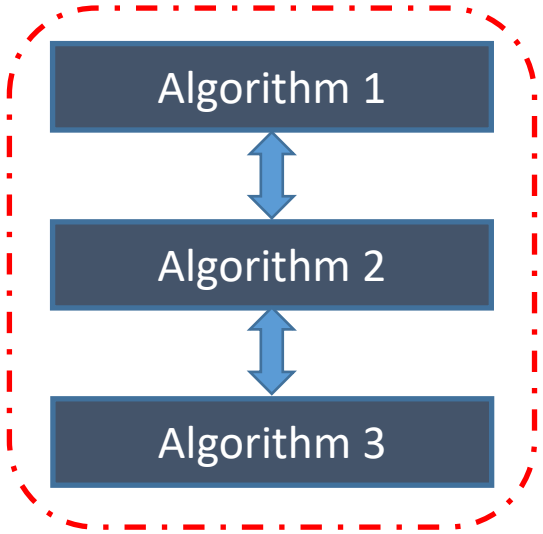
- Techniques in existing algorithms can be classified into several categories each of which have the same goal.
  - Example: Methods filtering candidates, methods optimizing the matching order.

Observation

- Techniques in existing algorithms can be classified into several categories each of which have the same goal.
  - Example: Methods filtering candidates, methods optimizing the matching order.
- The methods are closely related and all affect the evaluation performance.

Observation

- Techniques in existing algorithms can be classified into several categories each of which have the same goal.
  - Example: Methods filtering candidates, methods optimizing the matching order.
- The methods are closely related and all affect the evaluation performance.
- Previous studies regard each algorithm as a black box.
  - Hide effectiveness of individual techniques.



Our Work

- Study individual techniques in the algorithms within a common framework.
  - Compare and analyze individual techniques in existing algorithms.
  - Conduct extensive experiments to evaluate the effectiveness of the techniques.
  - Pinpoint techniques leading to the performance differences and make recommendation.

Our Work

- Study individual techniques in the algorithms within a common framework.
  - Compare and analyze individual techniques in existing algorithms.
  - Conduct extensive experiments to evaluate the effectiveness of the techniques.
  - Pinpoint techniques leading to the performance differences and make recommendation.

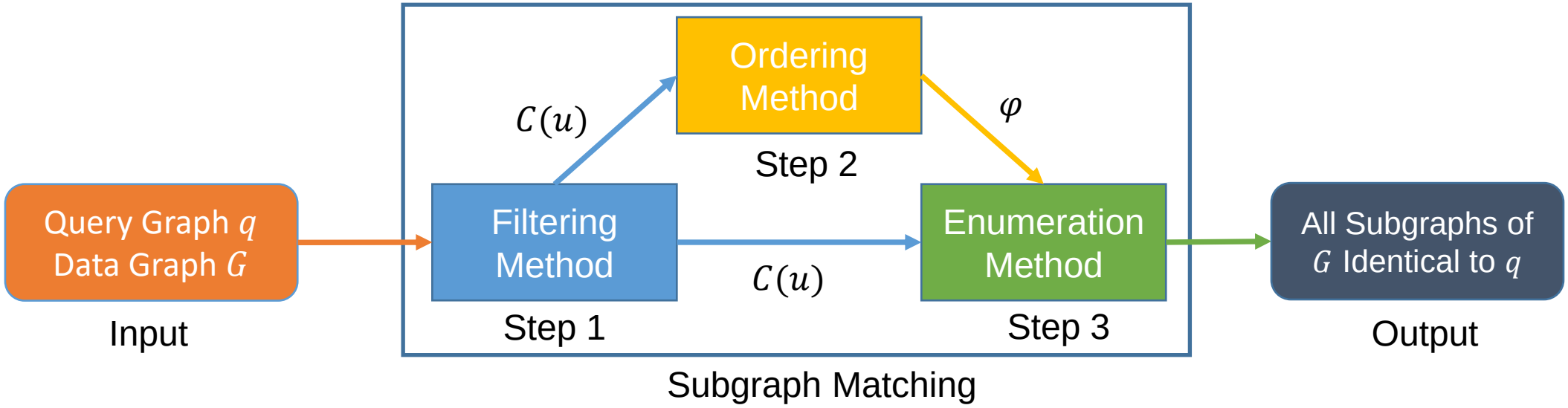
- Select seven algorithms from three different communities.
  - GraphQL [SIGMOD'08]
  - CFL [SIGMOD'16]
  - CECI [SIGMOD'19]
  - DP-iso [SIGMOD'19]

  } The preprocessing-enumeration algorithms

  - QuickSI [VLDB'08]
  - RI [BMC Bioinformatics'13]
  - VF2++ [Discrete Applied Mathematics'18]

  } The direct-enumeration algorithms

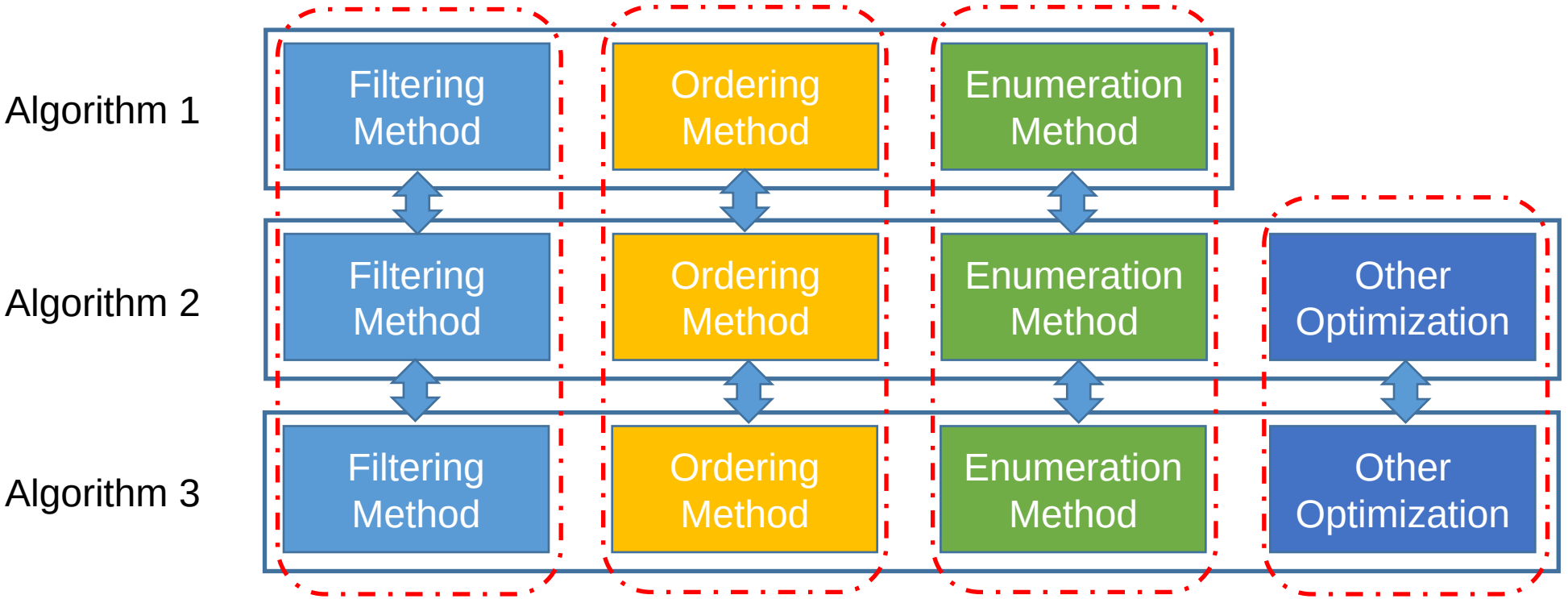
Common Framework



- Filtering Method: Given q and G, minimize candidate vertex sets $C(u)$ for each $u \in V(q)$.
  - $C(u)$: A set of data vertices $v \in V(G)$ that can be mapped to u.
- Ordering Method: Optimize the matching order φ based on the statistics of candidate vertex sets.
  - φ: A sequence of query vertices $V(q)$.
- Enumeration Method: Iteratively extend partial results M by mapping $u \in V(q)$ to $v \in C(u)$ along φ.
  - M: A dictionary storing mappings between query vertices to data vertices.

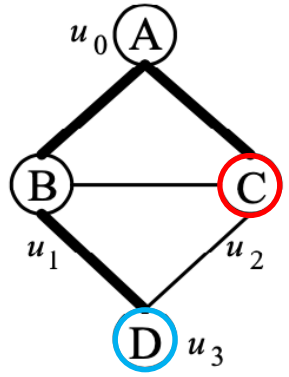
Principles of Our Study

- Study the performance of the algorithms from four aspects.
- When comparing one component, fix the others for fair comparison.

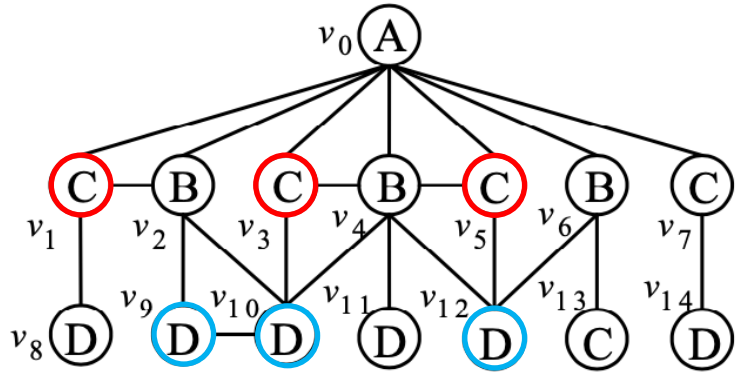


Filtering Method

- Basic Method: Filtering $C(u)$ based on the label $L(u)$ and degree $d(u)$ of u, i.e., $C(u) = \{v \in V(G) | L(v) = L(u) \wedge d(v) \geq d(u)\}$
  - Take u_2 and u_3 as examples: $C(u_2) = \{v_1, v_3, v_5\}, C(u_3) = \{v_9, v_{10}, v_{12}\}$



(a) Query graph q.

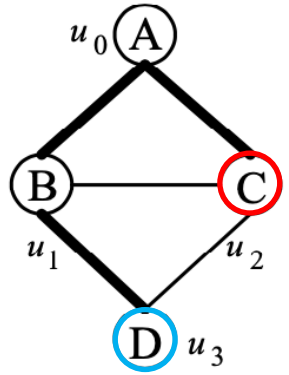


(b) Data graph G.

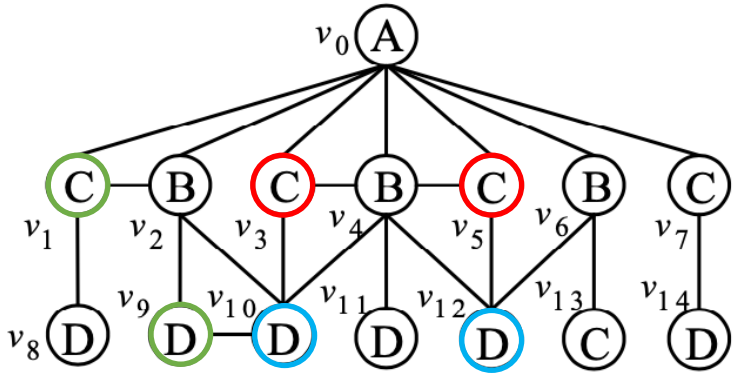
Filtering Method

- Filtering Rule: Given $v \in C(u)$, if there exists $u' \in N(u)$ such that $N(v) \cap C(u') = \emptyset$, then v can be removed from $C(u)$.

- Advanced Method: Filtering $C(u)$ with the rule along a sequence of $u \in V(q)$.
  - Example algorithms: GraphQL, CFL, CECI and DP-iso.
  - Major differences: The filtering sequence and the number of rounds repeated.



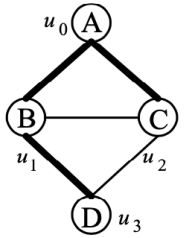
(a) Query graph q.



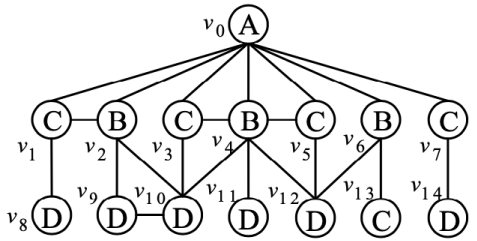
(b) Data graph G.

Filtering Method

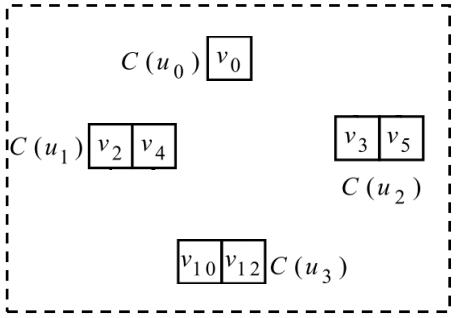
- Build an auxiliary data structure A to record edges between candidate vertex sets.
  - Serve the cardinality estimation in the ordering method.
  - Accelerate the subsequent enumeration method.



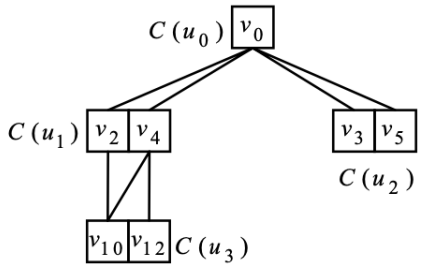
(a) Query graph q.



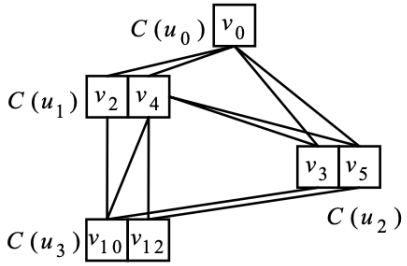
(b) Data graph G.



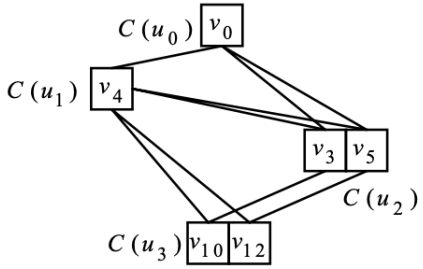
GraphQL



A of CFL



A of CECI



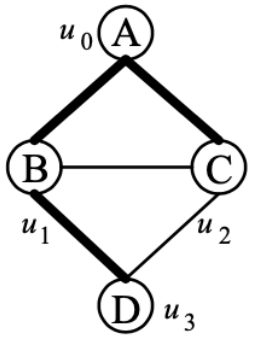
A of DP-iso

Ordering Method

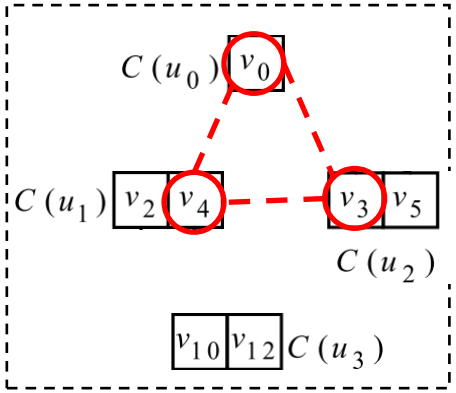
- Adopt the greedy method that (1) selects a start vertex; and (2) iteratively adds unselected query vertices to φ according to the cost estimation based on C and A.
  - The major difference is the cost function.
    - GraphQL: Select the vertex u with the minimum $|C(u)|$ at each step.
    - CFL/DP-iso: Select the path of q with the minimum number of embeddings in A at each step.

Enumeration Method

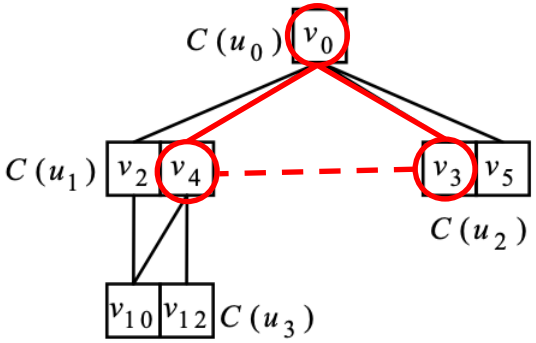
- Extend partial results by mapping $u \in V(q)$ to $v \in C(u)$ along φ with the assistance of A.
  - GraphQL: Probe G for all edge validation.
  - CFL: Probe G and A for the non-tree and tree edge validation, respectively.
  - DP-iso/CECI: Probe A for all edge validation.



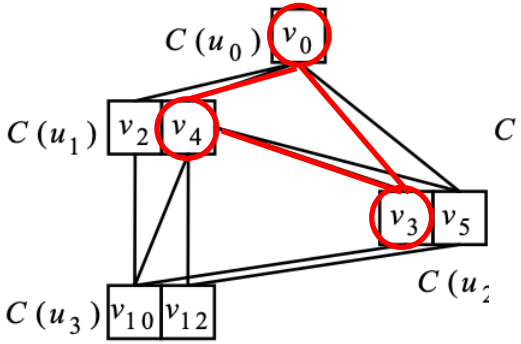
Query graph q



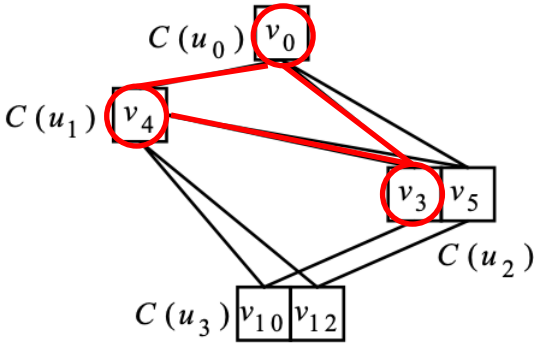
GraphQL



A of CFL



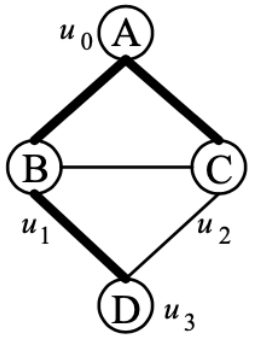
A of CECI



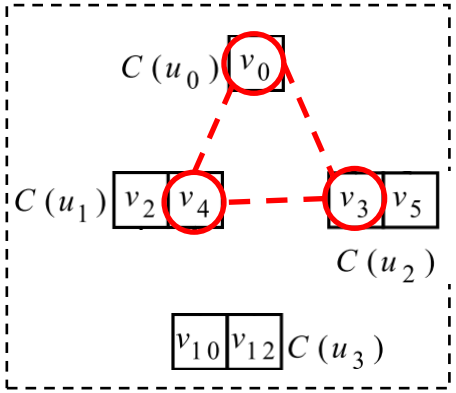
A of DP-iso

Enumeration Method

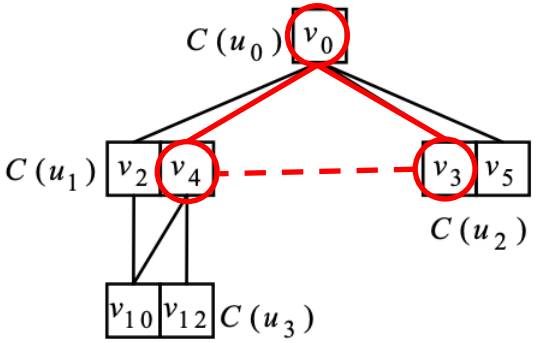
- Extend partial results by mapping $u \in V(q)$ to $v \in C(u)$ along φ with the assistance of A.
  - GraphQL: Probe G for all edge validation.
  - CFL: Probe G and A for the non-tree and tree edge validation, respectively.
  - DP-iso/CECI: Probe A for all edge validation.



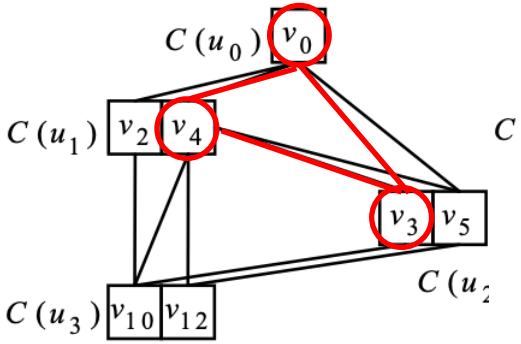
Query graph q



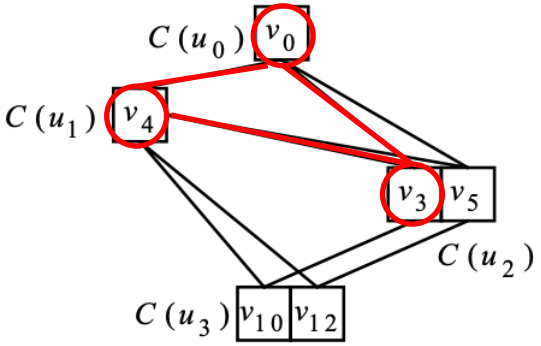
GraphQL



A of CFL



A of CECI



A of DP-iso

Recommendation: Use the DP-iso/CECI-style auxiliary data structure and enumeration method.

Optimization Method

- Failing set pruning: During the enumeration, utilize the information obtained from the explored part of the search tree to prune invalid partial results.
  - Proposed by DP-iso.
  - Other algorithms can adopt the optimization as well.

Experimental Setup

- All algorithms are implemented in C++ and run on a machine with 2.3GHz CPUs and 128GB RAM.
- Real-world data graphs:

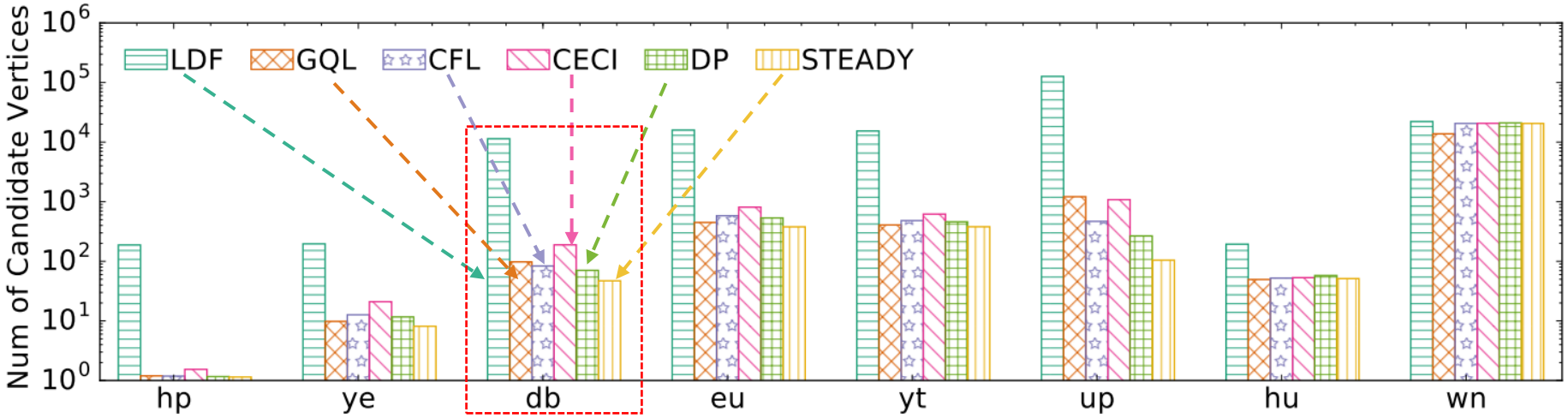
Category	Dataset	Name	$\|V\|$	$\|E\|$	$\|\Sigma\|$	d
Biology	Yeast	*ye*	3,112	12,519	71	8.0
	Human	*hu*	4,674	86,282	44	36.9
	HPRD	*hp*	9,460	34,998	307	7.4
Lexical	WordNet	*wn*	76,853	120,399	5	3.1
Citation	US Patents	*up*	3,774,768	16,518,947	20	8.8
Social	Youtube	*yt*	1,134,890	2,987,624	25	5.3
	DBLP	*db*	317,080	1,049,866	15	6.6
Web	eu2005	*eu*	862,664	16,138,468	40	37.4

- Query sets:
  - Query graphs are randomly extracted from the data graph.
  - Each query set contains 200 connected graphs with the same number of vertices.
  - Q_{iD} and Q_{is} denote dense ($d(q) \geq 3$) and sparse ($d(q) < 3$) query sets containing graphs with i vertices.
  - Each data graph has 1800 queries in total.

Dataset	Query Set	Default
Yeast, HPRD, US Patents, Youtube, DBLP, eu2005	Q_4, Q_{8D}, Q_{16D}, Q_{24D}, Q_{32D}, Q_{8S}, Q_{16S}, Q_{24S}, Q_{32S}	Q_{32D}, Q_{32S}
Human, WordNet	Q_4, Q_{8D}, Q_{12D}, Q_{16D}, Q_{20D}, Q_{8S}, Q_{12S}, Q_{16S}, Q_{20S}	Q_{20D}, Q_{20S}

Effectiveness of Filtering Methods

- **Metrics:** Num of Candidate Vertices = $\frac{1}{|Q|}\sum_{q\in Q}\frac{1}{|V(q)|}\sum_{u\in V(q)}|C(u)|$.
- **Finding:** GraphQL, CFL and DP-iso are competitive with each other, and they are close to STEADY.

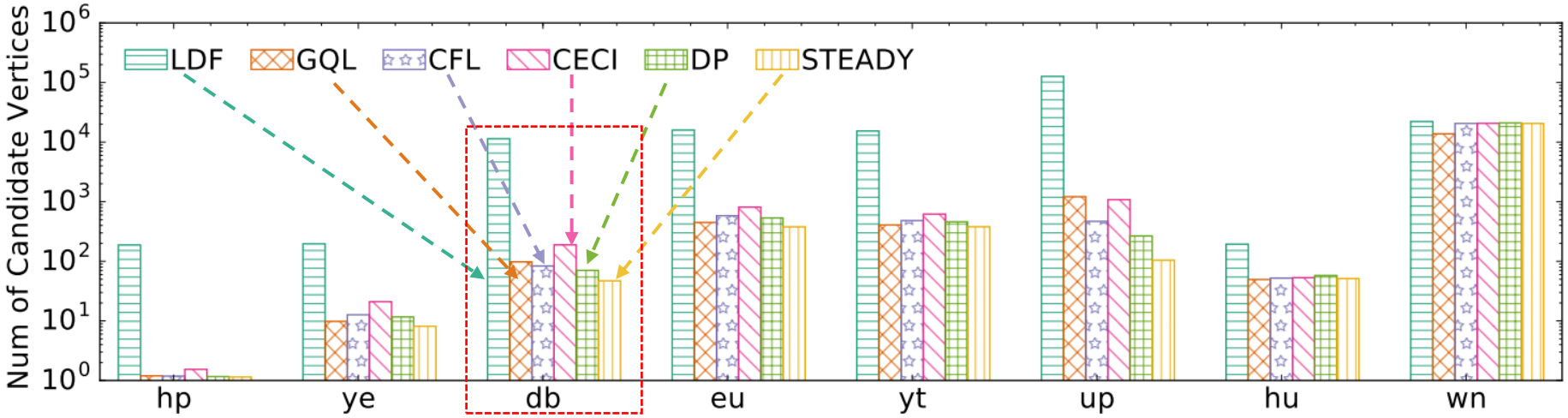


Varying datasets on dense query sets.

LDF: Label and degree filter.
GQL: Filtering of GraphQL.
CFL: Filtering of CFL.
CECI: Filtering of CECI.
DP: Filtering of DP-iso.
STEADY: Given $v \in C(u)$, it satisfies that $\forall u' \in N(u), N(v) \cap C(u') \neq \emptyset$.

Effectiveness of Filtering Methods

- **Metrics:** Num of Candidate Vertices = $\frac{1}{|Q|}\sum_{q\in Q}\frac{1}{|V(q)|}\sum_{u\in V(q)}|C(u)|$.
- **Finding:** GraphQL, CFL and DP-iso are competitive with each other, and they are close to STEADY.
- **Recommendation:** Adopt the filtering method of GraphQL/CFL/DP-iso to prune candidate vertex sets.

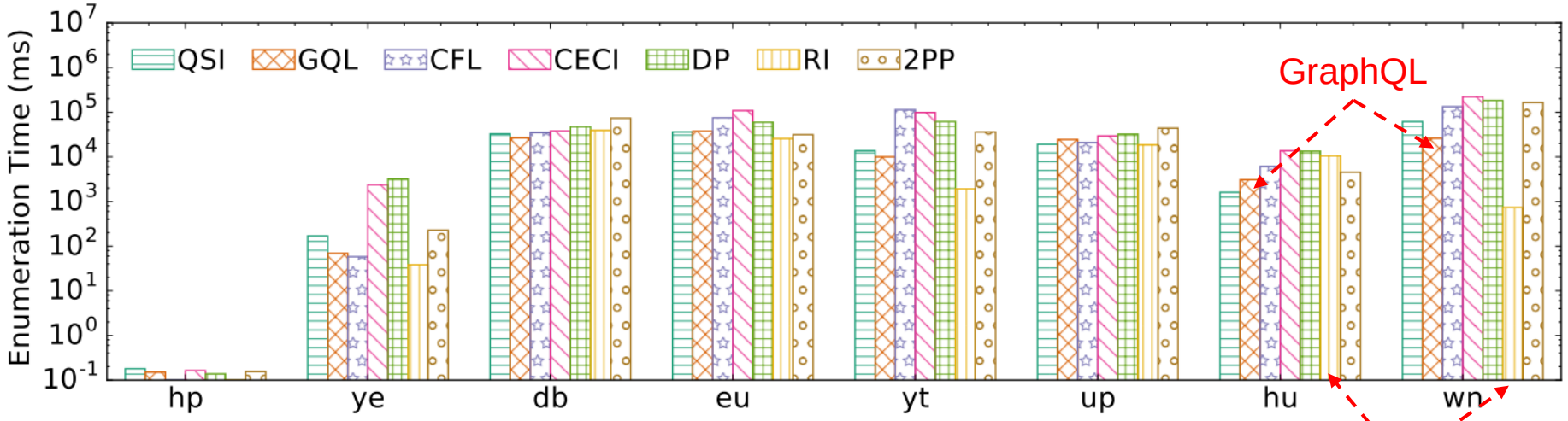


Varying datasets on dense query sets.

LDF: Label and degree filter.
GQL: Filtering of GraphQL.
CFL: Filtering of CFL.
CECI: Filtering of CECI.
DP: Filtering of DP-iso.
STEADY: Given $v \in C(u)$, it satisfies that $\forall u' \in N(u), N(v) \cap C(u') \neq \emptyset$.

Effectiveness of Ordering Methods

- **Setup:** Use the DP-iso/CECI-style auxiliary data structure and enumeration method and adopt candidate vertex sets of GraphQL.
- **Metrics:** Enumeration Time = $\frac{1}{|Q|}\sum_{q \in Q} T(A, q)$.
- **Finding:** GraphQL and RI are usually the most effective among competing methods.

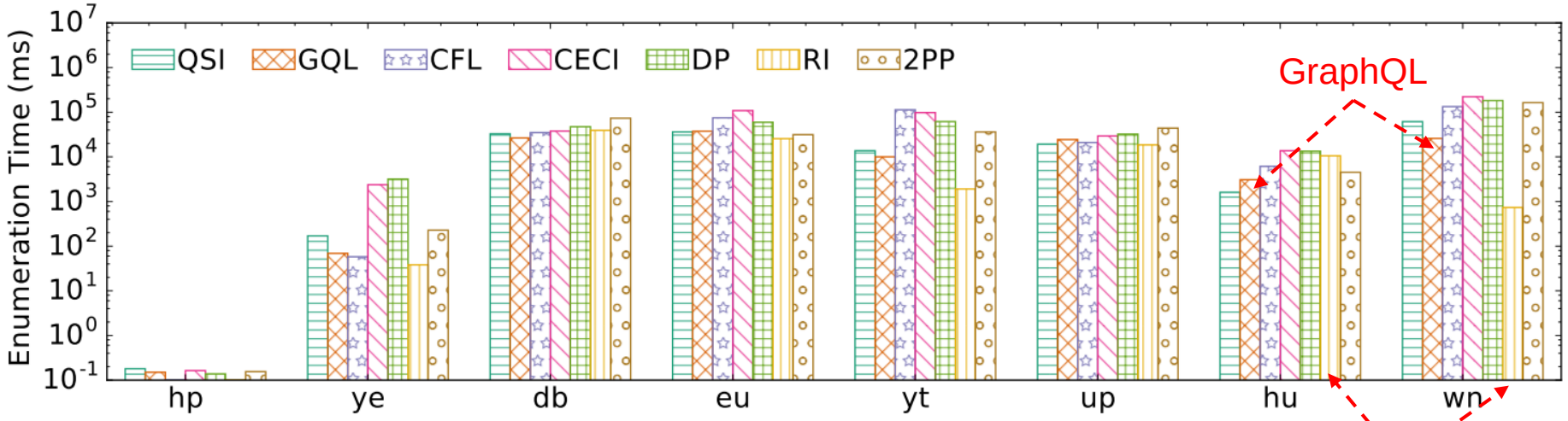


Varying datasets on dense query sets.

QSI: Ordering of QuickSI.
GQL: Ordering of GraphQL.
CFL: Ordering of CFL.
CECI: Ordering of CECI.
DP: Ordering of DP-iso.
RI: Ordering of RI.
2PP: Ordering of VF2++.

Effectiveness of Ordering Methods

- **Setup:** Use the DP-iso/CECI-style auxiliary data structure and enumeration method and adopt candidate vertex sets of GraphQL.
- **Metrics:** Enumeration Time = $\frac{1}{|Q|}\sum_{q \in Q} T(A, q)$.
- **Finding:** GraphQL and RI are usually the most effective among competing methods.
- **Recommendation:** Adopt GraphQL and RI on dense and sparse data graphs respectively.



Varying datasets on dense query sets.

QSI: Ordering of QuickSI.
GQL: Ordering of GraphQL.
CFL: Ordering of CFL.
CECI: Ordering of CECI.
DP: Ordering of DP-iso.
RI: Ordering of RI.
2PP: Ordering of VF2++.

Effectiveness of Failing Set Pruning

- **Setup:** Continue with the experiments on ordering methods and enable the failing set pruning.
- **Metrics:** Count the number of unsolved queries within 5 minutes.
- **Finding:** (1) Failing set pruning can significantly reduce the number of unsolved queries; and (2) all competing algorithms can generate ineffective matching orders.

Algorithm	*yt*		*up*		*hu*		*wn*	
	wo/fs	w/fs	wo/fs	w/fs	wo/fs	w/fs	wo/fs	w/fs
QSI	14	0	26	9	12	6	69	20
GQL	11	0	23	8	10	2	17	3
CFL	95	6	24	12	16	8	191	139
CECI	161	5	39	7	40	9	547	351
DP	70	6	40	13	30	20	307	221
RI	2	0	18	8	23	9	0	0
2PP	49	3	49	17	12	7	270	220
Fail-All	**0**	**0**	**7**	**3**	**2**	**0**	**0**	**0**

Number of unsolved queries among 1800 queries for each data graph.

wo/fs: Enumeration without the failing set pruning.
w/fs: Enumeration with the failing set pruning.
Fail-ALL: Number of queries that no competing algorithms can complete within 5 minutes.

Effectiveness of Failing Set Pruning

- **Setup:** Continue with the experiments on ordering methods and enable the failing set pruning.
- **Metrics:** Count the number of unsolved queries within 5 minutes.
- **Finding:** (1) Failing set pruning can significantly reduce the number of unsolved queries; and (2) all competing algorithms can generate ineffective matching orders.
- **Recommendation:** Enable failing set pruning for large queries.

Algorithm	*yt*		*up*		*hu*		*wn*	
	wo/fs	w/fs	wo/fs	w/fs	wo/fs	w/fs	wo/fs	w/fs
QSI	14	0	26	9	12	6	69	20
GQL	11	0	23	8	10	2	17	3
CFL	95	6	24	12	16	8	191	139
CECI	161	5	39	7	40	9	547	351
DP	70	6	40	13	30	20	307	221
RI	2	0	18	8	23	9	0	0
2PP	49	3	49	17	12	7	270	220
Fail-All	**0**	**0**	**7**	**3**	**2**	**0**	**0**	**0**

Number of unsolved queries among 1800 queries for each data graph.

wo/fs: Enumeration without the failing set pruning.
w/fs: Enumeration with the failing set pruning.
Fail-ALL: Number of queries that no competing algorithms can complete within 5 minutes.

Summary

- Compare and analyze individual techniques in seven algorithms from three communities within a common framework.

- Conduct extensive experiments to evaluate the effectiveness of each kind of methods respectively.

- Report our new findings and make the recommendation through experiments and analysis.

Checkout source code and datasets at: github.com/RapidsAtHKUST/SubgraphMatching

Outline

- Benchmark
  - Background
  - In-Memory Subgraph Matching: An In-Depth Study. SIGMOD 2020.

- Algorithms
  - RapidMatch: A Holistic Approach to Subgraph Query Processing. VLDB 2021.
  - PathEnum: Towards Real-Time Hop Constraint s-t Path Enumeration. SIGMOD 2021.

- Parallelization
  - LIGHT: Parallelizing Subgraph Query Processing. ICPADS 2018 & ICDE 2019.
  - ThunderRW: An In-Memory Graph Random Walk Engine. VLDB 2021.

RapidMatch: A Holistic Approach to Subgraph Query Processing

Shixuan Sun[1], Xibo Sun[2], Yulin Che[2], Qiong Luo[2], Bingsheng He[1]

[1]*National University of Singapore*

[2]*Hong Kong University of Science and Technology*

Two Trends of Methods on the Same Problem

Exploration-based Methods

CFL [SIGMOD'16], DP-iso [SIGMOD'19]

Native methods specifically designed for subgraph query processing.

Methodology

Join-based Methods

EmptyHeaded [SIGMOD'16], GraphFlow [VLDB'19]

Evaluating subgraph queries with worst-case optimal join (WCOJ).

Two Trends of Methods on the Same Problem

	Exploration-based Methods CFL [SIGMOD'16], DP-iso [SIGMOD'19]		**Join-based Methods** EmptyHeaded [SIGMOD'16], GraphFlow [VLDB'19]
Methodology	Native methods specifically designed for subgraph query processing.		Evaluating subgraph queries with worst-case optimal join (WCOJ).
Workload	Q with tens of vertices on G having thousands to millions of vertices.		Q with a few vertices (<10) on G having up to hundreds of millions of vertices.

Two Trends of Methods on the Same Problem

	Exploration-based Methods CFL [SIGMOD'16], DP-iso [SIGMOD'19]		**Join-based Methods** EmptyHeaded [SIGMOD'16], GraphFlow [VLDB'19]
	Native methods specifically designed for subgraph query processing.	**Methodology**	Evaluating subgraph queries with worst-case optimal join (WCOJ).
	Q with tens of vertices on G having thousands to millions of vertices.	**Workload**	Q with a few vertices (<10) on G having up to hundreds of millions of vertices.
	Optimizing query plans with greedy methods based on cardinality estimation.	**Query Plan Optimization**	Finding the optimal query plan based on cardinality estimation in a plan space.

Two Trends of Methods on the Same Problem

	Exploration-based Methods CFL [SIGMOD'16], DP-iso [SIGMOD'19]	**Join-based Methods** EmptyHeaded [SIGMOD'16], GraphFlow [VLDB'19]
Methodology	Native methods specifically designed for subgraph query processing.	Evaluating subgraph queries with worst-case optimal join (WCOJ).
Workload	Q with tens of vertices on G having thousands to millions of vertices.	Q with a few vertices (<10) on G having up to hundreds of millions of vertices.
Query Plan Optimization	Optimizing query plans with greedy methods based on cardinality estimation.	Finding the optimal query plan based on cardinality estimation in a plan space.
Input Filtering	Applying advanced filtering methods to reduce the input graph size.	Simply utilizing labels to pruning the input graph.

Problems Studied in Our Work

- **Q1.** Is one kind of methods inherently better than the other?
- **A1.** No, the complexity of result enumeration in state-of-the-art exploration-based methods can match that of WCOJ.

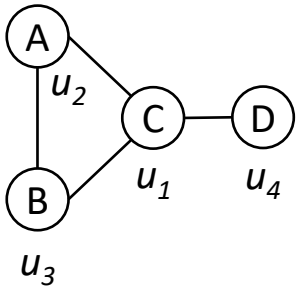
Problems Studied in Our Work

- **Q1.** Is one kind of methods inherently better than the other?
- **A1.** No, the complexity of result enumeration in state-of-the-art exploration-based methods can match that of WCOJ.

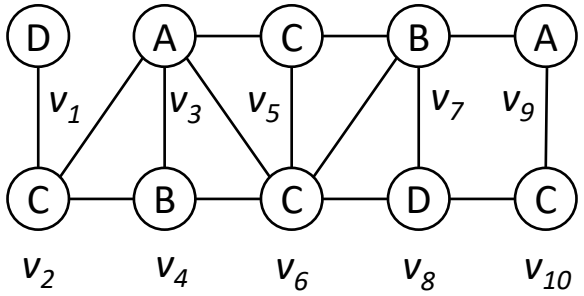
- **Q2:** How to design an approach to handle various workloads efficiently?

Evaluating Subgraph Query with Join

Subgraph Query



Query Graph Q.



Data Graph G.

Multi-way Join

$$Q := R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_2, u_3) \bowtie R(u_1, u_4)$$

$R(u_1, u_3)$

u_1	u_3
v_2	v_4
v_5	v_7
v_6	v_4
v_6	v_7

$R(u_1, u_4)$

u_1	u_4
v_2	v_1
v_6	v_8
v_{10}	v_8

$R(u_2, u_3)$

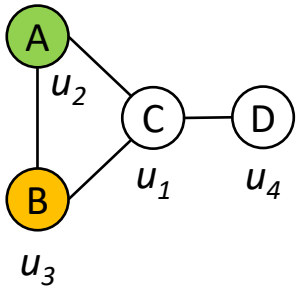
u_2	u_3
v_3	v_4
v_9	v_7
v_{10}	v_8

$R(u_1, u_2)$

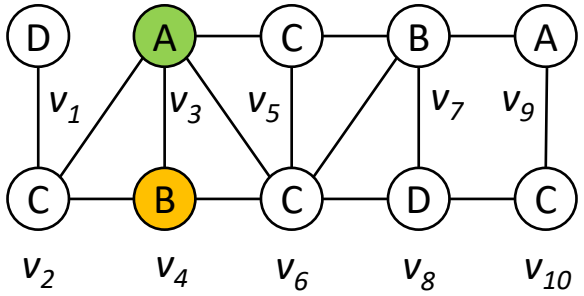
u_1	u_2
v_2	v_3
v_5	v_3
v_6	v_3
v_{10}	v_9

Evaluating Subgraph Query with Join

Input



Query Graph Q.



Data Graph G.

Relation Generation

$Q := R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_2, u_3) \bowtie R(u_1, u_4)$

$R(u_1, u_3)$

u_1	u_3
v_2	v_4
v_5	v_7
v_6	v_4
v_6	v_7

$R(u_1, u_4)$

u_1	u_4
v_2	v_1
v_6	v_8
v_{10}	v_8

$R(u_2, u_3)$

u_2	u_3
v_3	v_4
v_9	v_7
v_{10}	v_8

$R(u_1, u_2)$

u_1	u_2
v_2	v_3
v_5	v_3
v_6	v_3
v_{10}	v_9

Result Enumeration

Matching Order φ

u_2
u_3
u_1
u_4

Candidate Data Vertex

$\{v_3, v_9\}$

$R(u_2: v_3, u_3)$

$\downarrow$

$\{v_4\}$

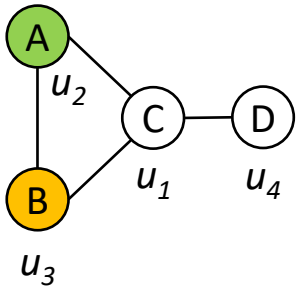
$\{?\}$

$\{?\}$

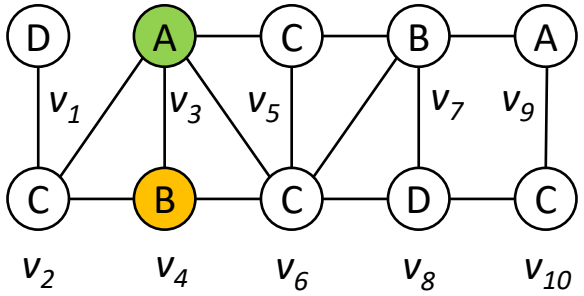
Notation:
$R(u: v, u')$: The neighbors of v in $R(u, u')$.

Evaluating Subgraph Query with Join

Input



Query Graph Q.



Data Graph G.

Relation Generation

$Q := R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_2, u_3) \bowtie R(u_1, u_4)$

$R(u_1, u_3)$

u_1	u_3
v_2	v_4
v_5	v_7
v_6	v_4
v_6	v_7

$R(u_1, u_4)$

u_1	u_4
v_2	v_1
v_6	v_8
v_{10}	v_8

$R(u_2, u_3)$

u_2	u_3
v_3	v_4
v_9	v_7
v_{10}	v_8

$R(u_1, u_2)$

u_1	u_2
v_2	v_3
v_5	v_3
v_6	v_3
v_{10}	v_9

Result Enumeration

Matching Order φ

u_2 — u_3 — u_1 — u_4

Candidate Data Vertex

$\{v_3, v_9\}$

$R(u_2: v_3, u_3)$

$\{v_4\}$

$R(u_2: v_3, u_1) \cap R(u_3: v_4, u_1)$

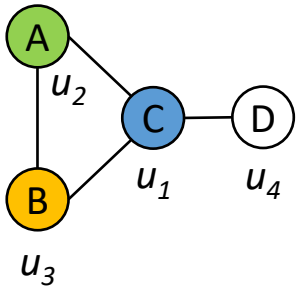
$\{v_2, v_6\}$

$\{?\}$

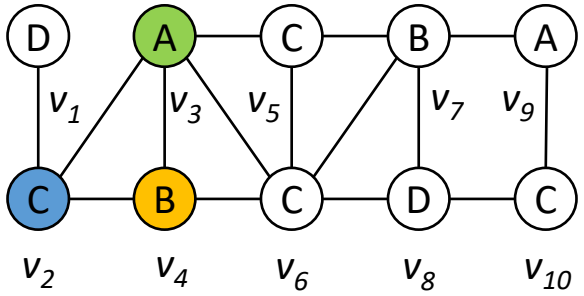
Notation:
$R(u: v, u')$: The neighbors of v in $R(u, u')$.

Evaluating Subgraph Query with Join

Input



Query Graph Q.



Data Graph G.

Relation Generation

$Q := R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_2, u_3) \bowtie R(u_1, u_4)$

$R(u_1, u_3)$

u_1	u_3
v_2	v_4
v_5	v_7
v_6	v_4
v_6	v_7

$R(u_1, u_4)$

u_1	u_4
v_2	v_1
v_6	v_8
v_{10}	v_8

$R(u_2, u_3)$

u_2	u_3
v_3	v_4
v_9	v_7
v_{10}	v_8

$R(u_1, u_2)$

u_1	u_2
v_2	v_3
v_5	v_3
v_6	v_3
v_{10}	v_9

Result Enumeration

Matching Order φ

u_2
u_3
u_1
u_4

Candidate Data Vertex

$\{v_3, v_9\}$

$R(u_2: v_3, u_3)$

$\{v_4\}$

$R(u_2: v_3, u_1) \bigcap R(u_3: v_4, u_1)$

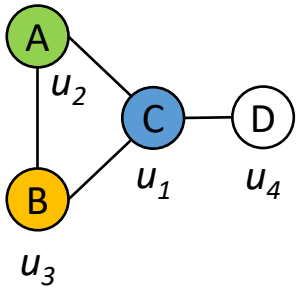
$\{v_2, v_6\}$

$\{?\}$

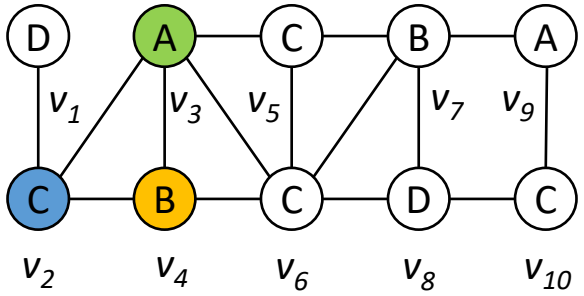
Notation:
$R(u: v, u')$: The neighbors of v in $R(u, u')$.

Evaluating Subgraph Query with Join

Input



Query Graph Q.



Data Graph G.

Relation Generation

$Q := R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_2, u_3) \bowtie R(u_1, u_4)$

$R(u_1, u_3)$

u_1	u_3
v_2	v_4
v_5	v_7
v_6	v_4
v_6	v_7

$R(u_1, u_4)$

u_1	u_4
v_2	v_1
v_6	v_8
v_{10}	v_8

$R(u_2, u_3)$

u_2	u_3
v_3	v_4
v_9	v_7
v_{10}	v_8

$R(u_1, u_2)$

u_1	u_2
v_2	v_3
v_5	v_3
v_6	v_3
v_{10}	v_9

Result Enumeration

Matching Order φ

u_2, u_3, u_1, u_4

Candidate Data Vertex

$\{v_3, v_9\}$

$R(u_2: v_3, u_3)$

$\{v_4\}$

$R(u_2: v_3, u_1) \cap R(u_3: v_4, u_1)$

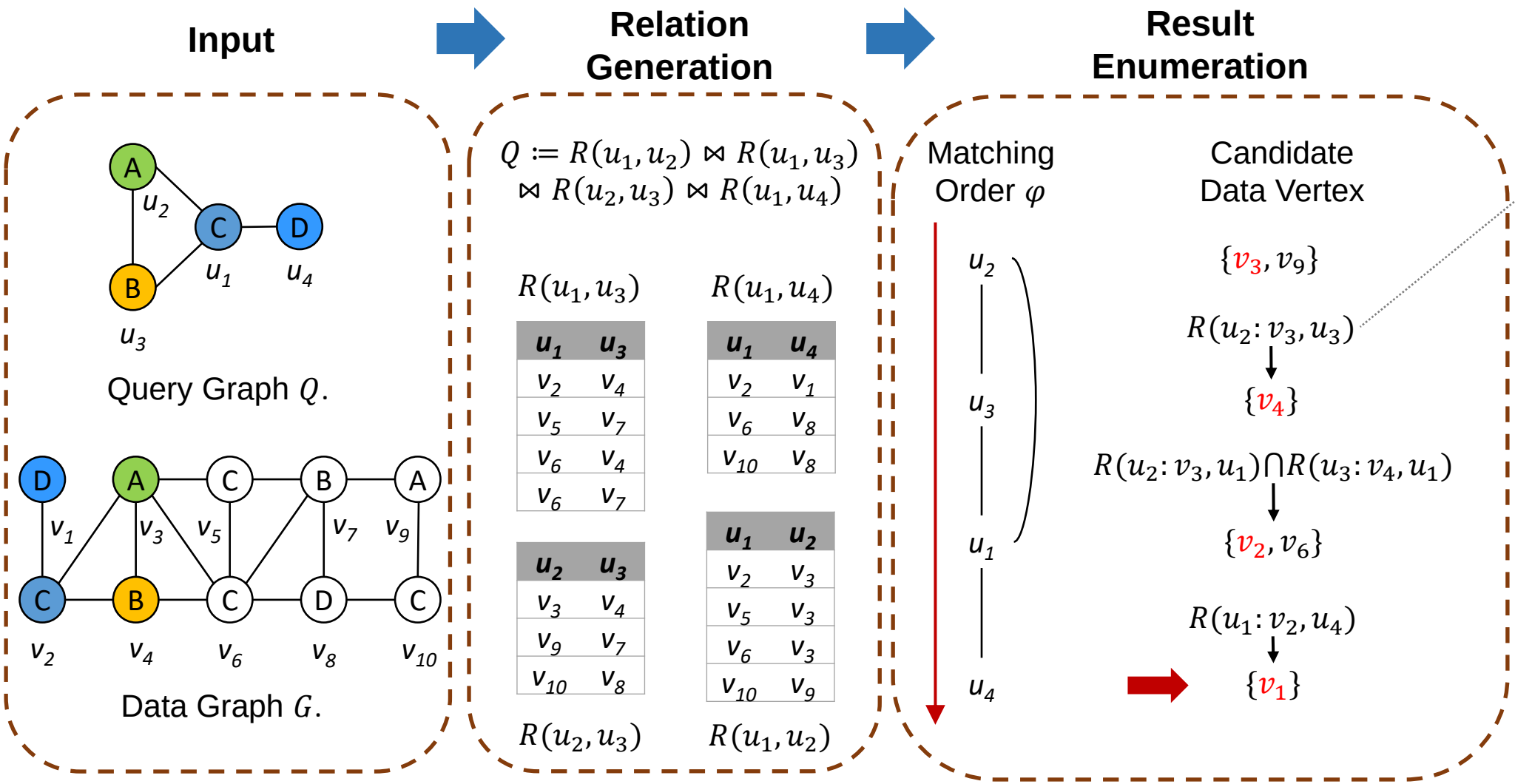
$\{v_2, v_6\}$

$R(u_1: v_2, u_4)$

$\{v_1\}$

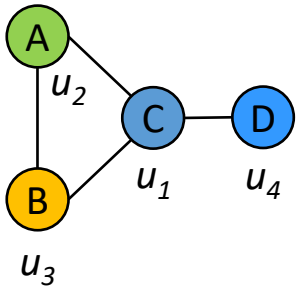
Notation:
$R(u: v, u')$: The neighbors of v in $R(u, u')$.

Evaluating Subgraph Query with Join

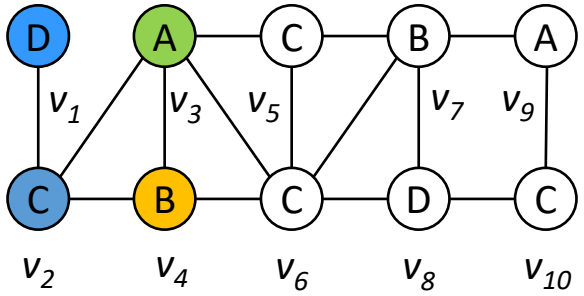


Evaluating Subgraph Query with Join

Input → **Relation Generation** → **Result Enumeration** → **Output**



Query Graph Q.



Data Graph G.

$Q := R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_2, u_3) \bowtie R(u_1, u_4)$

$R(u_1, u_3)$

u_1	u_3
v_2	v_4
v_5	v_7
v_6	v_4
v_6	v_7

$R(u_1, u_4)$

u_1	u_4
v_2	v_1
v_6	v_8
v_{10}	v_8

$R(u_2, u_3)$

u_2	u_3
v_3	v_4
v_9	v_7
v_{10}	v_8

$R(u_1, u_2)$

u_1	u_2
v_2	v_3
v_5	v_3
v_6	v_3
v_{10}	v_9

Matching Order φ

u_2, u_3, u_1, u_4

Candidate Data Vertex

$\{v_3, v_9\}$

$R(u_2 : v_3, u_3)$ → $\{v_4\}$

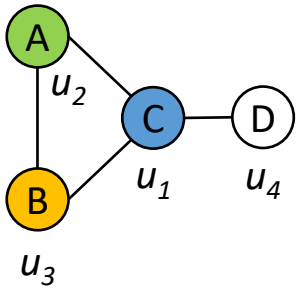
$R(u_2 : v_3, u_1) \cap R(u_3 : v_4, u_1)$ → $\{v_2, v_6\}$

$R(u_1 : v_2, u_4)$ → $\{v_1\}$

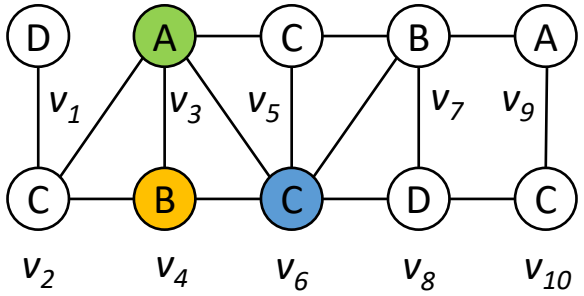
u_1	u_2	u_3	u_4
v_2	v_3	v_4	v_1

Evaluating Subgraph Query with Join

Input → **Relation Generation** → **Result Enumeration** → **Output**



Query Graph Q.



Data Graph G.

$Q := R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_2, u_3) \bowtie R(u_1, u_4)$

$R(u_1, u_3)$

u_1	u_3
v_2	v_4
v_5	v_7
v_6	v_4
v_6	v_7

$R(u_1, u_4)$

u_1	u_4
v_2	v_1
v_6	v_8
v_{10}	v_8

$R(u_2, u_3)$

u_2	u_3
v_3	v_4
v_9	v_7
v_{10}	v_8

$R(u_1, u_2)$

u_1	u_2
v_2	v_3
v_5	v_3
v_6	v_3
v_{10}	v_9

Matching Order φ

u_2, u_3, u_1, u_4

Candidate Data Vertex

$\{v_3, v_9\}$

$R(u_2 : v_3, u_3)$

$\{v_4\}$

$R(u_2 : v_3, u_1) \cap R(u_3 : v_4, u_1)$

$\{v_2, v_6\}$

$\{?\}$

u_1	u_2	u_3	u_4
v_2	v_3	v_4	v_1

Performance Factors

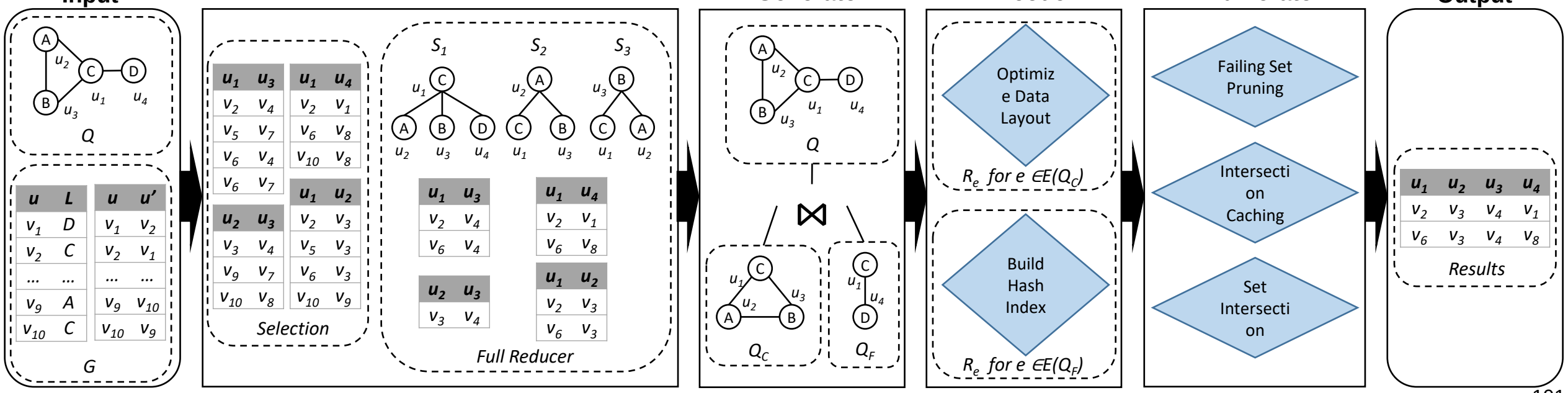
- The cardinality of the input relations.
- The effectiveness of the matching order.
- The efficiency of processing each intermediate result.

RapidMatch: A Holistic Approach to Subgraph Queries

Optimize the matching order to reduce the number of intermediate results.

Minimize the size of input relations.

Accelerate the efficiency of processing intermediate results.

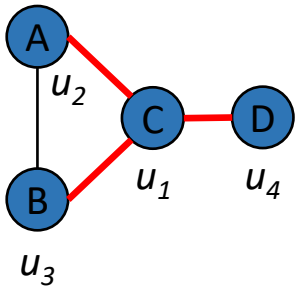


Relation Filter

- **Full Reducer:** A sequence of semi-joins to remove ***dangling tuples*** from an acyclic query.
  - Dangling tuples: the tuple that cannot appear in any results.

Notation:
S_u: The star rooted at a vertex u.

$$S_{u_1} := R(u_1, u_2) \bowtie R(u_1, u_3) \bowtie R(u_1, u_4)$$



Query Graph Q.

$R(u_1, u_3)$

u_1	u_3
v_2	v_4
v_5	v_7
v_6	v_4
v_6	v_7

$R(u_1, u_4)$

u_1	u_4
v_2	v_1
v_6	v_8
v_{10}	v_8

$R(u_1, u_2)$

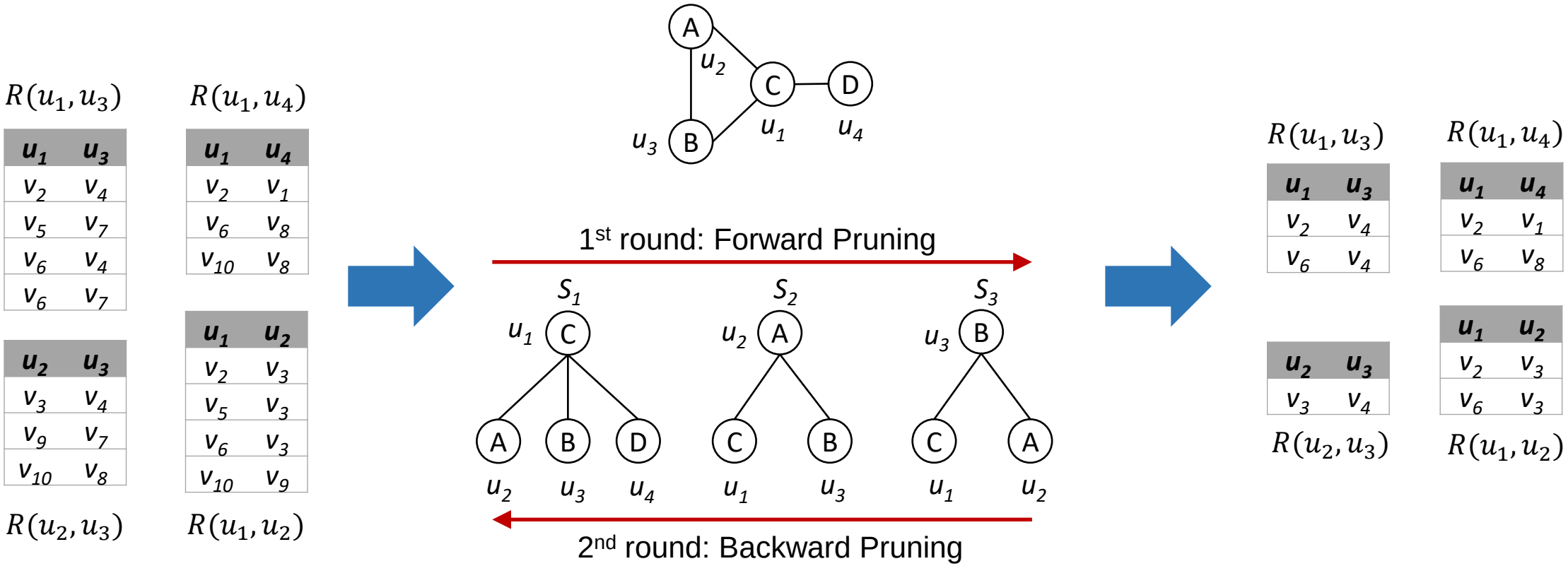
u_1	u_2
v_2	v_3
v_5	v_3
v_6	v_3
v_{10}	v_9

Results $R(S_{u_1})$

u_1	u_2	u_3	u_4
v_2	v_3	v_4	v_1
v_6	v_3	v_4	v_8
v_6	v_3	v_7	v_8

Relation Filter

- **Method:** Apply the full reducer on S_u for each query vertex u along an order δ.
  - 1st: conduct the filter along the order of δ, i.e., forward pruning.
  - 2nd: repeat the filter along the reverse order of δ, i.e., backward pruning.

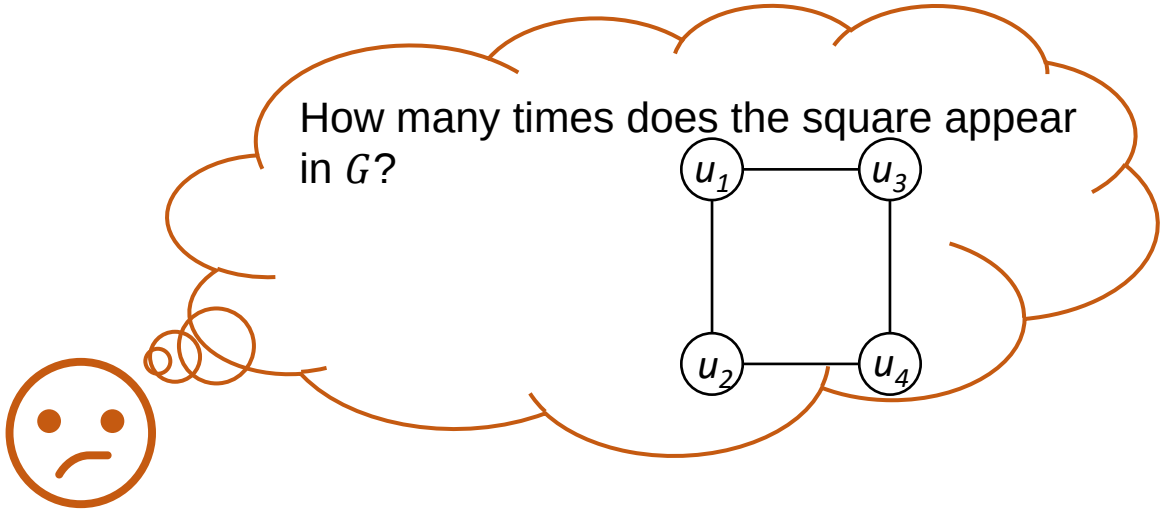


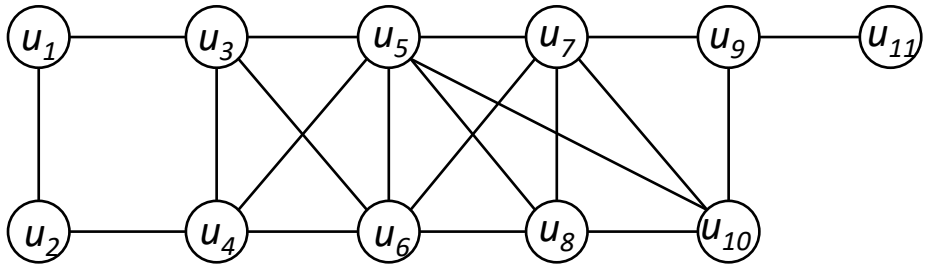
Traditional Join Plan Generator

- **Problem:** Optimize the matching order to minimize the number of intermediate results.
- **Existing Methods:**
  - **Task 1:** Estimate the cost given a matching order based on the ***cardinality estimation***.
  - **Task 2:** Find the order with the minimum cost in the ***plan space***.

Cardinality Estimation is Hard

- **Cardinality Estimation:** Estimate the number of a sub-structure of Q that appears in G.



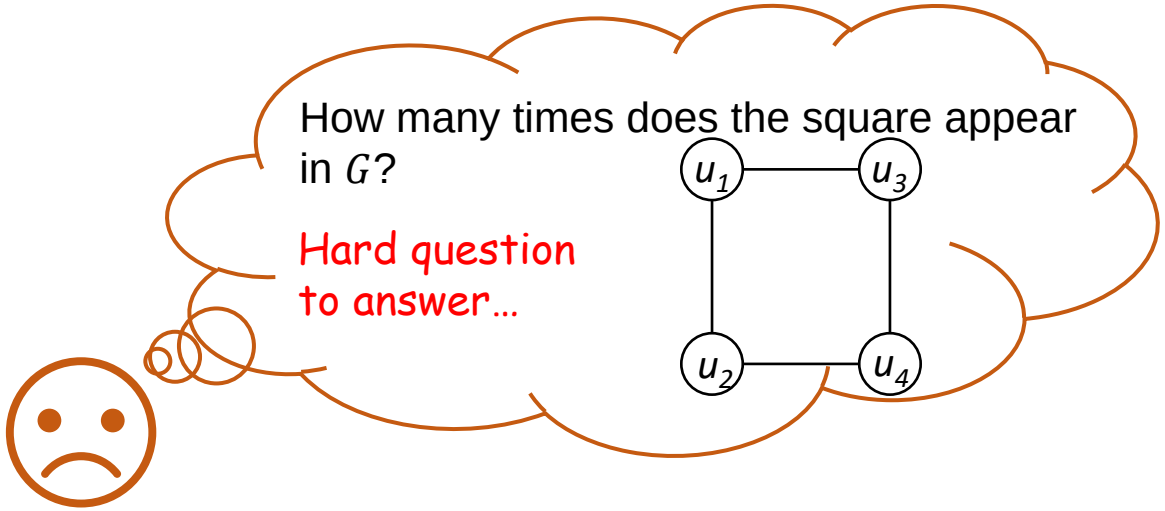


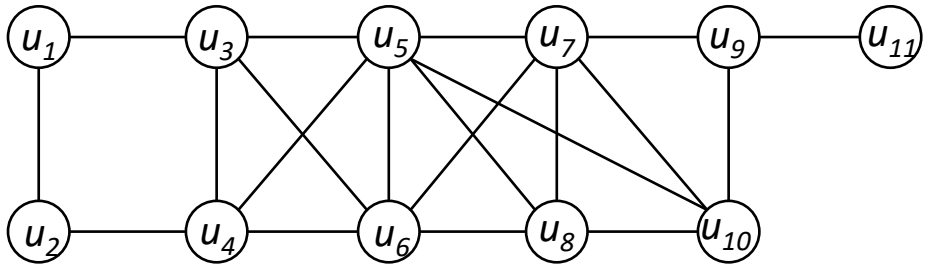
Query Graph Q.

Data Graph G.

Cardinality Estimation is Hard

- **Cardinality Estimation:** Estimate the number of a sub-structure of Q that appears in G.



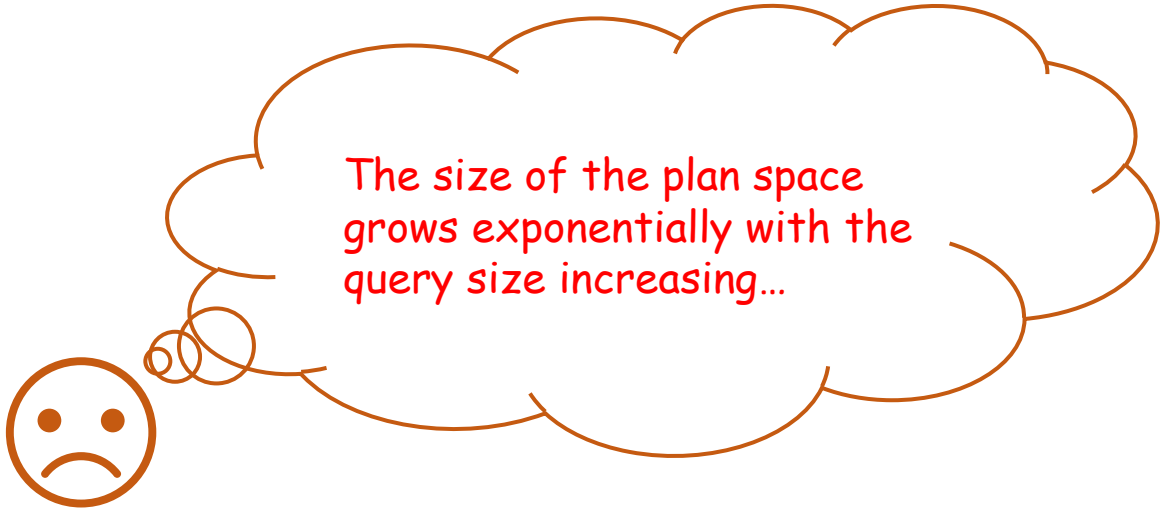


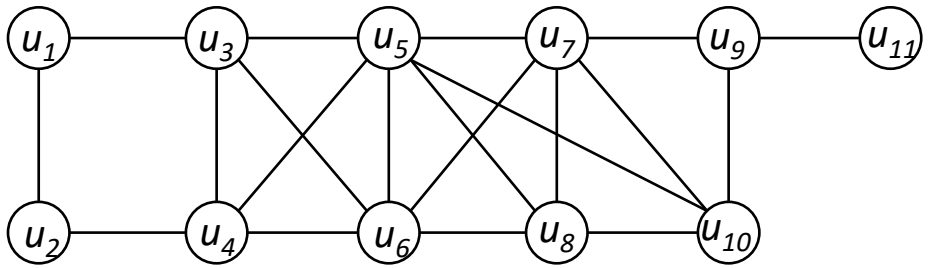
Query Graph Q.

Data Graph G.

Plan Space is Huge

- **Plan Space:** A set containing all valid join orders.



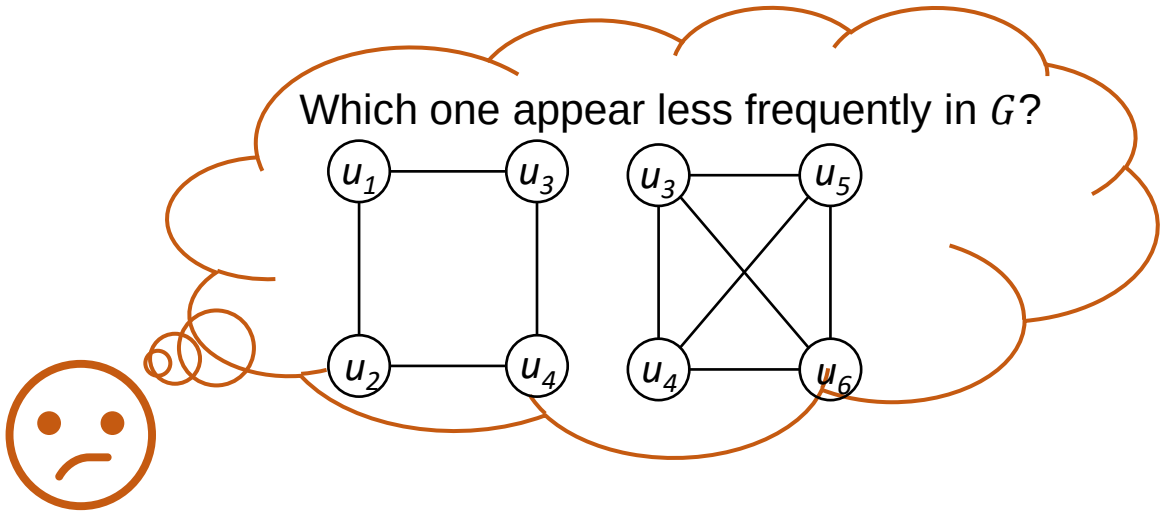


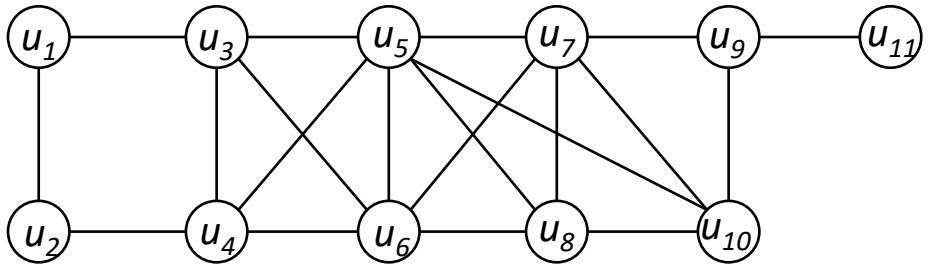
Query Graph Q.

The number of query vertices	The size of the plan space
8	40,320
9	362,880
10	3,628,800
11	39,916,800
…	…
16	20,922,789,888,000

Considering to extend φ by a vertex at one time only.

Considering a Simpler Problem

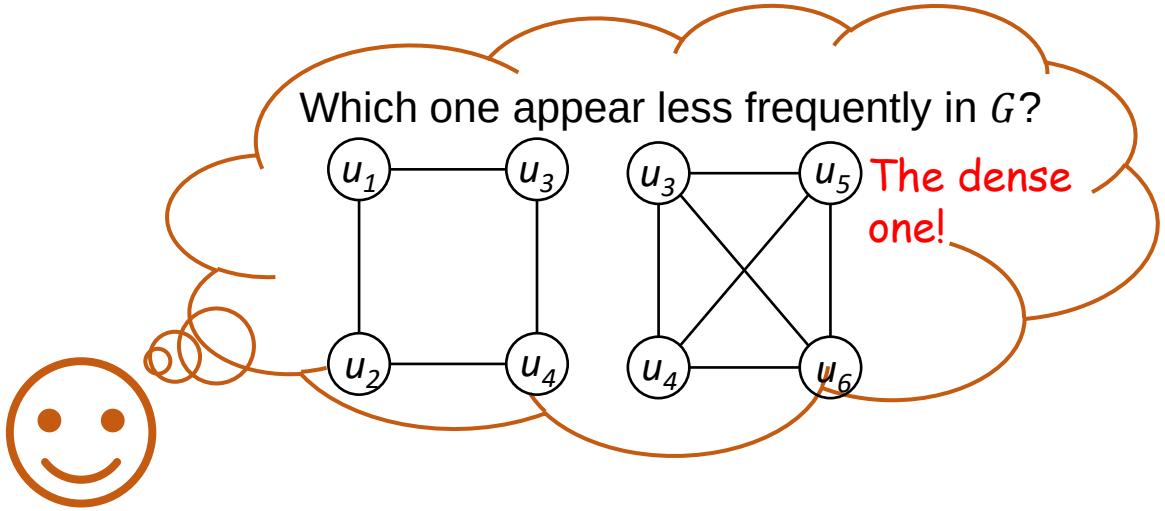


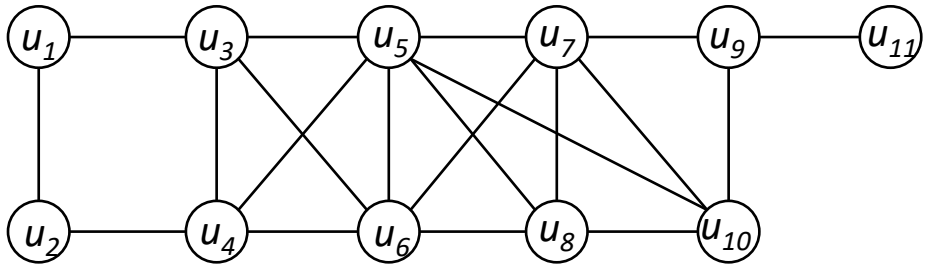


Query Graph Q.

Data Graph G.

Considering a Simpler Problem



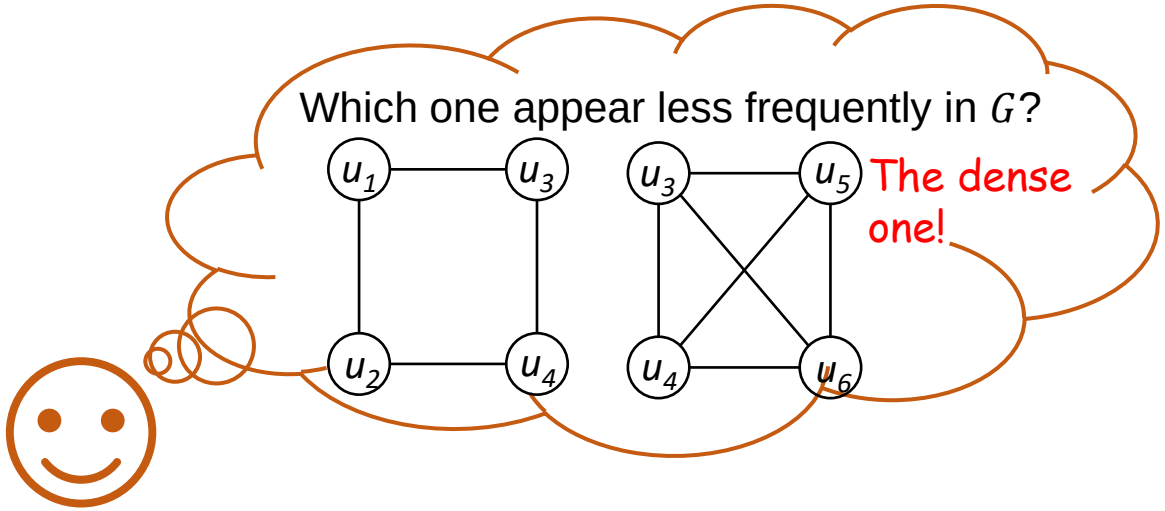


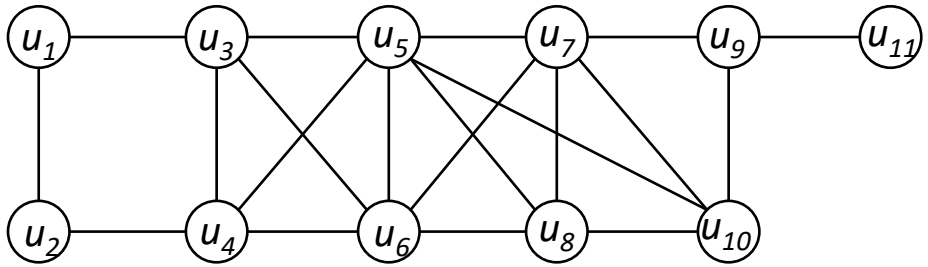
Query Graph Q.

Data Graph G.

Considering a Simpler Problem

Prioritizing dense sub-structures of Q can reduce the number of intermediate results.





Query Graph Q.

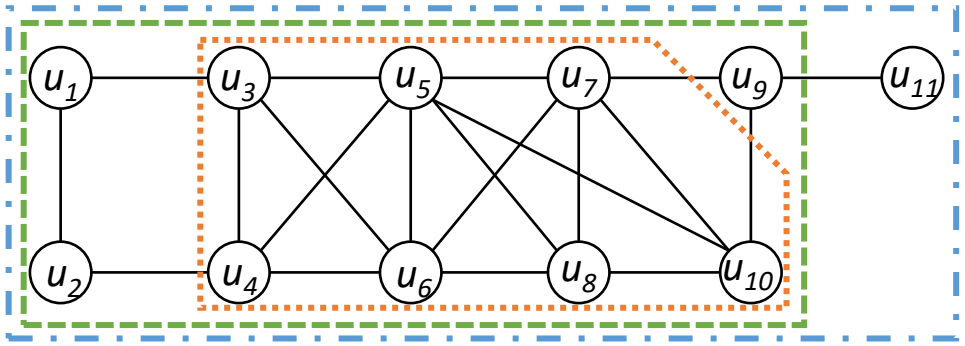
Data Graph G.

Join Plan Generator based on Graph Density

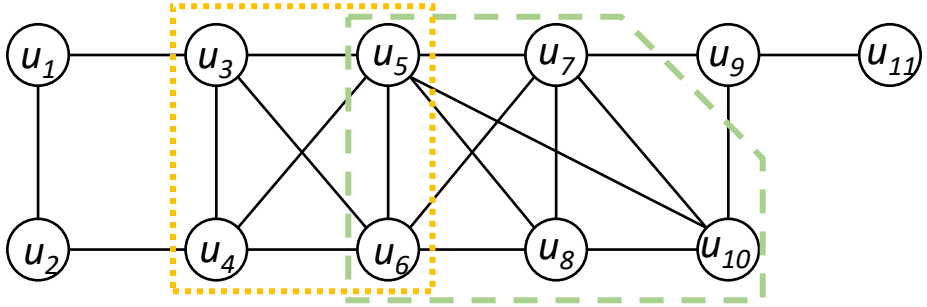
- Decompose Q into several subgraphs with different densities.
- Construct a tree where each node is a subgraph and the edge denotes the containment relationship.
- Traverse the tree to generate a matching order putting vertices in the dense part of Q at the beginning of the matching order.

Optimizing Matching Order based on Graph Density

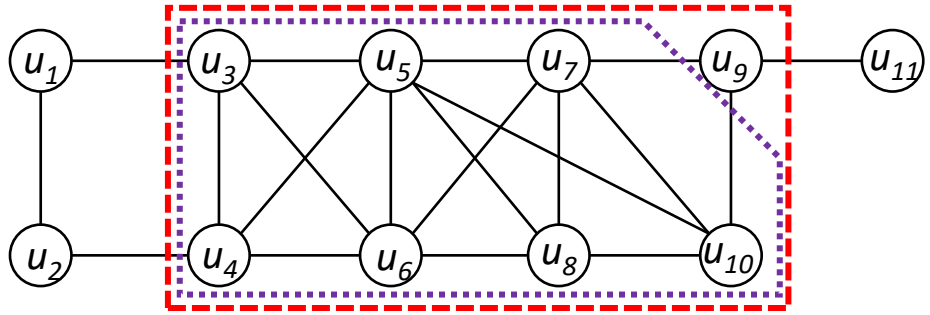
- **Nucleus Decomposition:** Find dense subgraphs at different level of hierarchies.
  - a nucleus χ is a connected subgraph satisfying density and connectivity constraints.
  - a nucleus forest $\mathcal{T}$ describes hierarchies based on nucleus containment relationship.



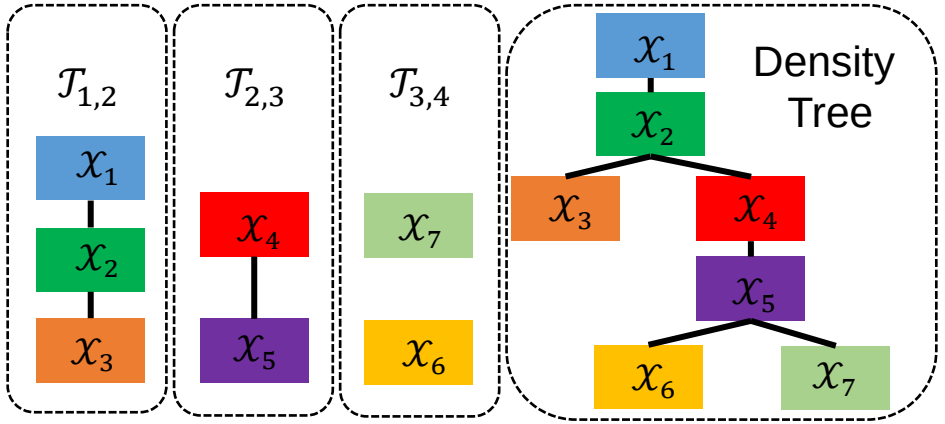
$r = 1, s = 2$



$r = 3, s = 4$



$r = 2, s = 3$



Theoretical Guarantee

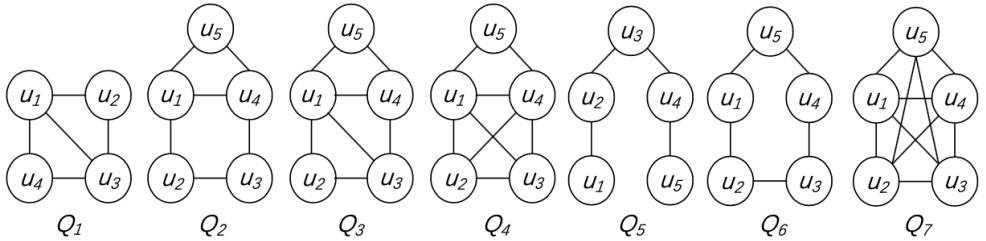
- For the query graph Q with an arbitrary structure, RapidMatch is worst-case optimal, i.e., the running time matches the maximum output size of Q.

- For the query graph Q with the acyclic structure, RapidMatch is instance optimal, i.e., the running time matches the number of results in G.

Experimental Setup

- **Data Graphs:** Seven real-world graphs with $|E(G)|$ varying from $86K$ to $42M$.

- **Query Graphs:** Both small and large query workloads.
  - **Small Queries:** Seven queries widely used in previous work.



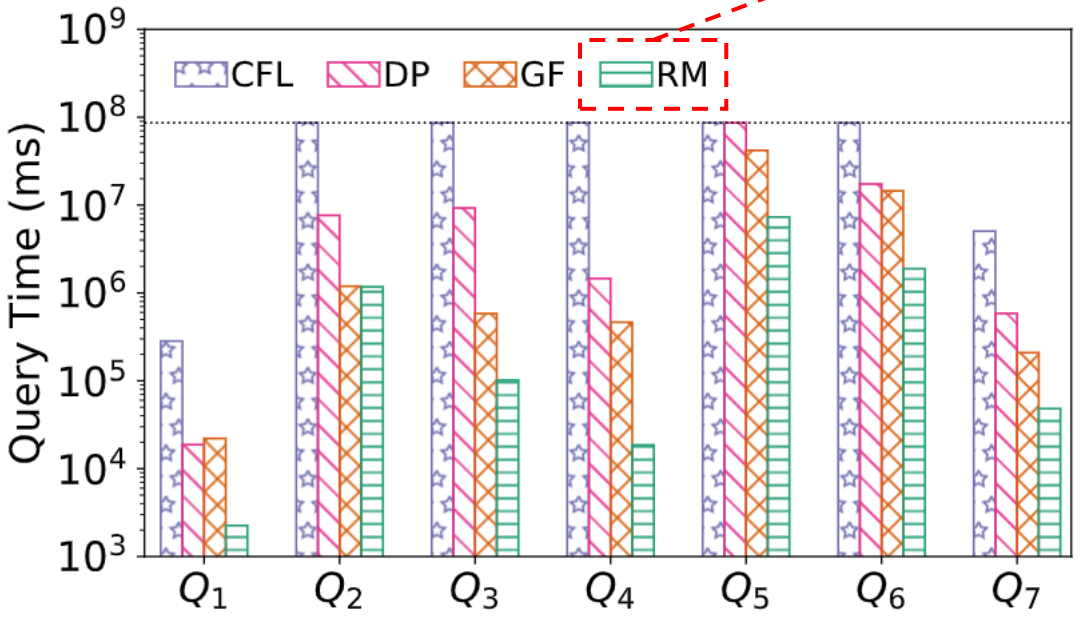
  - **Large Queries:** Ten query set each of which contains 200 queries.
    - $|V(Q)|$ varied from 4 to 32.

- **Counterparts:**
  - CFL [SIGMOD'16], DF [SIGMOD'19], GF [VLDB'19]

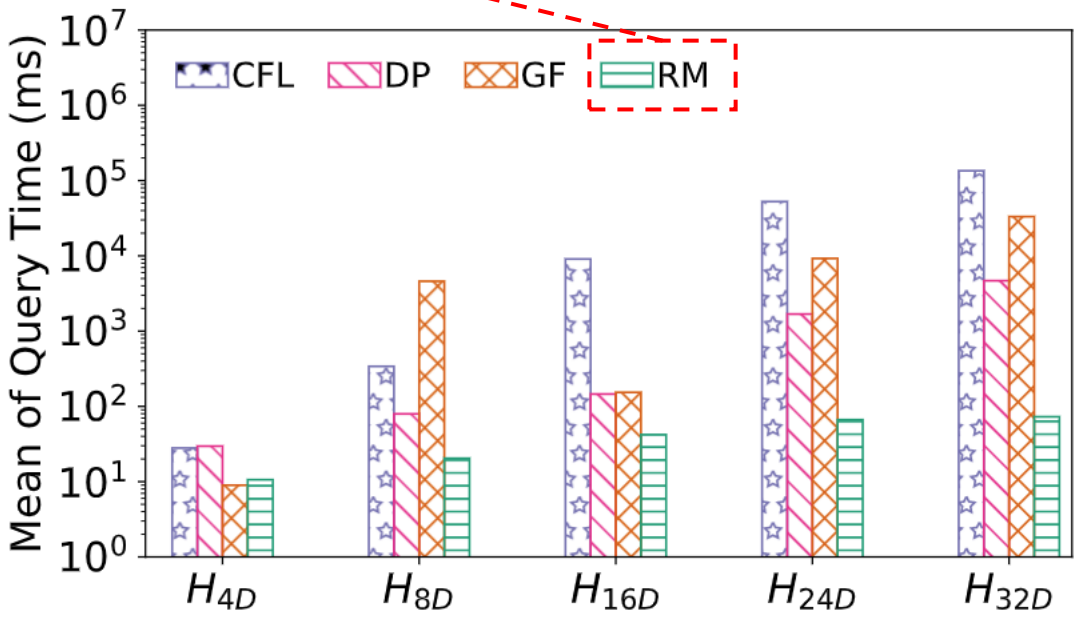
Experiment Results

- Our solution outperforms state of the art by orders of magnitude.

Our Solution: RapidMatch



Small queries on *eu2005* dataset
$|V| = 862{,}664$, $|\mathrm{E}| = 16{,}138{,}468$, $|\Sigma| = 4$



Large queries on *youtube* dataset
$|V| = 1{,}134{,}890$, $|\mathrm{E}| = 2{,}987{,}624$, $|\Sigma| = 25$

Notation:
$|\Sigma|$: The number of labels.

Summary

- We study exploration-based and join-based methods and bridge the gap between them.
- We propose a join-based engine that can efficiently evaluate various workloads.
- We conduct extensive experiments with various workloads to evaluate the effectiveness of our solution.
- Datasets and source code available at *github.com/RapidsAtHKUST/RapidMatch*.

PathEnum: Towards Real-Time Hop Constraint *s-t* Path Enumeration

Shixuan Sun, Yuhang Chen, Bingsheng He, Bryan Hooi

National University of Singapore

Walk and Path

- Walk W:
  - A sequence of vertices $(v_0, v_1, \ldots, v_l)$ such that $\forall 1 \leq i \leq l, e(v_{i-1}, v_i) \in E$.

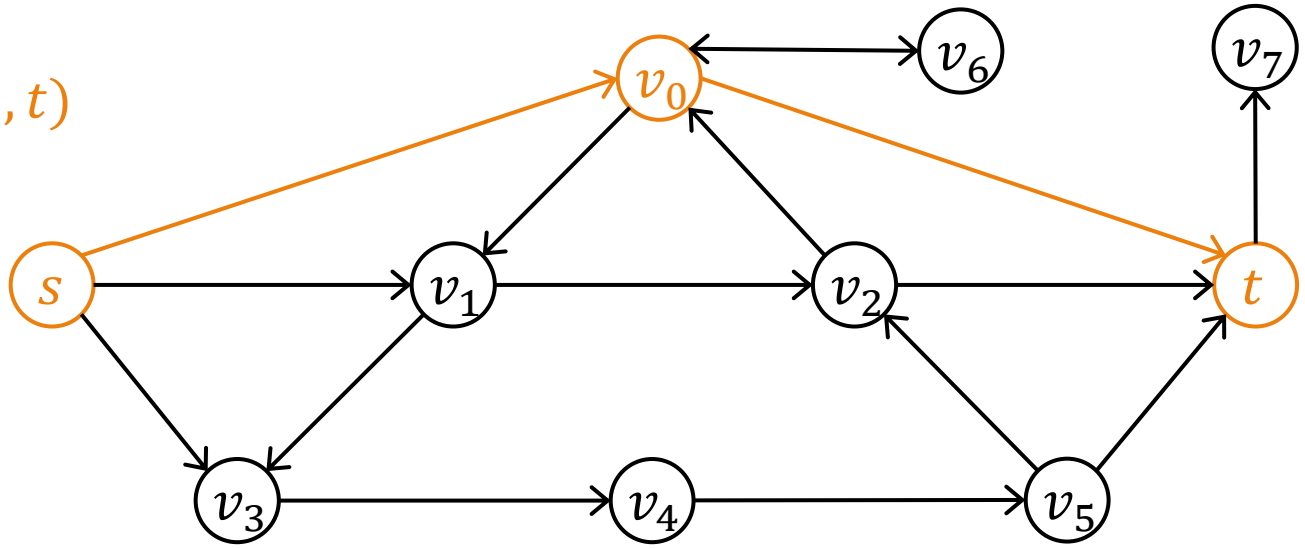
Walk and Path

- Walk W:
  - A sequence of vertices $(v_0, v_1, \dots, v_l)$ such that $\forall 1 \leq i \leq l, e(v_{i-1}, v_i) \in E$.
- Path P:
  - A walk with no duplicate vertices.

Walk and Path

- Walk W:
  - A sequence of vertices $(v_0, v_1, \ldots, v_l)$ such that $\forall 1 \leq i \leq l, e(v_{i-1}, v_i) \in E$.
- Path P:
  - A walk with no duplicate vertices.

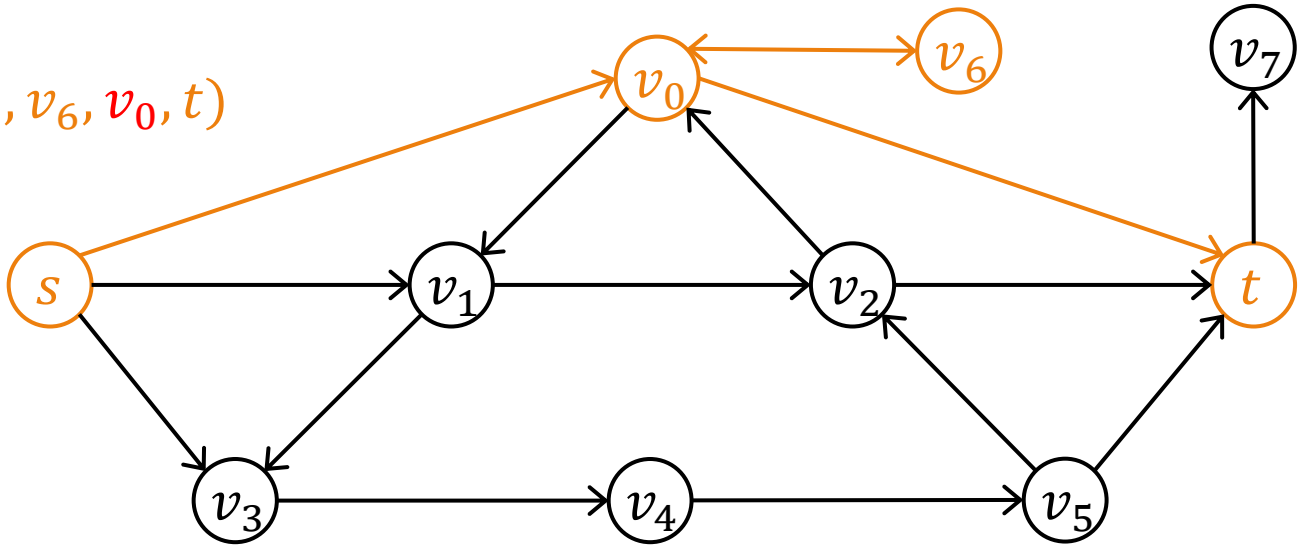
$W_1 = (s, v_0, t)$



Walk and Path

- Walk W:
  - A sequence of vertices $(v_0, v_1, \ldots, v_l)$ such that $\forall 1 \leq i \leq l, e(v_{i-1}, v_i) \in E$.
- Path P:
  - A walk with no duplicate vertices.

$W_2 = (s, v_0, v_6, v_0, t)$

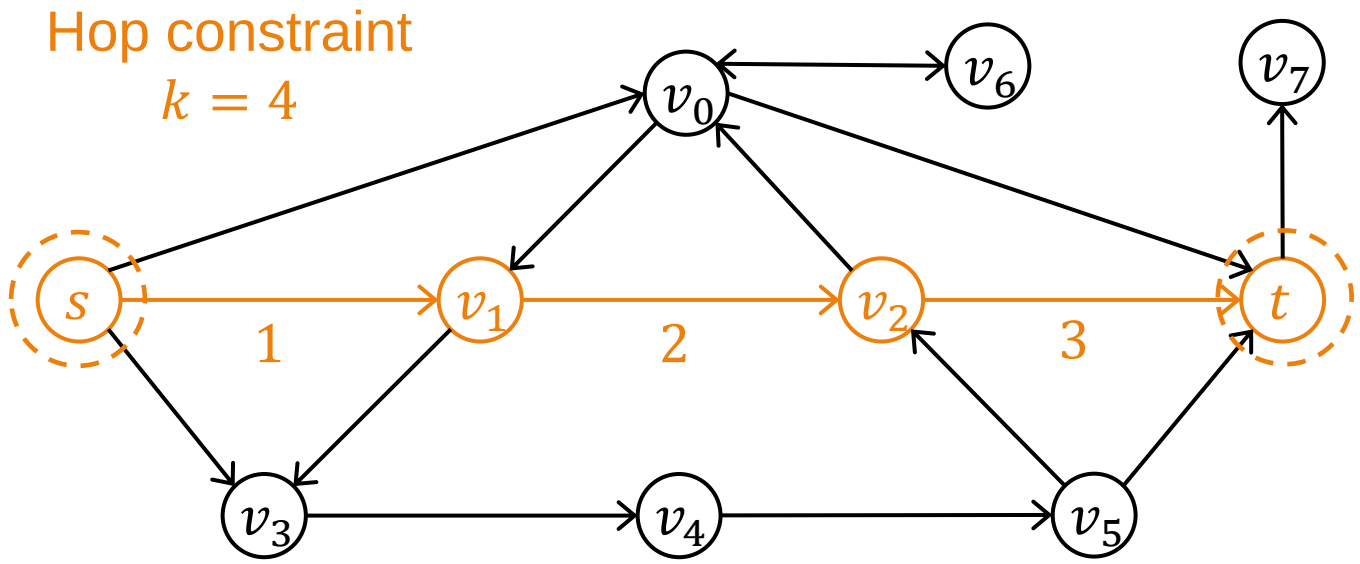


Problem Definition

- Hop constraint s-t path enumeration (HcPE):
  - Find all paths P from s to t such that the length $L(P) \leq k$.

Problem Definition

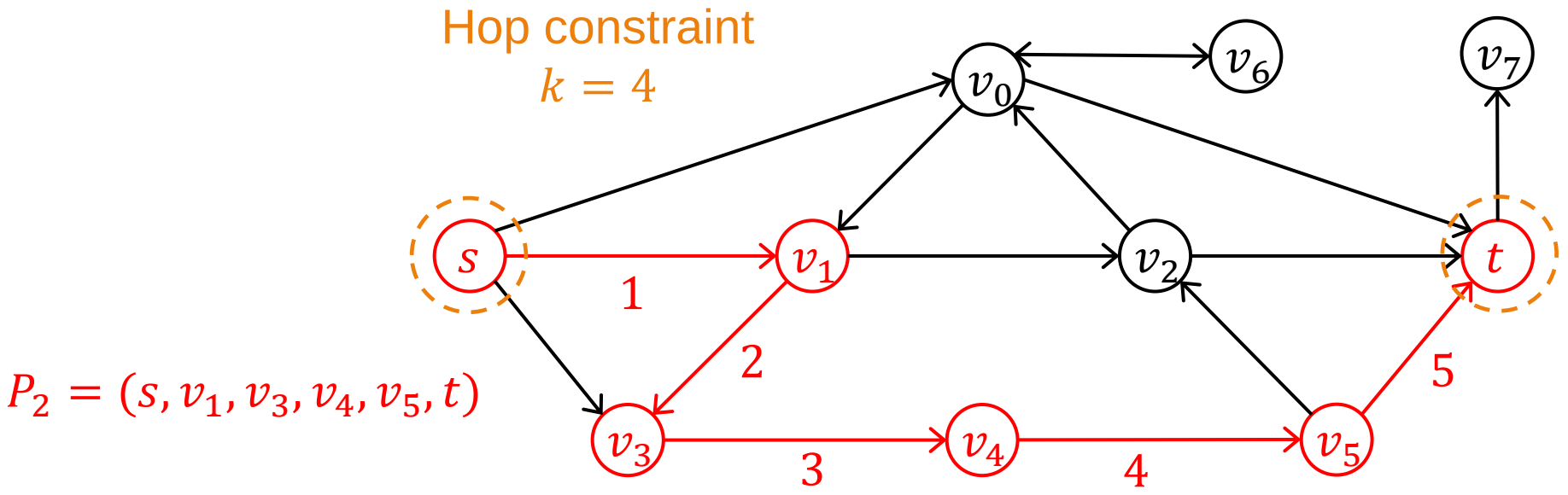
- Hop constraint s-t path enumeration (HcPE):
  - Find all paths P from s to t such that the length $L(P) \leq k$.



$P_1 = (s, v_1, v_2, t)$

Problem Definition

- Hop constraint s-t path enumeration (HcPE):
  - Find all paths P from s to t such that the length $L(P) \leq k$.



Applications

- Detecting money laundering [FATF'13, AAAI'20]
  - Money transactions among bank accounts.
  - Find transaction paths between suspicious accounts.
  - k is relatively small (e.g., $k = 2$).

Applications

- Detecting money laundering [FATF'13, AAAI'20]
  - Money transactions among bank accounts.
  - Find transaction paths between suspicious accounts.
  - k is relatively small (e.g., $k = 2$).

- Detecting e-commerce merchant fraud [VLDB'18]
  - Activities among individual users in online shopping.
  - Find cycles triggered by activities between users.
  - k is relatively small (e.g., $k = 6$).

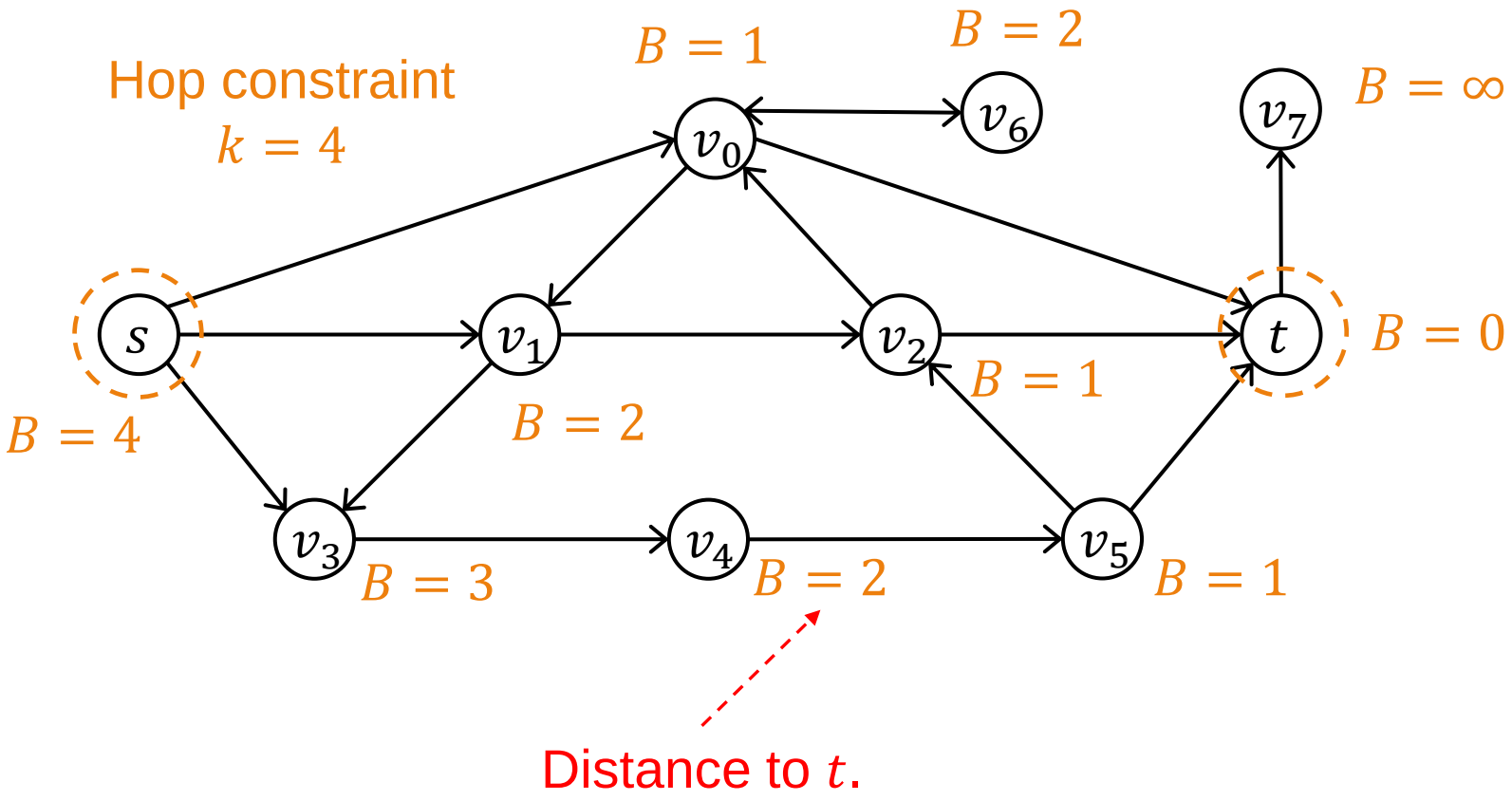
Challenges

- Applications have rigid real-time requirement.
- Search space can be large with k increasing.
- Query time of different queries varies greatly.

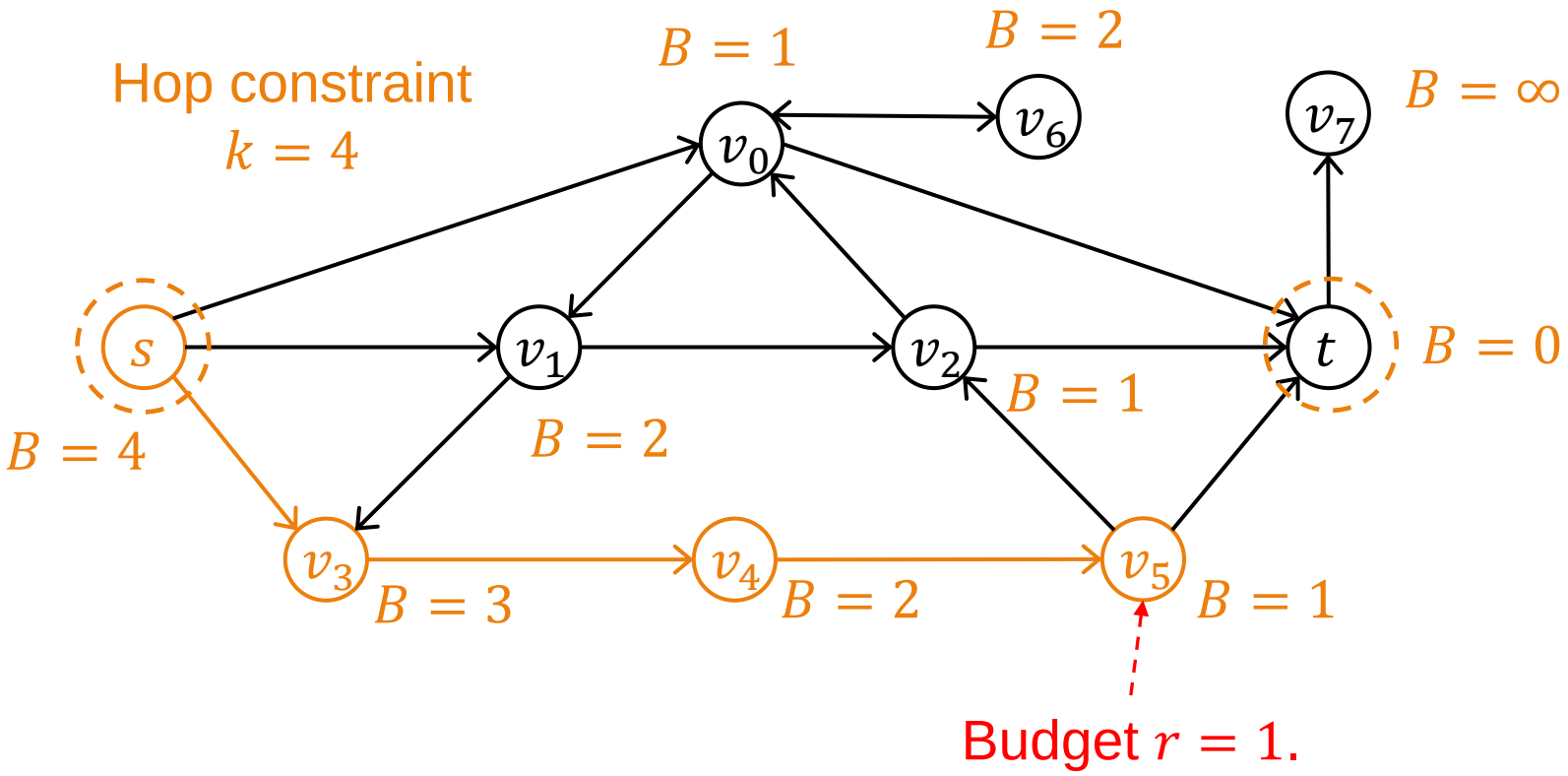
Existing Solutions

- A depth-first search (DFS) based framework [VLDB'20].
  - Enumerate all results by executing a backtracking search from s on G.
  - Prune invalid paths with barriers.
  - Update barriers dynamically to achieve polynomial delay.

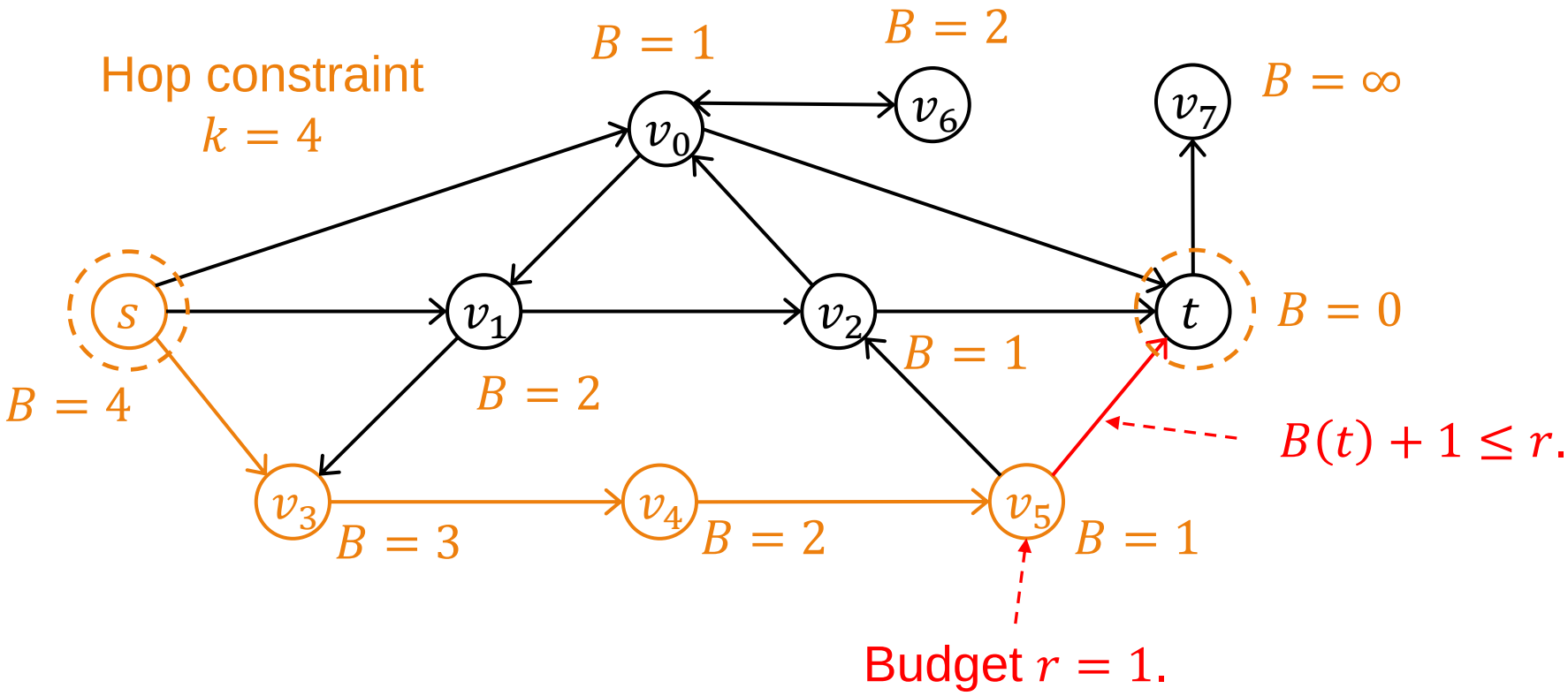
Barrier Initialization



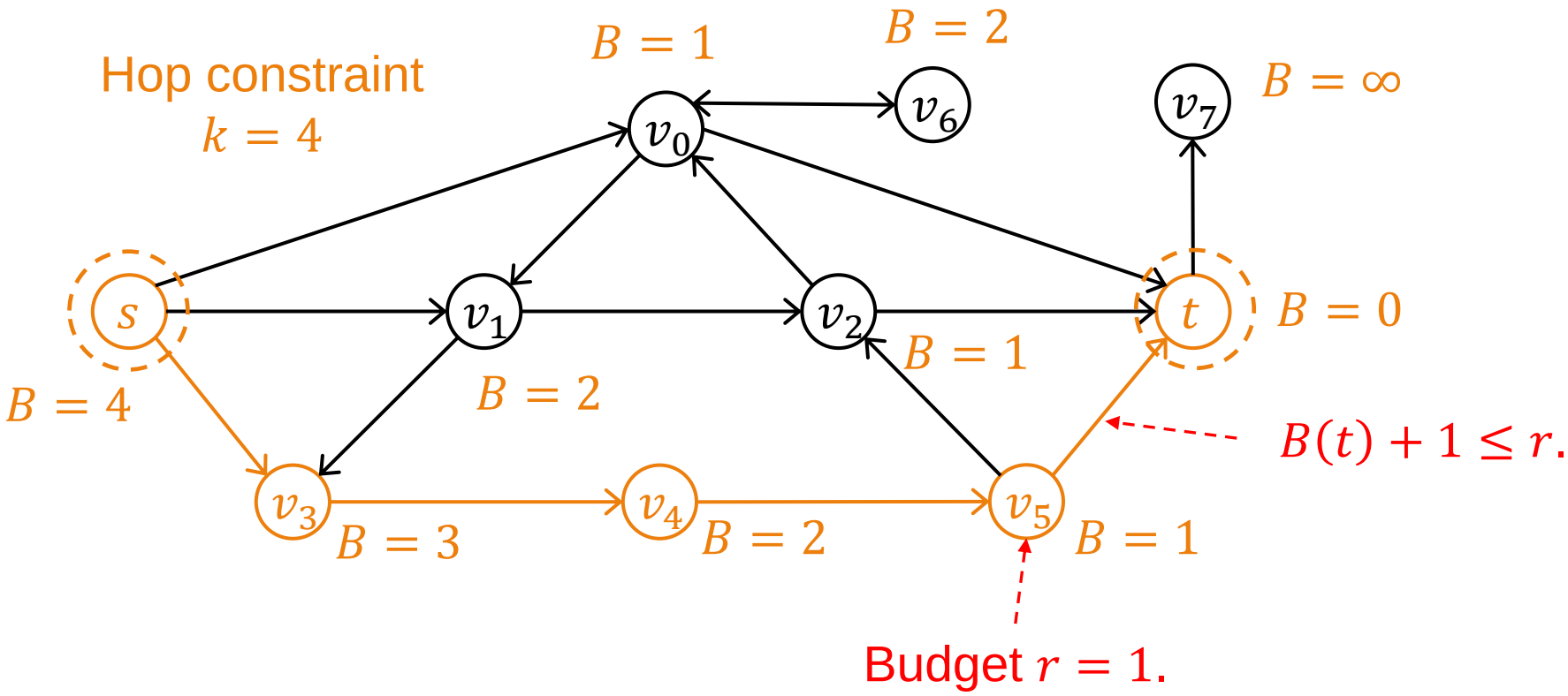
DFS on Graph


Hop constraint
$k = 4$
$B = 1$
$B = 2$
$B = \infty$
$B = 0$
$B = 4$
$B = 2$
$B = 1$
$B = 3$
$B = 2$
$B = 1$
v_0
v_1
v_2
v_3
v_4
v_5
v_6
v_7
s
t
Budget $r = 1$.

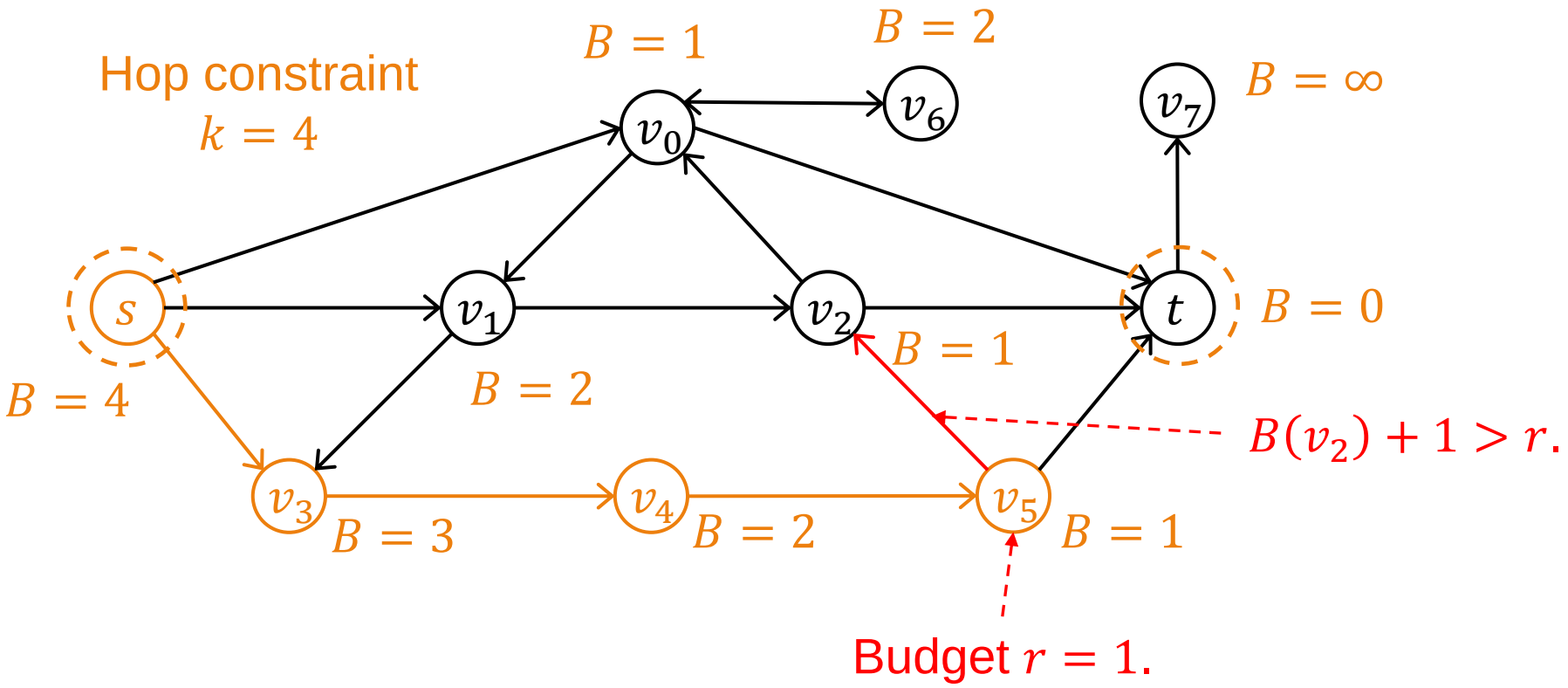
DFS on Graph


Hop constraint
$k = 4$
$B = 1$
$B = 2$
$B = \infty$
v_0
v_6
v_7
s
v_1
v_2
t
$B = 0$
$B = 4$
$B = 2$
$B = 1$
$B(t) + 1 \leq r.$
v_3
v_4
v_5
$B = 3$
$B = 2$
$B = 1$
Budget $r = 1$.

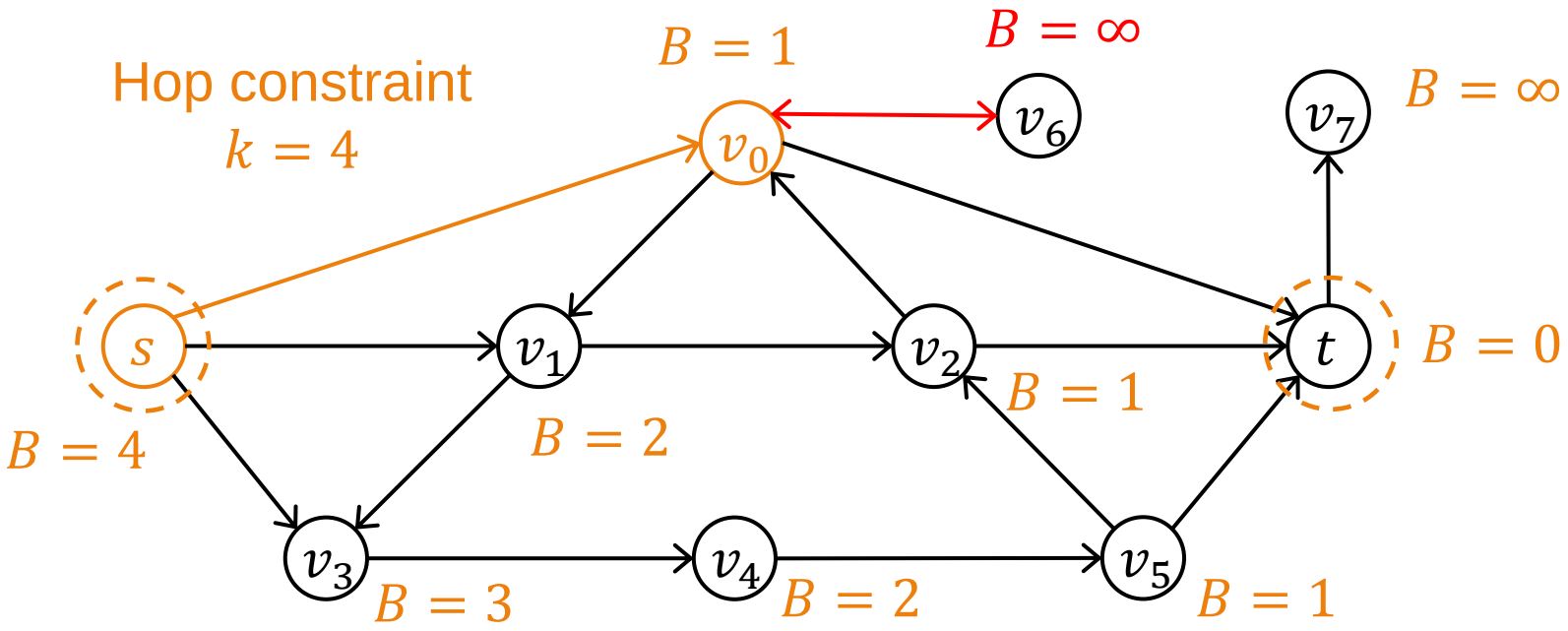
DFS on Graph


Hop constraint
$k = 4$
$B = 1$
$B = 2$
$B = \infty$
v_0
v_6
v_7
s
v_1
v_2
t
$B = 0$
$B = 4$
$B = 2$
$B = 1$
$B(t) + 1 \leq r.$
v_3
v_4
v_5
$B = 3$
$B = 2$
$B = 1$
Budget $r = 1$.

DFS on Graph


Hop constraint
$k = 4$
$B = 1$
$B = 2$
$B = \infty$
$B = 0$
$B = 4$
$B = 2$
$B = 1$
$B = 3$
$B = 2$
$B = 1$
$B(v_2) + 1 > r.$
Budget $r = 1$.
s
v_0
v_1
v_2
v_3
v_4
v_5
v_6
v_7
t

Barrier Update



Issues

- Barrier update incurs high overhead.
- Invalid edges involve in the search.
- Lack a model to optimize the search order.

Issues

- Barrier update incurs high overhead.
- Invalid edges involve in the search.
- Lack a model to optimize the search order.

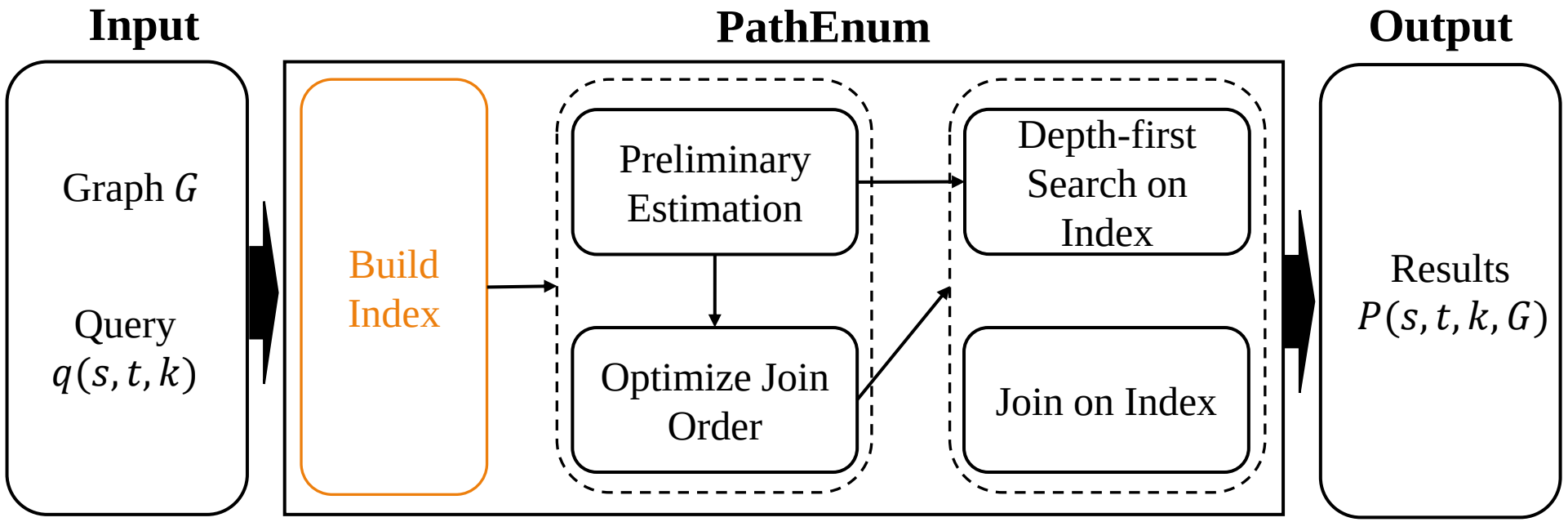
Fail to meet the rigid time-constraint in real-world applications!

Our Solution

PathEnum:
Keep the search ~~simple but efficient~~
simple and efficient!

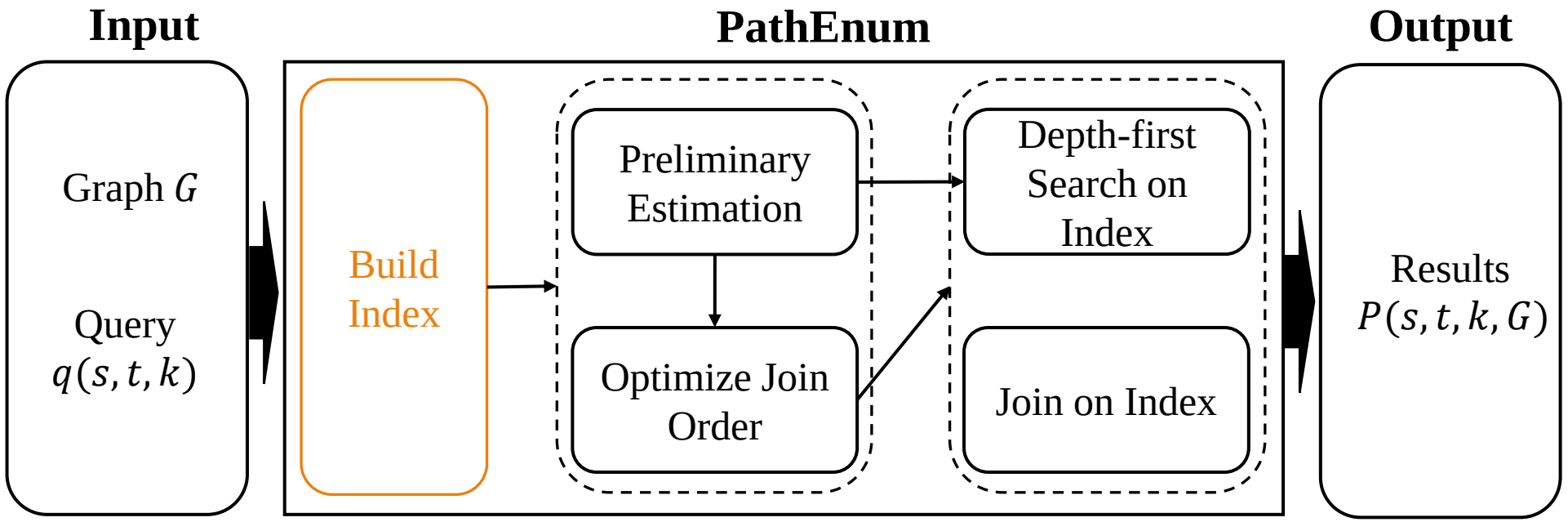
Design

- Build a light-weight index I by executing BFS from s and t.



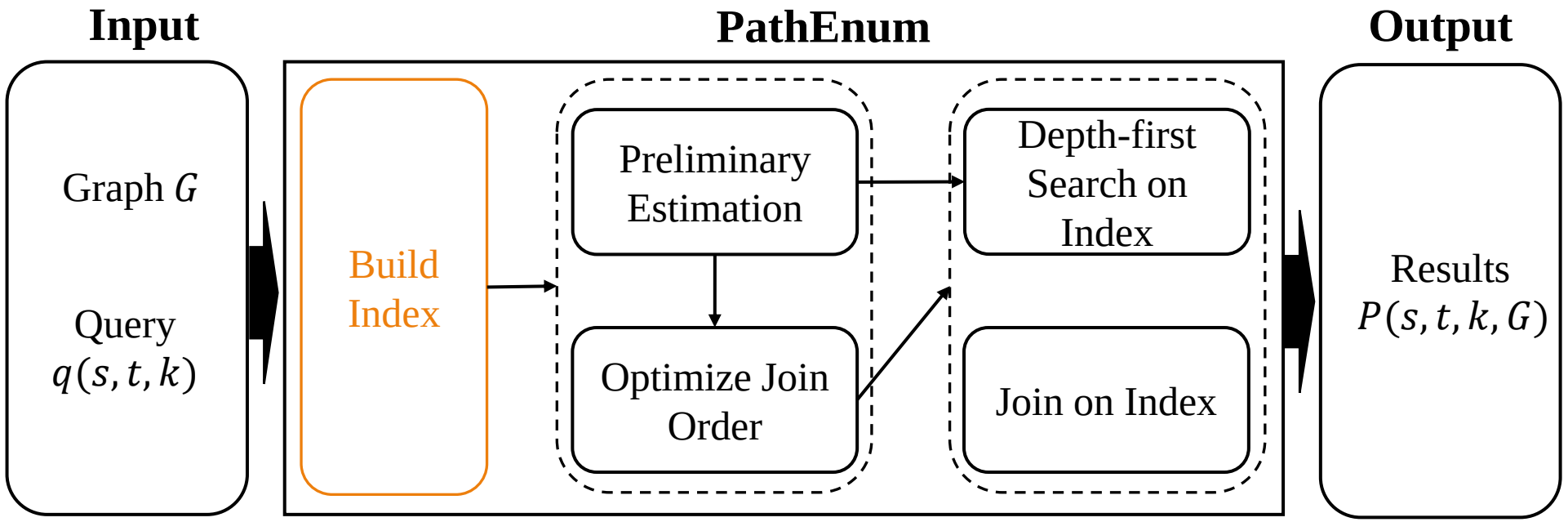
Design

- Build a light-weight index I by executing BFS from s and t.
  - $I(i)$: candidate vertices that can appear at position i of P from s to t.



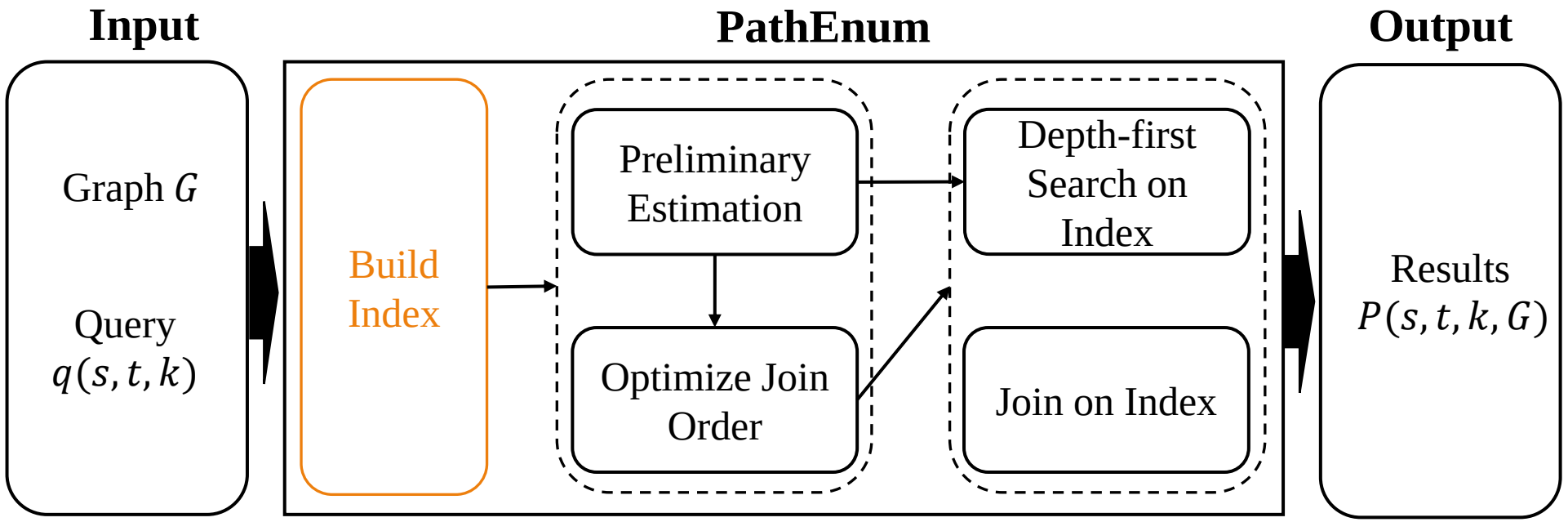
Design

- Build a light-weight index I by executing BFS from s and t.
  - $I(i)$: candidate vertices that can appear at position i of P from s to t.
  - $I(v, i)$: neighbors v' of v such that $B(v') \leq i$.



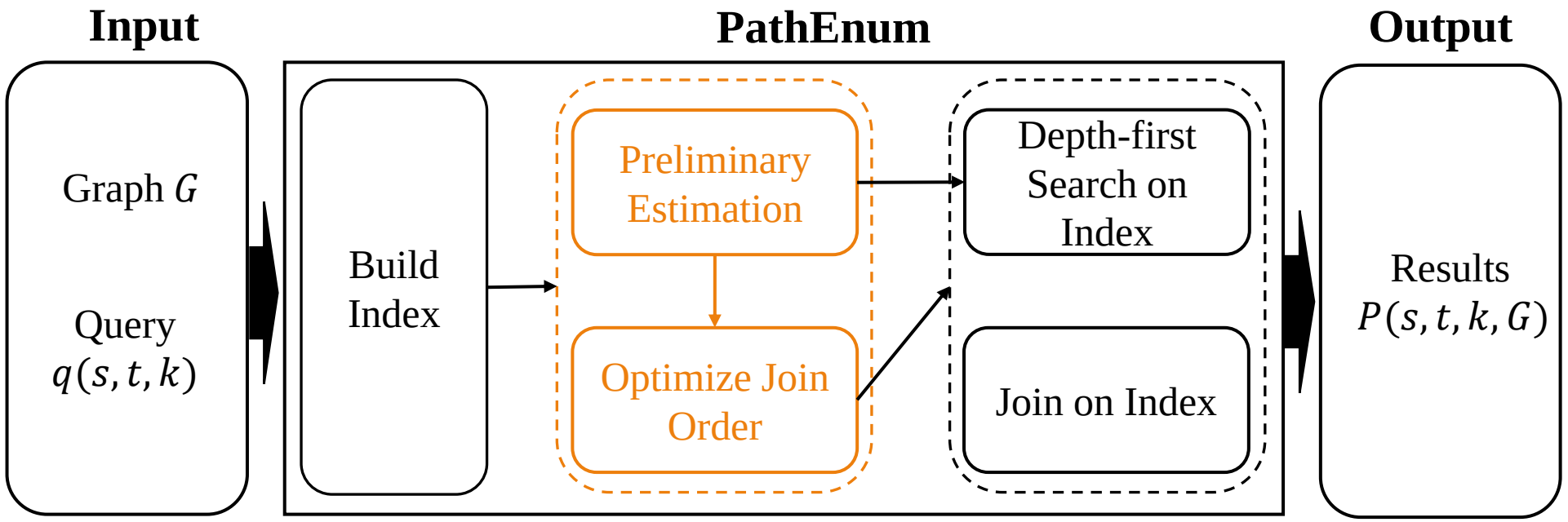
Design

- Build a light-weight index I by executing BFS from s and t.
  - $I(i)$: candidate vertices that can appear at position i of P from s to t.
  - $I(v, i)$: neighbors v' of v such that $B(v') \leq i$.
  - Time complexity: $O(|V| + |E|)$.



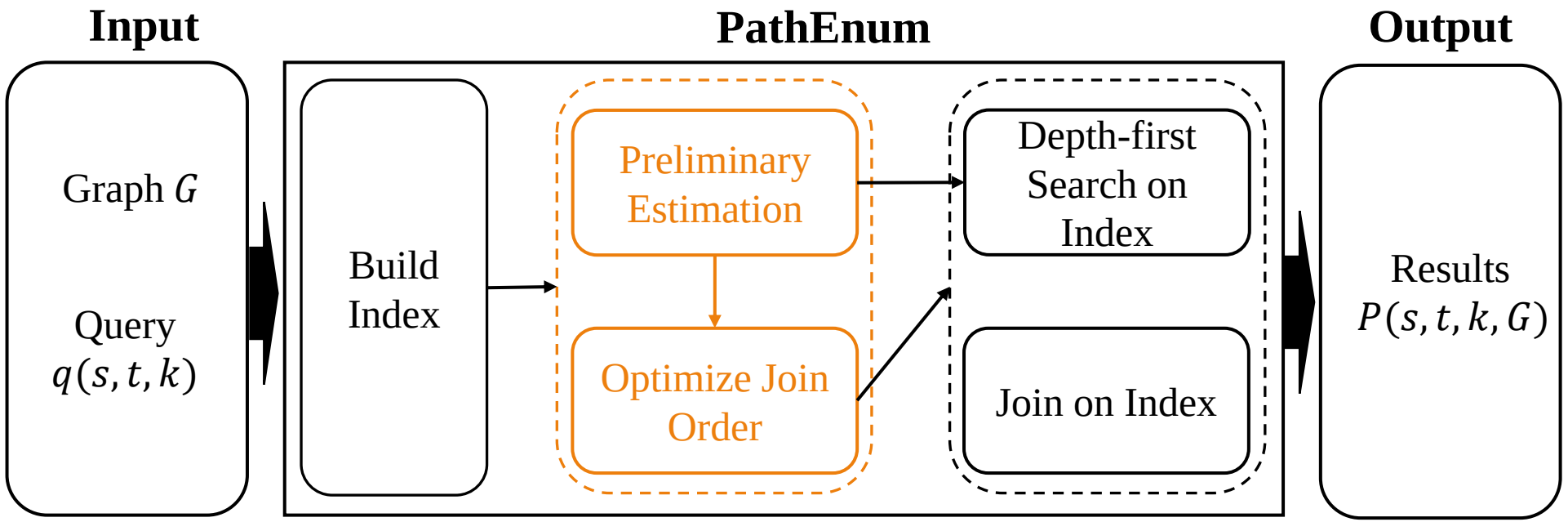
Design

- Optimize the search order with a join-based model.



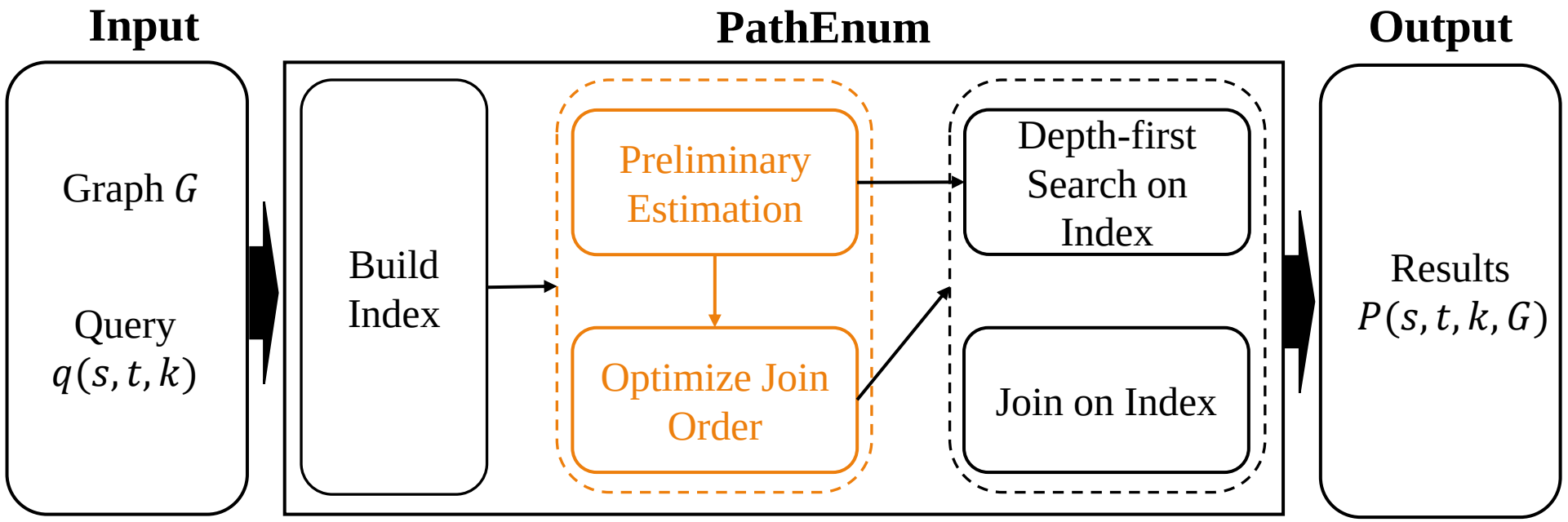
Design

- Optimize the search order with a join-based model.
  - Preliminary: roughly but quickly estimate the cost of the search.



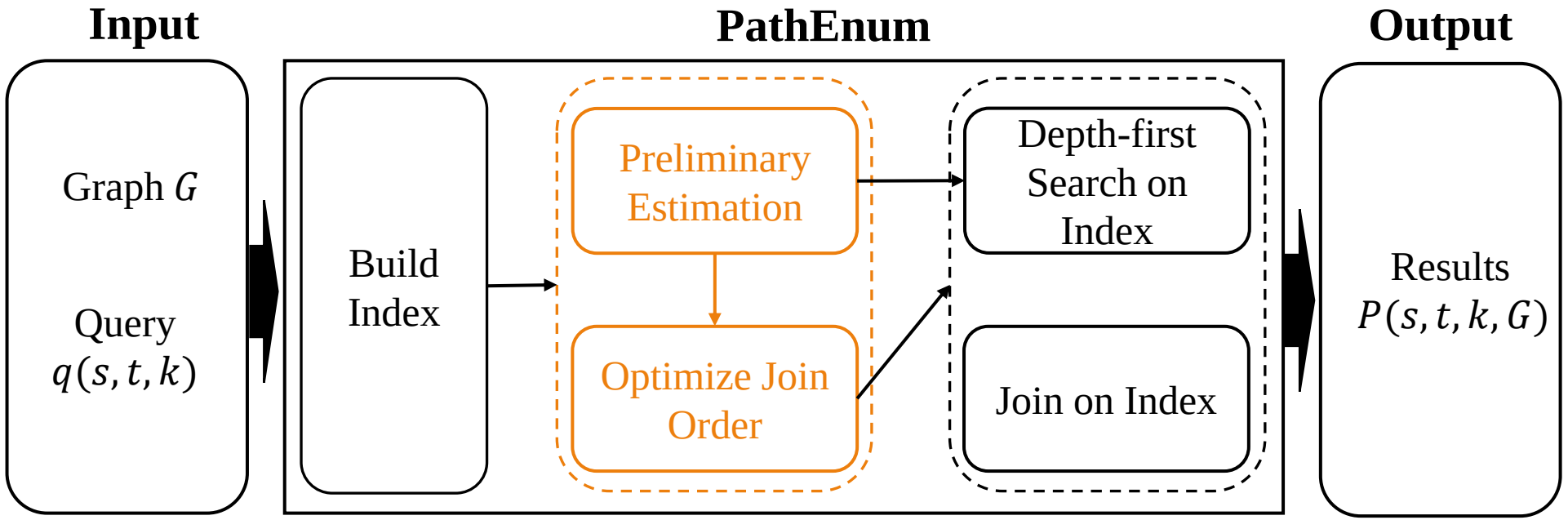
Design

- Optimize the search order with a join-based model.
  - Preliminary: roughly but quickly estimate the cost of the search.
  - Full-fledged: optimize the order with a dynamic programming method.



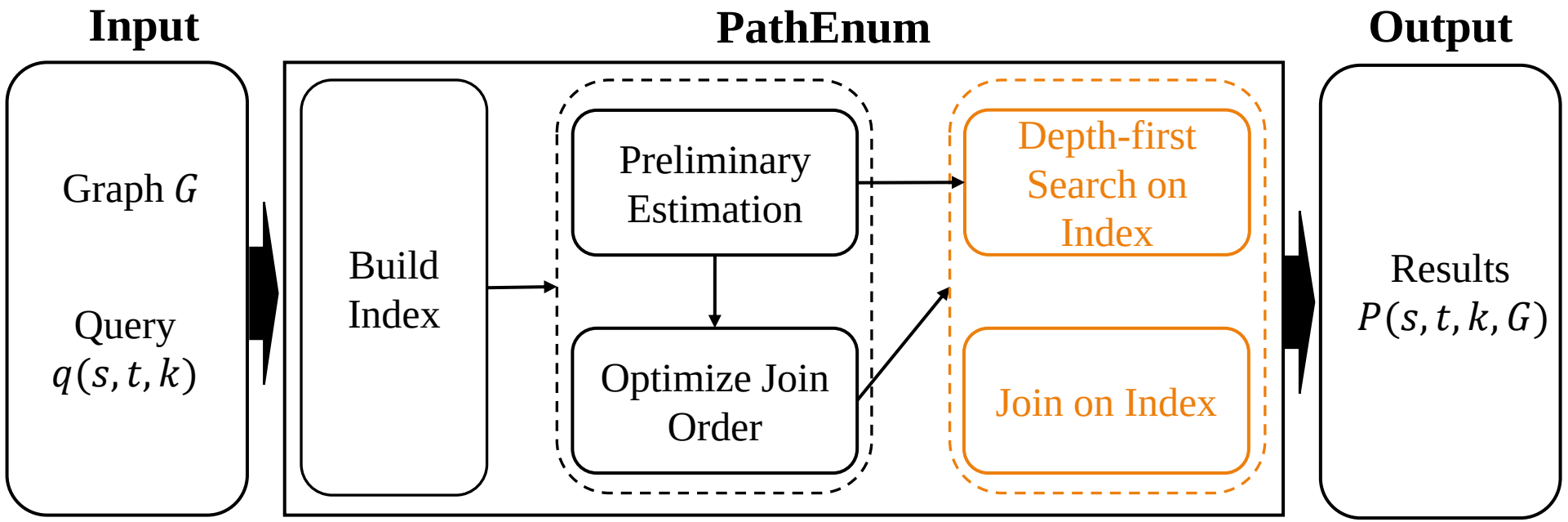
Design

- Optimize the search order with a join-based model.
  - Preliminary: roughly but quickly estimate the cost of the search.
  - Full-fledged: optimize the order with a dynamic programming method.
  - Time complexity: $O(k^2)$ and $O(k \times |E|)$.



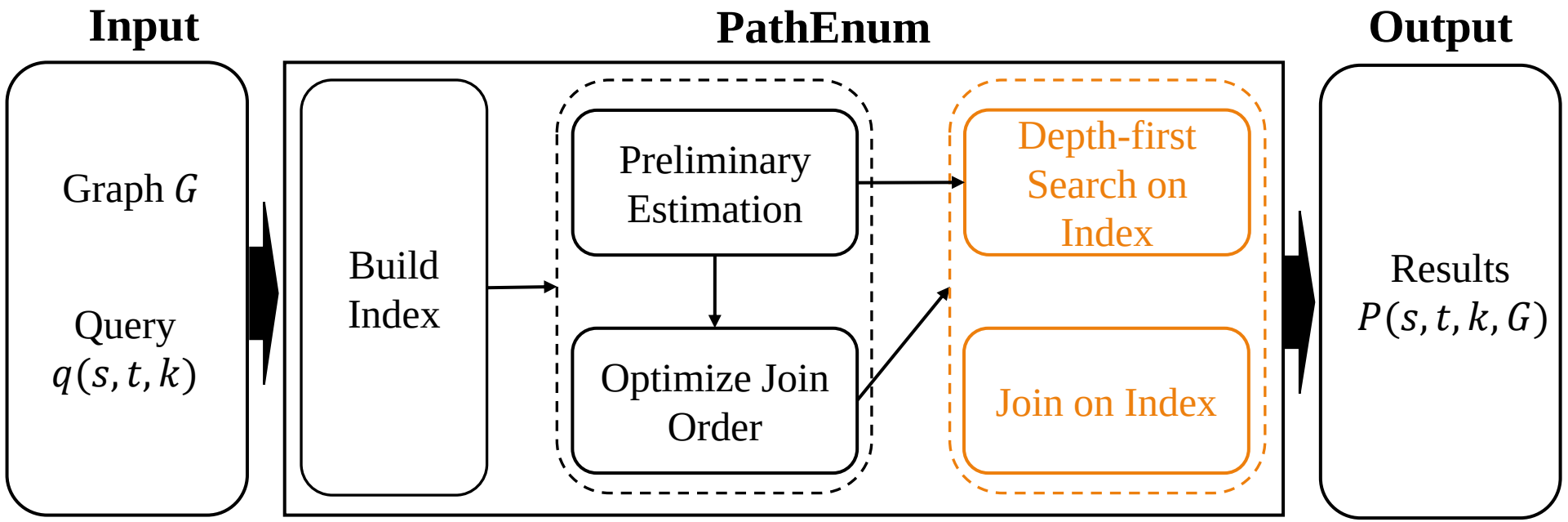
Design

- Search on the index based on the guidance of the optimizer.



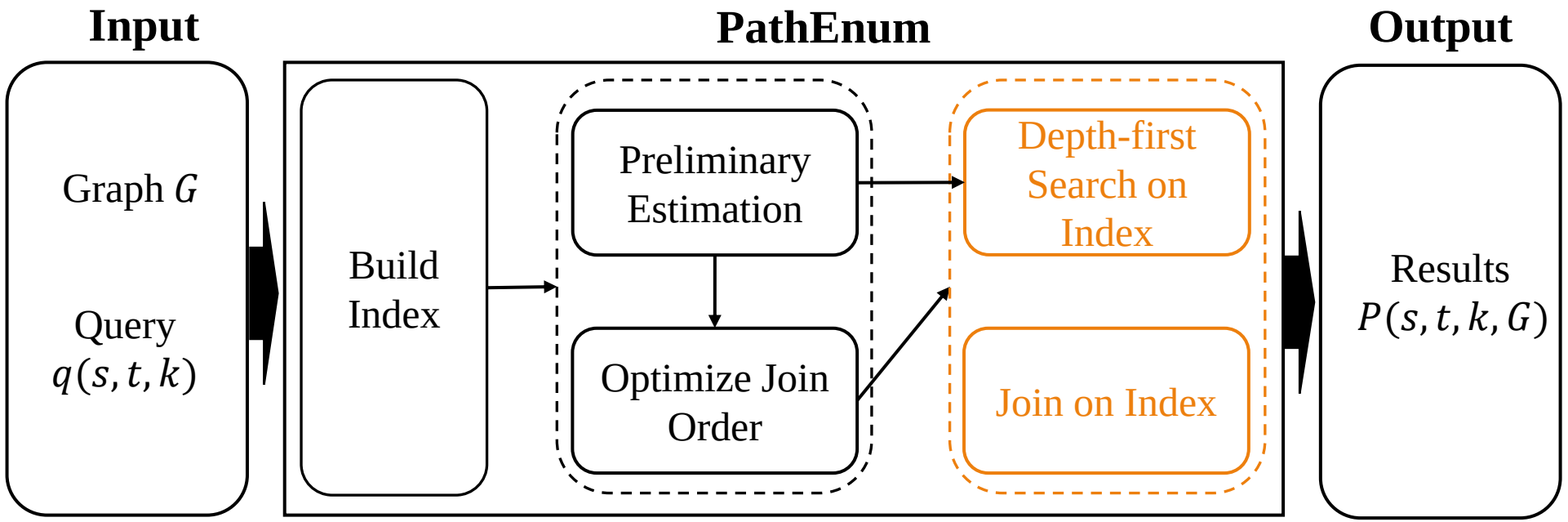
Design

- Search on the index based on the guidance of the optimizer.
  - Perform a DFS on the index from s.



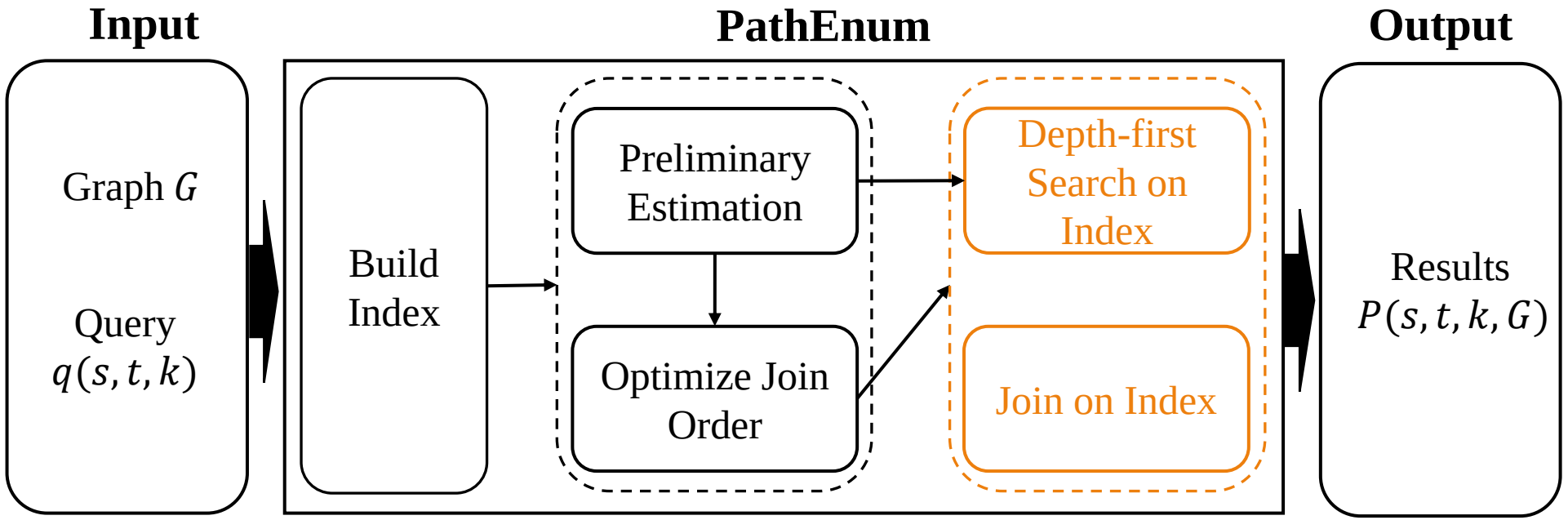
Design

- Search on the index based on the guidance of the optimizer.
  - Perform a DFS on the index from s.
  - Perform binary joins on the index.



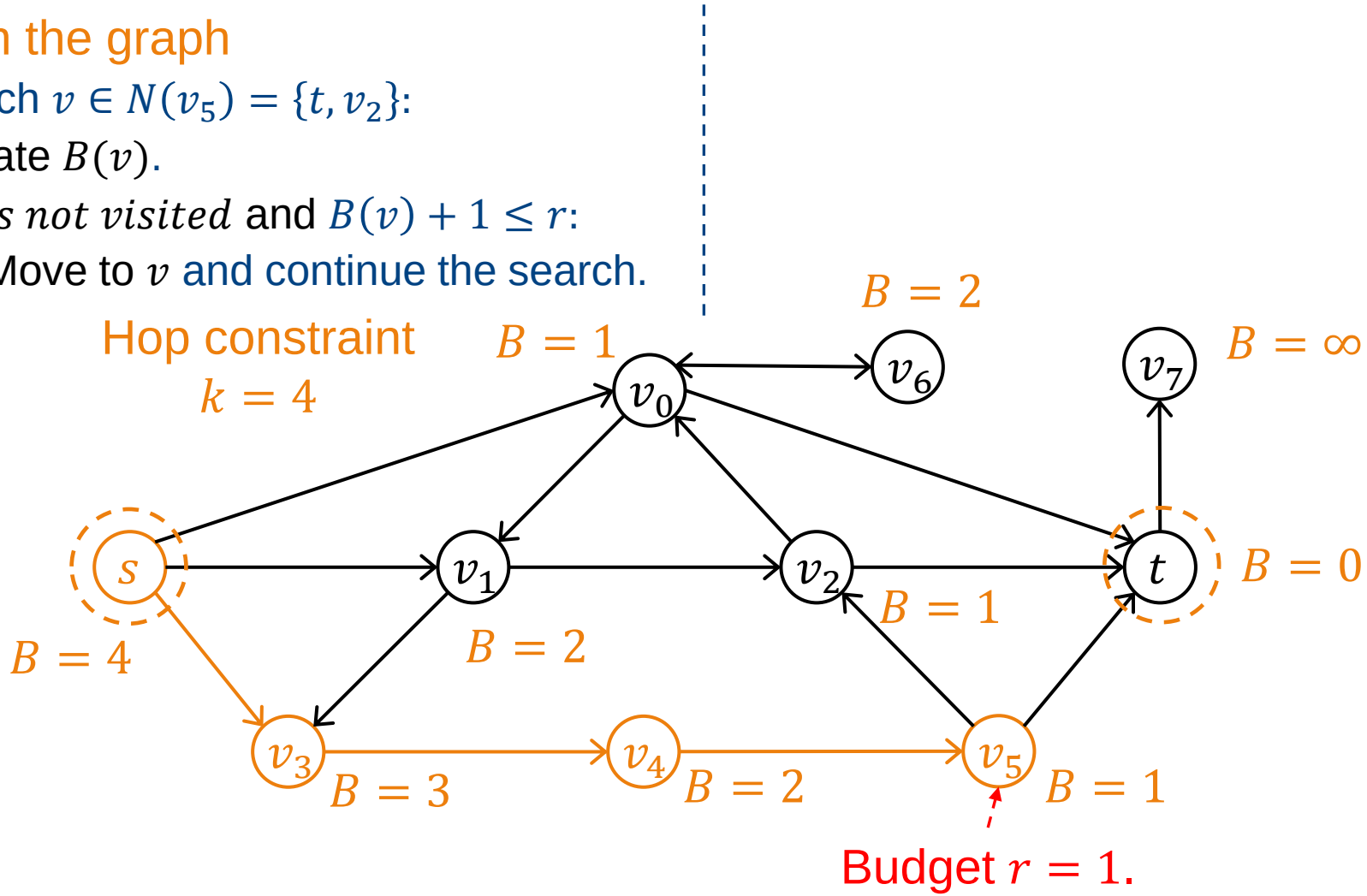
Design

- Search on the index based on the guidance of the optimizer.
  - Perform a DFS on the index from s.
  - Perform binary joins on the index.
  - Time complexity: $O(k \times |\delta_W|)$ (δ_W: walks W from s to t such that $L(W) \leq k$).



Comparison of Search

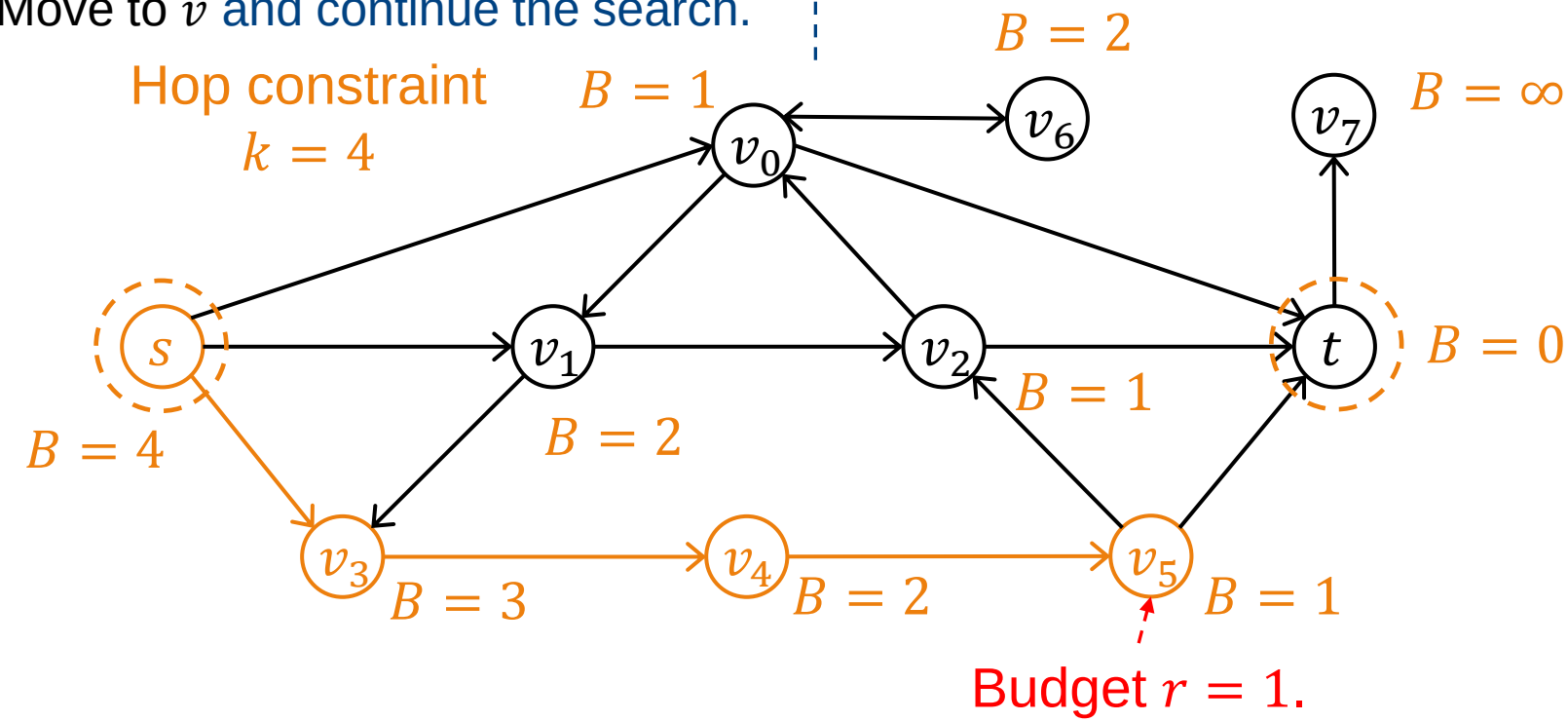
- Search on the graph
  - For each $v \in N(v_5) = \{t, v_2\}$:
    Update $B(v)$.
    If $v\ is\ not\ visited$ and $B(v) + 1 \leq r$:
      Move to v and continue the search.



Comparison of Search

- Search on the graph
  - For each $v \in N(v_5) = \{t, v_2\}$:
    Update $B(v)$.
    If v *is not visited* and $B(v) + 1 \leq r$:
    Move to v and continue the search.

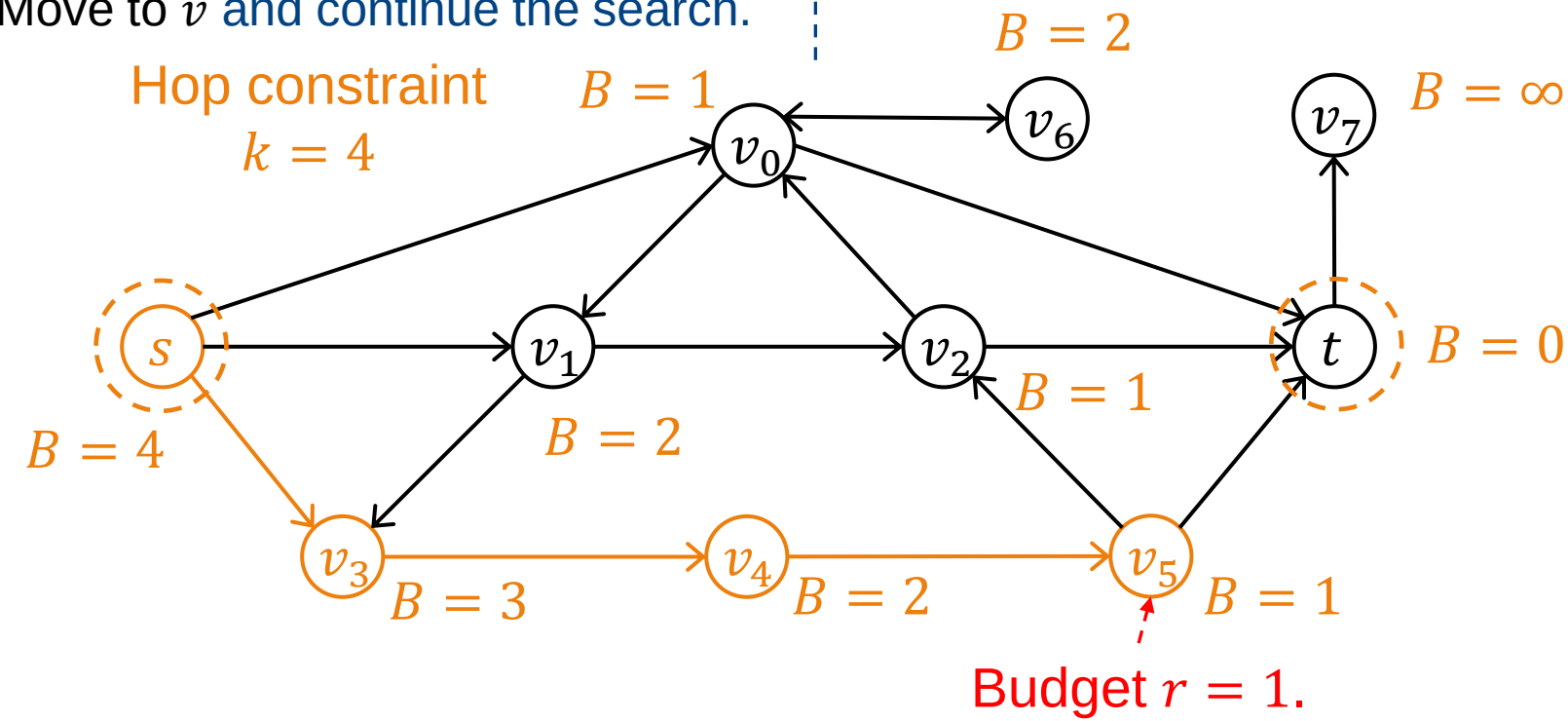
- Search on the index
  - For each $v \in I(v_5, r-1) = \{t\}$:
    If v *is not visited*:
    Move to v and continue the search.



Comparison of Search

- Search on the graph
  - For each $v \in N(v_5) = \{t, v_2\}$:
    Update $B(v)$.
    If v *is not visited* and $B(v) + 1 \leq r$:
    Move to v and continue the search.

- Search on the index
  - For each $v \in I(v_5, r-1) = \{t\}$:
    If v *is not visited*:
    Move to v and continue the search.

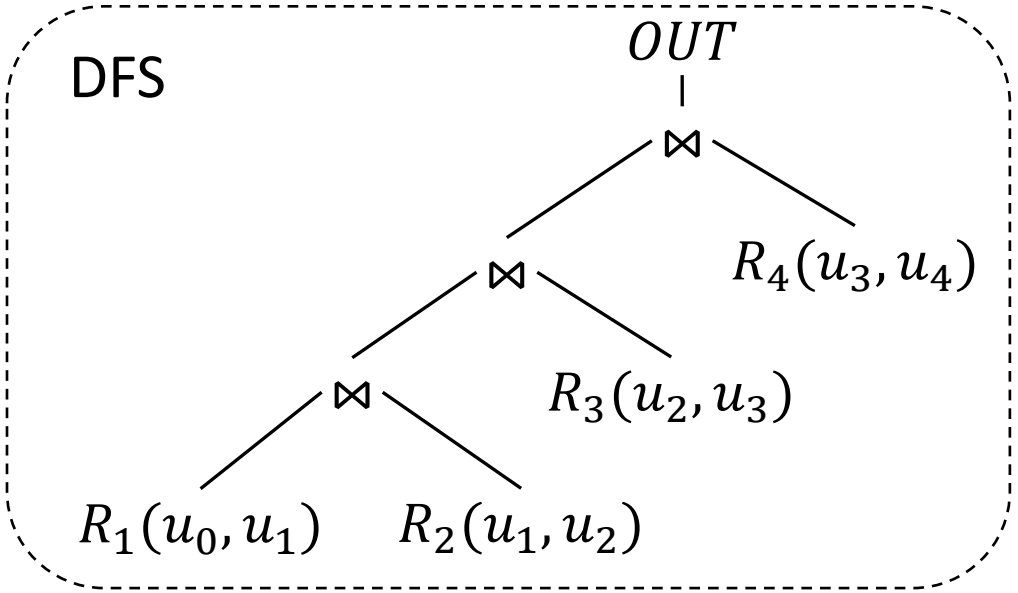


Comparison of Search Order

- Model a HcPE query as a chain join.
  - $Q \coloneqq R_1(u_0, u_1) \bowtie R_2(u_1, u_2) \bowtie R_3(u_2, u_3) \bowtie R_4(u_3, u_4)$

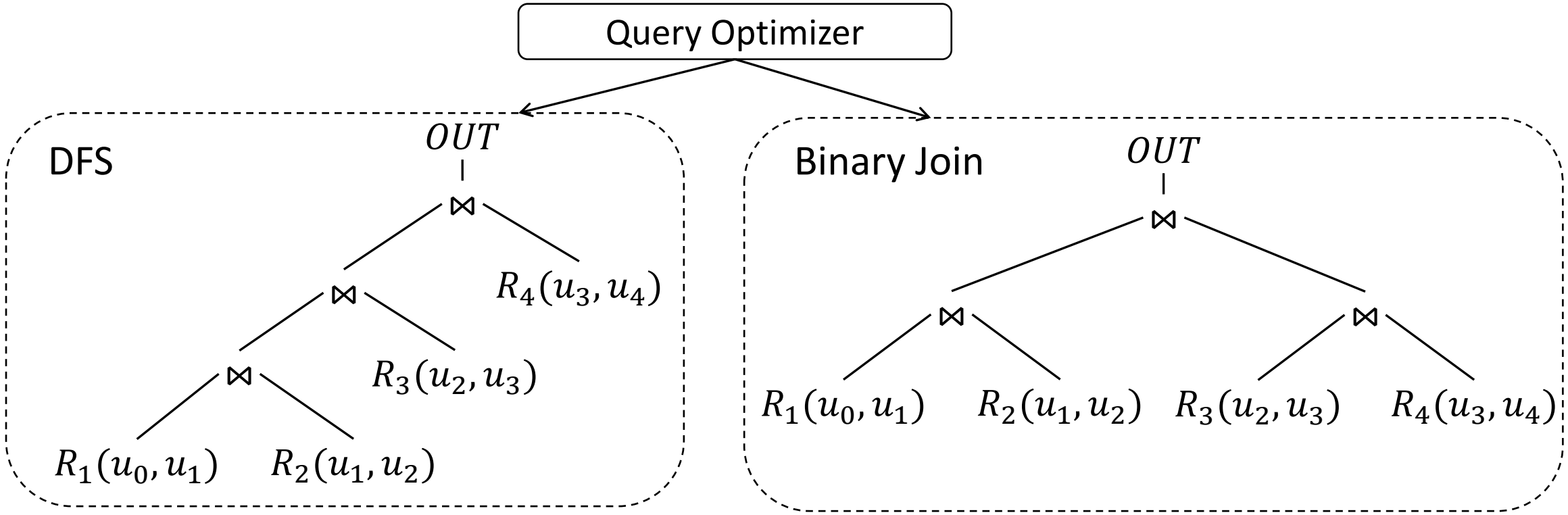
Comparison of Search Order

- Model a HcPE query as a chain join.
  - $Q \coloneqq R_1(u_0, u_1) \bowtie R_2(u_1, u_2) \bowtie R_3(u_2, u_3) \bowtie R_4(u_3, u_4)$



Comparison of Search Order

- Model a HcPE query as a chain join.
  - $Q := R_1(u_0, u_1) \bowtie R_2(u_1, u_2) \bowtie R_3(u_2, u_3) \bowtie R_4(u_3, u_4)$



Recap

- Existing Solutions:

- PathEnum:

Recap

- Existing Solutions:
  - Conduct filtering in the search to achieve polynomial delay.

- PathEnum:
  - Build a light-weight index to keep the search simple and efficient.

Recap

- Existing Solutions:
  - Conduct filtering in the search to achieve polynomial delay.
  - Perform a DFS on the graph and dynamically update the barrier.

- PathEnum:
  - Build a light-weight index to keep the search simple and efficient.
  - Search on the index with the guidance of a cost-based query optimizer.

Recap

- Existing Solutions:
  - Conduct filtering in the search to achieve polynomial delay.
  - Perform a DFS on the graph and dynamically update the barrier.
  - $O(k \times |E| \times |\delta_P|)$, δ_P denotes paths P from s to t such that $L(P) \leq k$.

- PathEnum:
  - Build a light-weight index to keep the search simple and efficient.
  - Search on the index with the guidance of a cost-based query optimizer.
  - $O(k \times |\delta_W|)$, δ_W denotes walks W from s to t such that $L(W) \leq k$.

Experimental Setup

- Workload:
  - 14 real-world graphs with $|E|$ varying from $314K$ to $17M$.
  - 1000 queries randomly generated.
  - k varies from 3 to 8 and the default value is 6.
- Metrics:
  - Response time: the elapsed time on finding 1000 results.
  - Query time: the elapsed time on completing the query.
- Counterpart:
  - BC-DFS/BC-JOIN [VLDB'20].
- Open Source:
  - *https://github.com/Xtra-Computing/PathEnum*

Summary of Results

- Response time:
  - 14.2 - 358.5X speedup.
  - Less than 1 second (generally less than 100 ms).

Summary of Results

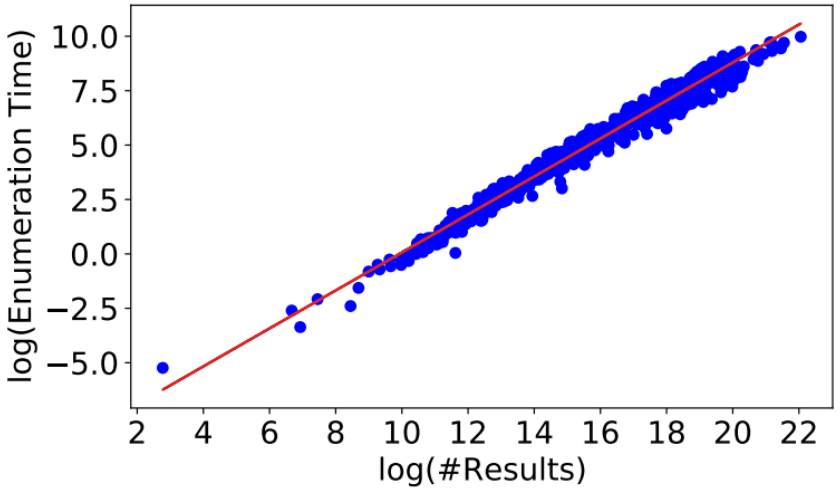
- Response time:
  - 14.2 - 358.5X speedup.
  - Less than 1 second (generally less than 100 ms).

- Query time:
  - 1.9 - 240.7X speedup.
  - Improve the throughput from around 10^5 to 10^8 results/per second.

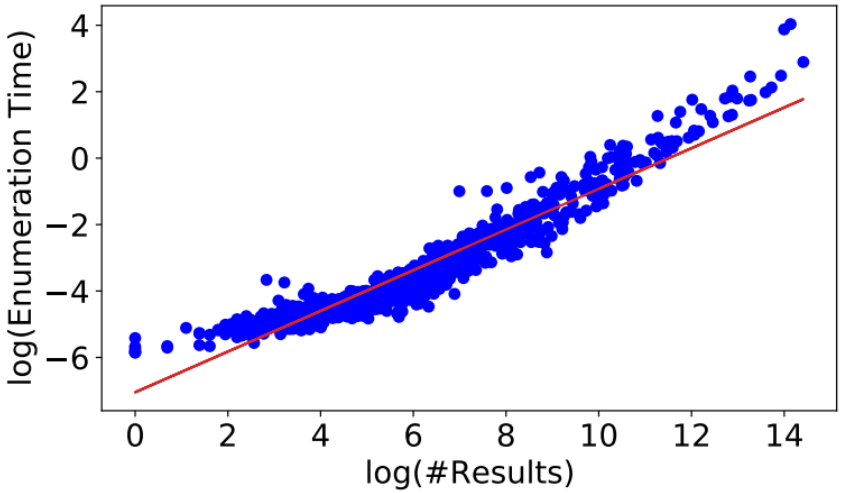
Summary of Results

- Response time:
  - 14.2 - 358.5X speedup.
  - Less than 1 second (generally less than 100 ms).
- Query time:
  - 1.9 - 240.7X speedup.
  - Improve the throughput from around 10^5 to 10^8 results/per second.
- Query time variance:
  - From 0.1 ms to several minutes.

Why are some queries time consuming?



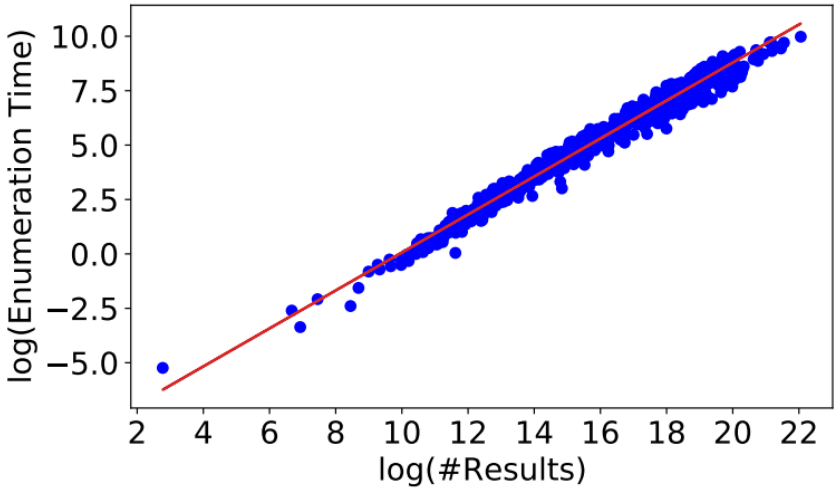
Epinsion ($|V| = 75K, |E| = 508K$)



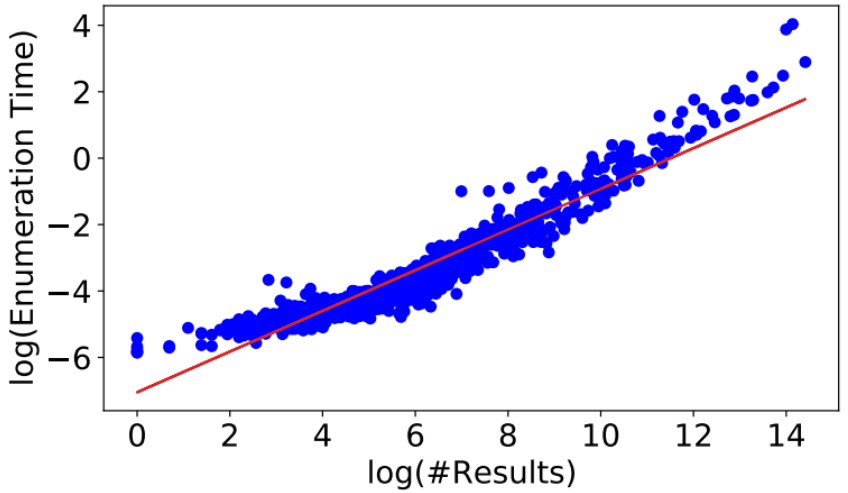
Google ($|V| = 876K, |E| = 5M$)

Why are some queries time consuming?

- Enumeration time is closely related to the number of results.
- Some queries have a huge number of results.



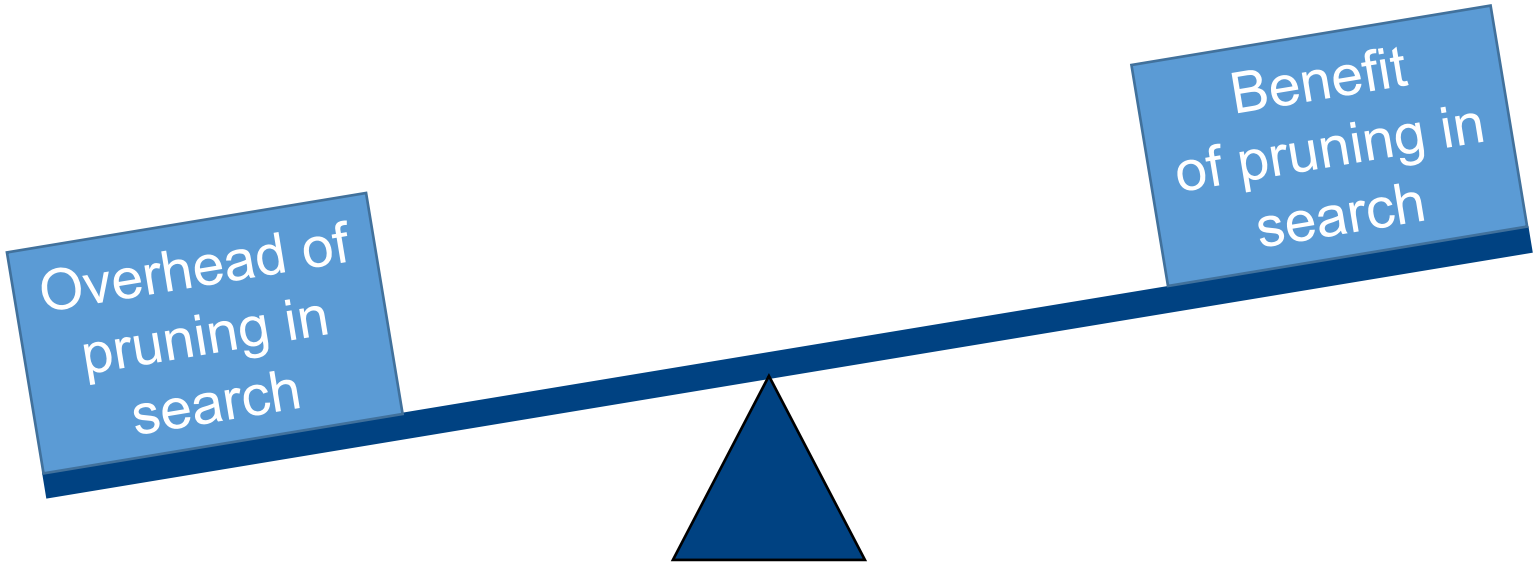
Epinsion ($|V| = 75K, |E| = 508K$)



Google ($|V| = 876K, |E| = 5M$)

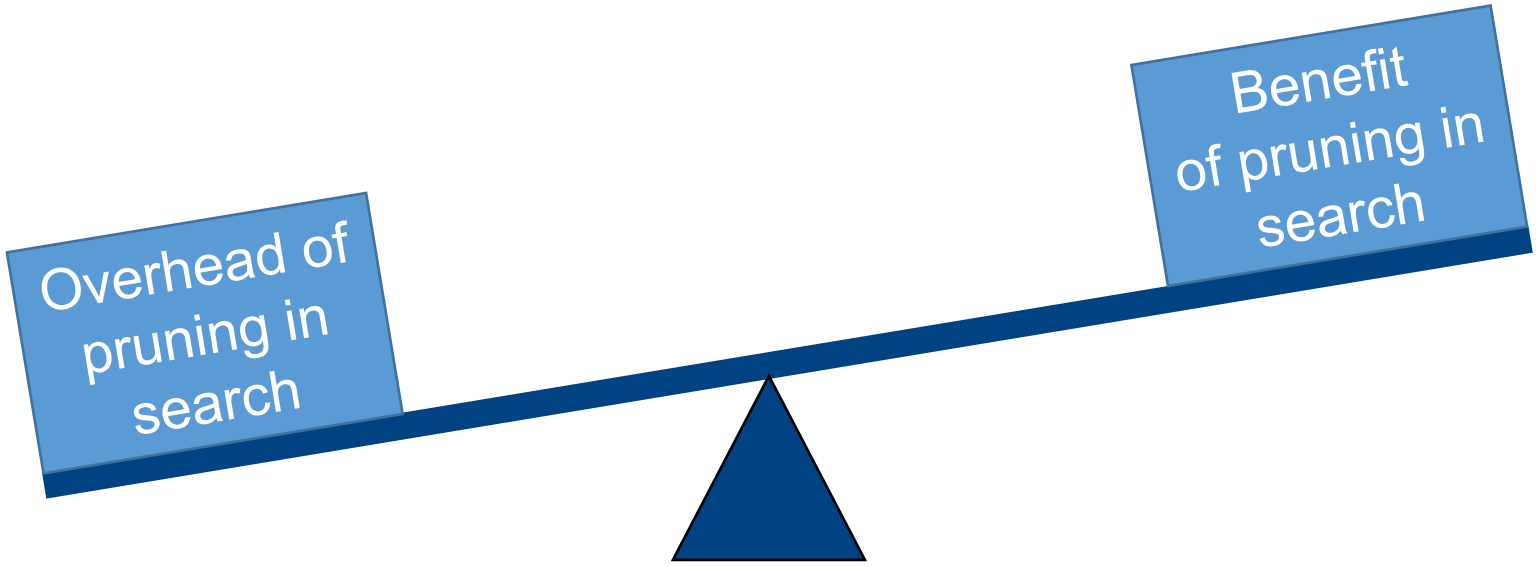
Takeaway

- Keep the search simple and efficient.



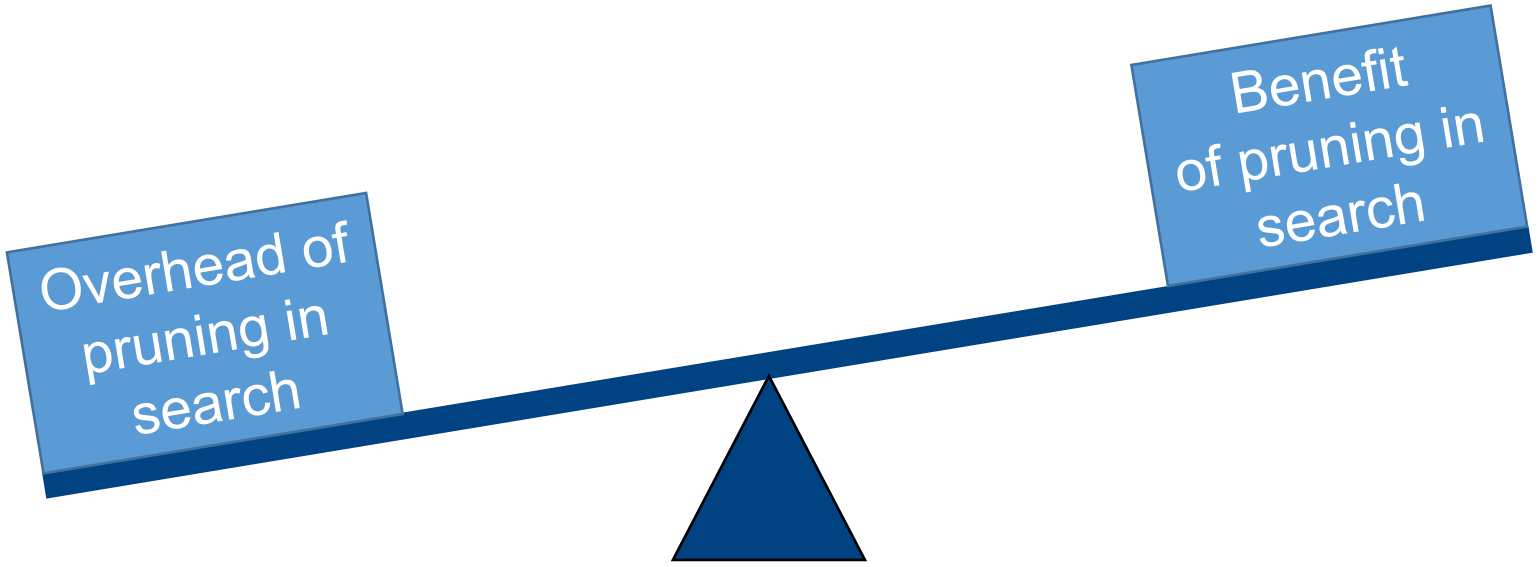
Takeaway

- Keep the search simple and efficient.
- Query-dependent index can significantly improve the performance.



Takeaway

- Keep the search simple and efficient.
- Query-dependent index can significantly improve the performance.
- Query time is closely related to the number of results.



Summary

- *PathEnum*, an efficient approach for HcPE.

- *PathEnum*'s key components include
  - A light-weight index for input query.
  - A two-level query optimizer with a join-based cost model.
  - A search engine on the index.

- Up to two orders of magnitude speedup over state of the art.

Future Work

- Scalability evaluation with a graph with 2 billion edges.

Future Work

- Scalability evaluation with a graph with 2 billion edges.
  - Achieve a high throughput (up to 10^7 results/second).

Future Work

- Scalability evaluation with a graph with 2 billion edges.
  - Achieve a high throughput (up to 10^7 results/second).
  - The response time can be long because of the BFS (up to tens of seconds)…

Future Work

- Scalability evaluation with a graph with 2 billion edges.
  - Achieve a high throughput (up to 10^7 results/second).
  - The response time can be long because of the BFS (up to tens of seconds)…

How to reduce the response time on very large graphs?

Selected References

[FATF'13] Financial Action Task Force. 2013. FATF Report: Money Laundering and Terrorist Financing Vulnerabilities of Legal Professionals. Paris: FATF (2013).

[AAAI'20] Xiangfeng Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hooi, He Huang, and Xueqi Cheng. 2020. FlowScope: Spotting Money Laundering Based on Graphs. In AAAI. 4731–4738.

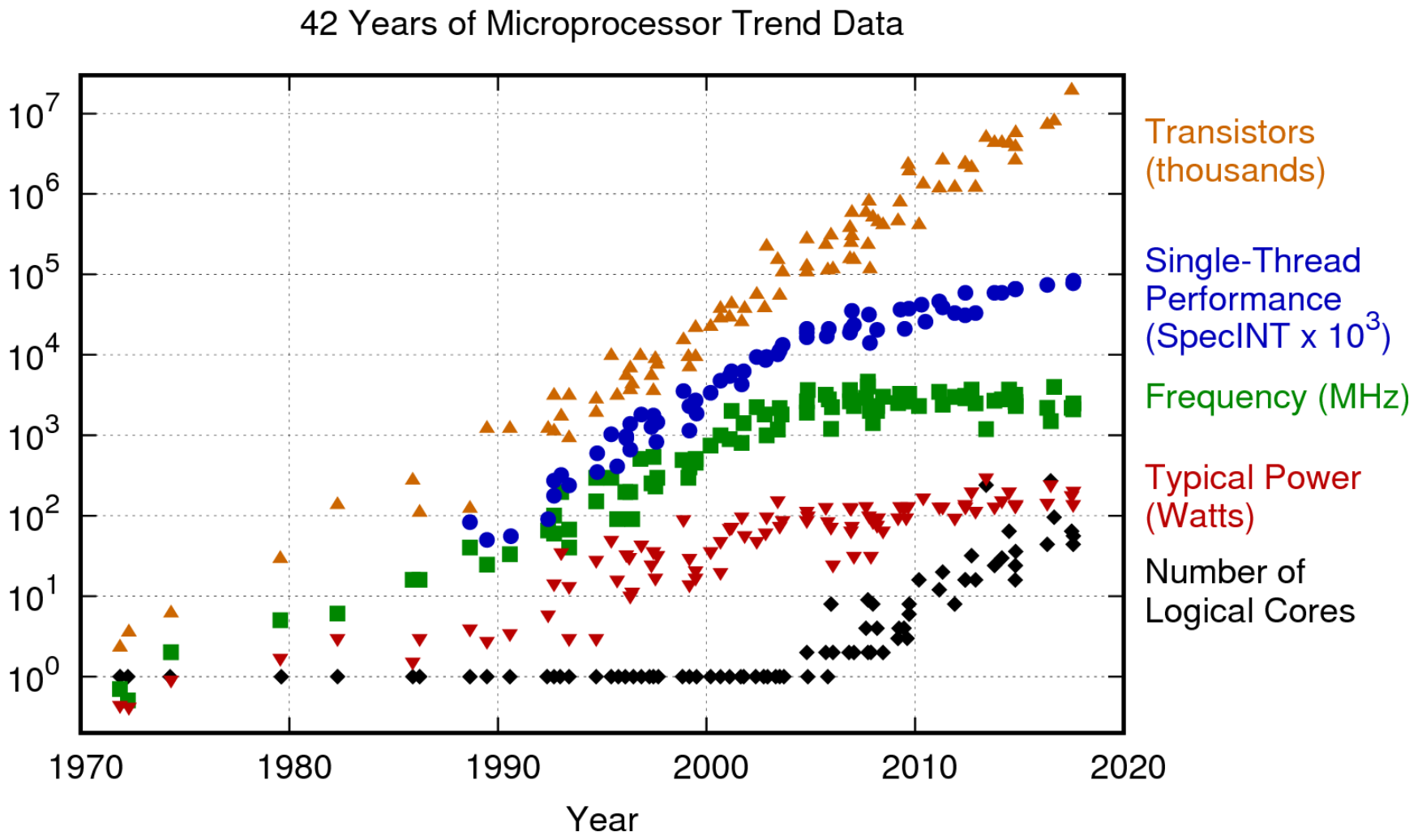
[VLDB'18] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. Proceedings of the VLDB Endowment 11, 12 (2018).

[VLDB'20] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Towards bridging theory and practice: hop-constrained st simple path enumeration. Proceedings of the VLDB Endowment 13, 4 (2019).

Outline

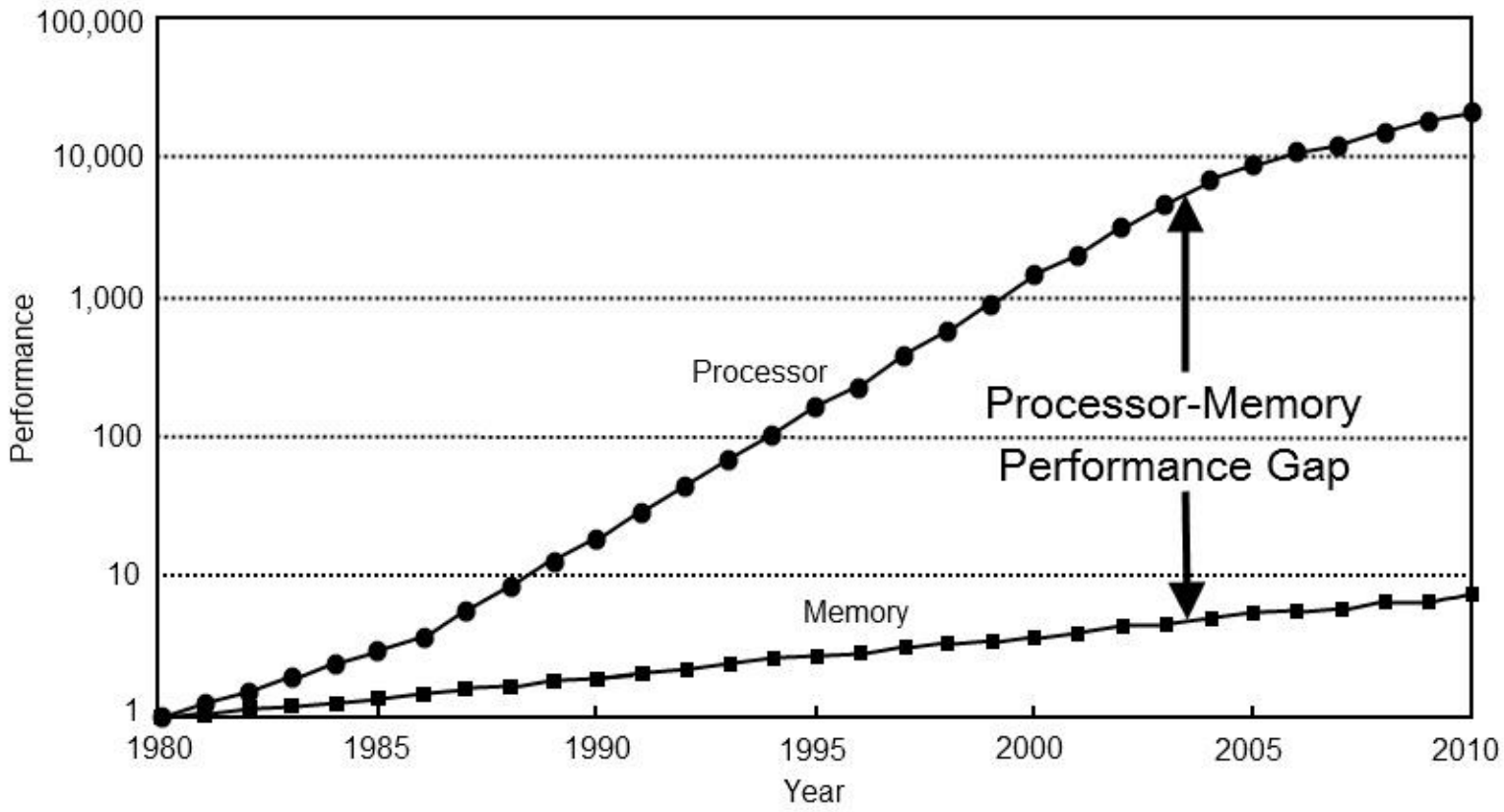
- Benchmark
  - Background
  - In-Memory Subgraph Matching: An In-Depth Study. SIGMOD 2020.
- Algorithms
  - RapidMatch: A Holistic Approach to Subgraph Query Processing. VLDB 2021.
  - PathEnum: Towards Real-Time Hop Constraint s-t Path Enumeration. SIGMOD 2021.
- Parallelization
  - LIGHT: Parallelizing Subgraph Query Processing. ICPADS 2018 & ICDE 2019.
  - ThunderRW: An In-Memory Graph Random Walk Engine. VLDB 2021.

Multi(Many)-Core Era

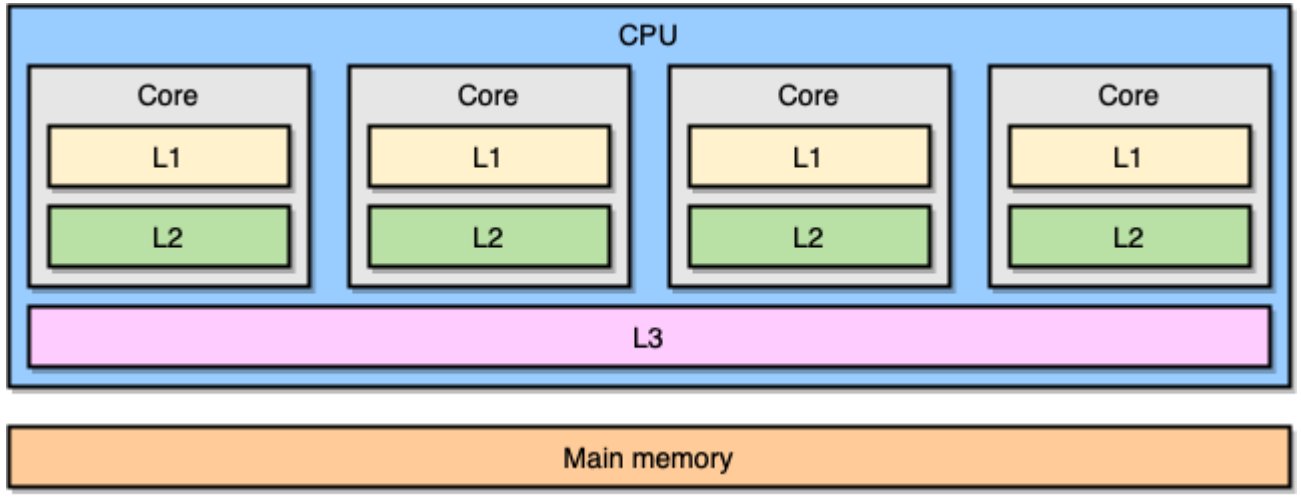


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Performance Gap Between Processor and Memory



Modern Processor Architecture



L1 cache hit latency:
5 cycles / 2.6 GHz = **1.92 ns**

L2 cache hit latency:
11 cycles / 2.6 GHz = **4.23 ns**

L3 cache hit latency:
34 cycles / 2.6 GHz = **13.08 ns**

Memory access latency:
L3 + Memory Access = **~60-100 ns**

Up to 50X performance gap!

Figure source: https://teivah.medium.com/go-and-cpu-caches-af5d32cc5592.

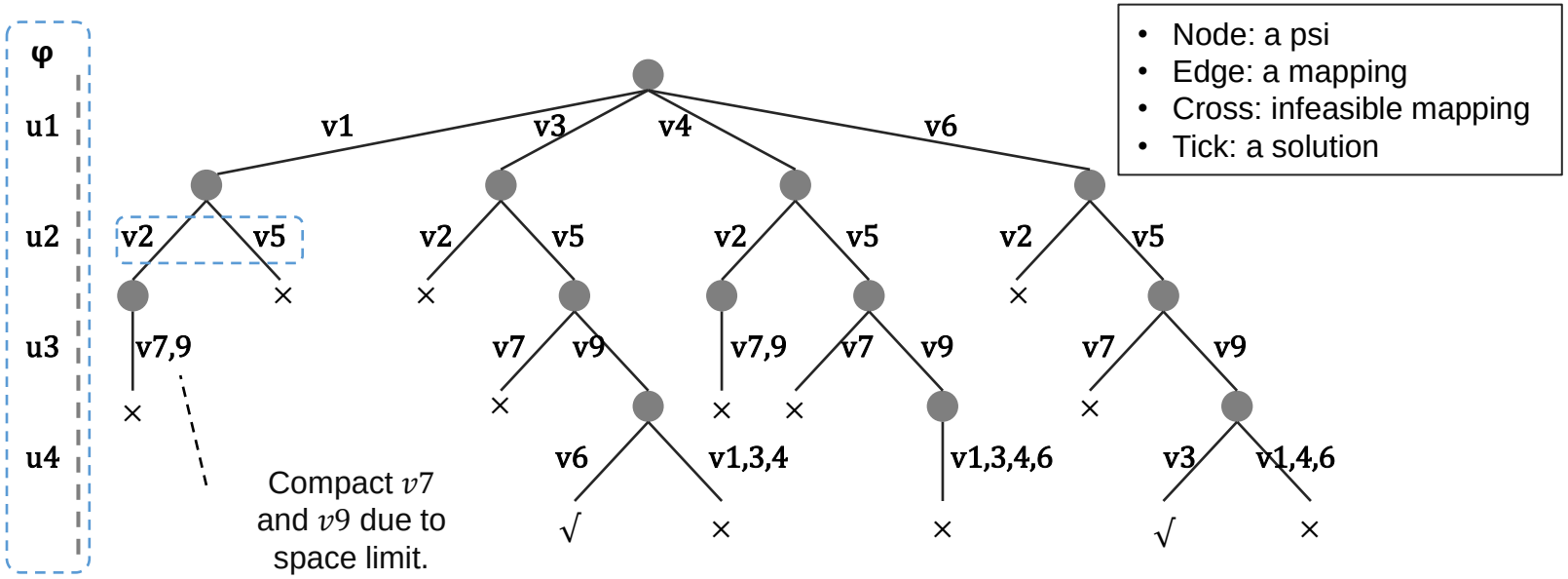
Parallelizing Subgraph Query Processing on a Single Machine

Shixuan Sun, Qiong Luo. ICPADS 2018.

Shixuan Sun, Yulin Che, Lipeng Wang, Qiong Luo. ICDE 2019.

Research Focus of Sequential Algorithms

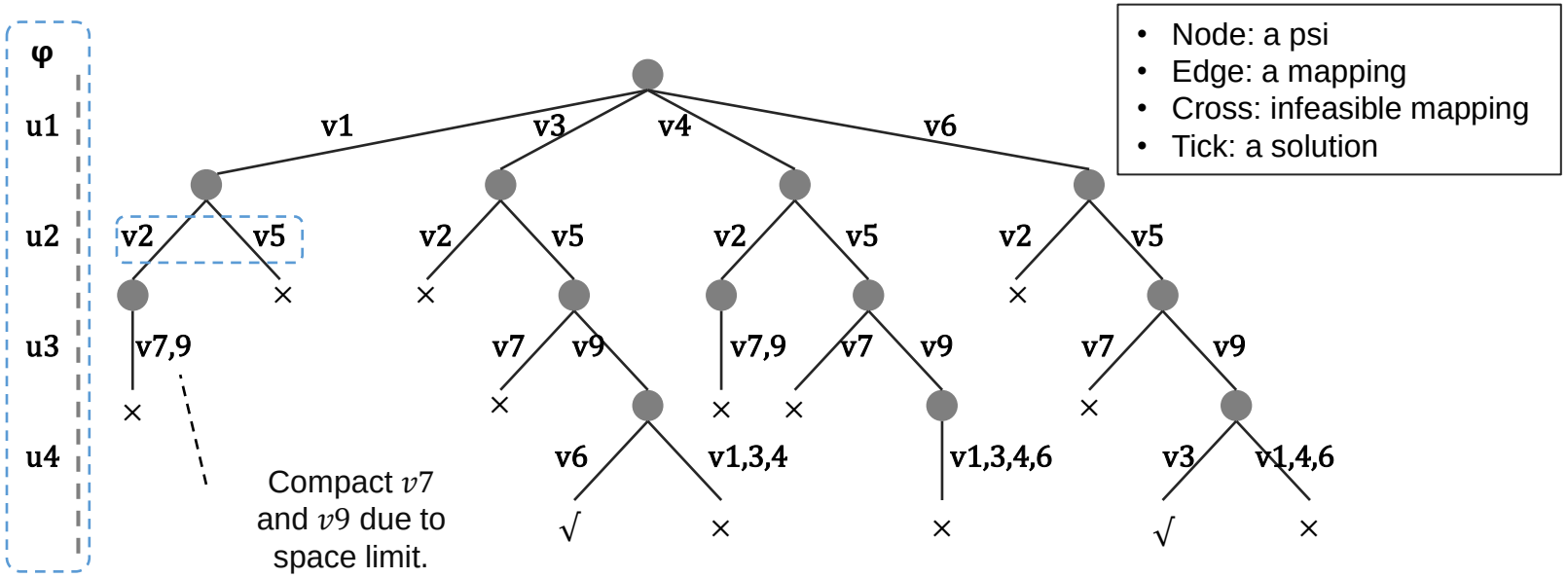
- Optimize the matching order.
- Minimize the search breadth (branches) of each state.



Research Focus of Sequential Algorithms

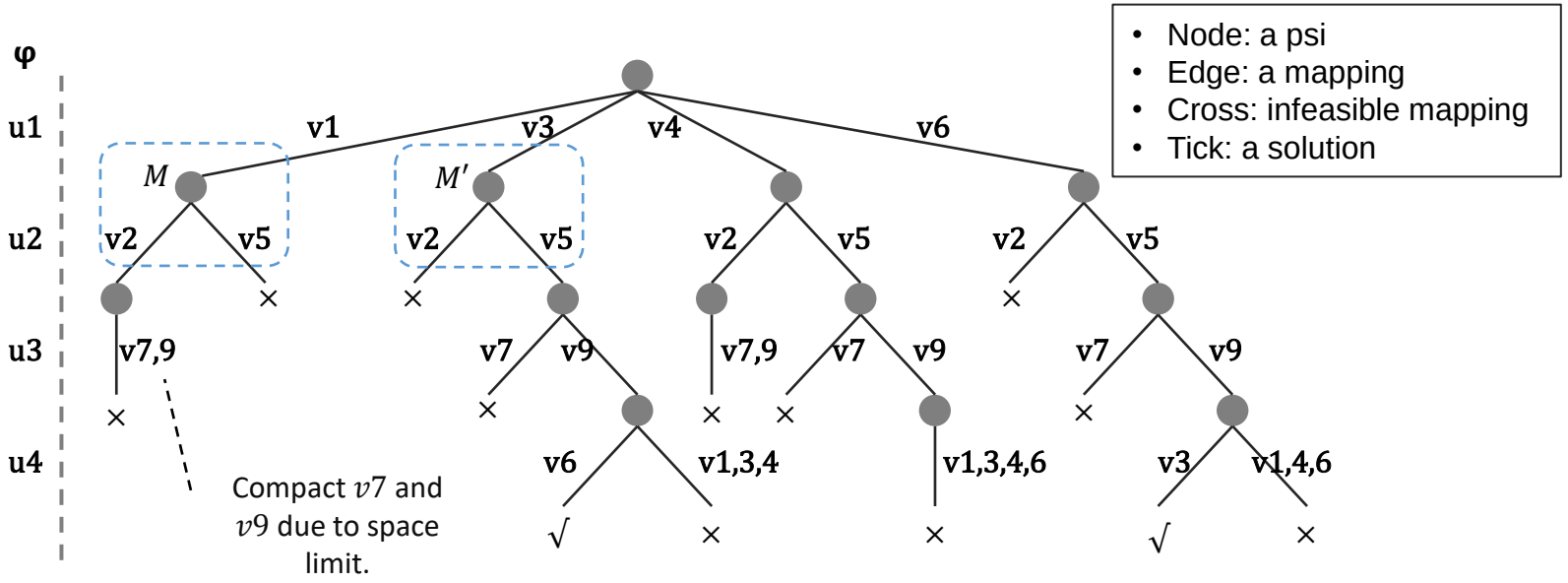
- Optimize the matching order.
- Minimize the search breadth (branches) of each state.

We focus on efficiently exploring the tree in parallel.



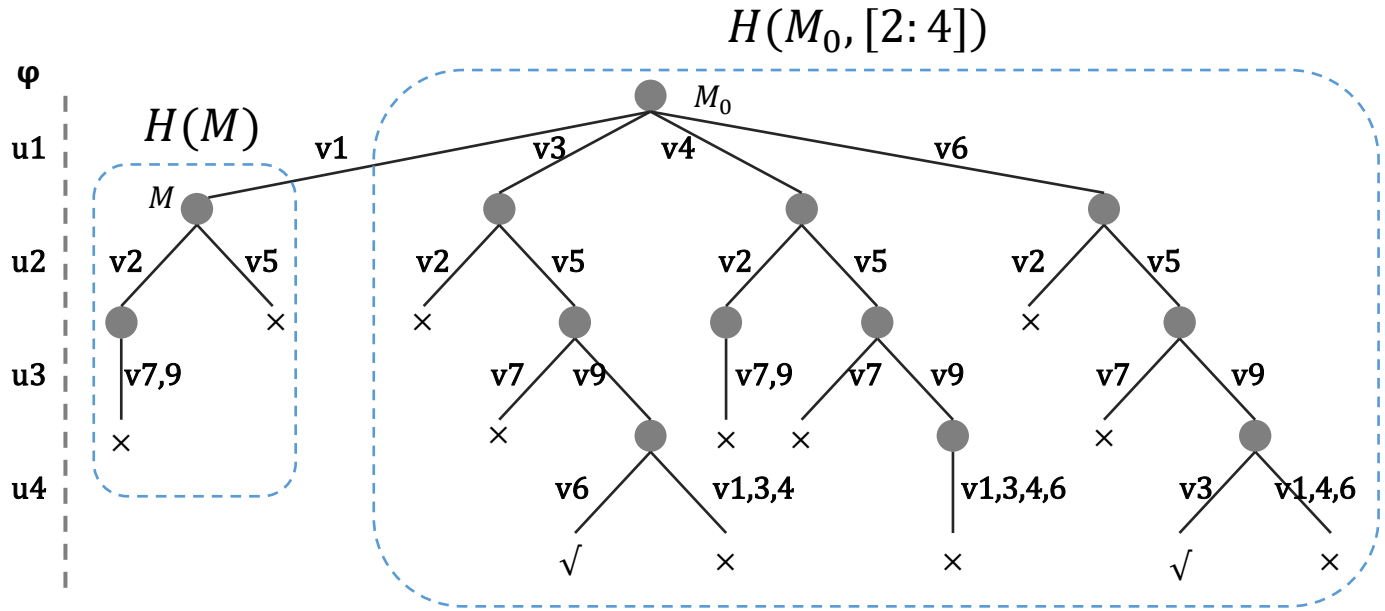
Fine-Grained Parallelism

- **Observation:** Each node (state) can be expanded independently.
- **Solution:** Regard each node as the basic task unit.
- **Cons:**
  - The fine-grained parallel method results in a large number of light weight tasks.
  - The approach can incur a high communication overhead.



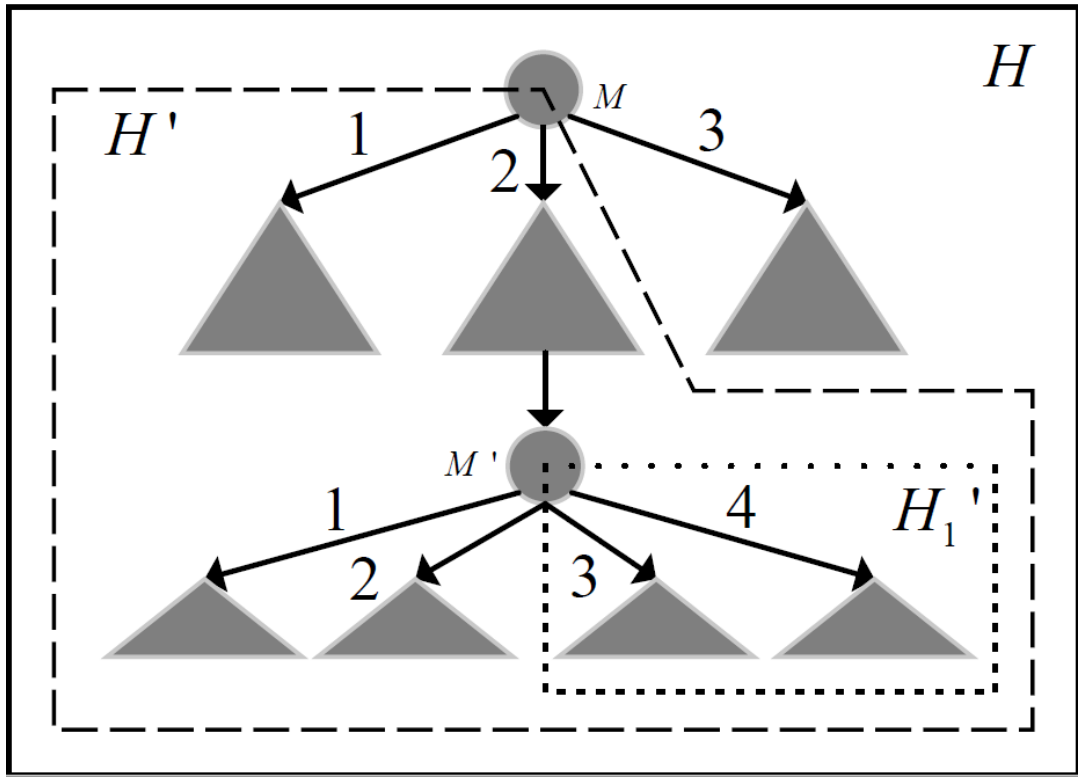
Coarse-Grained Parallelism

- **Observation:** The subtree rooted at a node can be explored independently.
- **Solution:** Regard the subtree rooted at M, denoted as $H(M)$, as a parallel task. $H(M)$ can be further divided into more fine grained ones by taking part of the candidates, denoted as $H(M, [i:j])$.



Parallel Task

- We take coarse-grained tasks instead of fine-grained ones.
  - Expand each subtree independently in a depth-first search method.
  - Example: H, H' and $H_1{}'$ can be explored concurrently by different workers.



Load Balancing

- It is hard to assign equal amounts of workload to workers at the beginning (static load balancing), because H is constructed on the fly and irregular.

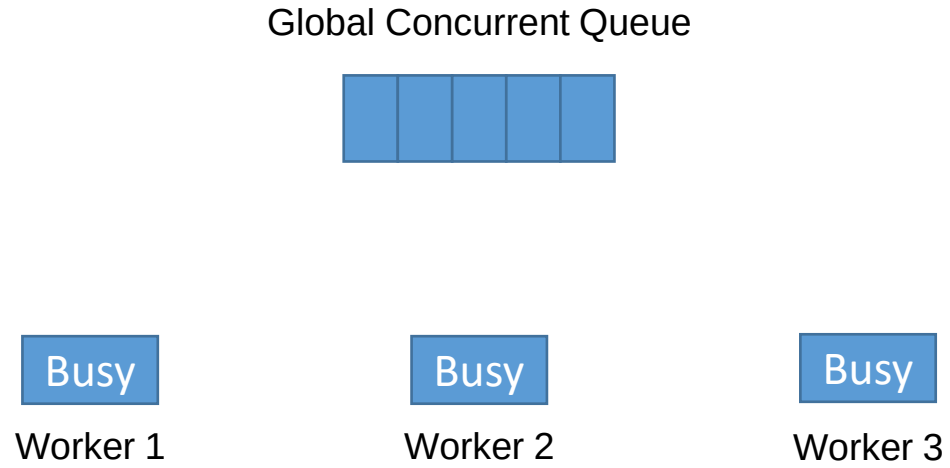
- We design a dynamic load balancing approach to resolve the load imbalance problem.

Load Balancing

- Adopt a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- Adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
  - Busy workers will donate part of its task when they find that the queue is empty and there are idle workers.

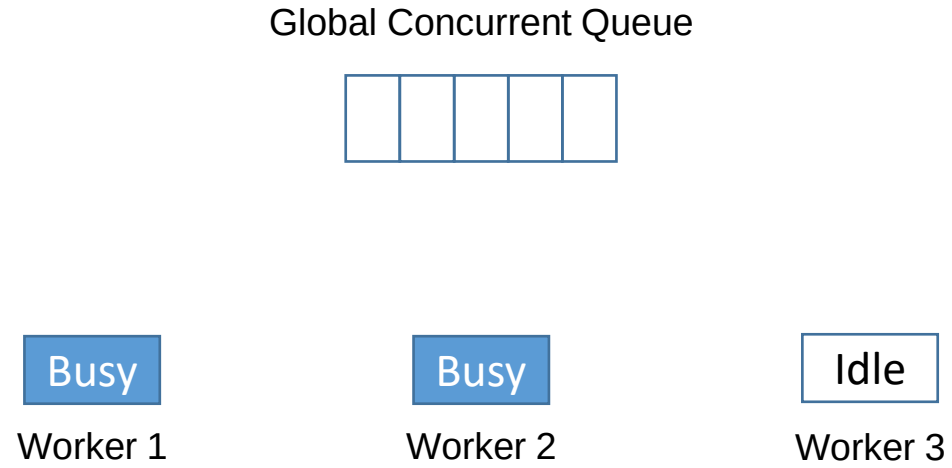
Load Balancing

- Adopt a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- Adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
  - Busy workers will donate part of its task when they find that the queue is empty and there are idle workers.



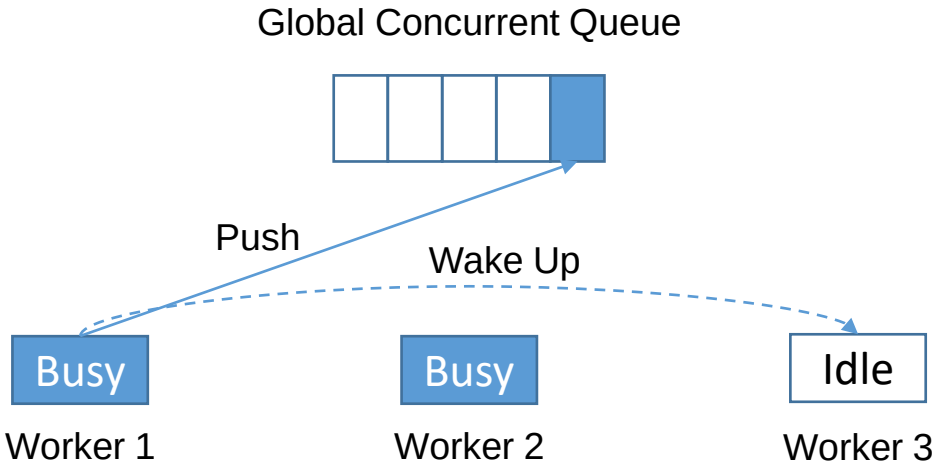
Load Balancing

- Adopt a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- Adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
  - Busy workers will donate part of its task when they find that the queue is empty and there are idle workers.



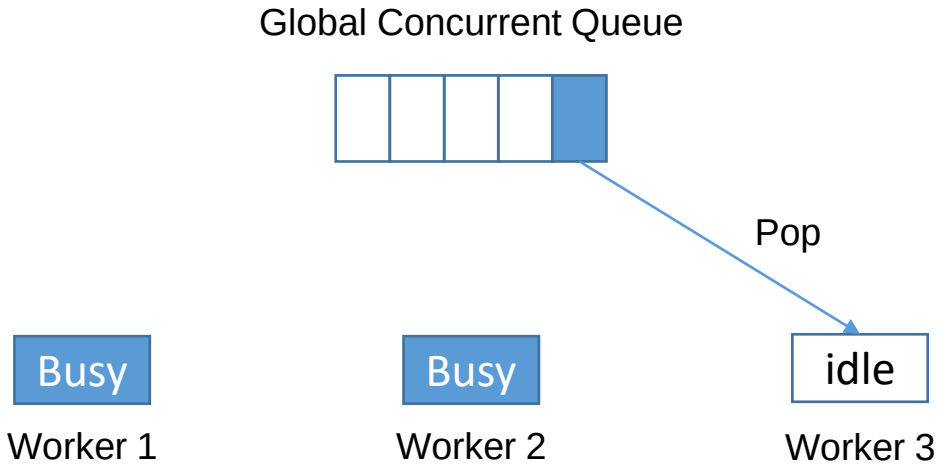
Load Balancing

- Adopt a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- Adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
  - Busy workers will donate part of its task when they find that the queue is empty and there are idle workers.



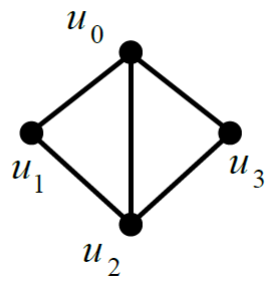
Load Balancing

- Adopt a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- Adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
  - Busy workers will donate part of its task when they find that the queue is empty and there are idle workers.

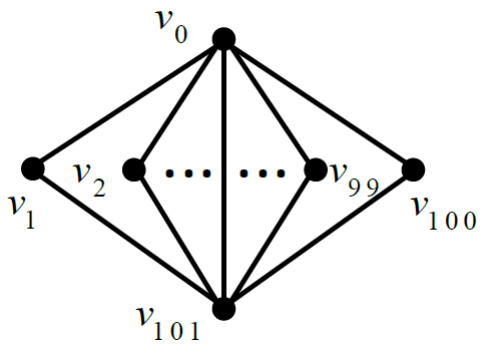


We find that there is a large amount of redundant computation in the unlabeled graph enumeration.

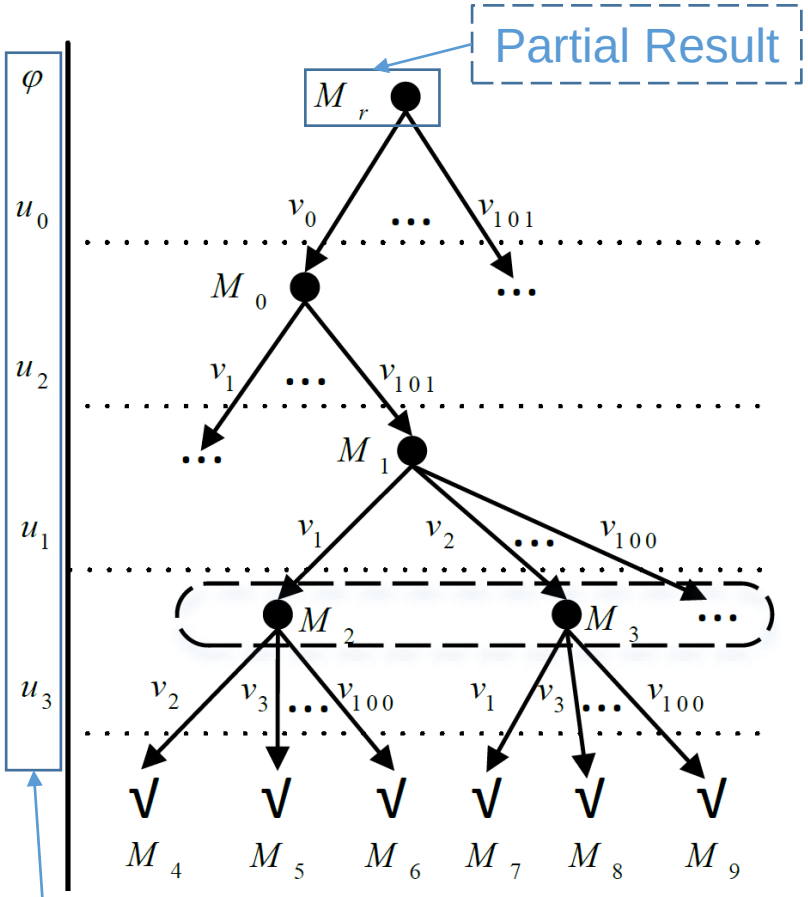
Example of SE



Query graph q



Data graph G

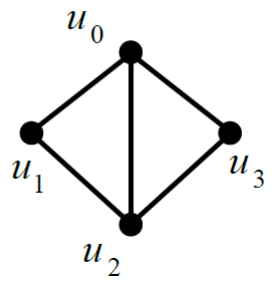


Search tree of SE

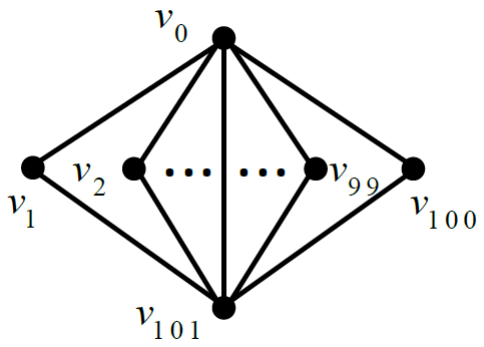
Given $u \in V(q)$ and φ, the backward neighbors $N_+^{\varphi}(u)$ of u contains the neighbors of u positioned before u in φ.

Example: $N_+^{\varphi}(u_1) = \{u_0, u_2\}$

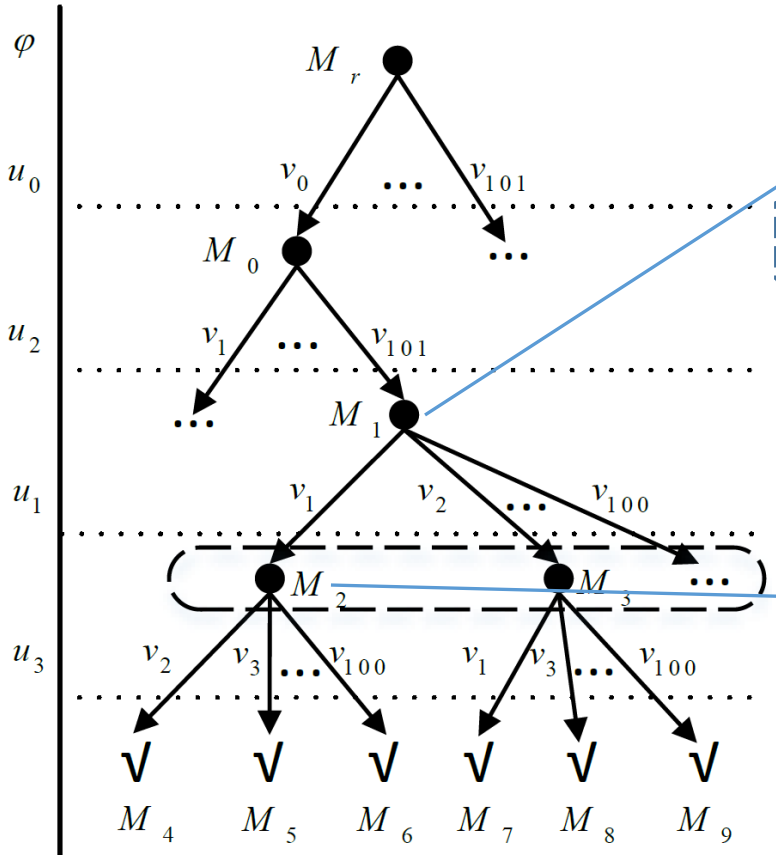
Example of SE



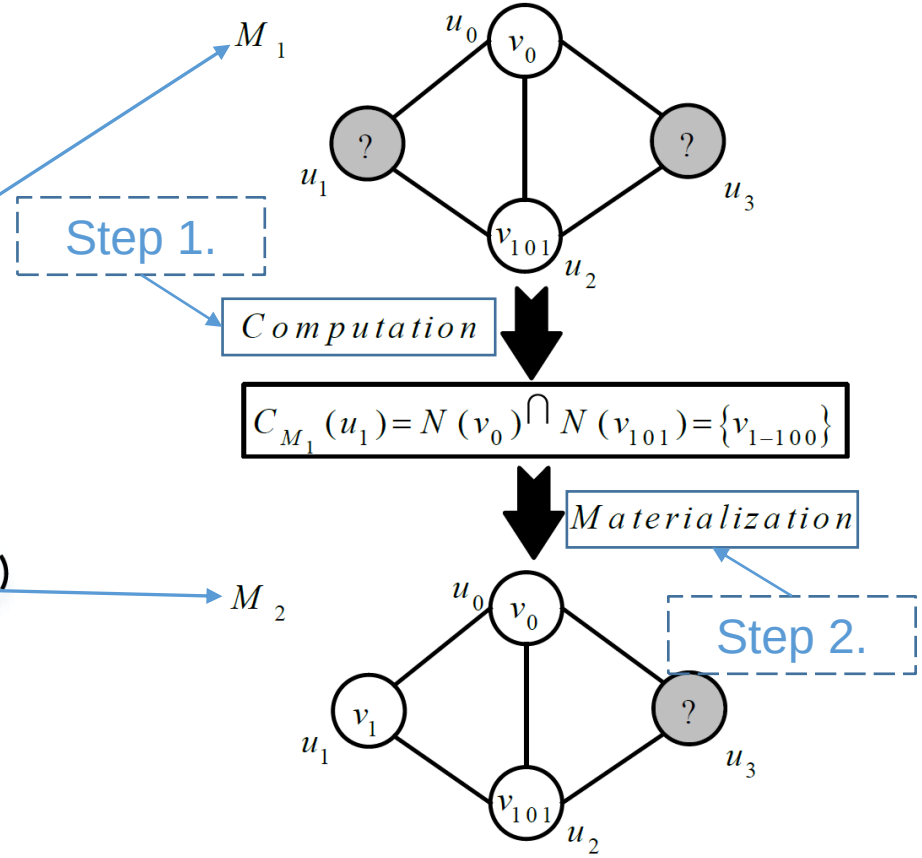
Query graph q



Data graph G

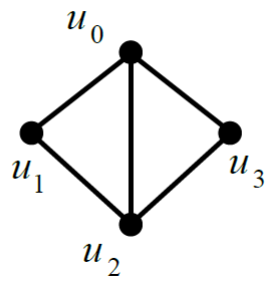


Search tree of SE

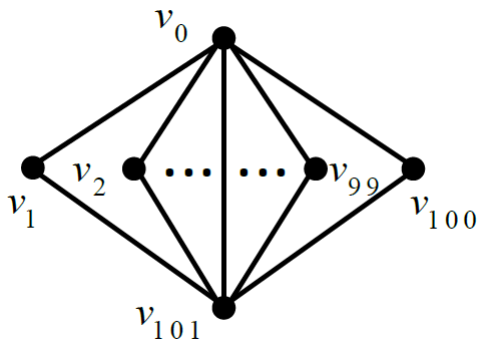


Expand a partial result

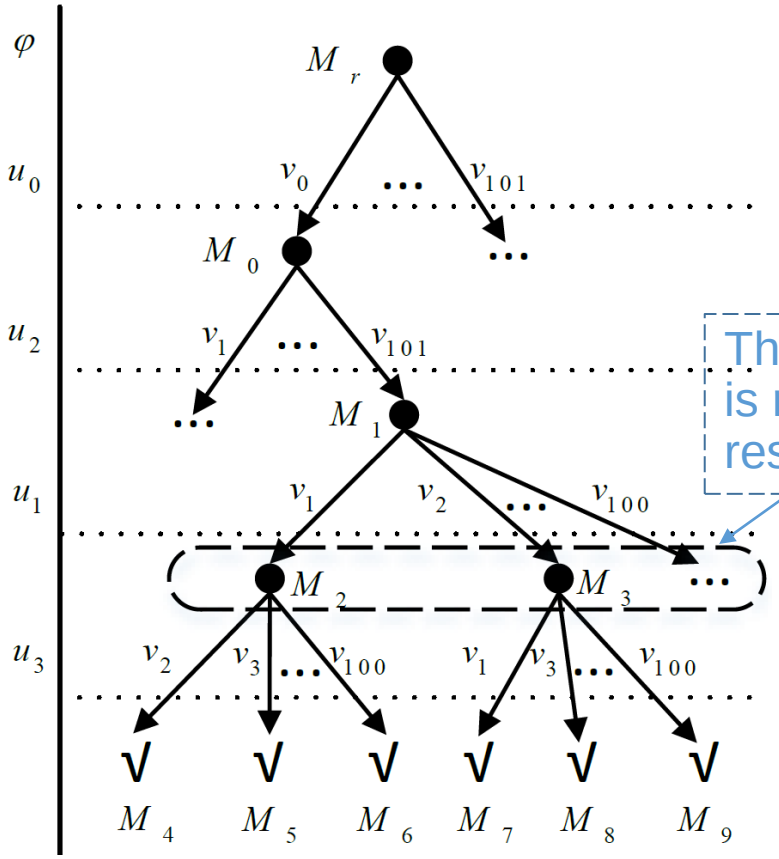
Observation One



Query graph q



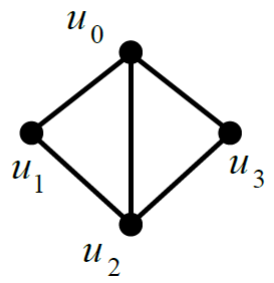
Data graph G



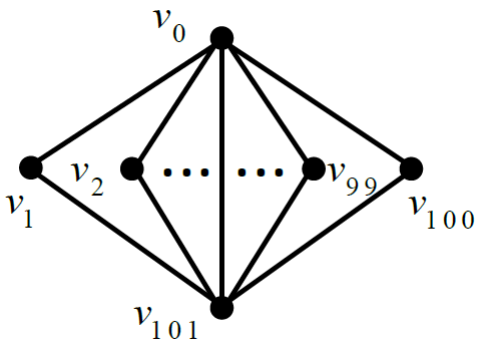
Search tree of SE

The same set intersection $N(v_0) \cap N(v_{101})$ is repeated in the computation of partial results in the dashed rectangle for u_3.

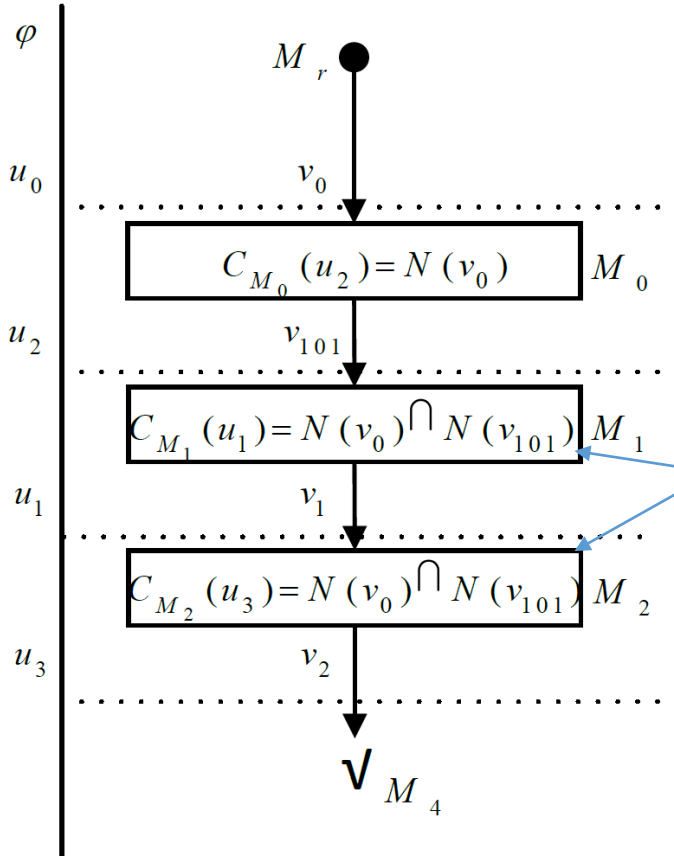
Observation Two



Query graph q



Data graph G



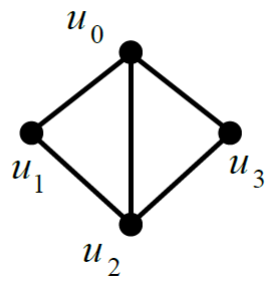
Search path of SE

Given partial results M_1 and M_2, the same set intersection $N(v_0) \cap N(v_{101})$ is repeated in the computation of candidates of u_1 and u_3.

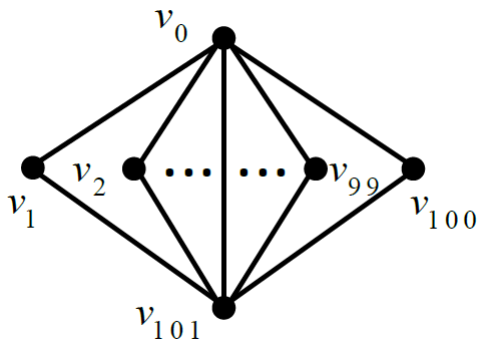
Lazy Materialization

- We propose the lazy materialization subgraph enumeration algorithm, called **LIGHT**.
  - Separate the computation and the materialization.
  - Keep the order of the computation unchanged.
  - Delay the materialization until some computation requires it.

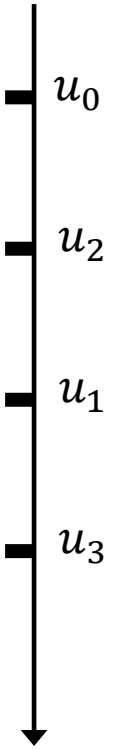
Example of Lazy Materialization



Query graph q

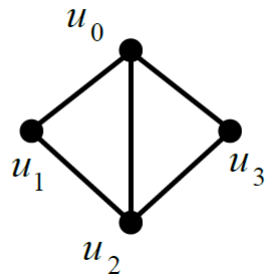


Data graph G

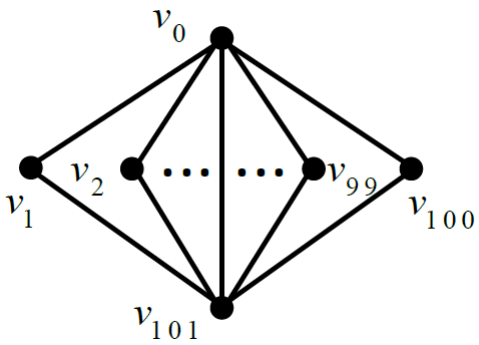


Enumeration order

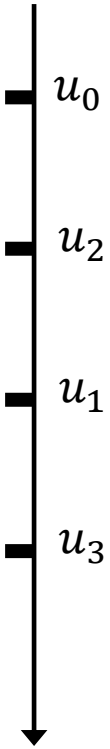
Example of Lazy Materialization



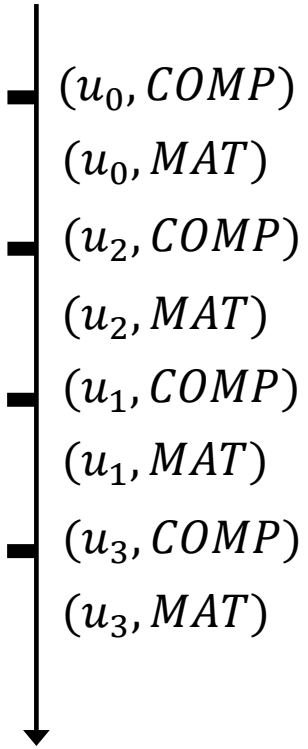
Query graph q



Data graph G

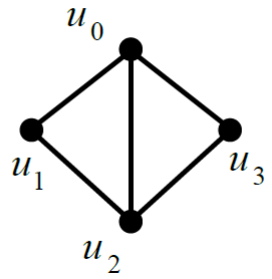


Enumeration order

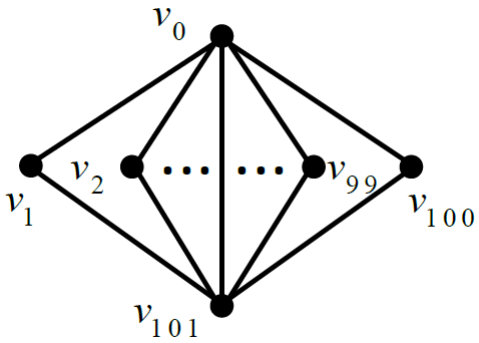


Operation order of SE

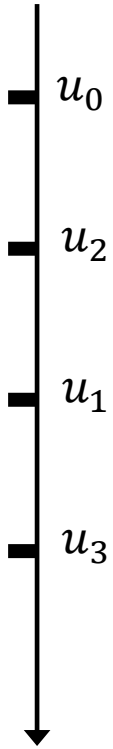
Example of Lazy Materialization



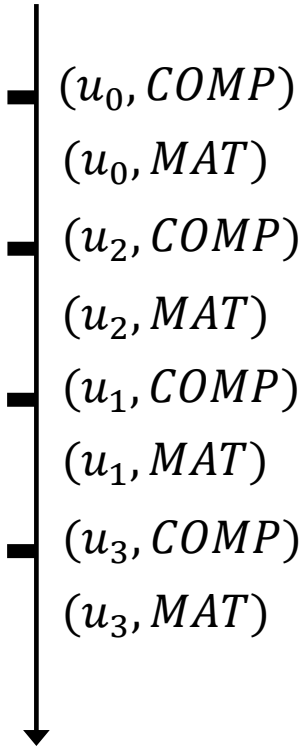
Query graph q



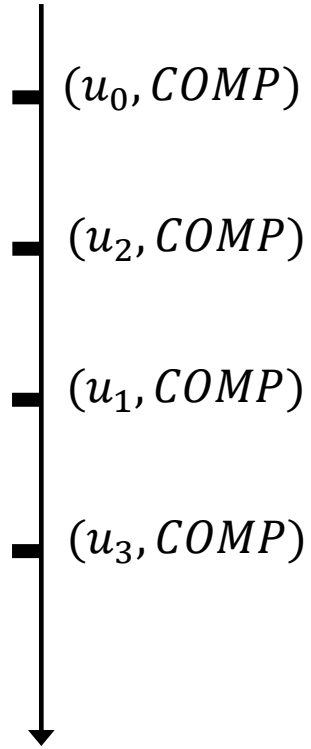
Data graph G



Enumeration order

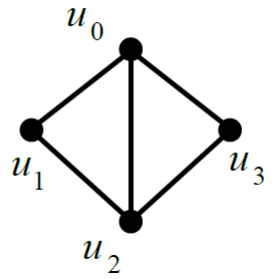


Operation order of SE

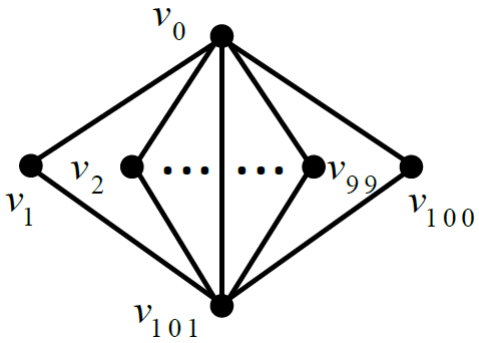


Operation order of LIGHT

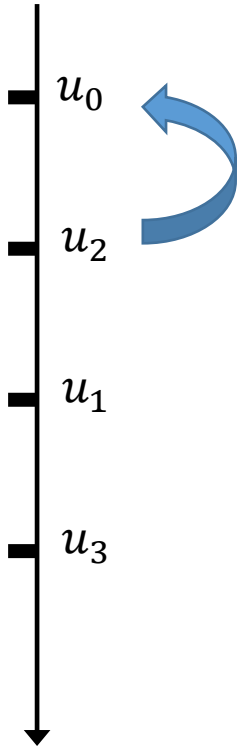
Example of Lazy Materialization



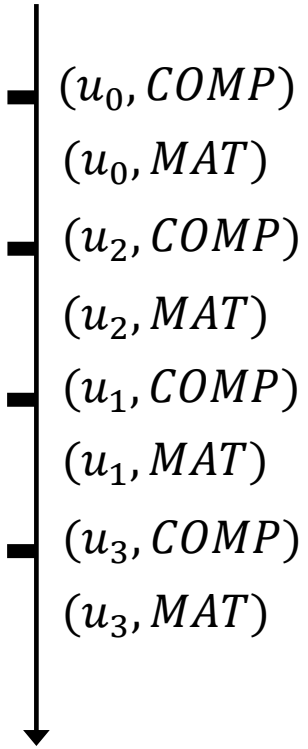
Query graph q



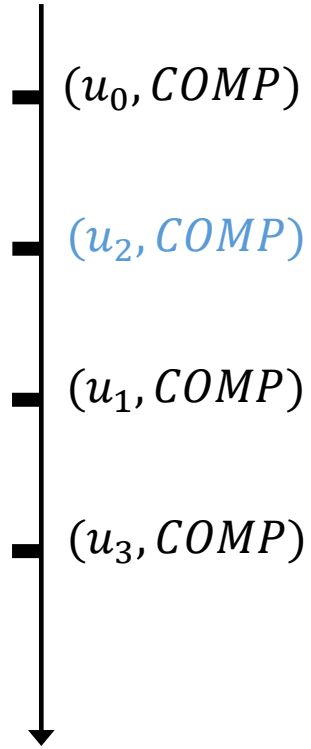
Data graph G



Enumeration order

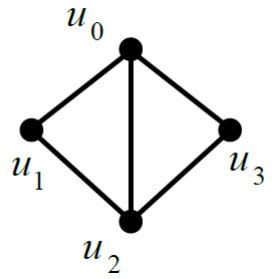


Operation order of SE

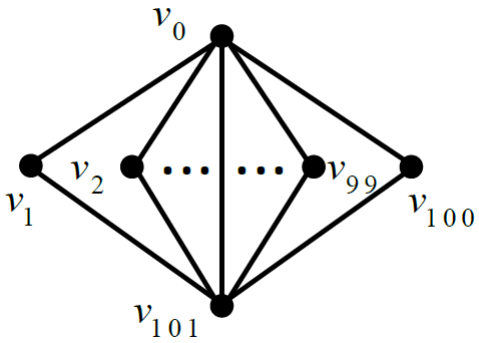


Operation order of LIGHT

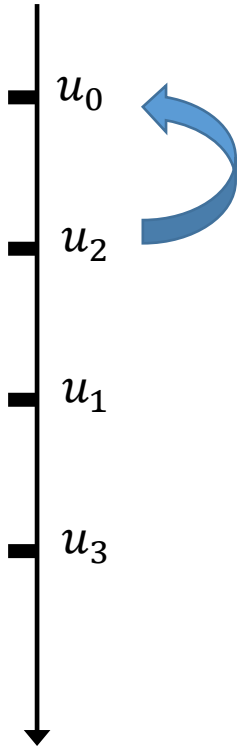
Example of Lazy Materialization



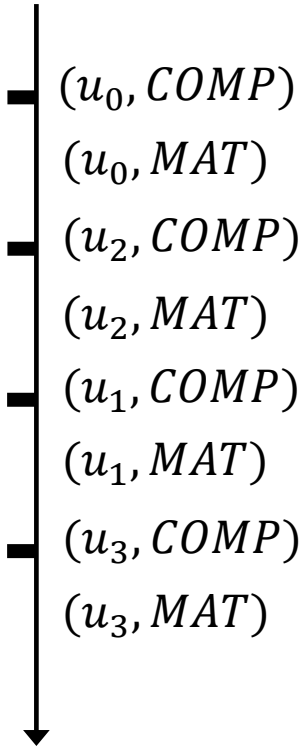
Query graph q



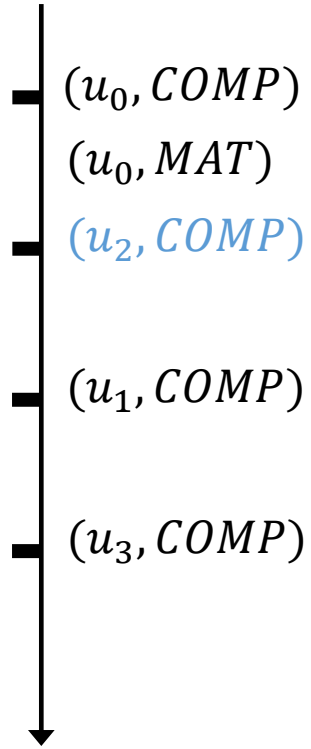
Data graph G



Enumeration order

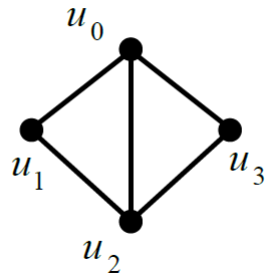


Operation order of SE

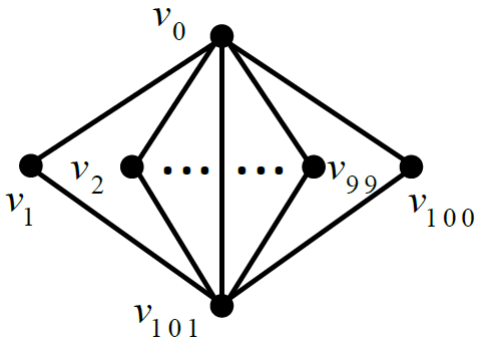


Operation order of LIGHT

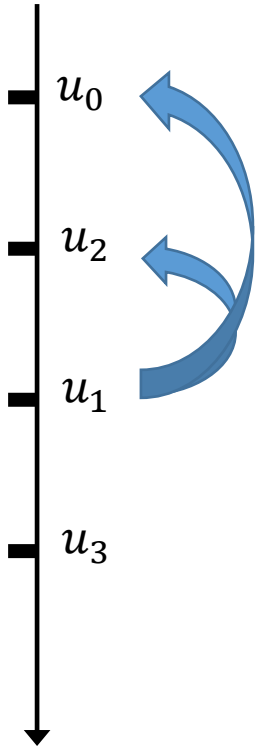
Example of Lazy Materialization



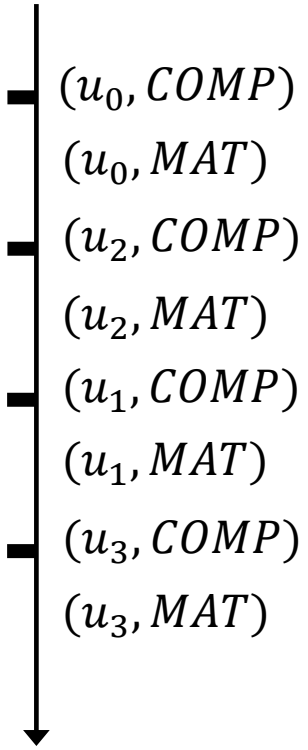
Query graph q



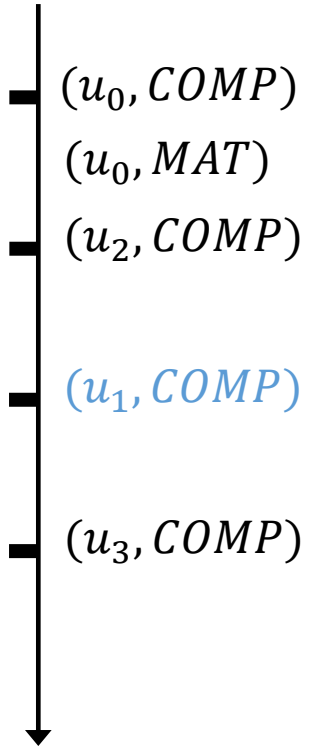
Data graph G



Enumeration order

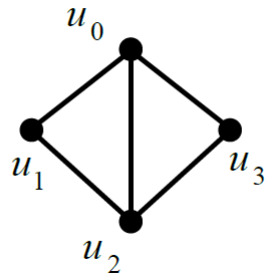


Operation order of SE

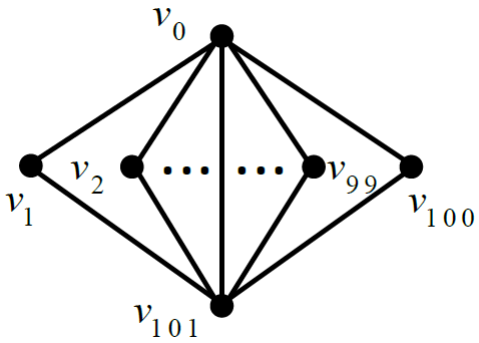


Operation order of LIGHT

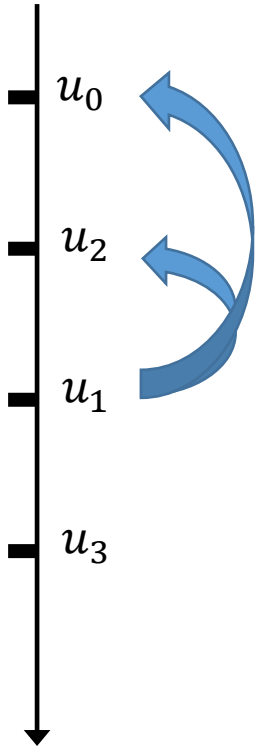
Example of Lazy Materialization



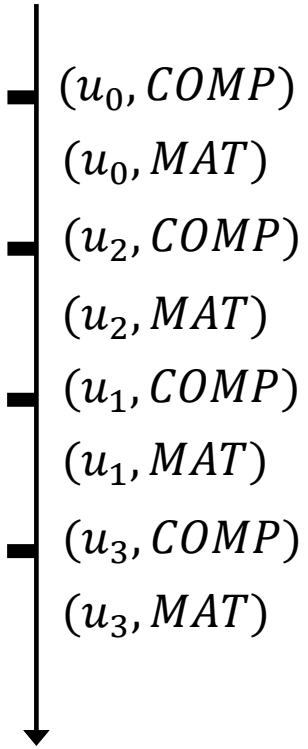
Query graph q



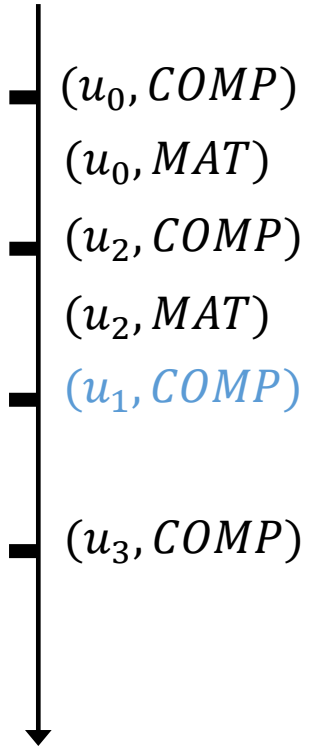
Data graph G



Enumeration order

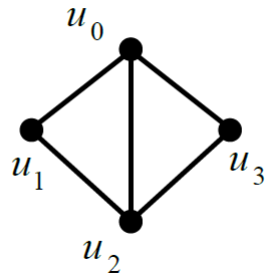


Operation order of SE

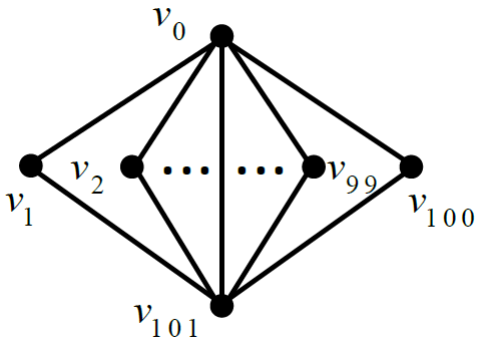


Operation order of LIGHT

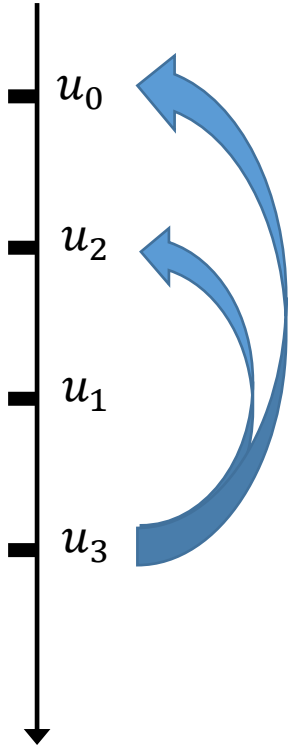
Example of Lazy Materialization



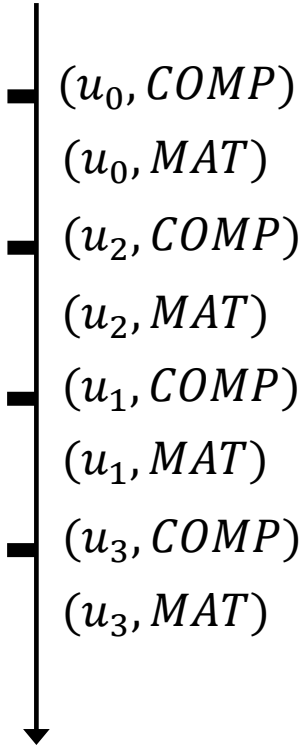
Query graph q



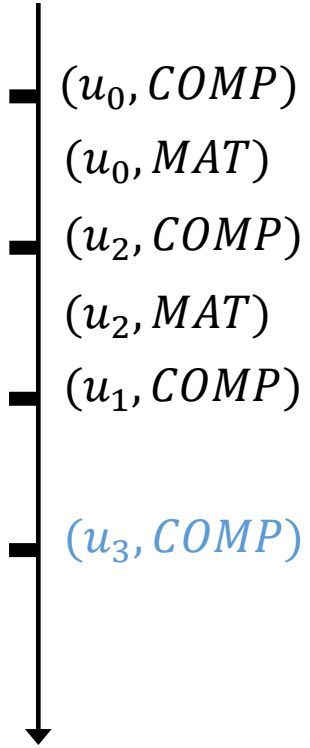
Data graph G



Enumeration order

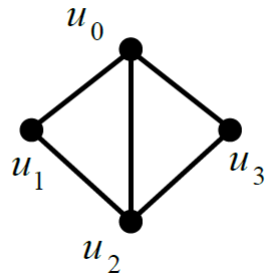


Operation order of SE

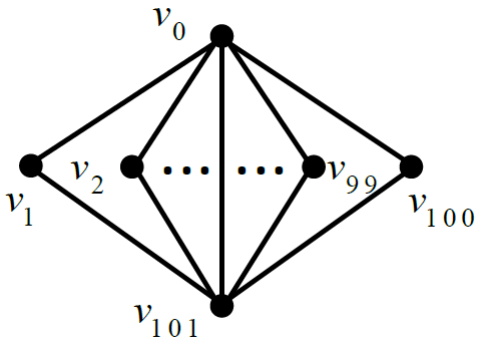


Operation order of LIGHT

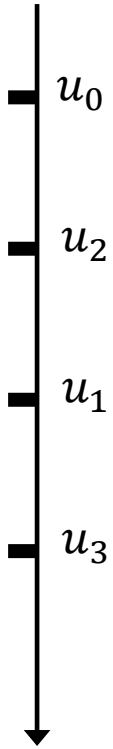
Example of Lazy Materialization



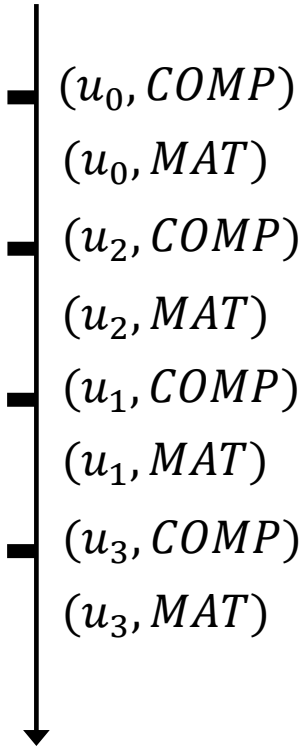
Query graph q



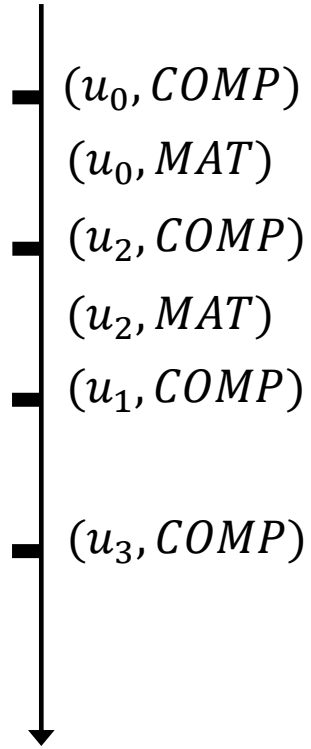
Data graph G



Enumeration order

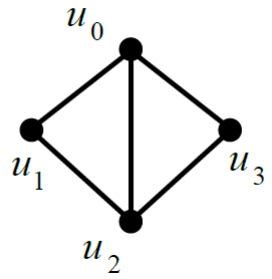


Operation order of SE

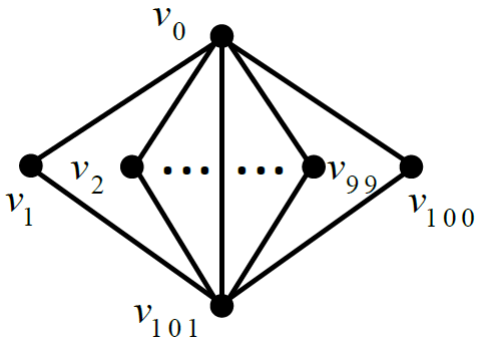


Operation order of LIGHT

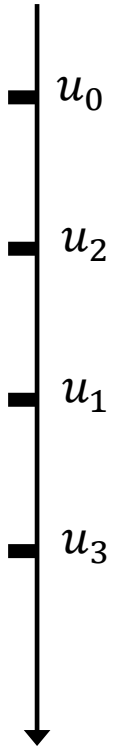
Example of Lazy Materialization



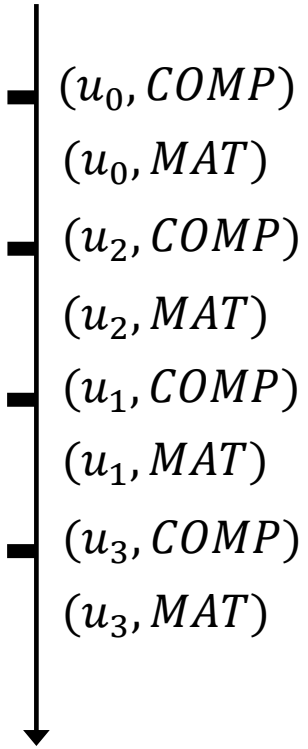
Query graph q



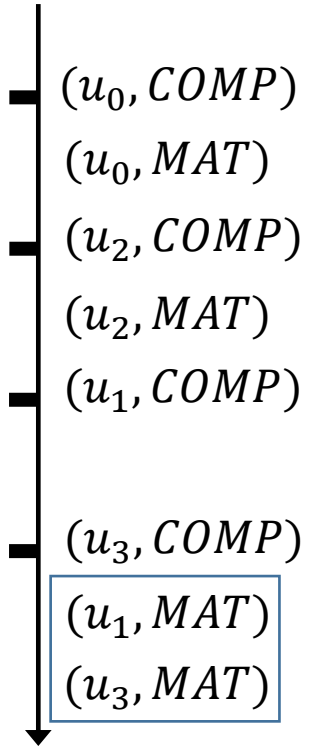
Data graph G



Enumeration order

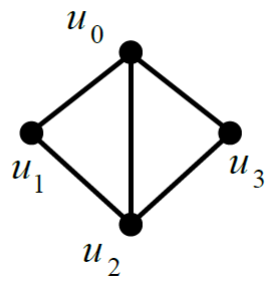


Operation order of SE

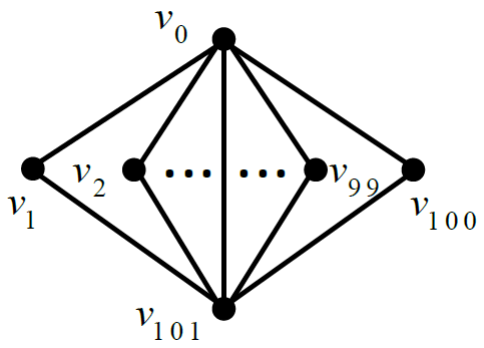


Operation order of LIGHT

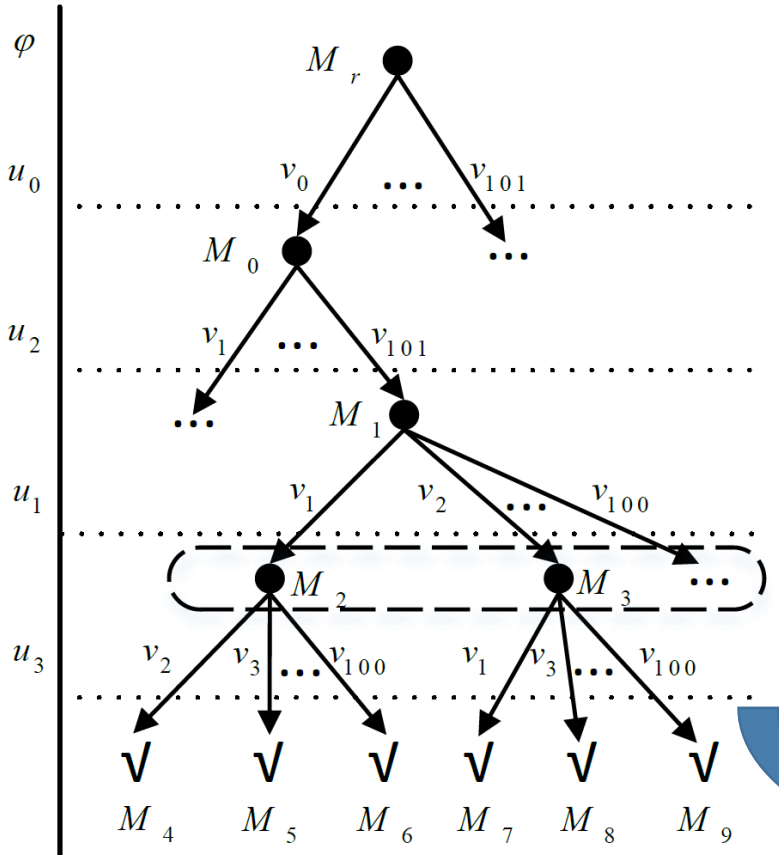
Example of Lazy Materialization



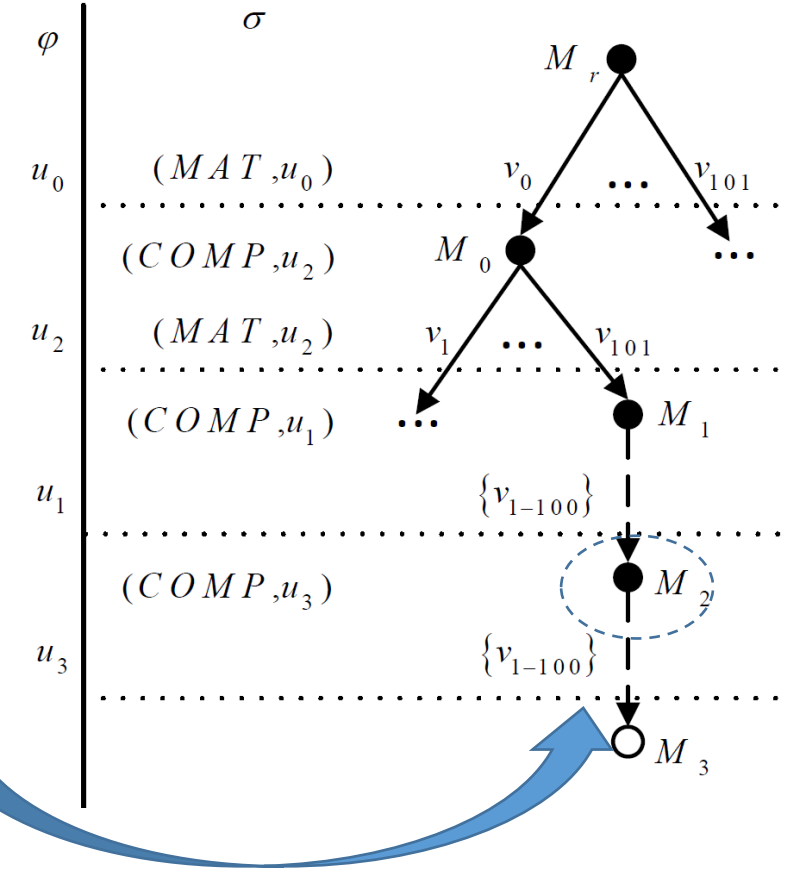
Query graph q



Data graph G

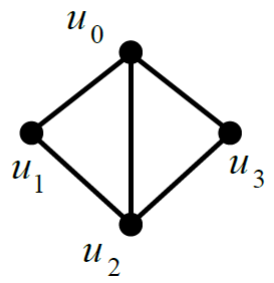


Search tree of SE

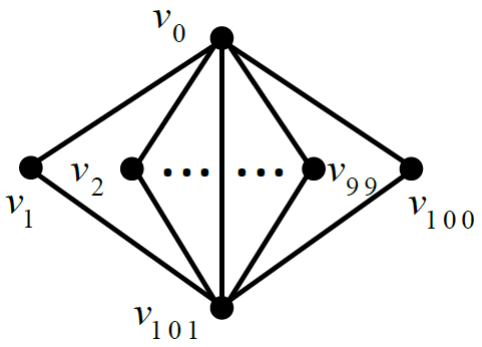


Search tree of LIGHT

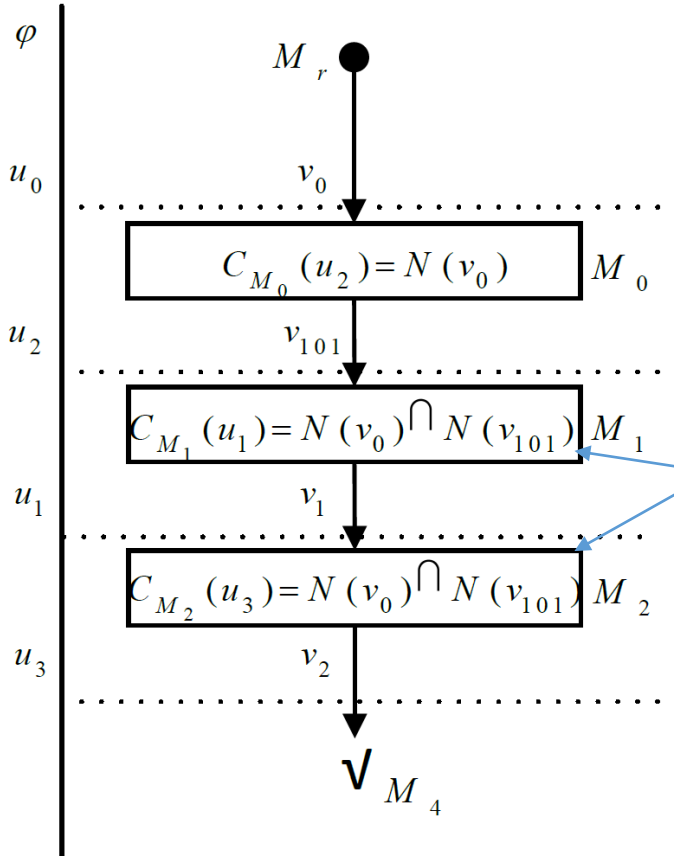
Observation Two



Query graph q



Data graph G



Search path of SE

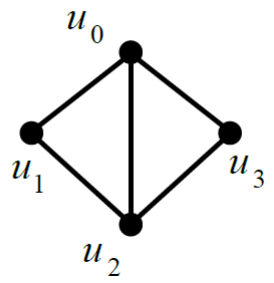
Given partial results M_1 and M_2, the same set intersection $N(v_0) \cap N(v_{101})$ is repeated in the computation of candidates of u_1 and u_3.

MSC based Candidate Sets Computation

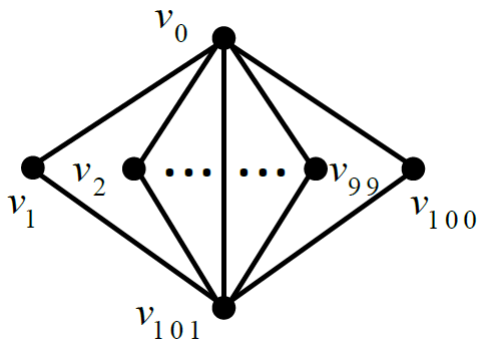
- Compute the candidate set of $u \in \varphi$ by utilizing candidate sets of $u' \in X(u)$ where $X(u)$ contains all query vertices before u in φ.

- Convert it to the minimum set cover (MSC) problem.
  - **Input**: $U = N_+^{\varphi}(u)$, $S = \{\{u'\} | u' \in U\} \cup \{N_+^{\varphi}(u') | N_+^{\varphi}(u') \subseteq N_+^{\varphi}(u) \wedge u' \in X(u)\}$.
  - **Output:** The smallest sub-collection S' of S whose union equals U.

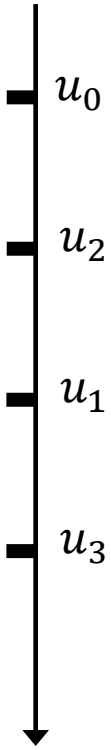
Example of MSC



Query graph q



Data graph G



Enumeration order

$N_+^{\varphi}(u_3) = \{u_0, u_2\}$

$X(u_3) = \{u_0, u_1, u_2\}$

MSC Input:

$N_+^{\varphi}(u_1)$

$U = \{u_0, u_2\}$

$S = \{\{u_0\}, \{u_2\}, \{u_0, u_2\}\}$

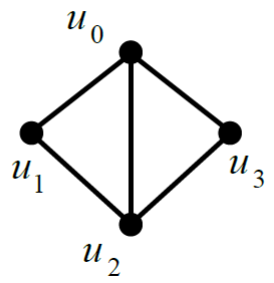
MSC Output:

$S' = \{\{u_0, u_2\}\}$

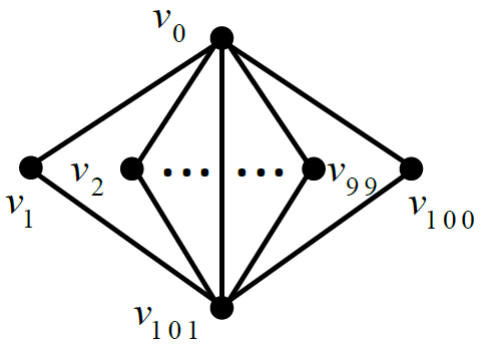
$C_M(u_3) = C_M(u_1)$

Compute candidate set of u_3

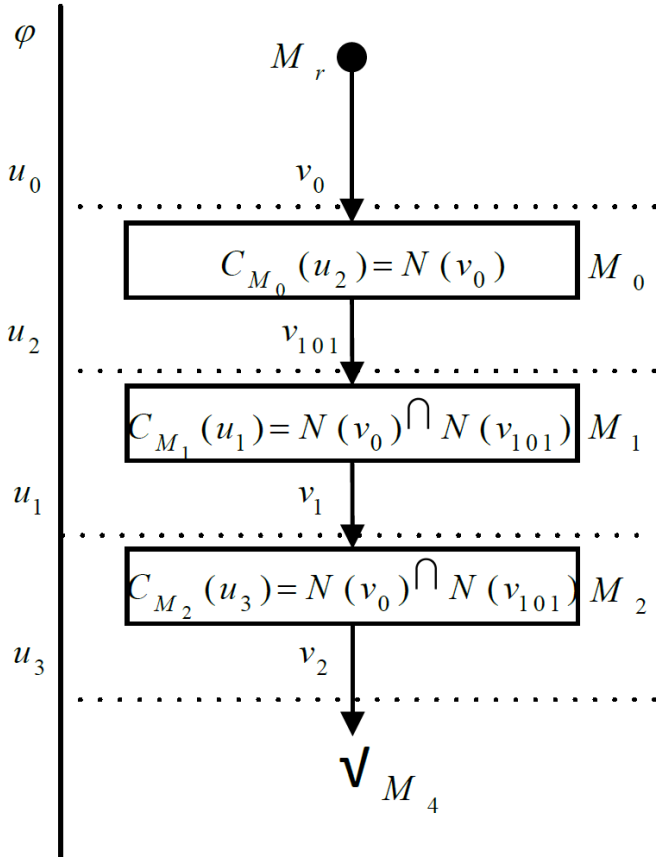
Example of MSC



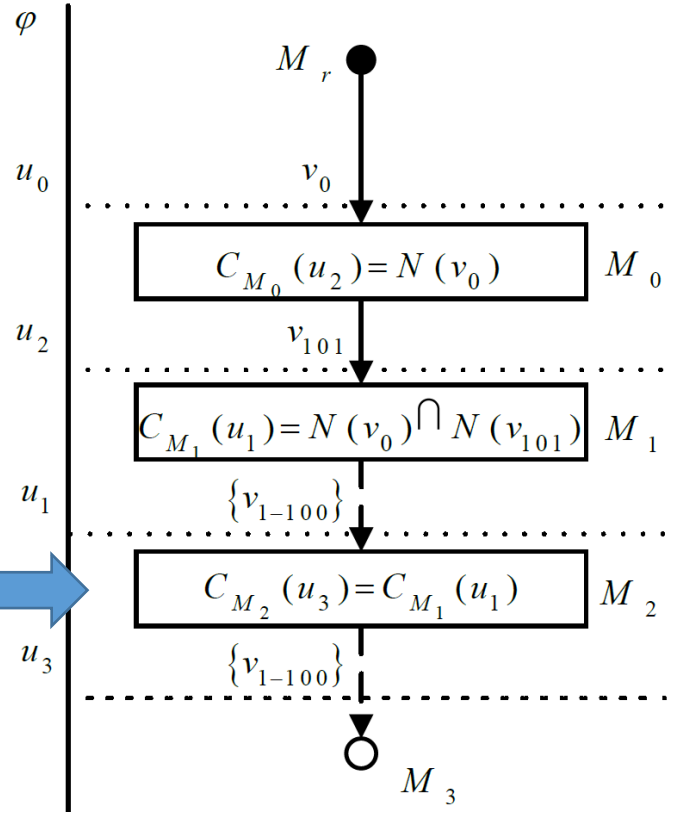
Query graph q



Data graph G



Search path of SE



Search path of LIGHT

Parallel Implementation

- Utilize both vector registers and multiple cores in modern CPUs.
  - Parallelize set intersections with SIMD (Single-Instruction-Multiple-Data) instructions.
  - Parallelize the exploration of the search tree with multi-threading.

Experimental Setup

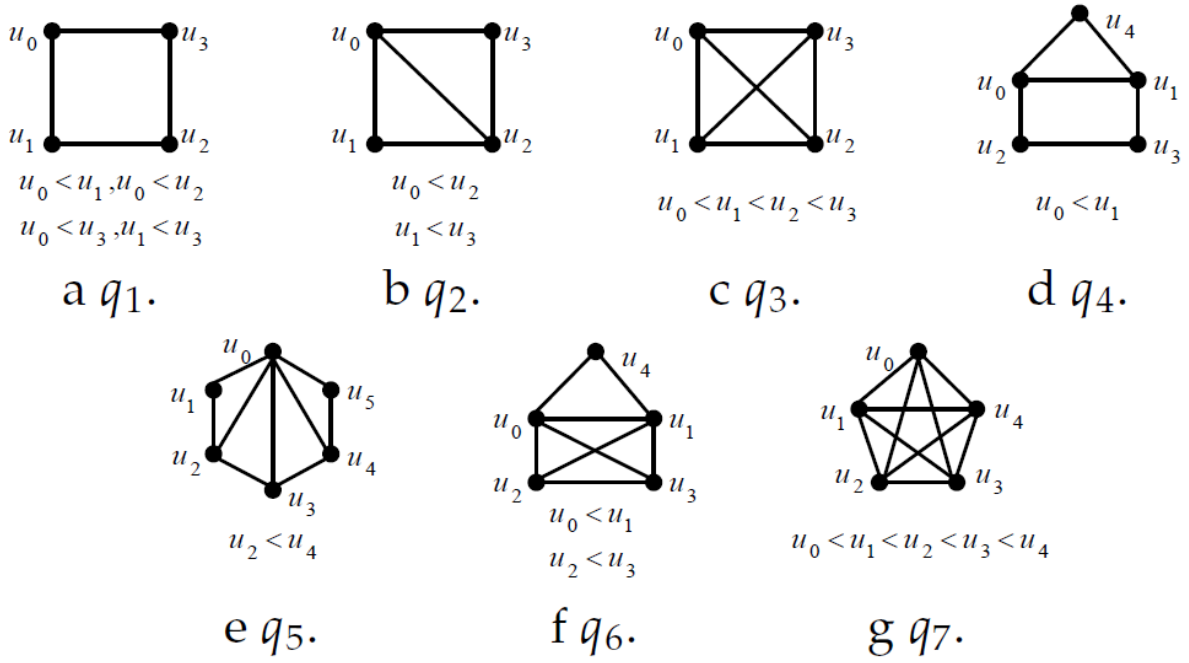
- **Experimental Environment:**
  - Implemented in C++ and compiled with icpc 16.0.0.
  - A machine equipped with 20 cores (2 Intel Xeon E5-2650 v3 @ 2.30GHz CPUs), 64GB RAM and 1TB HDD.
  - Use the AVX2 (256-bit) instruction set and execute with 64 threads.

- **Data Graphs:**

Dataset	Name	N (million)	M (million)	Memory (GB)
youtube	*yt*	3.22	9.38	0.09
eu-2005	*eu*	0.86	19.24	0.15
live-journal	*lj*	4.85	68.48	0.53
com-orkut	*ot*	3.07	117.19	0.89
uk-2002	*uk*	18.52	298.11	2.30
friendster	*fs*	65.61	1,806.07	13.71

Experimental Setup

- **Query Graphs:**



Comparison with SE

- T_{SE} and T_{LIGHT} are the serial execution time of SE and LIGHT respectively.
- T_{SE+P} and $T_{LIGHT+P}$ are the parallel execution time of SE and LIGHT respectively.
- $Overall\ Speedup = \frac{T_{SE}}{T_{LIGHT+P}}$.

Dataset	*yt*			*lj*		
Pattern	q_2	q_4	q_6	q_2	q_4	q_6
T_{SE}	645	176,181	4,448	677	232,800	34,090
T_{SE+P}	22	4,034	115	15.9	6,949	1,425
T_{LIGHT}	31	3,309	43	26	3,497	285
$T_{LIGHT+P}$	0.3	56	0.9	0.9	80	8.7
Speedup	2,150X	3,146X	4,942X	752X	2,910X	3,918X

Comparison with SE (seconds).

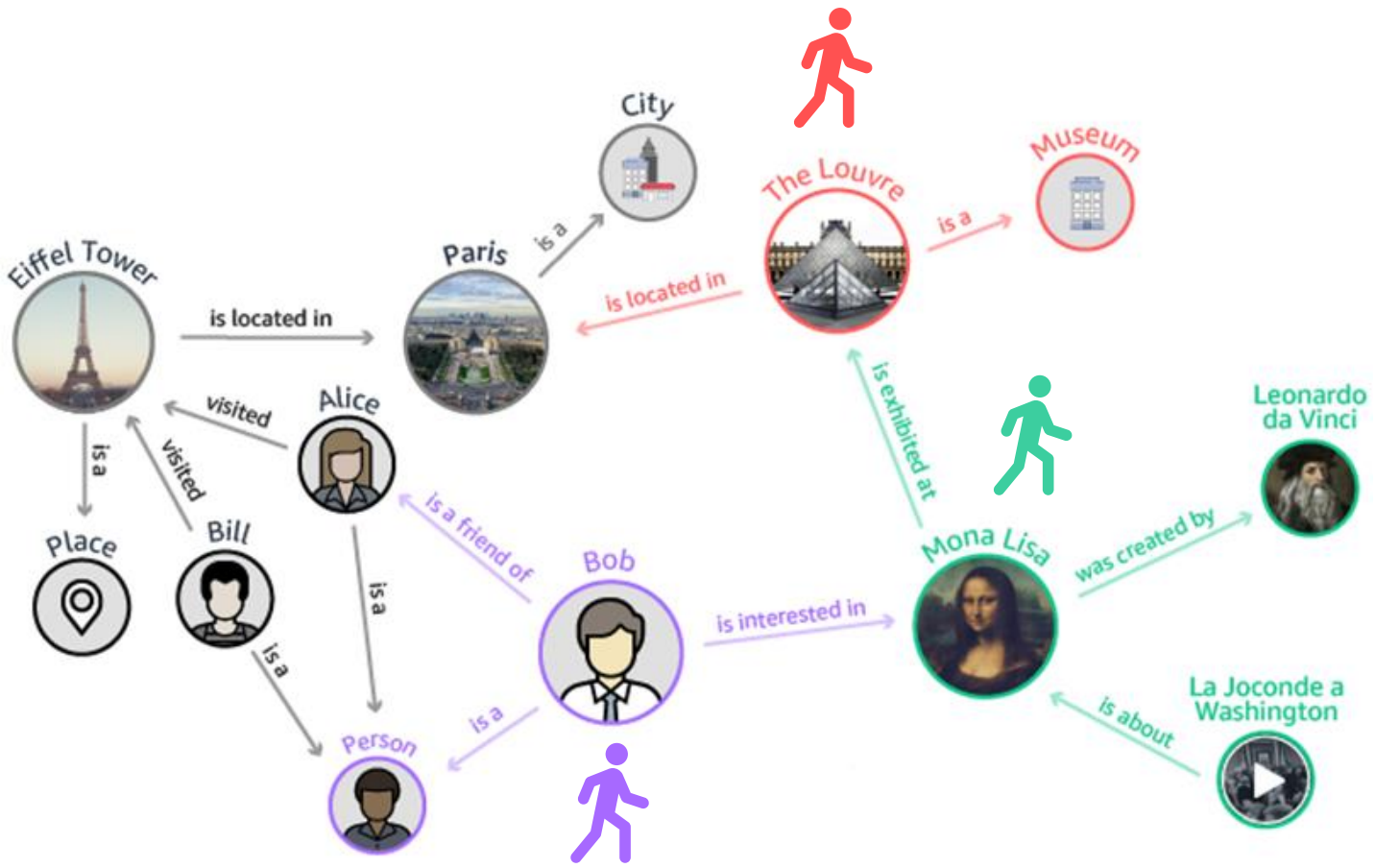
ThunderRW: An In-Memory Graph Random Walk Engine

Shixuan Sun[1], Yuhang Chen[1], Shengliang Lu[1], Bingsheng He[1], Yuchen Li[2]

[1]National University of Singapore

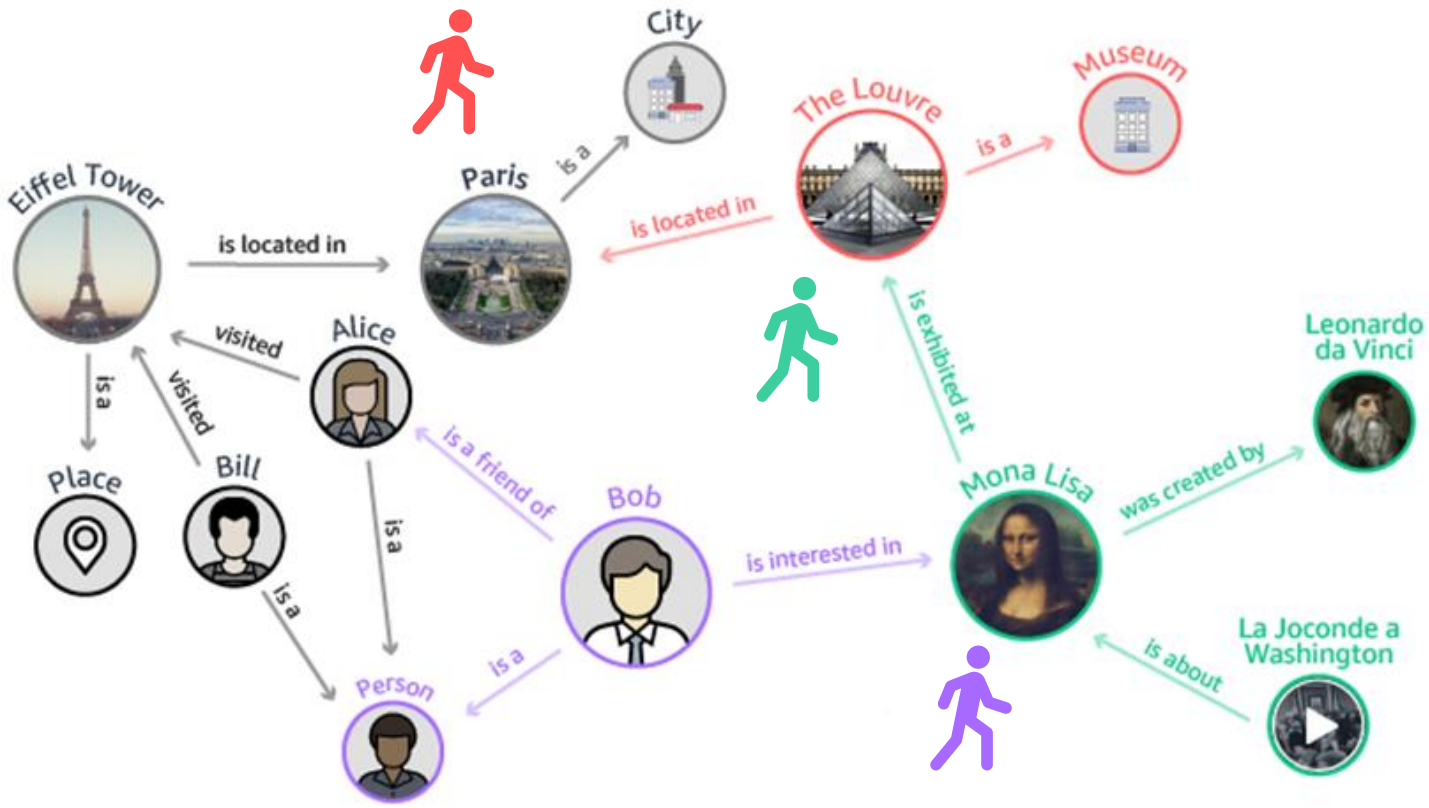
[2]Singapore Management University

Graph Random Walk (RW)



- **Input:**
  - A graph G
  - A set $\boldsymbol{Q}$ of walkers
- **Action:**
  - Each walker Q wanders in G independently
  - Q randomly select a neighbor from the current residing vertex at each step
  - Stop when satisfying a specific termination condition
- **Output:**
  - The walk sequence of each walker in $\boldsymbol{Q}$

Graph Random Walk (RW)



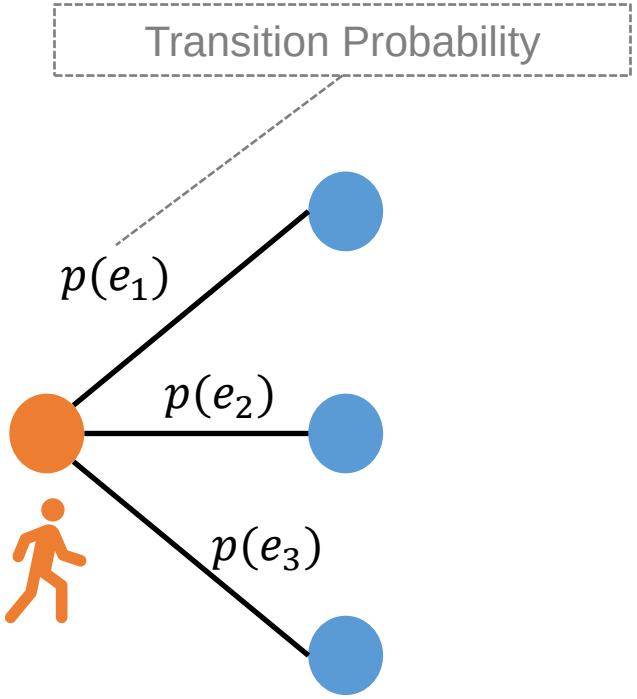
- **Input:**
  - A graph G
  - A set $\boldsymbol{Q}$ of walkers
- **Action:**
  - Each walker Q wanders in G independently
  - Q randomly select a neighbor from the current residing vertex at each step
  - Stop when satisfying a specific termination condition
- **Output:**
  - The walk sequence of each walker in $\boldsymbol{Q}$

Usage of Graph Random Walk

- Graph processing applications
  - Network community profiling
  - Graphlet concentration
- Graph ranking applications
  - Personalized page rank
  - SimRank
- Graph embedding applications
  - DeepWalk
  - Node2Vec

Categorization of Graph Random Walks

- RW algorithms mainly differ in the ***neighbor selection*** step.

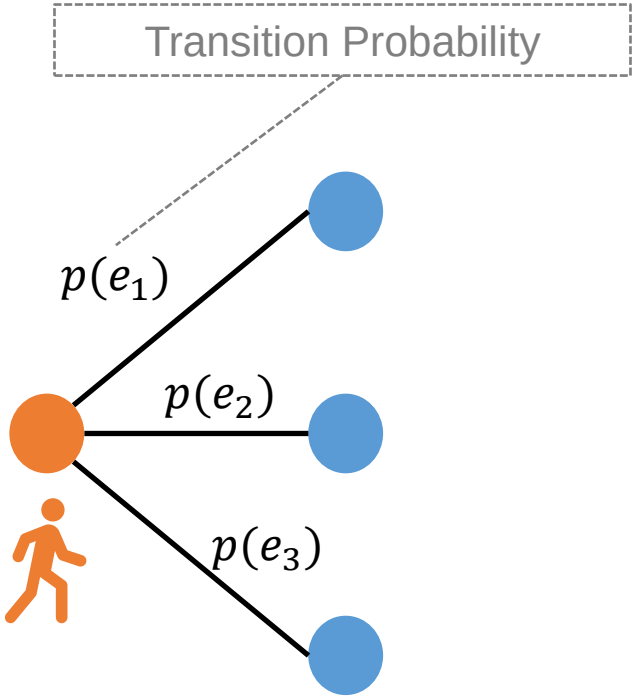


Categorization of Graph Random Walks

- RW algorithms mainly differ in the ***neighbor selection*** step.

- Categorization based on ***transition probability p*** properties.
  - ***Unbiased***: p is the same.
  - ***Static:*** p is fixed in execution.
  - ***Dynamic:*** p depends on the state of a walker.

Biased



Properties of Graph Random Walk

- **Input:**
  - A graph G
  - A set $\boldsymbol{Q}$ of walkers
- **Action:**
  - Each walker Q wanders in G independently
  - Q randomly select a neighbor from the current residing vertex at each step
  - Stop when satisfying a specific termination condition
- **Output:**
  - The walk sequence of each walker in $\boldsymbol{Q}$

Limited Data Parallelism within One Query

Properties of Graph Random Walk

- **Input:**
  - A graph G
  - A set $\boldsymbol{Q}$ of walkers
- **Action:**
  - Each walker Q wanders in G independently
  - Q randomly select a neighbor from the current residing vertex at each step
  - Stop when satisfying a specific termination condition
- **Output:**
  - The walk sequence of each walker in $\boldsymbol{Q}$

Limited Data Parallelism within One Query

Massive Queries Executing Simultaneously

Properties of Graph Random Walk

Method	Front End	Bad Spec.	Core Bound	Memory Bound	Retiring
BFS	11.6%	9.1%	20.8%	40.6%	18.0%
SSSP	9.1%	12.5%	24.9%	36.9%	16.6%
PPR	0.6%	0.7%	15.8%	**73.1%**	9.7%
DeepWalk	1.0%	3.9%	16.7%	**69.7%**	8.7%

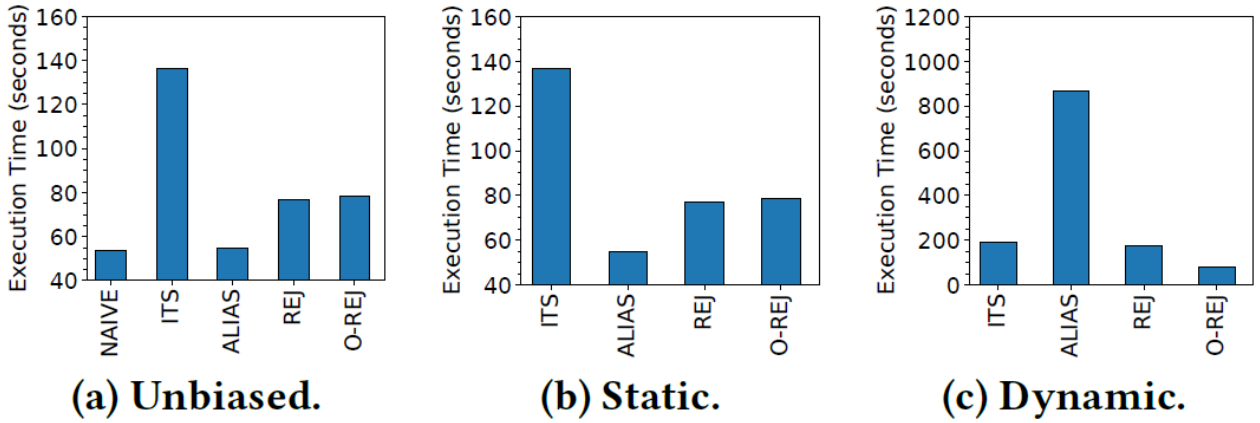
Comparison of pipeline slot breakdown between traditional graph algorithms and RW algorithms (Measured by Intel VTune Profiler).

Limited Data Parallelism within One Query

Massive Queries Executing Simultaneously

Frequent Memory Stalls due to Random Memory Access

Properties of Graph Random Walk



Effectiveness of sampling methods on different types of random walks. *NAÏVE:* a simple uniform sampling method; *ITS:* inverse transformation sampling; *ALIAS:* alias sampling; *REJ:* rejection sampling; *O-REJ:* a variant of rejection sampling.

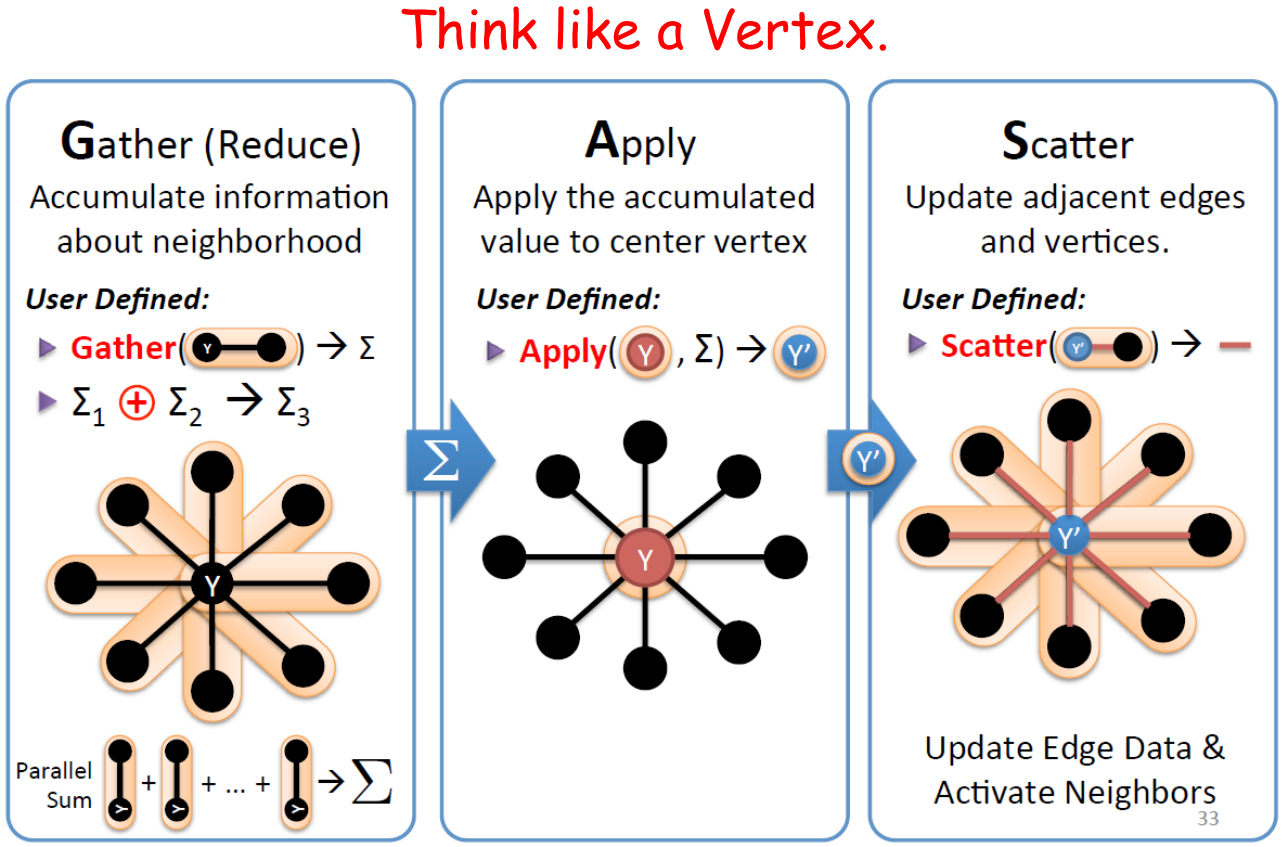
Limited Data Parallelism within One Query

Massive Queries Executing Simultaneously

Frequent Memory Stalls due to Random Memory Access

Significant Impact of Sampling Methods in Neighbor Selection

Characteristics of Graph Computing Frameworks



Optimized for Single Query:
BFS, SSSP, CC etc.

Abstraction from View of Data:
vertex, edge, subgraph etc.

Exploiting Data Parallelism:
process vertices or edges in parallel.

[Figure Source: PowerGraph, OSDI'12].

When Existing Graph Computing Frameworks Meet Graph Random Walk…

Optimized for Single Query

Abstraction from View of Data

Exploiting Data Parallelism

Inherent Conflicts

Limited Data Parallelism within One Query

Massive Queries Executing Simultaneously

Frequent Memory Stalls due to Random Memory Access

Significant Impact of Sampling Methods in Neighbor Selection

ThunderRW: An In-Memory Graph Random Walk Engine

- Users can ***easily*** implement variant ***graph random walk*** based algorithms.

Hyperparameters

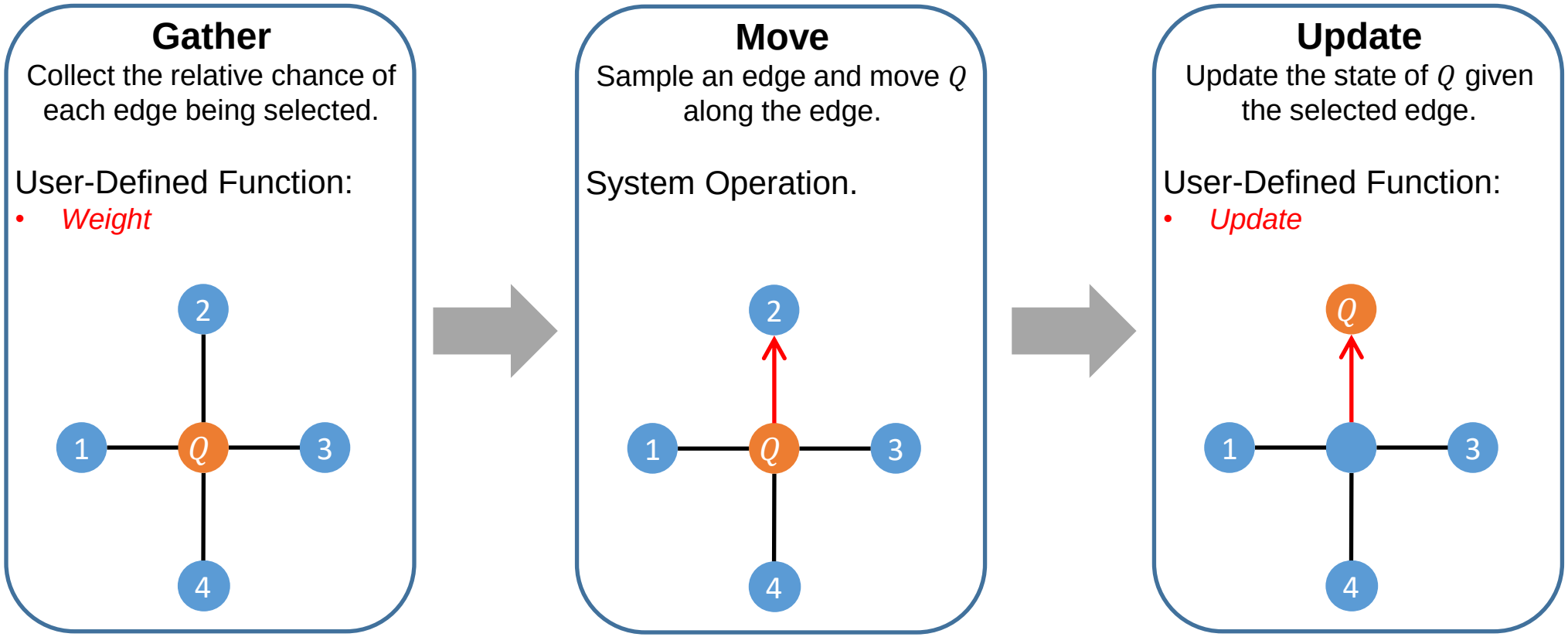
User-defined Functions

```
WalkerType walker_type = WalkerType::Dynamic;
SamplingMethod sampling_method = SamplingMethod::O-REJ;
double Weight(Walker Q, Edge e) {
    if (Q.length == 0) return max(1.0 / a, 1.0, 1.0 / b);
    else if (e.dst == Q.prev) return 1.0 / a;
    else if (IsNeighbor(e.dst, Q.prev)) return 1.0;
    else return 1.0 / b;
}
bool Update(Walker Q, Edge e) {
    return Q.length == target_length;
}
double MaxWeight() {
    return max(1.0 / a, 1.0, 1.0 / b);
}
```

Listing 1: Node2Vec sample code.

Step-Centric Model

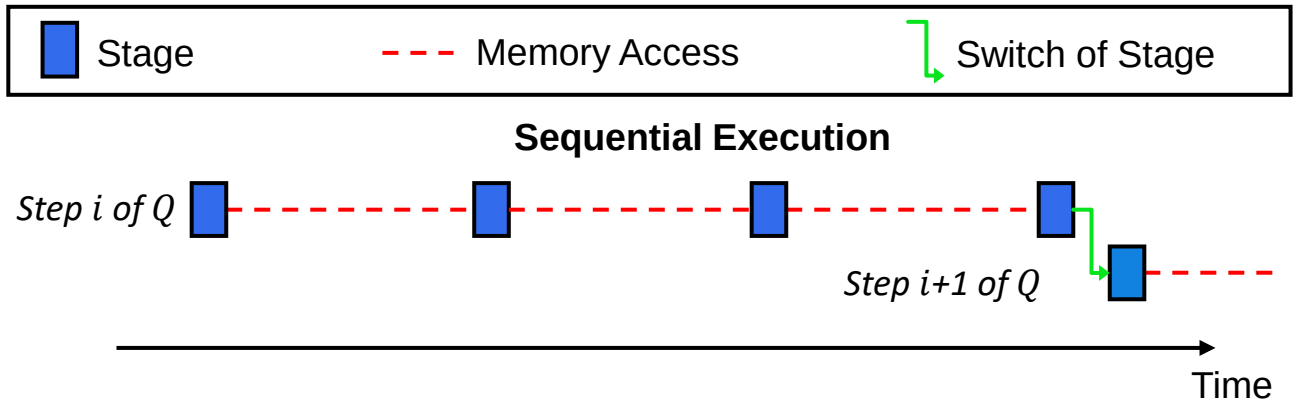
- Think like a ***"walker"*** and factor ***a step*** into ***Gather-Move-Update (GMU)*** operations.
- Apply GMU operations to each walker ***in parallel***.



Step-Interleaving Technique

- Resolve ***cache stalls*** caused by irregular memory access by ***software prefetching***.
  - Modern CPUs can issue ***multiple outstanding*** memory request.

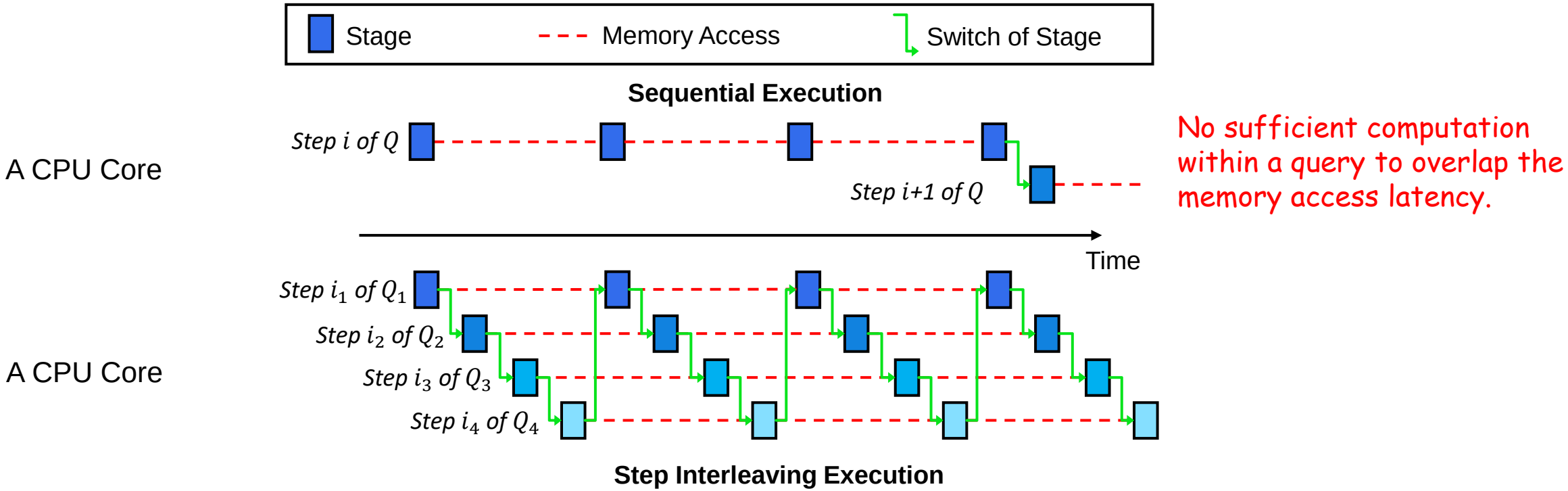
A CPU Core



No sufficient computation within a query to overlap the memory access latency.

Step-Interleaving Technique

- Resolve ***cache stalls*** caused by irregular memory access by ***software prefetching***.
  - Modern CPUs can issue ***multiple outstanding*** memory request.



No sufficient computation within a query to overlap the memory access latency.

Experiment Setup

Method	PPR	DeepWalk	Node2Vec	MetaPath
Baseline	✓	✓	✓	✓
KnightKing [SOSP'19]	✓	✓	✓	×
GraphWalker [USENIX ATC'20]	✓	×	×	×
ThunderRW	✓	✓	✓	✓

- **Workloads:** 12 graphs with $|E|$ varying from $1.85M$ to $1.81B$.
- **Environment:** A Linux Server with a CPU with 10 cores and $220\ GB$ RAM.

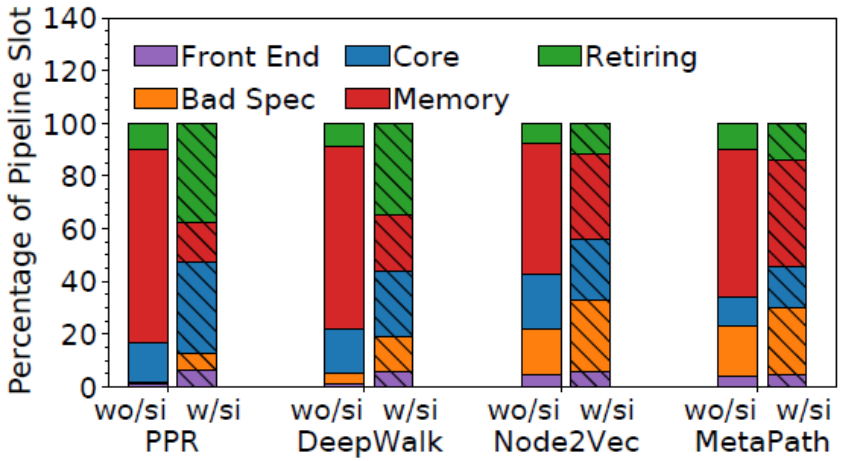
Summary of Results

- **Comparison with the baseline method:**
  - $\mathbf{8.6 - 3333.1X}$ speedup.
- **Comparison with existing systems:**
  - $\mathbf{1.7 - 14.6X}$ speedup.
- **Throughput:**
  - $\mathbf{3 \times 10^8}$ in terms of steps per second.

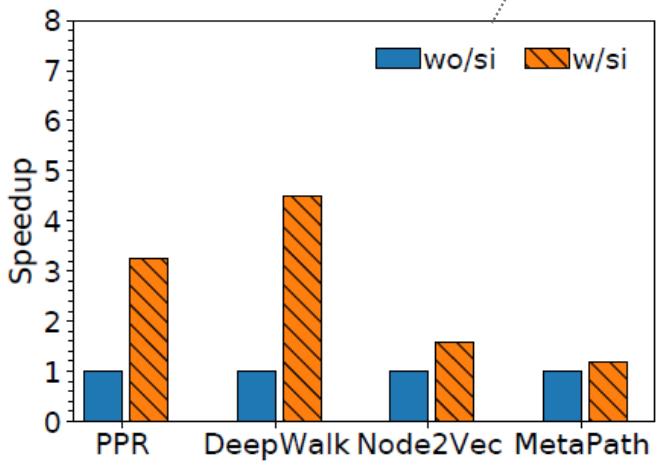
Evaluation of Step Interleaving

- Reduce ***memory bound*** from $\mathbf{73.1\%}$ to $\mathbf{15.0\%}$.
- Speed up queries by up to $\mathbf{4.8X}$.
- Improvement can be ***limited*** for ***high-order*** random walks.

Notation:
wo/si: Disable step interleaving.
w/si: Enable step interleaving.



(a) Pipeline slot breakdown.



(b) Speedup.

Experiment results on *livejournal* dataset, $|V| = 4.85M, |E| = 68.99M$

Summary

- We study the design and implementation of an ***in-memory graph RW engine***.

- We propose ***ThunderRW***, an efficient in-memory RW engine.
  - ***Step-Centric Model:*** Abstract the computation from the local view of moving a step.
  - ***Step-Interleaving Technique:*** Hide memory latency by executing multiple queries alternatively.

- Source code publicly available at *github.com/Xtra-Computing/ThunderRW*.

Conclusions

- Generic Benchmark Framework
  - Metrics, baseline, design guidelines
- Algorithmic Optimization
  - A holistic approach to arbitrary subgraph queries
  - Real-time processing for hop-constrained s-t path queries
- Hardware Utilization
  - Parallelizing query evaluation with multi-cores and vector registers
  - Efficient in-memory random walk engine with cache optimization

Thanks!
Q&A

Hybrid Set Intersection Method

- The neighbor set of a vertex is stored as a sorted array in which each element is a 32-bit integer.

- Adopt a hybrid set intersection method to ensure that the cost of a set intersection operation is proportional to the size of the smaller set.
  - **Input:** Two neighbor sets $N(u)$ and $N(v)$ where $|N(u)| \geq |N(v)|$.
  - **Output:** $N(u) \cap N(v)$
    1. If $|N(u)|/|N(v)| \leq threshold$, then use the merge-based set intersection. ($O(|N(u)| + |N(v)|)$)
    2. Otherwise, use the Galloping search based method. ($O(|N(v)| \times \log |N(u)|)$)