香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Rapids @ HKUST

Parallelizing Recursive Backtracking Based Subgraph Matching on a Single Machine

Shixuan SUN and Qiong LUO

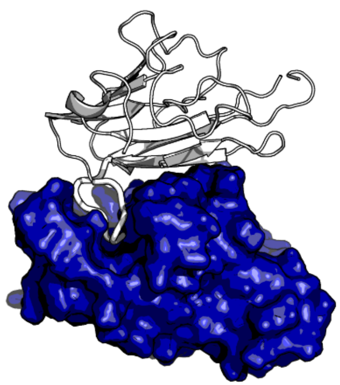
Department of Computer Science and Engineering
Hong Kong University of Science and Technology

Background

Applications

RDF queries

Computer aided design



Protein interaction studies

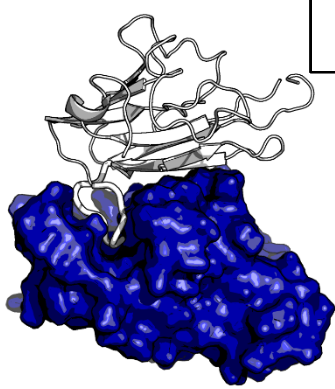
Social network analysis

Applications

RDF queries

Computer aided design

Subgraph Matching

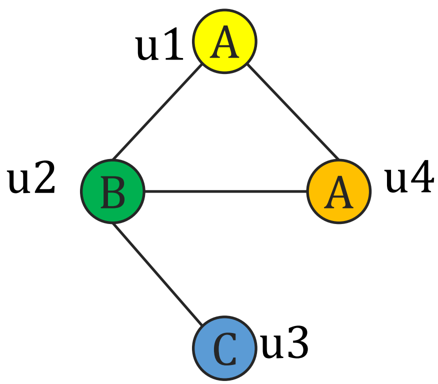


Protein interaction studies

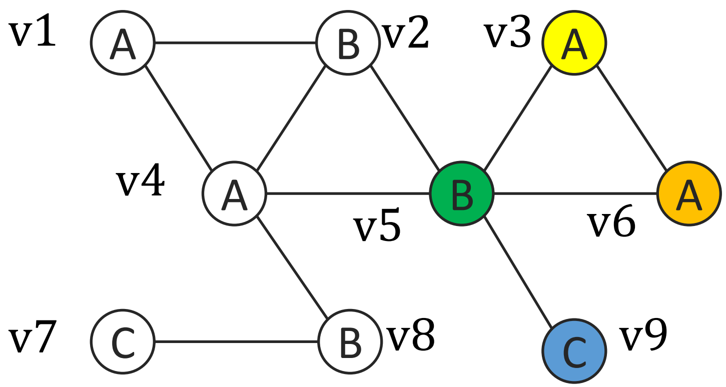
Social network analysis

Subgraph Matching

- Given a query graph q and a data graph G, find all subgraphs in G that are identical to q.
  - **Note:** q is connected, and much smaller than G.
  - **Complexity:** NP-hard.



(a). Query graph q



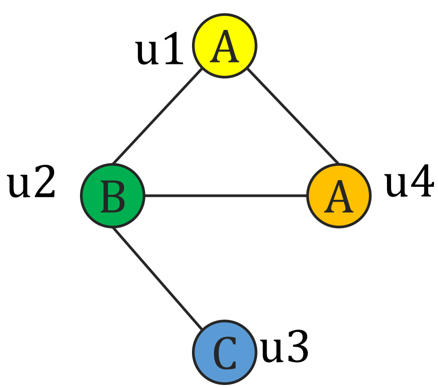
(b). Data graph G

$$f1 = \{(u1, v3), (u2, v5), (u3, v9), (u4, v6)\}$$

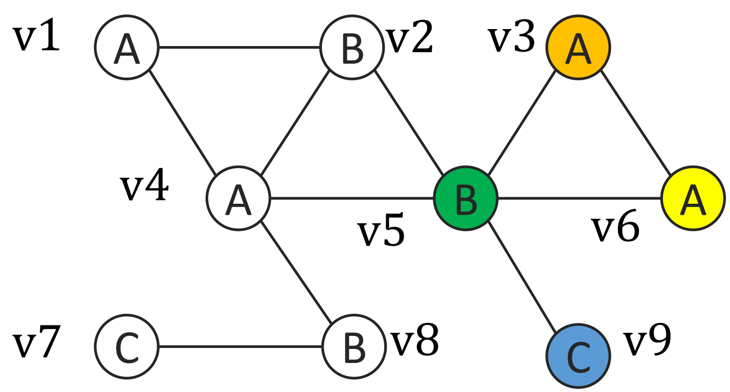
(c). The results of subgraph matching

Subgraph Matching

- Given a query graph q and a data graph G, find all subgraphs in G that are identical to q.
  - **Note:** q is connected, and much smaller than G.
  - **Complexity:** NP-hard.



(a). Query graph q



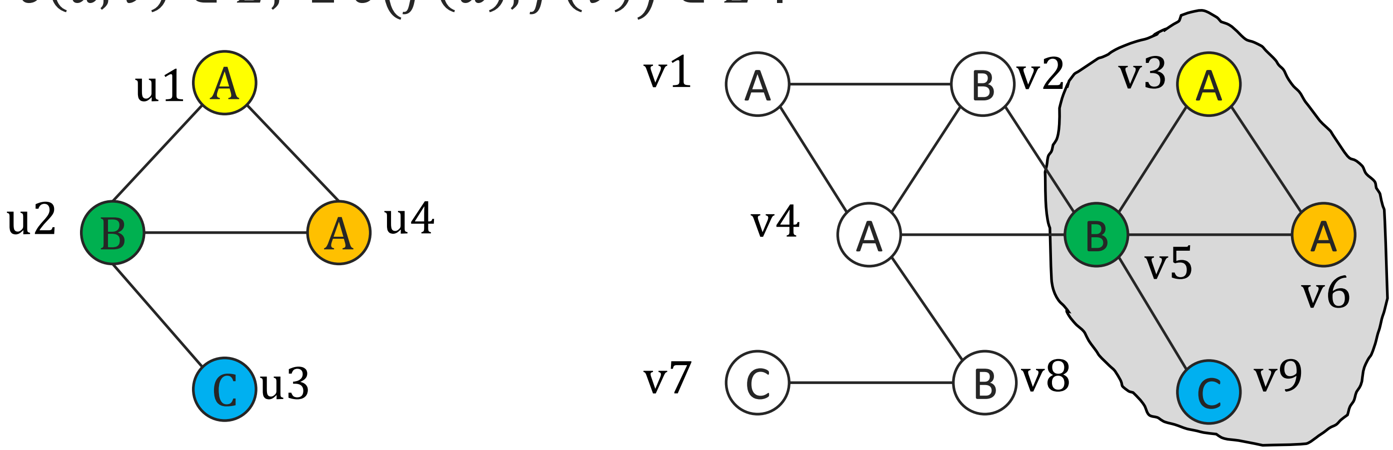
(b). Data graph G

$$f1 = \{(u1, v3), (u2, v5), (u3, v9), (u4, v6)\}$$
$$f2 = \{(u1, v6), (u2, v5), (u3, v9), (u4, v3)\}$$

(c). The results of subgraph matching

Subgraph Isomorphism

- Given a query graph $q = (V, E, \Sigma, L)$ and a data graph $G = (V', E', \Sigma', L')$, a subgraph isomorphism is an injective function f from $V \rightarrow V'$ that satisfies:
  1) $\forall\, u \in V,\ L(u) = L'\big(f(u)\big)$;
  2) $\forall\, e(u, v) \in E,\ \exists\, e\big(f(u), f(v)\big) \in E'$.



(a). Query graph q (b). Data graph G

$$f = \{(u1, v3), (u2, v5), (u3, v9), (u4, v6)\}$$

(c). A subgraph isomorphism from q to G

Motivation

- Due to the hardness of subgraph matching, existing algorithms often take a long time to process big data graphs.
  - Conducting subgraph matching on the Youtube dataset containing over one million vertices takes more than one thousand seconds.
- Existing parallel algorithms either achieve limited speedups or easily run out of memory.
  - pRI's speedup over the sequential RI is limited to less than 10 times on a machine of 16 CPU cores [11].
  - PGX has to maintain 3.2×10^{10} partial results at one iteration, which consumes all the memory space [10].
- A commodity machine nowadays has considerable parallel computation capabilities.
  - There are up to tens of cores in one processor.

Motivation

- Due to the hardness of subgraph matching, these algorithms often take a long time to process big data graphs.
    - Conducting subgraph matching on the Youtube dataset containing over one million vertices takes more than one thousand seconds.
- Existing parallel algorithms either achieve limited speedups or easily run out of memory.
    - pRI's speedup over the sequential RI is limited to less than 10 times on a machine of 16 CPU cores [11].
    - PGX has to maintain 3.2×10^{10} partial results at one iteration, which consumes all the memory space [10].
- A commodity machine nowadays has considerable parallel computation capabilities.
    - There are up to tens of cores in one processor.

We propose to parallelize subgraph matching on a single machine.

Existing Algorithms

Algorithms	Methodology	Execution	Year Published
Ullmann[1]	Backtracking	Serial	1976
VF2[2]	Backtracking	Serial	2004
QSI[3]	Backtracking	Serial	2008
GQL[4]	Backtracking	Serial	2008
GADDI[5]	Backtracking	Serial	2009
Spath[6]	Backtracking	Serial	2010
TurboISO[7]	Backtracking	Serial	2013
CFL[8]	Backtracking	Serial	2016
Stwig[9]	Join	Parallel, Distributed	2012
PGX[10]	Backtracking	Parallel, CPU	2014
pRI[11]	Backtracking	Parallel, CPU	2017
GpSM[12]	Join	Parallel, GPU	2015

Recursive Backtracking based Subgraph Matching

- **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, which is a permutation of query vertices;
     - QSI [3] adopts the infrequent-label first ordering strategy;
     - GQL [4] adopts the left-deep join ordering strategy;
     - CFL [8] adopts the tree-based ordering strategy;
  2. Obtain a candidate set $u.C$ for every vertex $u \in V(q)$, which contains the data vertices that can be mapped to u;
     - The neighborhood signature filter and the pseudo tree isomorphism filter;
  3. Enumerate all solutions by extending partial results recursively along the matching order π.

We propose an efficient parallel subgraph matching framework (PSM) to parallelize backtracking based subgraph matching algorithms on a single machine.

Challenges

- Abstract backtracking based subgraph matching algorithms into an uniform model.
- Find a suitable granularity of parallelism in subgraph matching.
- Achieve load balance and reduce overhead introduced by parallelization.

State Space Tree

State Space Tree Exploration

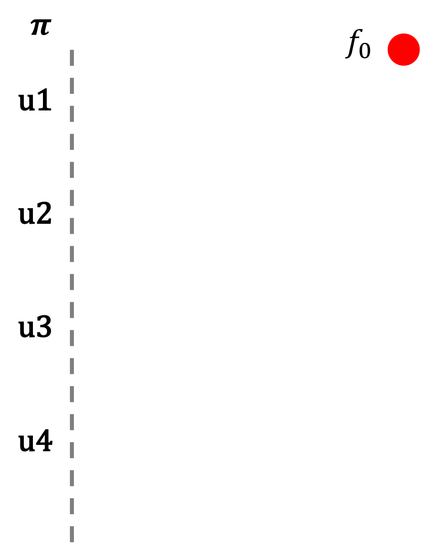
$u1.C = \{v1, v3, v4, v6\}$

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

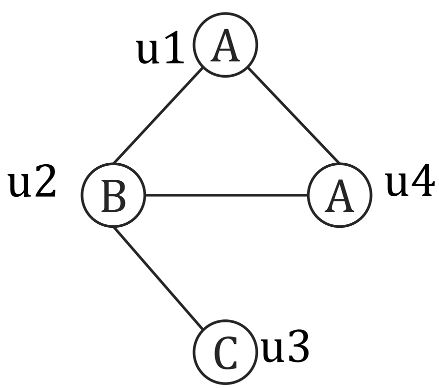
$u3.C = \{v7, v9\}$

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

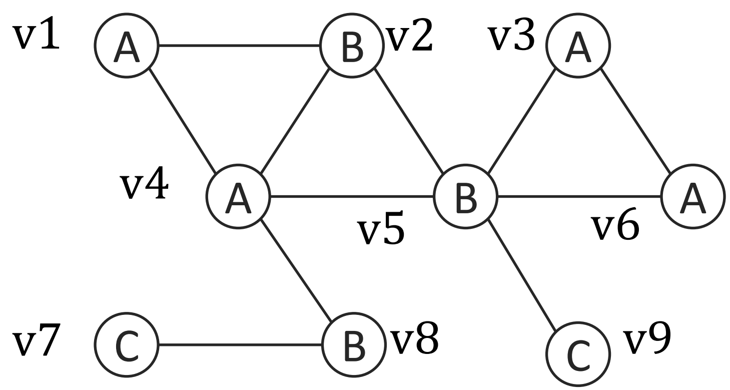
$f_0 = \{\}$



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



(a). Query graph q



(b). Data graph G

State Space Tree Exploration

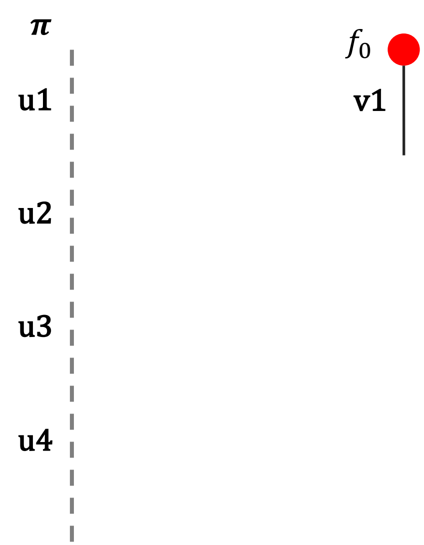
$u1.C = \{v1, v3, v4, v6\}$

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

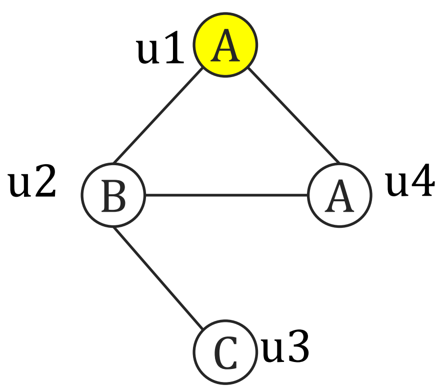
$u3.C = \{v7, v9\}$

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

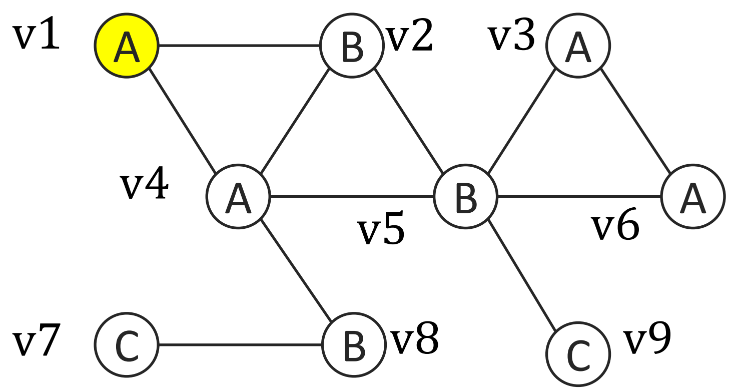
$f_0 = \{\}$



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



(a). Query graph q



(b). Data graph G

State Space Tree Exploration

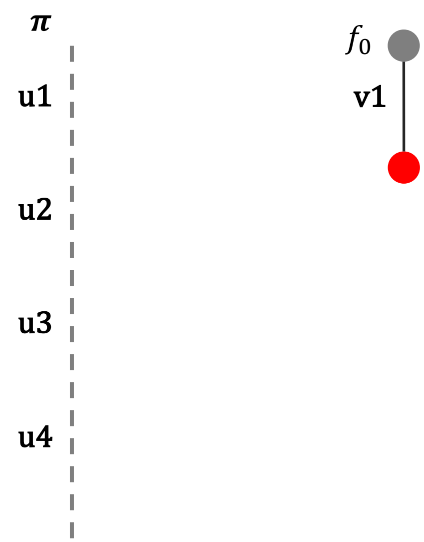
$$u1.C = \{v1, v3, v4, v6\}$$

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

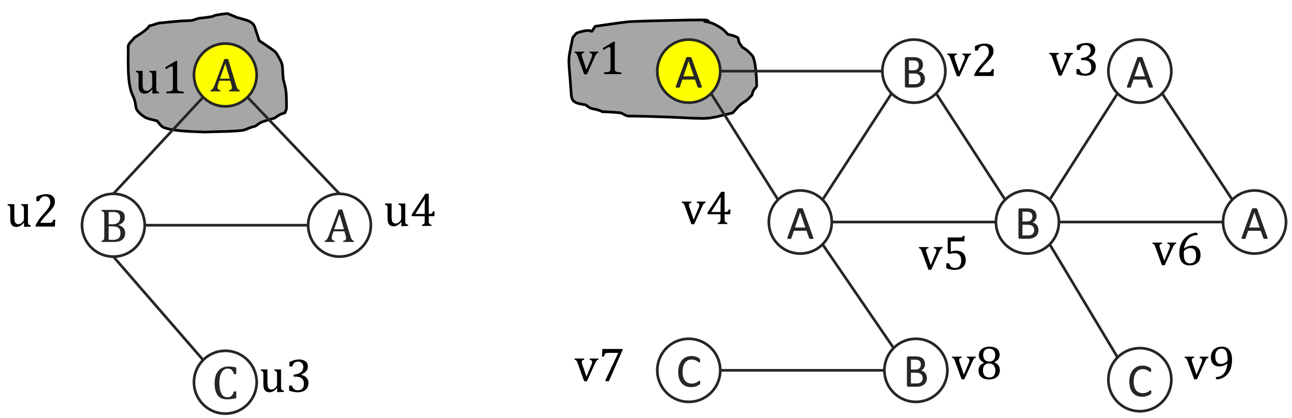
$$u3.C = \{v7, v9\}$$

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

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



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



(a). Query graph q (b). Data graph G

State Space Tree Exploration

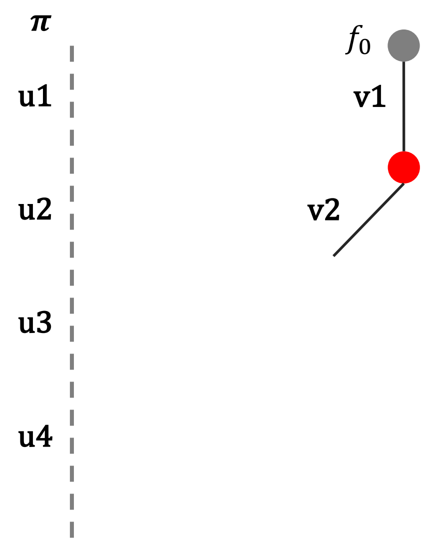
$u1.C = \{v1, v3, v4, v6\}$

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

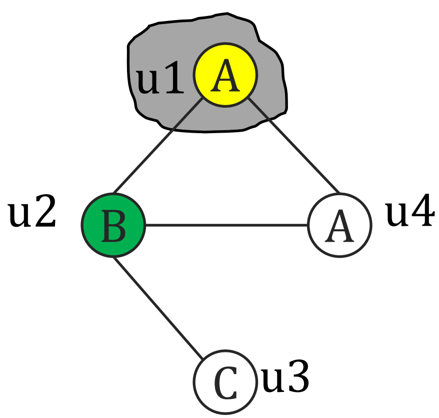
$u3.C = \{v7, v9\}$

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

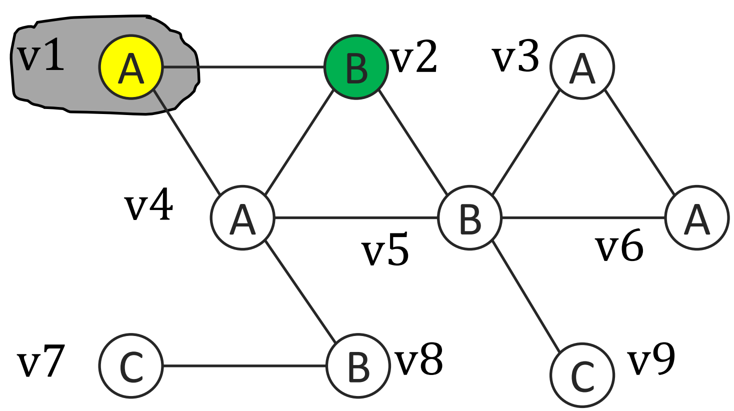
$f_1 = \{(u1, v1)\}$



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



(a). Query graph q



(b). Data graph G

State Space Tree Exploration

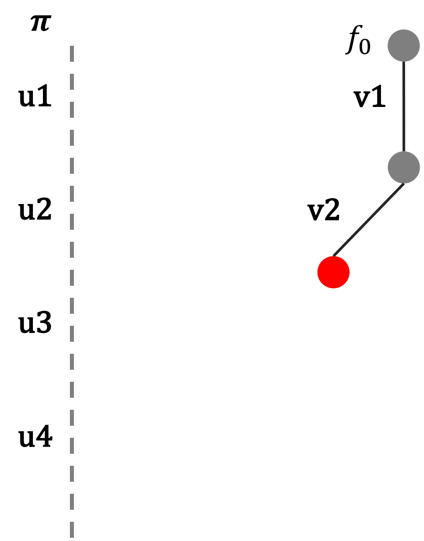
$u1.C = \{v1, v3, v4, v6\}$

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

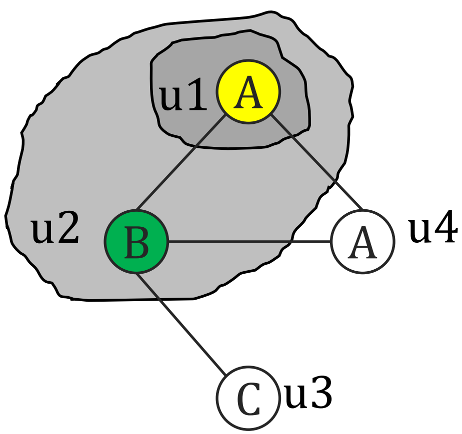
$u3.C = \{v7, v9\}$

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

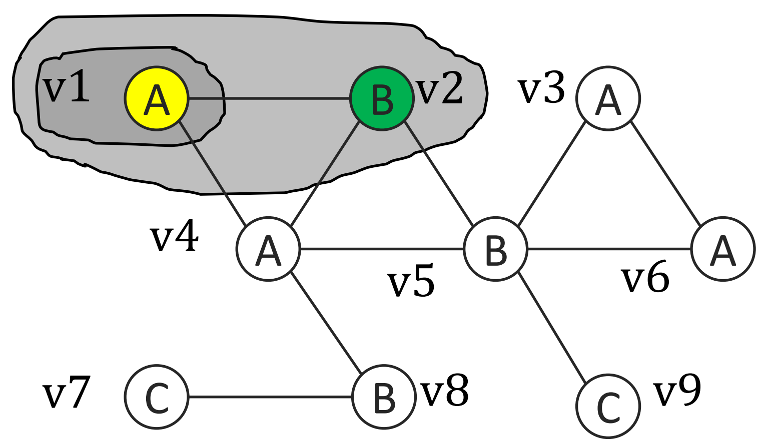
$f_2 = \{(u1, v1), (u2, v2)\}$



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



(a). Query graph q



(b). Data graph G

State Space Tree Exploration

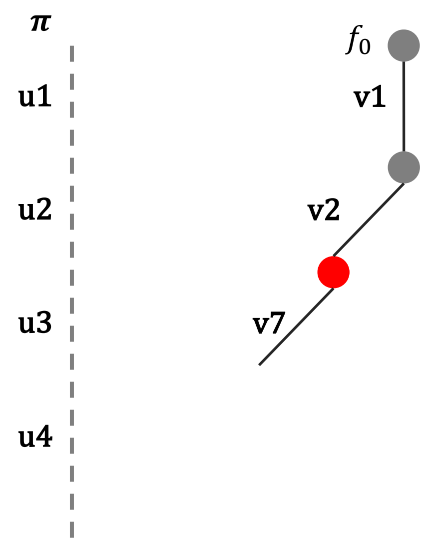
$u1.C = \{v1, v3, v4, v6\}$

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

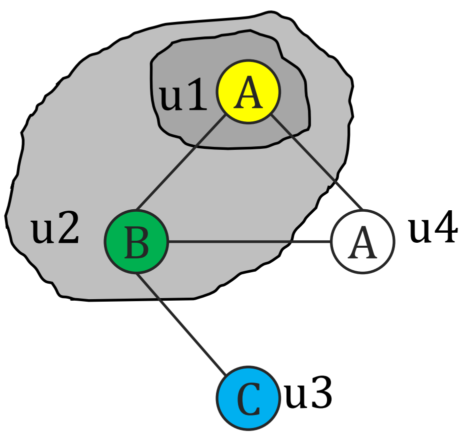
$u3.C = \{v7, v9\}$

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

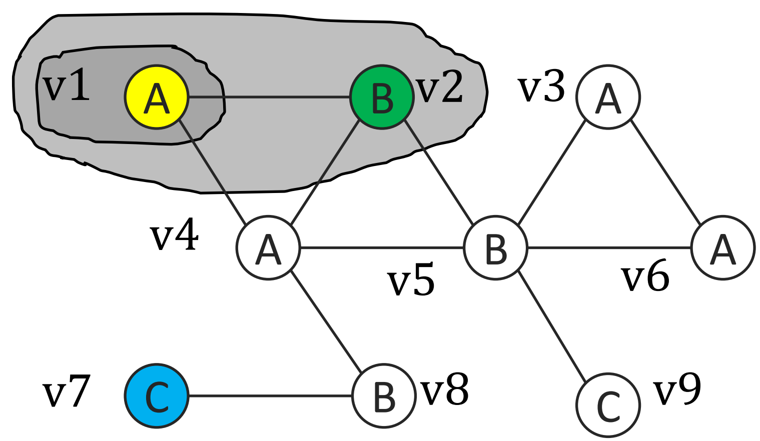
$f_2 = \{(u1, v1), (u2, v2)\}$



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



(a). Query graph q



(b). Data graph G

State Space Tree Exploration

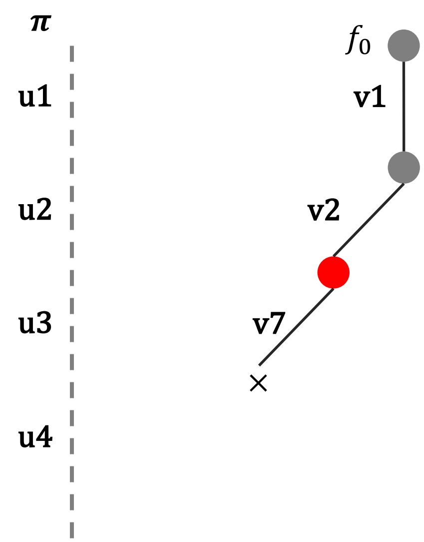
$u1.C = \{v1, v3, v4, v6\}$

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

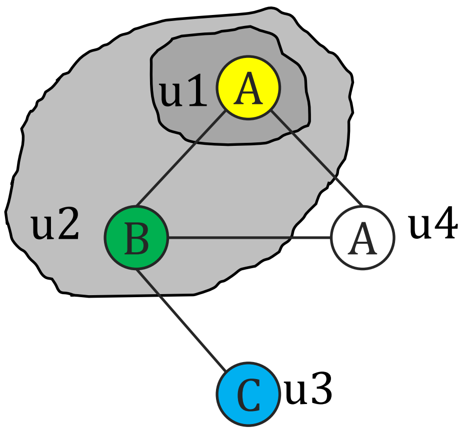
$u3.C = \{v7, v9\}$

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

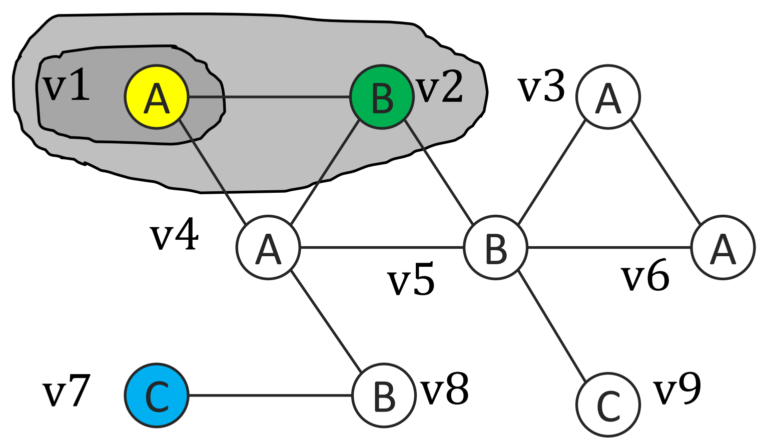
$f_2 = \{(u1, v1), (u2, v2)\}$



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



(a). Query graph q



(b). Data graph G

State Space Tree Exploration

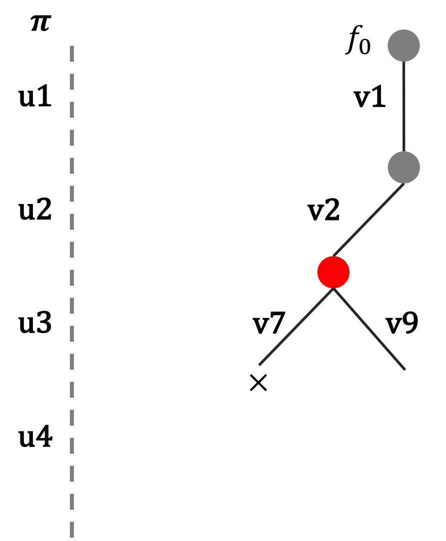
$u1.C = \{v1, v3, v4, v6\}$

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

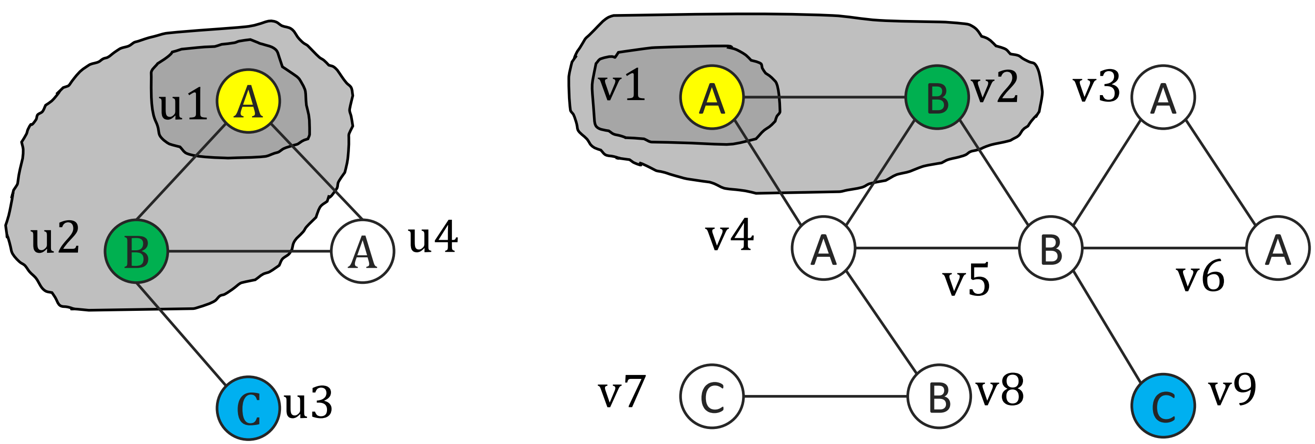
$u3.C = \{v7, v9\}$

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

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



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



(a). Query graph q (b). Data graph G

State Space Tree Exploration

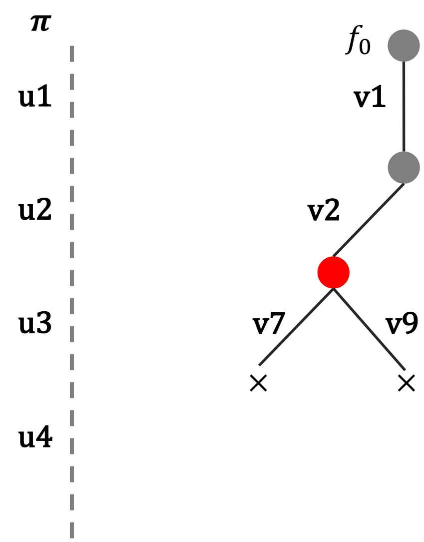
$u1.C = \{v1, v3, v4, v6\}$

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

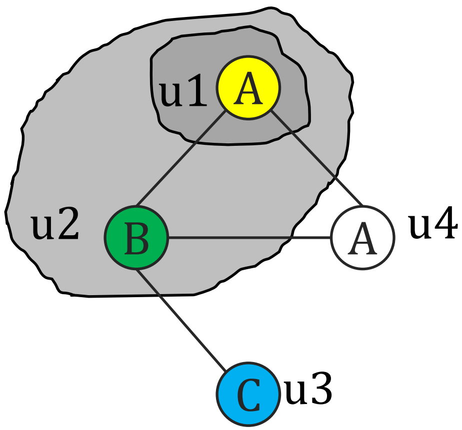
$u3.C = \{v7, v9\}$

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

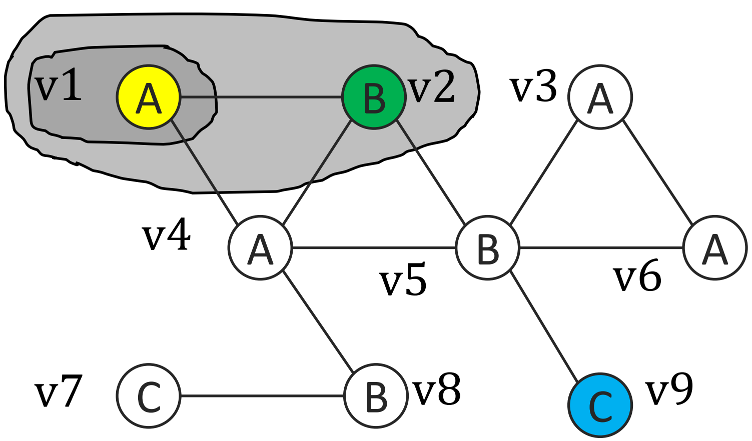
$f_2 = \{(u1, v1), (u2, v2)\}$



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



(a). Query graph q



(b). Data graph G

State Space Tree Exploration

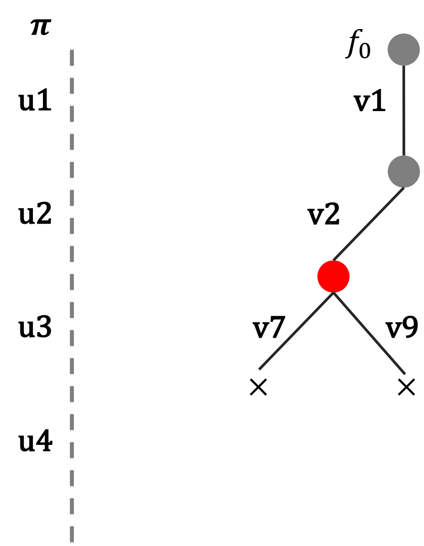
$u1.C = \{v1, v3, v4, v6\}$

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

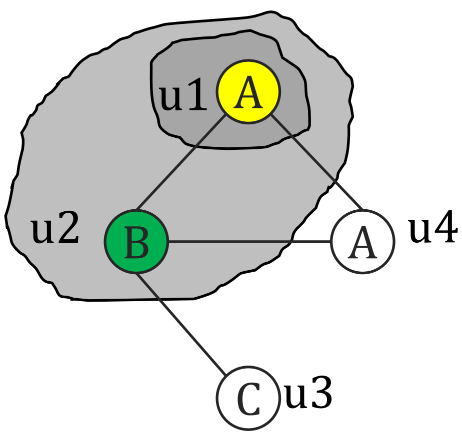
$u3.C = \{v7, v9\}$

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

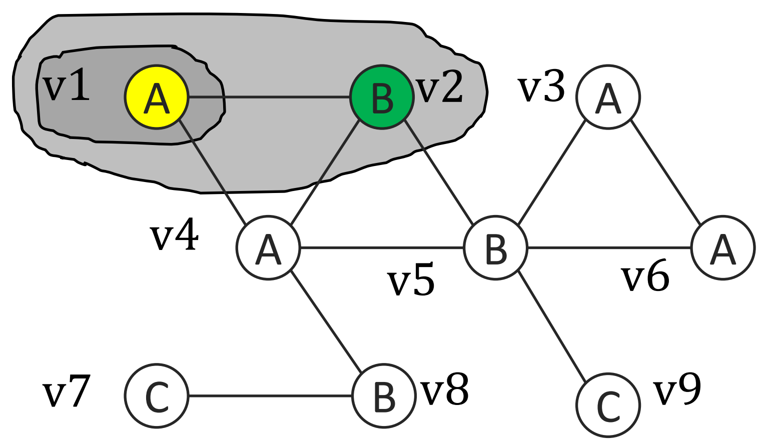
$f_2 = \{(u1, v1), (u2, v2)\}$



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



(a). Query graph q



(b). Data graph G

State Space Tree Exploration

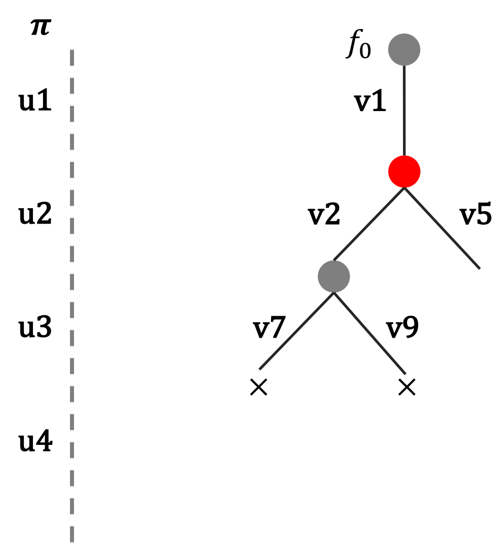
$u1.C = \{v1, v3, v4, v6\}$

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

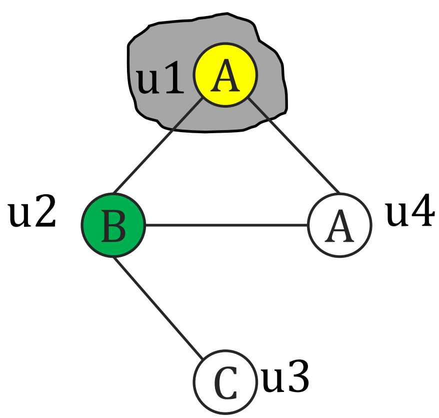
$u3.C = \{v7, v9\}$

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

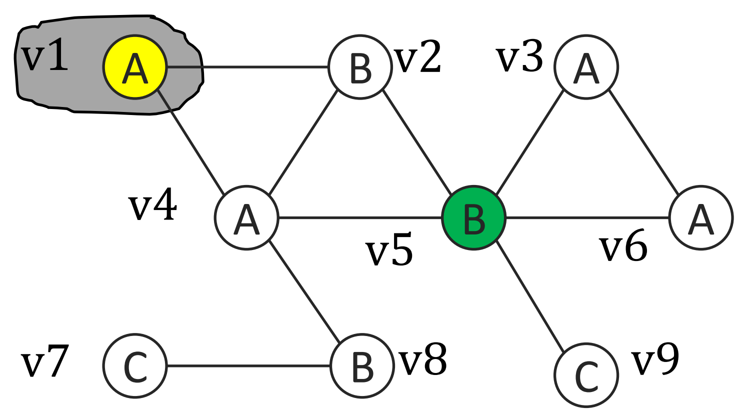
$f_1 = \{(u1, v1)\}$



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



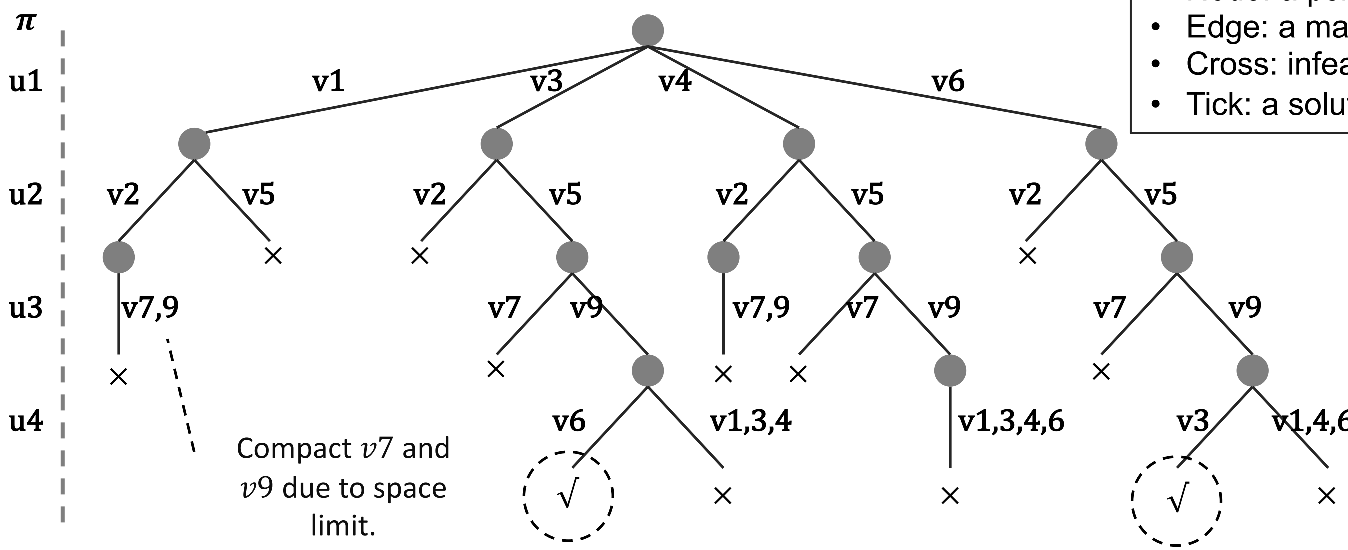
(a). Query graph q

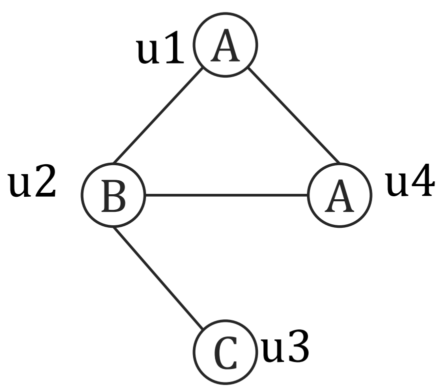


(b). Data graph G

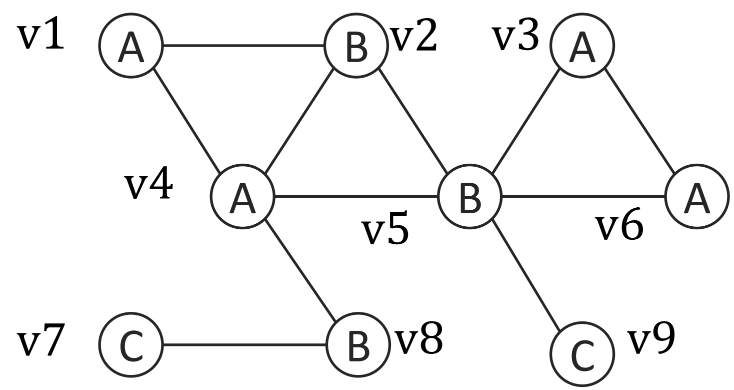
State Space Tree Exploration

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





(a). Query graph q



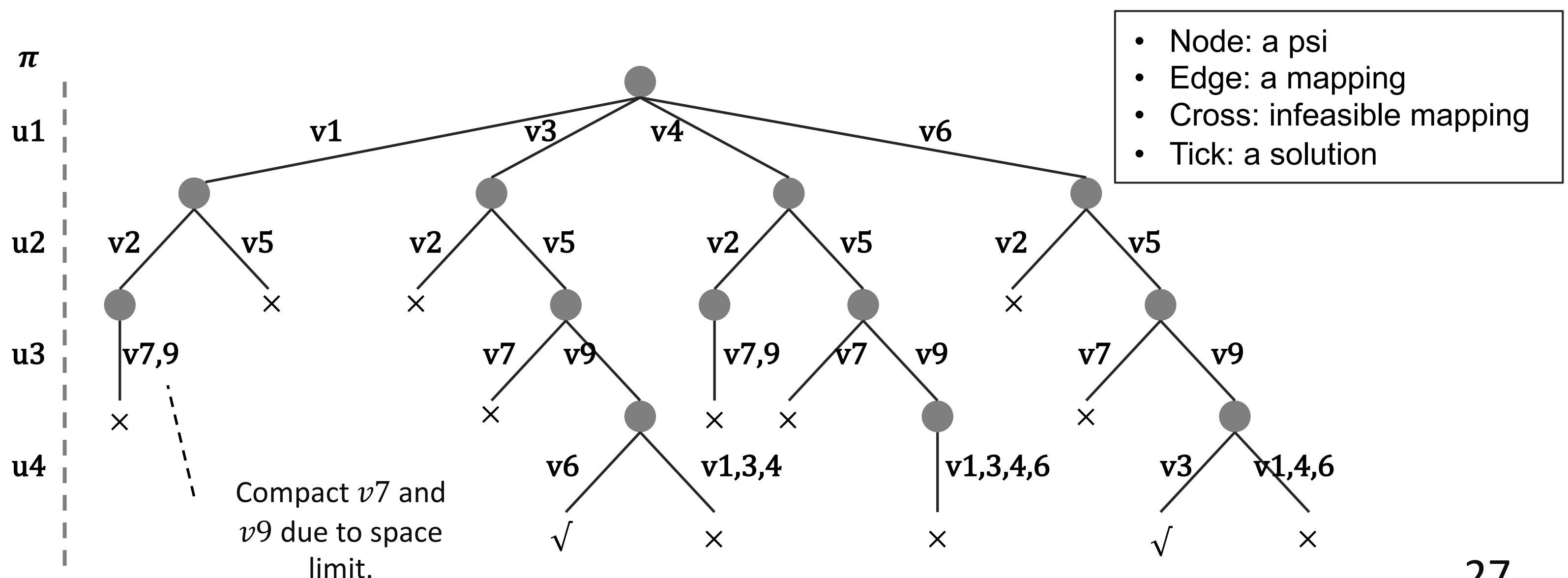
(b). Data graph G

Properties of the State Space Tree

$$|H_i| = \begin{cases} 1 & i = 0 \\ \prod_{j=0}^{i-1} b_j & 0 < i \leq |\pi| \end{cases}$$

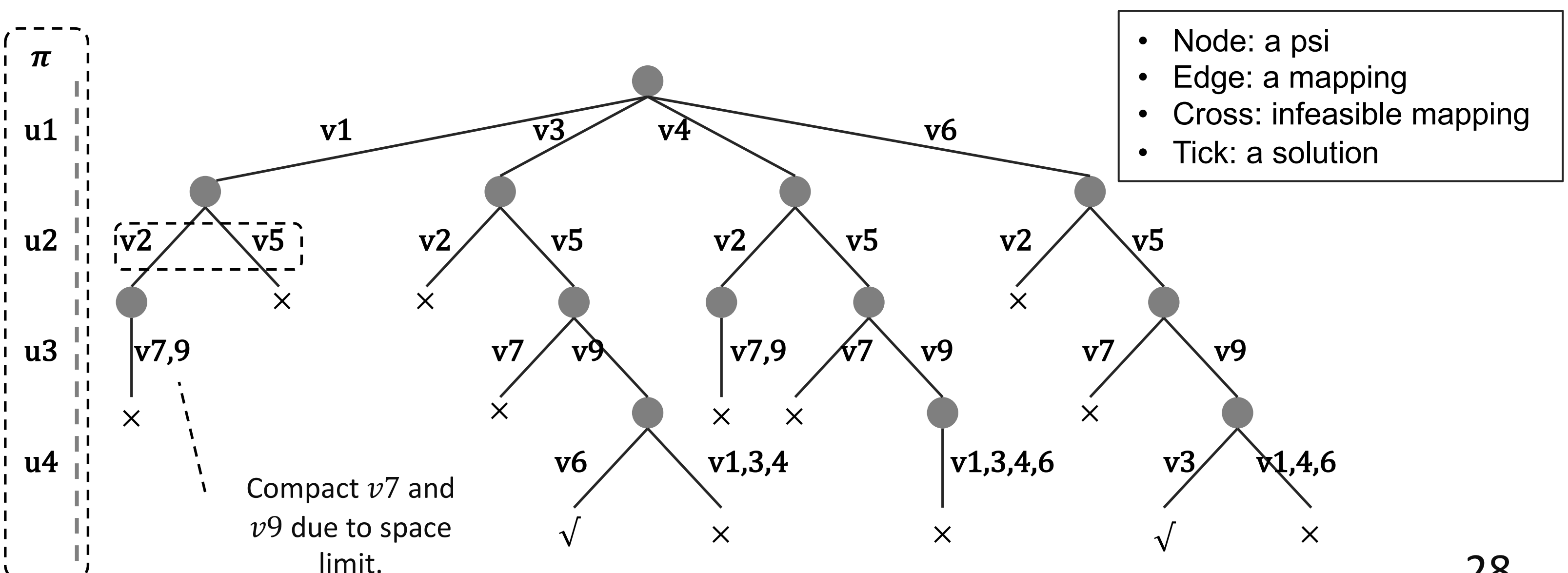
$|H_i|$ is the number of nodes at depth i in the state space tree H. b_j is the average branching factor of nodes at depth i in H.

- There is an **exponential** number of nodes in H.
- The tree H has an **irregular** shape.
- H is **flat**, i.e., $|\pi| \ll \max_{0 \leq i \leq |\pi|} |H_i|$.



Research Focus of Sequential Algorithms

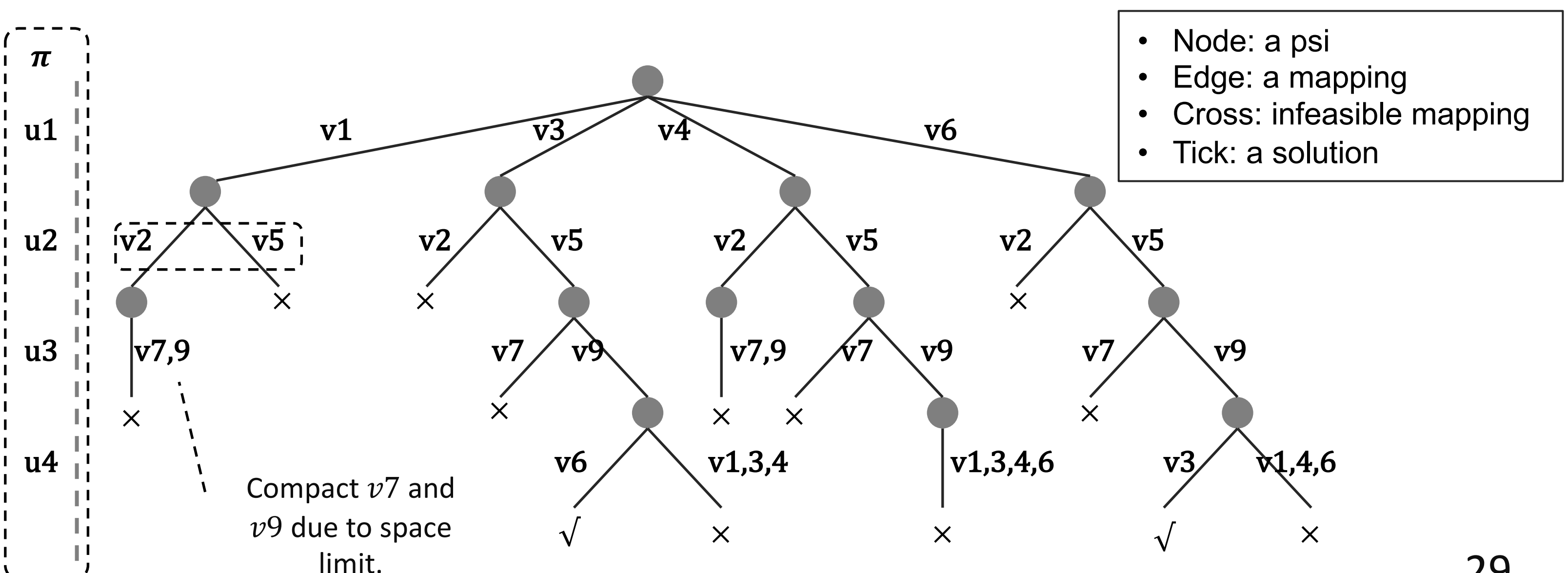
- Optimize the matching order.
- Minimize the search breadth (branches) of each state.



Research Focus of Sequential Algorithms

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

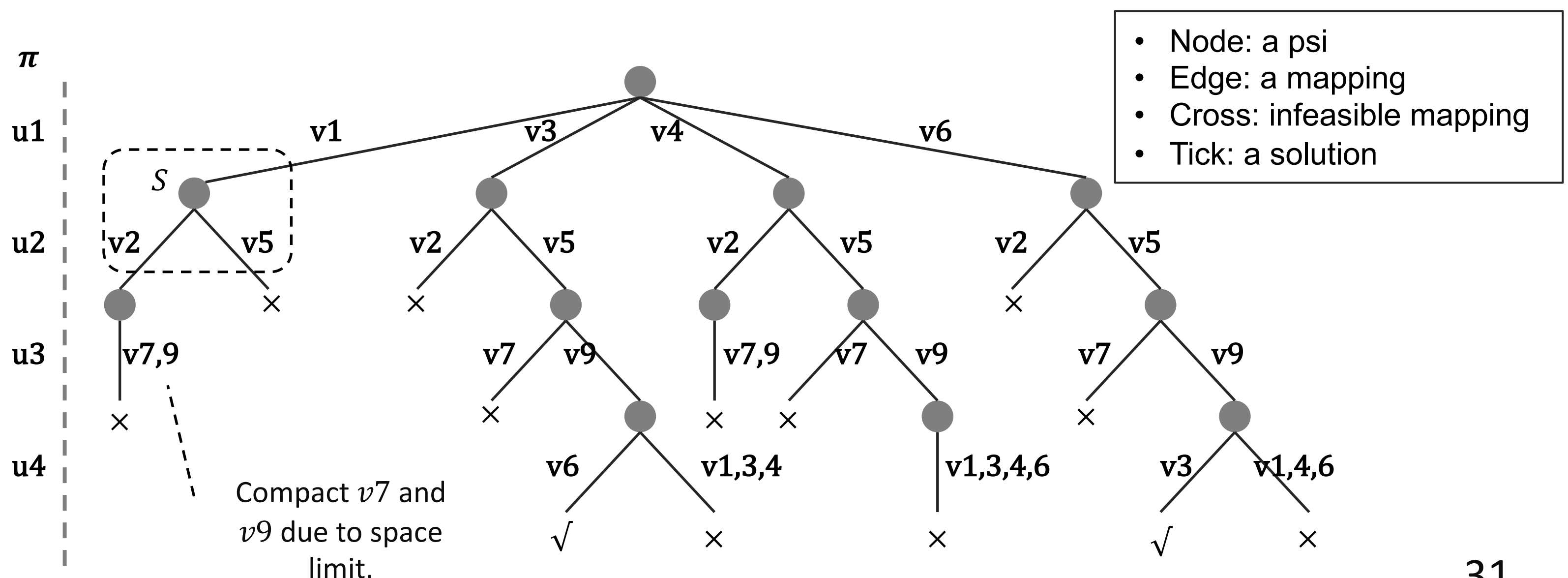
The focus of our paper is to explore the tree in parallel.



Design of Parallel Subgraph Matching (PSM)

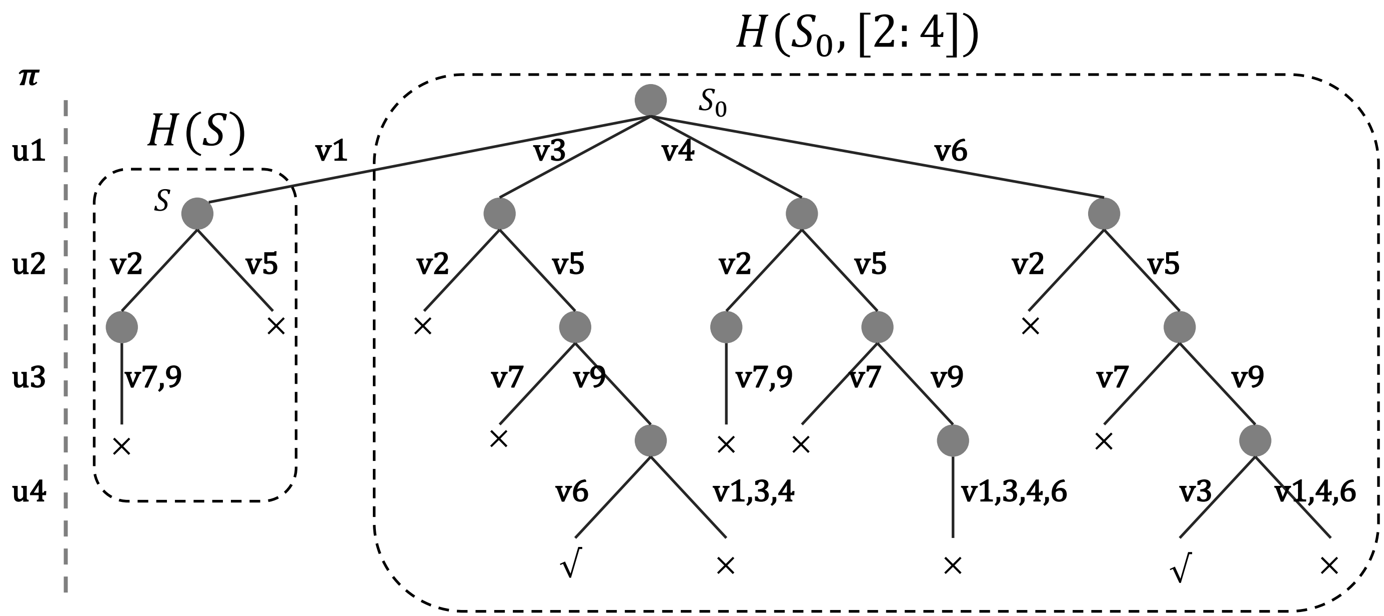
Parallel Task – Fine-Grained Parallelism

- **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.



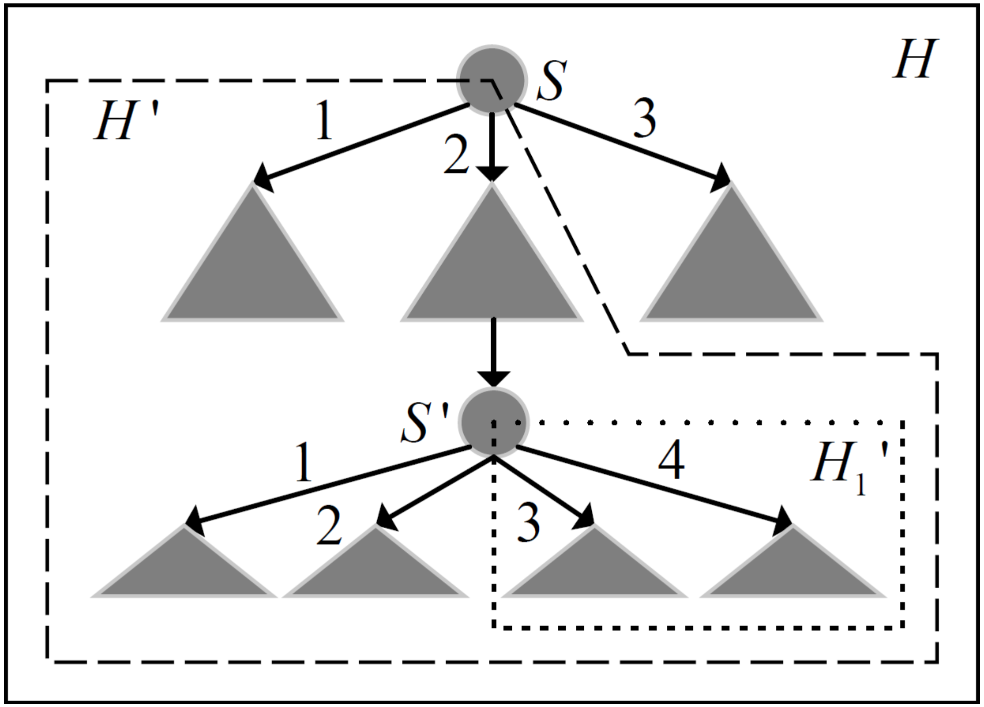
Parallel Task – Coarse-Grained Parallelism

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



Parallel Task – Coarse-Grained Parallelism

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



Load Balancing

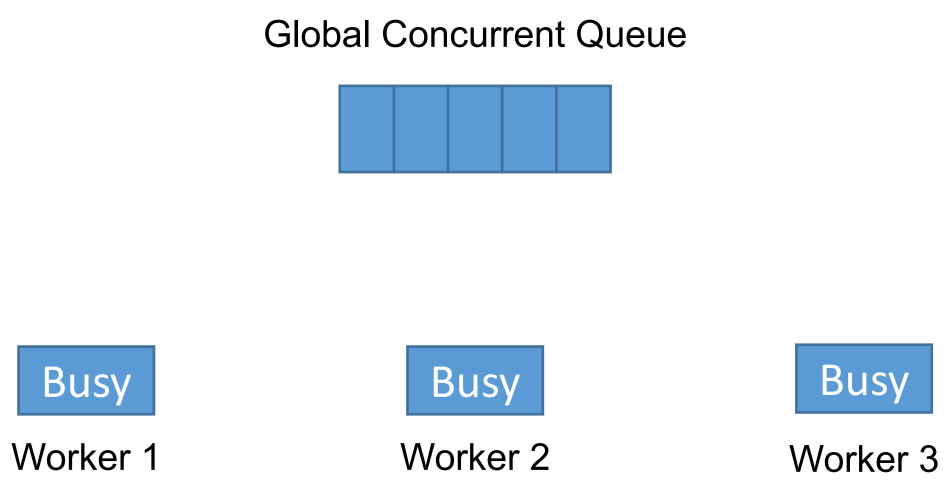
- It is hard to assign equal amounts of workload to workers at the beginning (**static load balancing**), because H is constructed on the fly and irregular.
- PSM designs a **dynamic load balancing** approach to resolve the load imbalance problem.

Load Balancing – Communication Model

- PSM adopts a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- PSM adopts 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.

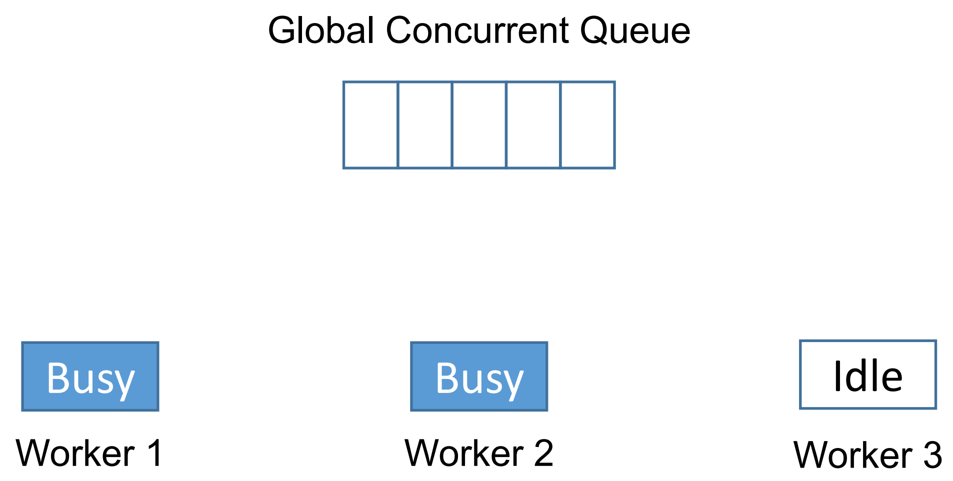
Load Balancing – Communication Model

- PSM adopts a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- PSM adopts 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.



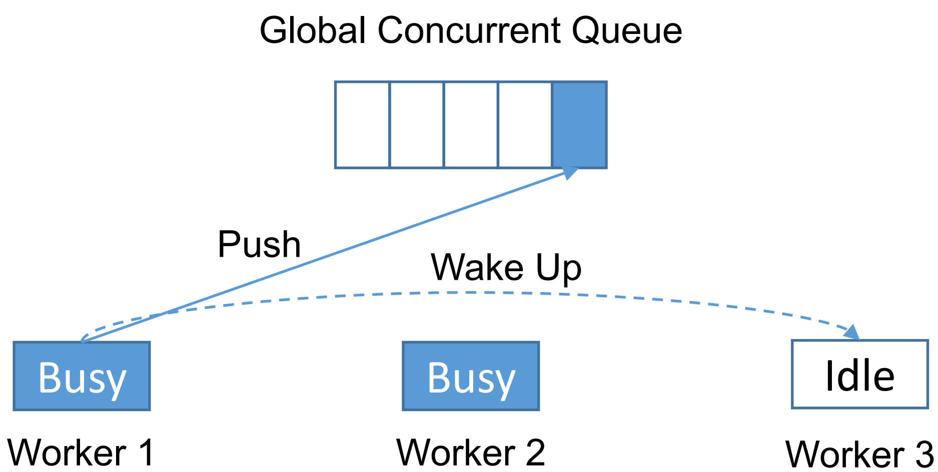
Load Balancing – Communication Model

- PSM adopts a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- PSM adopts 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.



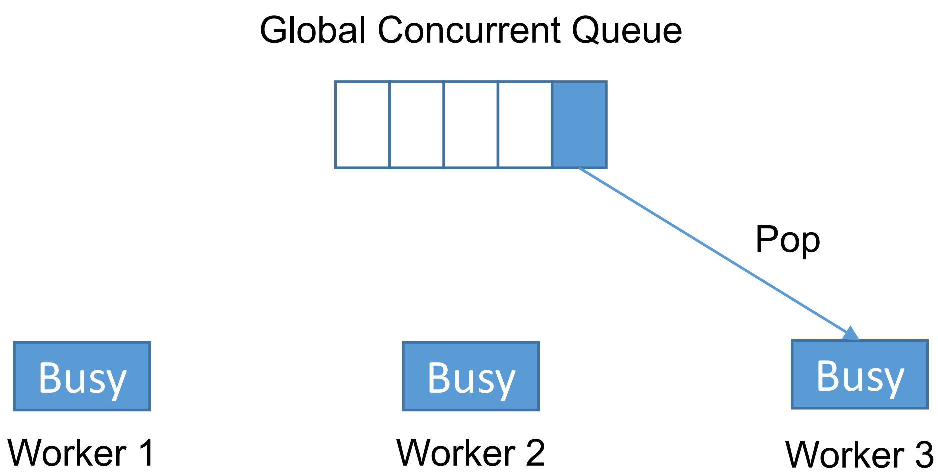
Load Balancing – Communication Model

- PSM adopts a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- PSM adopts 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 – Communication Model

- PSM adopts a decentralized communication model, i.e., PSM has no master responsible for assigning tasks.
- PSM adopts 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 – Task Split

- **Goal:** Divide the task into two subtasks with nearly equal workload.

Load Balancing – Task Split

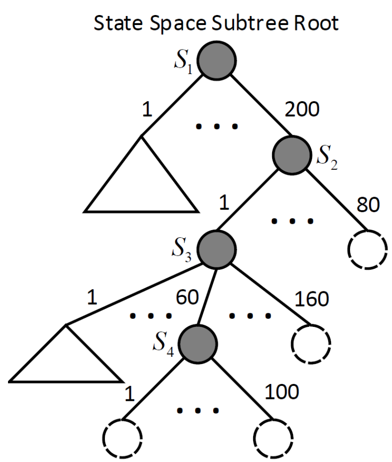
- **Goal:** Divide the task into two subtasks with nearly equal workload.
- **Challenge:** As the state space tree is constructed **on the fly** and **irregular**, it is hard to estimate the workload of a task.

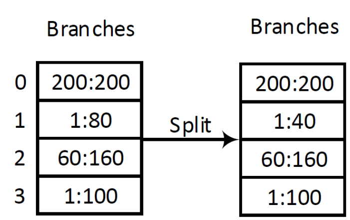
Load Balancing – Task Split

- **Goal:** Divide the task into two subtasks with nearly equal workload.
- **Challenge:** As the state space tree is constructed **on the fly** and **irregular**, it is hard to estimate the workload of a task.
- **Heuristic:** As the state space tree grows **exponentially**, the workload of the subtree rooted at a shallow depth is much more than that of one rooted at a deep depth.

Load Balancing – Task Split

- **Goal:** Divide the task into two subtasks with nearly equal workload.
- **Challenge:** As the state space tree is constructed **on the fly** and **irregular**, it is hard to estimate the workload of a task.
- **Heuristic:** As the state space tree grows **exponentially**, the workload of the subtree rooted at a shallow depth is much more than that of one rooted at a deep depth.
- **Solution :** Split the branches of a state close to the subtree root evenly to generate a new task.





We obtain a new task $H(S_2, [41,80])$.

Evaluation

Experimental Setup

- **Algorithms Under Study:**
  - **pQSI:** QuickSI [3] (VLDB'08) parallelized with PSM;
  - **pGQL:** GraphQL [4] (SIGMOD'08) parallelized with PSM;
  - **pCFL:** CFL [8] (SIGMOD'16) parallelized with PSM;
  - **PGX [10]:** A parallel BFS approach proposed in GRADES'14;
  - **pRI [11]:** A parallel approach proposed in IPDPS'17;
- **Experimental Environment**:
  - All algorithms are implemented in C++. The source code is compiled with g++ 4.9.3 with –O3 flag enabled.
  - We conduct experiments on a 64-bit Linux machine with 64GB RAM and two Intel Xeon E5-2650 v3 CPUs each of which has ten 2.30GHz physical cores (**20 workers by default**).

Experimental Setup

- **Real World Datasets:**

Dataset	$\|V\|$	$\|E\|$	$\|\Sigma\|$	Avg. Degree
Yeast	3,112	12,519	71	8.04
WordNet	76,853	120,399	5	3.13
Youtube	1,134,890	2,987,624	25	5.27
US Patents	3,774,768	16,518,948	20	8.75

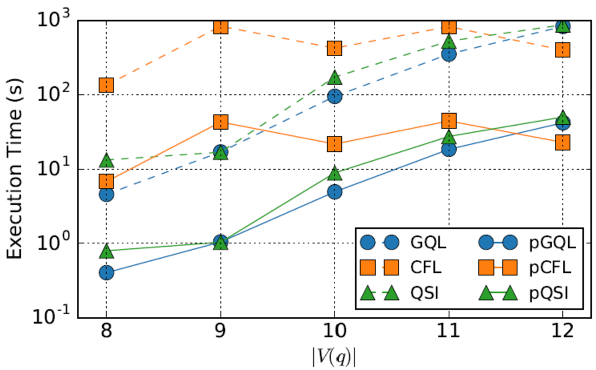
$|V|$ is the number of vertices.
$|E|$ is the number of edges.
$|\Sigma|$ is the number of distinct labels.

- **Query Datasets:**

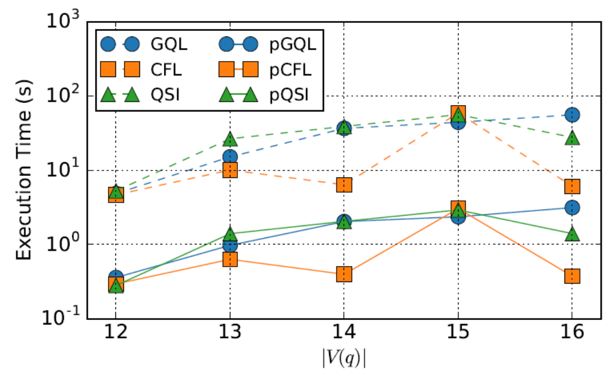
Dataset	Query Set
Yeast, Youtube, US Patents	$q_{12}, q_{13}, q_{14}, q_{15}, q_{16}$
WordNet	$q_8, q_9, q_{10}, q_{11}, q_{12}$

Comparison with Sequential Counterparts

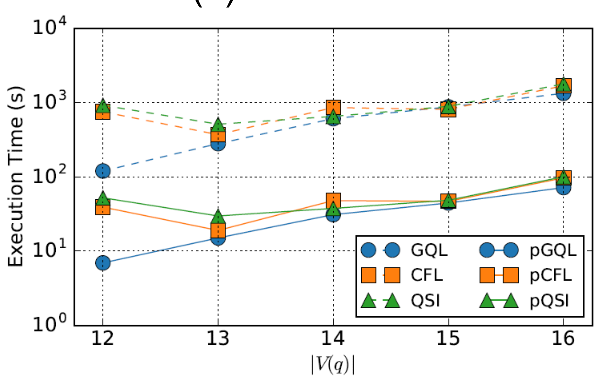
- The parallel algorithms with PSM achieve a speedup of **15.5X-19.5X** over the original sequential algorithms.



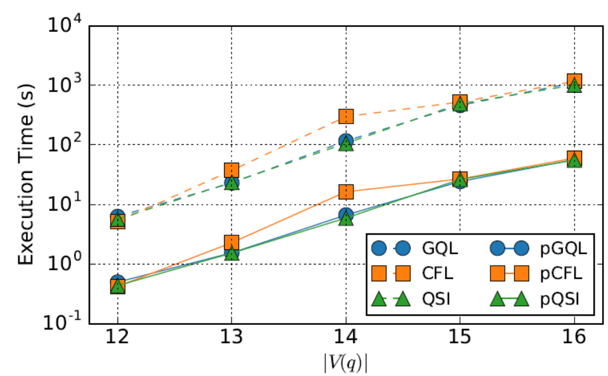
(a). WordNet



(b). Yeast



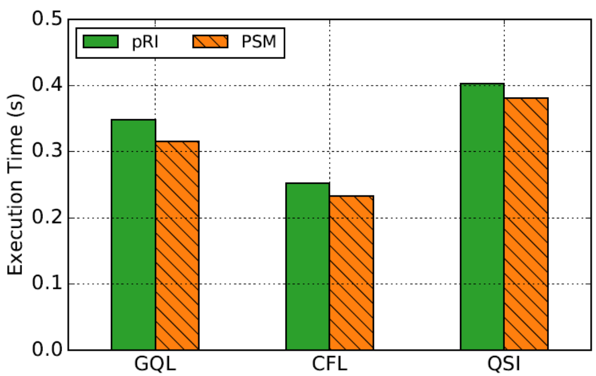
(c). Youtube



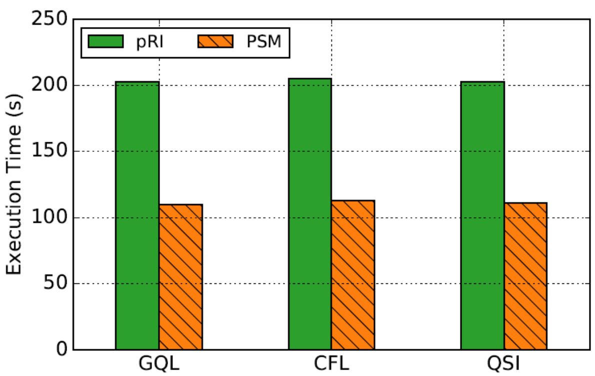
(d). US Patents

Comparison with Existing Parallel Algorithms

- PGX explores the search tree with parallel BFS method. It runs out of the memory due to the exponential number of states. We omit its experiment results.
- pRI takes each state as the parallel task and explores the search tree in DFS.
- For the fair of comparison, we use the same matching order and filtering methods in PGX and pRI with that in PSM.



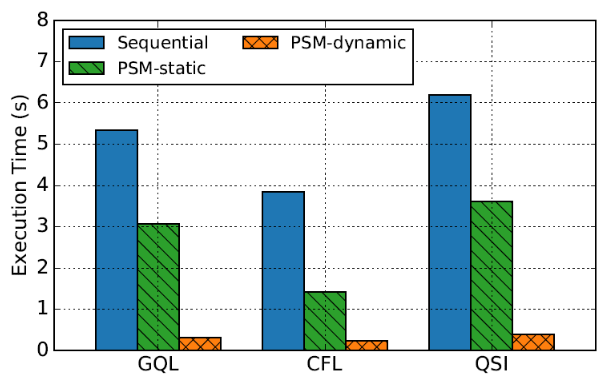
(a). Yeast



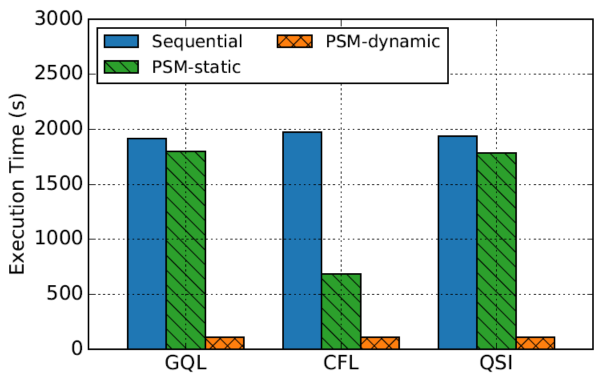
(b). Youtube

Evaluate the Dynamic Load Balancing of PSM

- **Static Load Balancing:** Assign the states at depth 1 of the state space tree evenly to workers.
- **Dynamic Load Balancing:** The load balancing strategy proposed in PSM.



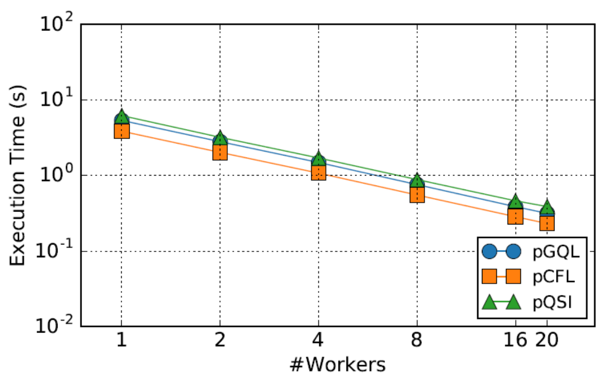
(a). Yeast



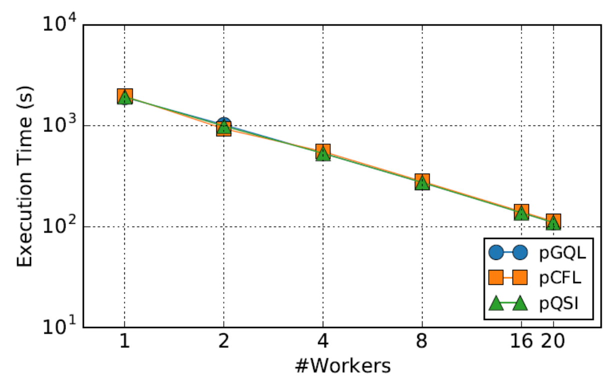
(b). Youtube

Evaluate the Scalability of PSM

- PSM achieve almost linear speedups on the two datasets.
- The speedups with 20 workers are about 16X on Yeast and 17.4X on Youtube.



(a). Yeast



(b). Youtube

Memory Consumption

- The memory consumption of the auxiliary data structures and the candidate sets is very small.
  - **Note:** We find the results without materializing the results into file systems.

	Yeast		Youtube	
	Task Queue	Candidates	Task Queue	Candidates
pGQL	2.969 KB	0.0387 MB	2.969 KB	0.2257 MB
pCFL	2.969 KB	0.0443 MB	2.969 KB	0.2538 MB
pQSI	2.969 KB	0.0366 MB	2.969 KB	0.2309 MB

Conclusion

Conclusion

- We propose a parallel subgraph matching framework called PSM to accelerate backtracking subgraph matching algorithms.
- Extensive experiments on a variety of real world datasets demonstrate the efficiency and robustness of PSM.

References

[1]. Ullmann, Julian R. "An algorithm for subgraph isomorphism." Journal of the ACM (JACM) 23.1 (1976): 31-42.
[2]. Cordella, Luigi P., et al. "A (sub) graph isomorphism algorithm for matching large graphs." IEEE transactions on pattern analysis and machine intelligence 26.10 (2004): 1367-1372.
[3]. Shang, Haichuan, et al. "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism." Proceedings of the VLDB Endowment 1.1 (2008): 364-375.
[4]. He, Huahai, and Ambuj K. Singh. "Graphs-at-a-time: query language and access methods for graph databases." Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008.
[5]. Zhang, Shijie, Shirong Li, and Jiong Yang. "GADDI: distance index based subgraph matching in biological networks." Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. ACM, 2009.
[6]. Zhao, Peixiang, and Jiawei Han. "On graph query optimization in large networks." Proceedings of the VLDB Endowment 3.1-2 (2010): 340-351.
[7]. Han, Wook-Shin, Jinsoo Lee, and Jeong-Hoon Lee. "Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases." Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013.
[8]. Bi, Fei, et al. "Efficient subgraph matching by postponing cartesian products." Proceedings of the 2016 International Conference on Management of Data. ACM, 2016.
[9]. Sun, Zhao, et al. "Efficient subgraph matching on billion node graphs." Proceedings of the VLDB Endowment 5.9 (2012): 788-799.
[10]. Raman, Raghavan, et al. "Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism." Proceedings of Workshop on GRAph Data management Experiences and Systems. ACM, 2014.

References

[11]. R. Kimming, H. Meyerhenke, and D. Strash. “Shared memory parallel subgraph enumeration.“ IPDPS Workshop. 2017.
[12]. Tran, Ha-Nguyen, Jung-jae Kim, and Bingsheng He. "Fast subgraph matching on large graphs using graphics processors." International Conference on Database Systems for Advanced Applications. Springer International Publishing, 2015.
[13]. Lee, Jinsoo, et al. "An in-depth comparison of subgraph isomorphism algorithms in graph databases." Proceedings of the VLDB Endowment. Vol. 6. No. 2. VLDB Endowment, 2012.

Thanks!