HiPerMotif: Novel Parallel Subgraph Isomorphism in Large-Scale Property Graphs

Mohammad Dindoost, Oliver Alvarado Rodriguez, Bartosz Bryg, Ioannis Koutis, and David A. Bader

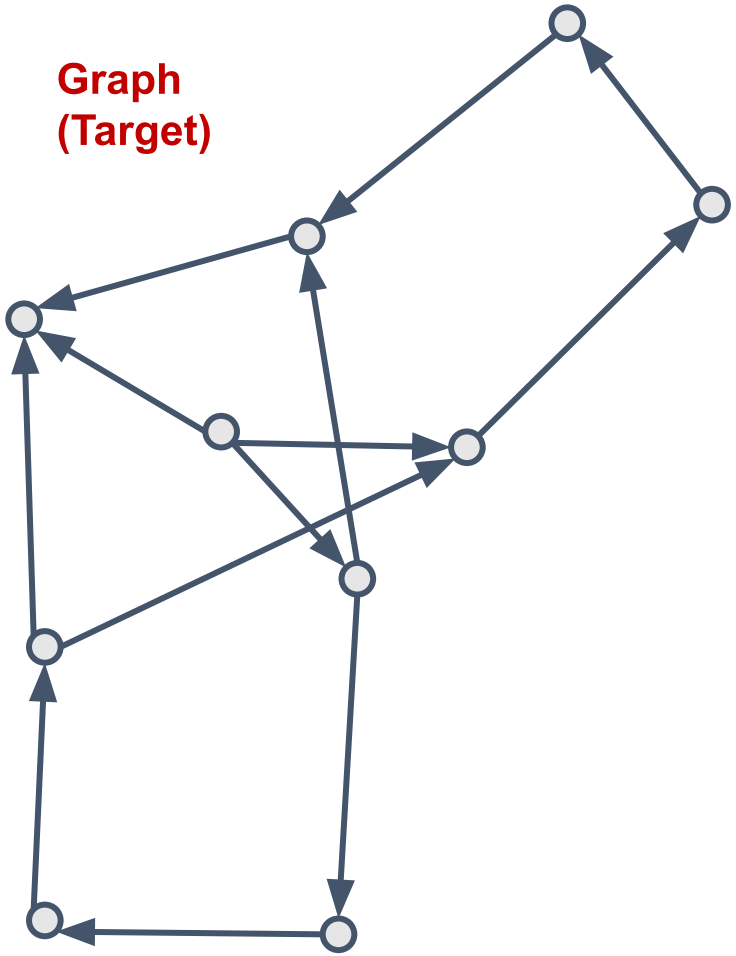
ChapelCon '25

October 9, 2025

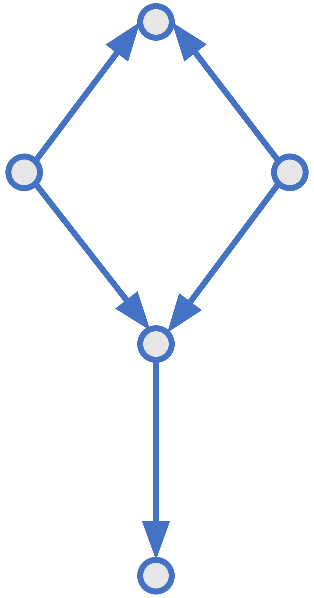
NJIT
New Jersey Institute of Technology

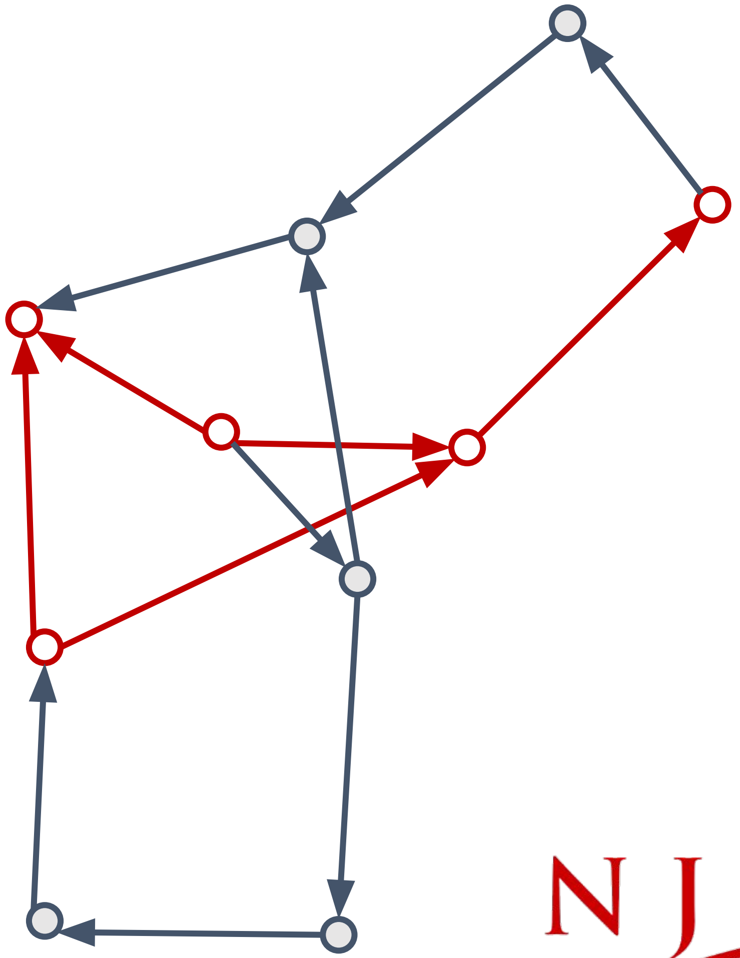
Problem Statement

Graph (Target)



Pattern (Motif)





NJIT New Jersey Institute of Technology

Motivation

- **Subgraph Isomorphism Challenge**

  Identifying small pattern graphs within larger graphs is a complex and computationally heavy problem in graph theory.(NP-Complete)

- **Applications Across Domains**

  Subgraph isomorphism impacts neuroscience, biology, social networks, cybersecurity, and fraud detection.

- **Scalability Issues**

  Traditional algorithms struggle with large graphs due to exhaustive search causing slow runtimes and memory failures.

- **HiPerMotif Solution**

  HiPerMotif introduces a hybrid parallel algorithm improving initialization and scaling for large graph analysis. (Chapel-Arachne)

NJIT New Jersey Institute of Technology

Research Background

[Luigi P. Cordella, et al.]

- Ullmann's algorithm, LAD (Labeled Distance), RI / RI-Plus / RI-DS, TurboISO, Glasgow Subgraph Solver, VF2, …
- NetworkX, DotMotif, iGraph, …
- VF2 is widely used.
- Great potential to be parallel.
- In neuroscience there are some tools already adapted it.

A Bird's-Eye View of Arachne

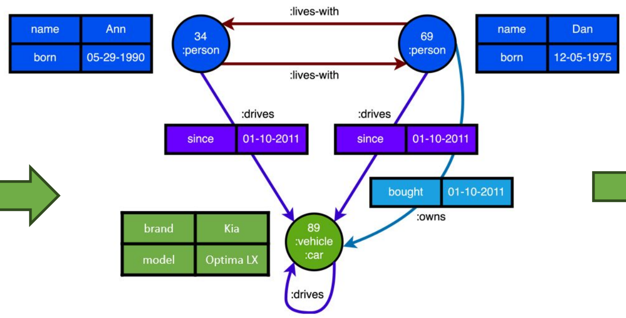
id	label	name	born	brand	model
34	person	Ann	1990	NULL	NULL
69					
89					
89					

src id	dst id	relationship	since	bought
34	69	lives-with	NULL	NULL
69	34	lives-with	NULL	NULL
34	89	drives	2011	NULL
69	89	drives	2011	NULL
69	89	owns	NULL	2011
89	89	drives	NULL	NULL

Load in large CSVs, HDF5s, Parquets, matrix market files, etc.

read_matrix_market_file()
add_edges_from()
rmat()
gnp()

Convert dataframes to graphs or generate your own synthetic graphs.



Work with your data as a graph.

bfs_layers()
subgraph_isomorphism()
triangle_counting()
subgraph_view()

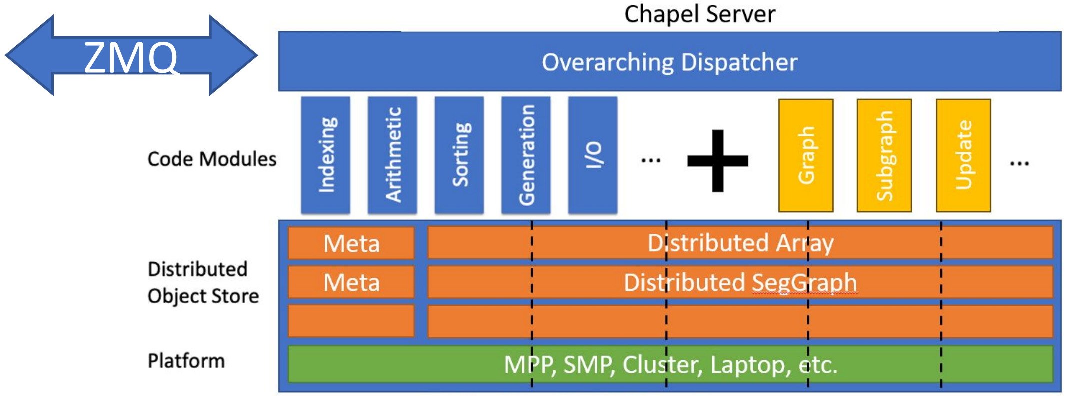
Perform analysis or filter for NetworkX, iGraph, or graph-tool.

User edits a Python script or a Jupyter Notebook.

User

```
import arkouda as ak
import arachne as ar

## Get src and dst from input file.

graph = ar.PropGraph()

## Generate label_df and relationships_df from input file.

graph.load_edge_attributes(relationships_df)
graph.load_node_attributes(label_df)

## User generates labels_to_find and relationships_to_find.
returned_nodes = graph.node_attributes["column"] == 1
returned_edges = graph.edge_attributes["column"] == 2

subgraph_src = ak.in1d(returned_edges[0], returned_nodes)
subgraph_dst = ak.in1d(returned_edges[1], returned_nodes)

kept_edges = subgraph_src & subgraph_dst

subgraph_src = subgraph_src[kept_edges]
subgraph_dst = subgraph_dst[kept_edges]

subgraph = ar.Graph()
subgraph.add_edges_from(subgraph_src, subgraph_dst)
## Run some other operations on subgraph!
```

Easily usable through NetworkX-like API.



Original image source: https://chapel-lang.org/CHIUW/2020/Reus.pdf was modified for this presentation

Runs on any hardware, data stays in the back-end, user calls API through Python: powerful and productive. (Image credit: [Reus 2020])

OPEN SOURCE: https://github.com/Bears-R-Us/arkouda-njit & https://github.com/Bears-R-Us/arkouda
PAPERS & TALKS AT: IEEE HPEC, IEEE HiPC, IEEE IPDPS, ACM PPoPP, SIAM PP, Nature, & MDPI Algorithms

NJIT New Jersey Institute of Technology

HiPerMotif

- **Edge-Centric Initialization**

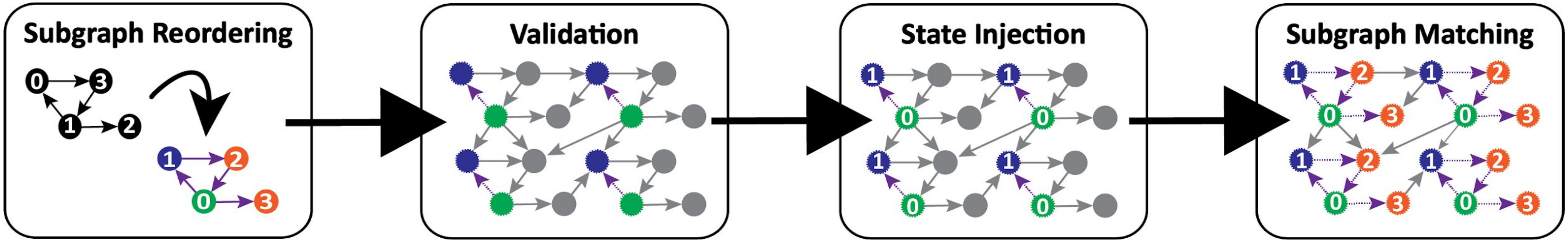
  HiPerMotif begins by identifying and validating all first-edge mappings, skipping empty initial mappings.
- **Pattern Graph Reordering**

  Structural reordering prioritizes vertices with high connectivity to optimize the matching process.
- **Parallel Edge Validation**

  Each edge mapping is validated independently, enabling natural parallelization for efficiency.
- **Framework Implementation**

  Implemented within Arkouda/Arachne, HiPerMotif scales efficiently for massive dataset analysis.

HiPerMotif



Allows us to reduce the search space by starting off with vertices that are structurally significant by number of out-neighbors.

Allows us to reduce the search space further by allowing the algorithm to dynamically pick initial vertices and edges that are semantically valuable.

We force subgraph searching to “time travel” and skip unneeded states generated by vertices 0 and 1 of the subgraph and rather start at depth 2.

Match the full subgraphs.

NJIT New Jersey Institute of Technology

Performance (Synthetic and Real-World Graphs)

- **Evaluation on Synthetic Graphs**

  HiPerMotif was tested on Erdős-Rényi, Barabási-Albert, and Watts-Strogatz graph models with varied densities and sizes.

- **Testing on Real-World Datasets**

  Datasets included neuroscience connectomes, communication and social networks, plus a massive human cortex graph.

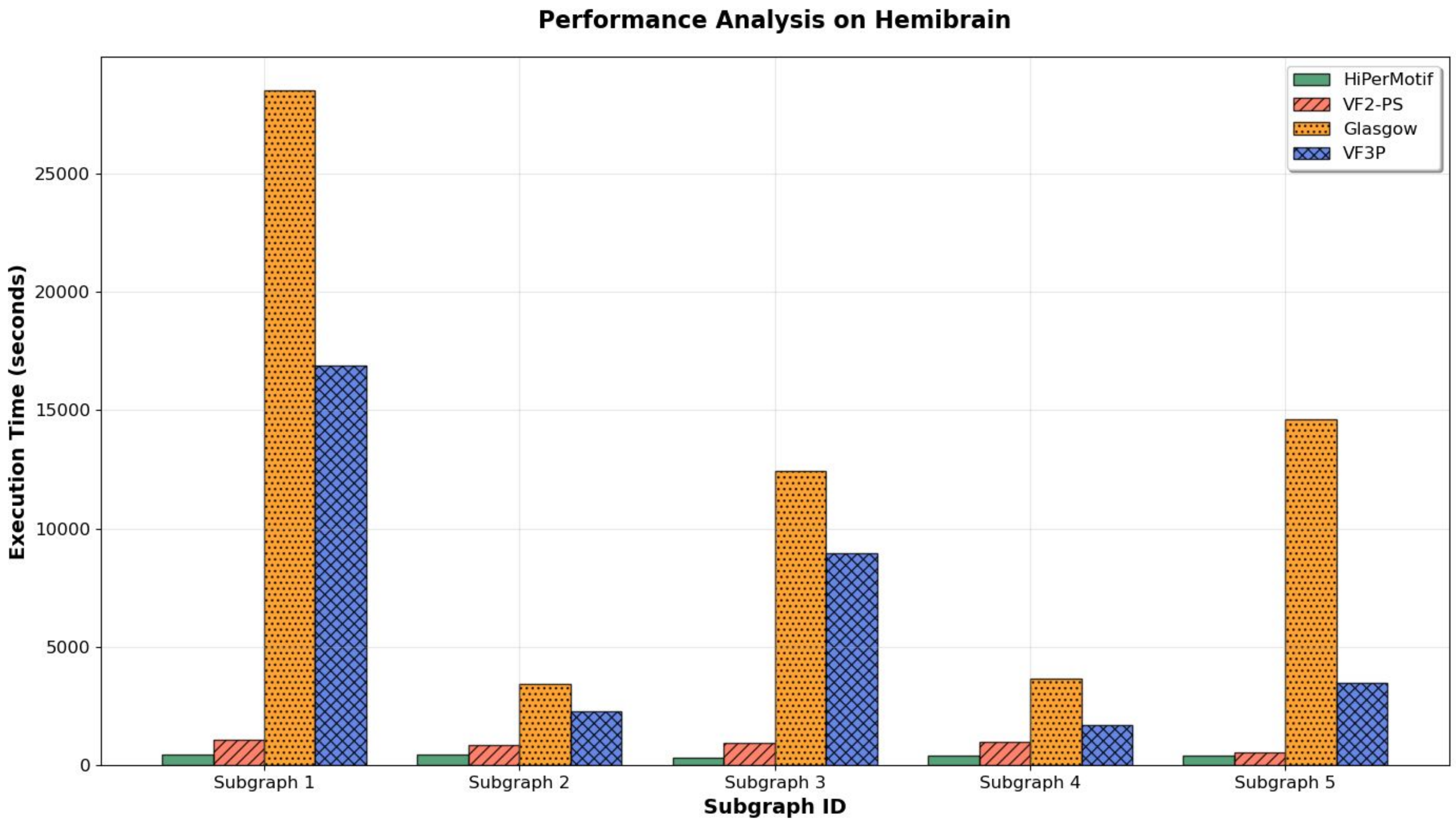
- **Superior Performance Metrics**

  HiPerMotif achieved up to **66×** speedup and processed large graphs where baselines failed due to memory limits.

- **Impact of Structural Reordering**

  Structural reordering strategy alone contributed up to **5×** speedup, enhancing HiPerMotif's efficiency.

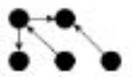
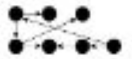
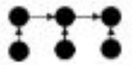
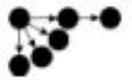
Neuroscience (Hemibrain Dataset)



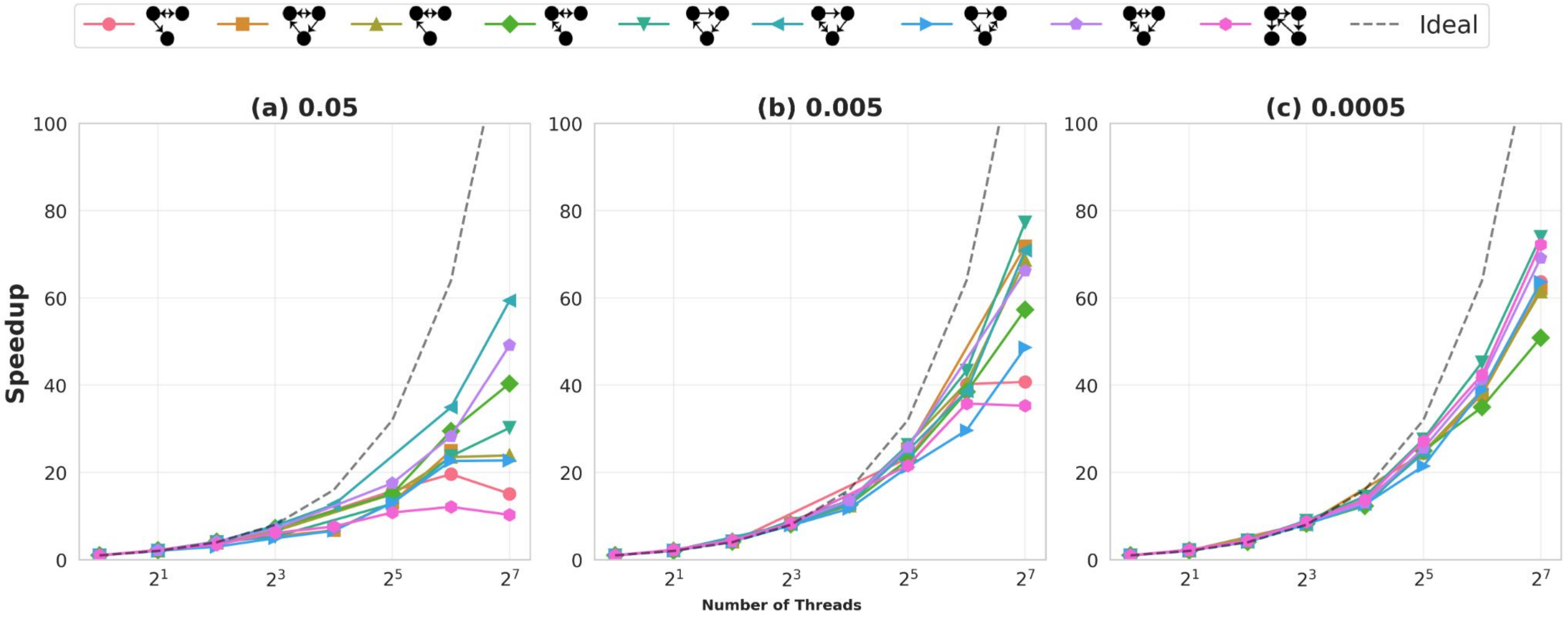
- All Motifs created randomly from 3 to 20 nodes
- Up to 66X speedups
- McCreesh et al, The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants
- Carletti et al, A parallel algorithm for subgraph isomorphism

H01 Dataset (Large-Scale Network)

50K vertices and 150 Millions edges
representing
a cubic millimeter of human cortex.

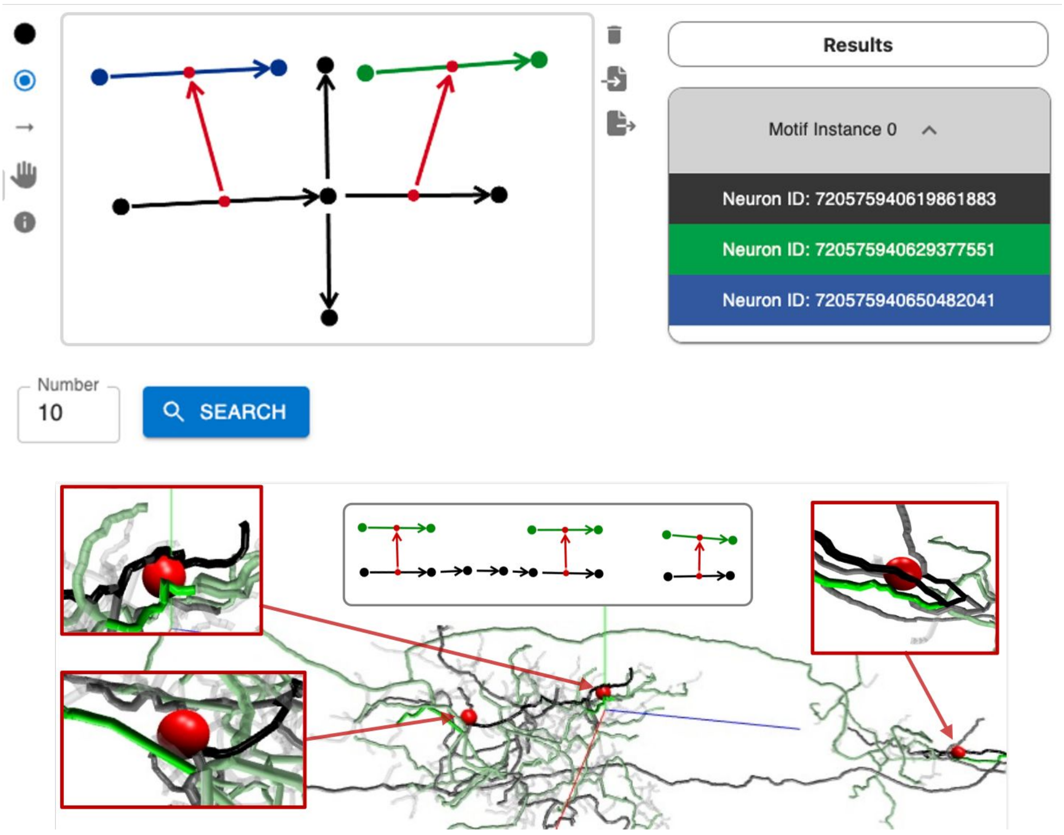
Motif	H01 (seconds)
	571.94
	1011.62
	21.23
	363.54
	1209.82

Parallel speedup

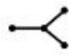
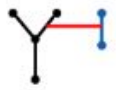
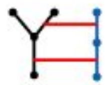
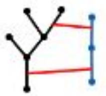
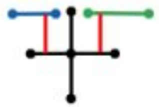
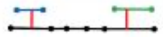


MOMO: Use Case

Our Collaboration with Harvard



M. Shewarega, J. Troidl, et al. / ToMo

Subgraphs	Arachne (s)	NetworkX (s)
	2.48	336.45
	3.62	173.75
	2.88	5,980.54
	339.46	16,436.85
	1.56	435.07
	78.77	810.23
	4.10	1,018.23
	38.06	>12,000

Dataset: 13,000 neurons with over 500,000 synaptic connections

using Arachne-HiPerMotif vs NetworkX VF2: Up to **2,000 X** faster!

NJIT
New Jersey Institute of Technology

Thank you all for your attention.

HiPerMotif is open-source and available on GitHub, so feel free to explore the code, try it out, and reach out with any feedback.

I'd be happy to take any questions you have.

VF Family (VF, VF2, VF2+, VF2++, VF3P)

[V. Carletti, P. Foggia, M. Vento, A. Juttner, P. Madarasi, A. Saggese, C. Sansone, at al]

Algorithm 1 *A high level description of VF2*

procedure VF2(Mapping $\mathfrak{m}$, ProblemType PT)
    if $\mathfrak{m}$ covers V_1 **then**
        Output($\mathfrak{m}$)
    else
        Compute the set $P_{\mathfrak{m}}$ of the candidate pairs for extending $\mathfrak{m}$
        for all $p \in P_{\mathfrak{m}}$ **do**
            if $\mathrm{Cons}_{PT}(p, \mathfrak{m}) \wedge \neg\mathrm{Cut}_{PT}(p, \mathfrak{m})$ **then**
                call VF2($extend(\mathfrak{m}, p)$, PT)

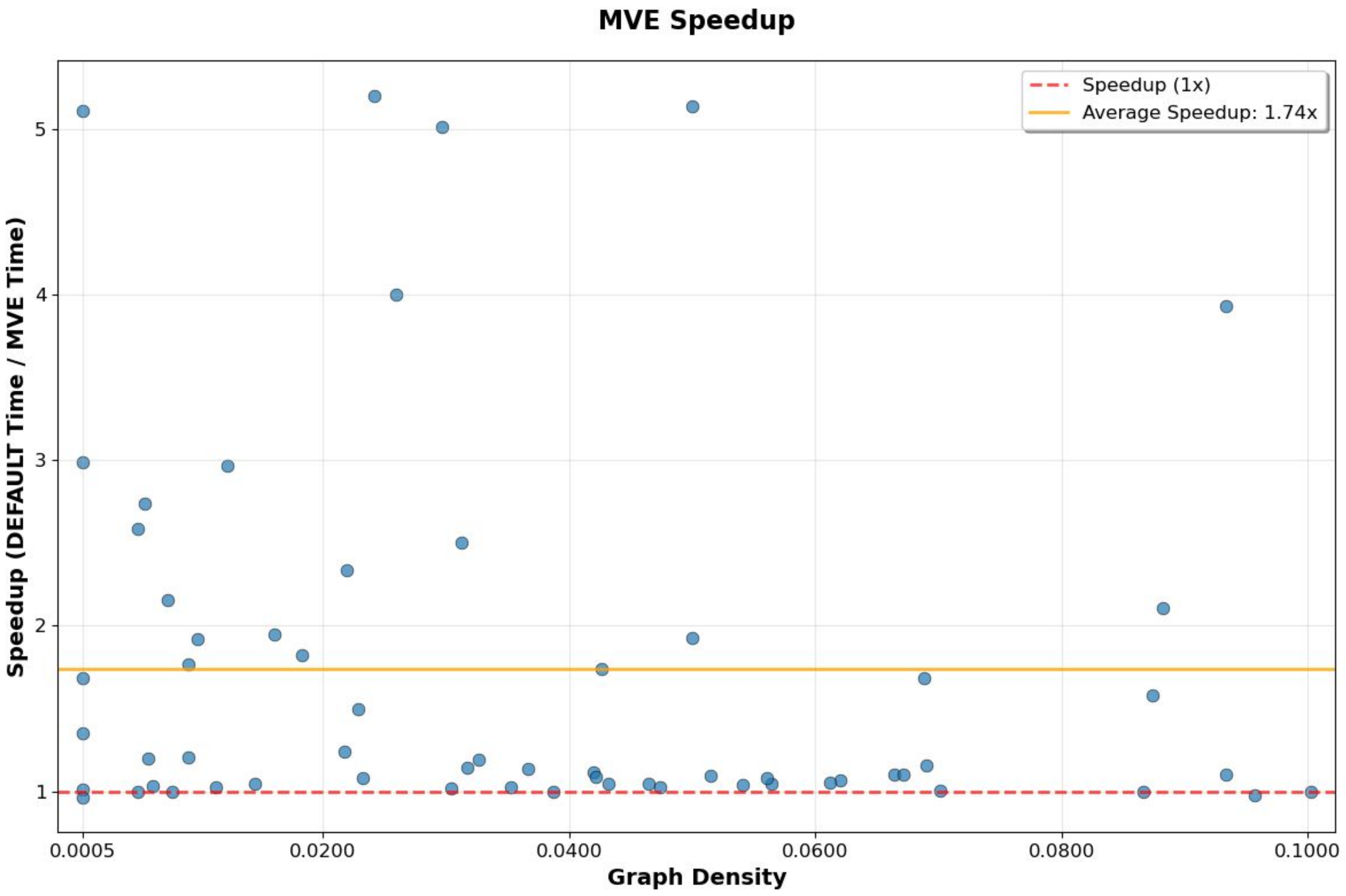
```
PROCEDURE Match(s)
   INPUT:  an intermediate state s; the initial state s0 has M(s0)=∅
   OUTPUT: the mappings between the two graphs

   IF M(s) covers all the nodes of G2 THEN
     OUTPUT M(s)
   ELSE
     Compute the set P(s) of the pairs candidate for inclusion in M(s)
     FOREACH p in P(s)
       IF the feasibility rules succeed for the inclusion of p in M(s) THEN
         Compute the state s´ obtained by adding p to M(s)
         CALL Match(s′)
       END IF
     END FOREACH
     Restore data structures
   END IF
END PROCEDURE Match
```

- Two vectors, core_1 and core_2, whose dimensions correspond to the number of nodes in G_1 and G_2, respectively, containing the current mapping; in particular, core_1[n] contains the index of the node paired with n, if n is in $M_1(s)$, and the distinguished value NULL_NODE otherwise. The same encoding is used for core_2.
- Four vectors, in_1, out_1, in_2, out_2, whose dimensions are equal to the number of nodes in the corresponding graphs, describing the membership of the terminal sets. In particular, in_1[n] is nonzero if n is either in $M_1(s)$ or in $T_1^{in}(s)$; similar definitions hold for the other three vectors. The actual value stored in the vectors is the depth in the SSR tree of the state in which the node entered the corresponding set.

Core_1	|G1|
Core_2	|G2|
T_in_1 T_out_1	|G1| |G1|
T_in_2 T_out_2	|G2| |G2|

NJIT New Jersey Institute of Technology

Structural Reordering(Helps us to prune faster)



Challenges in Traditional Algorithms

- **Inefficient Candidate Generation**

  Traditional algorithms generate large search spaces due to inefficient candidate selection, increasing computation.

- **Rigid Vertex-Ordering Heuristics**

  Fixed vertex-ordering heuristics fail to adapt dynamically to varying graph structures and patterns.

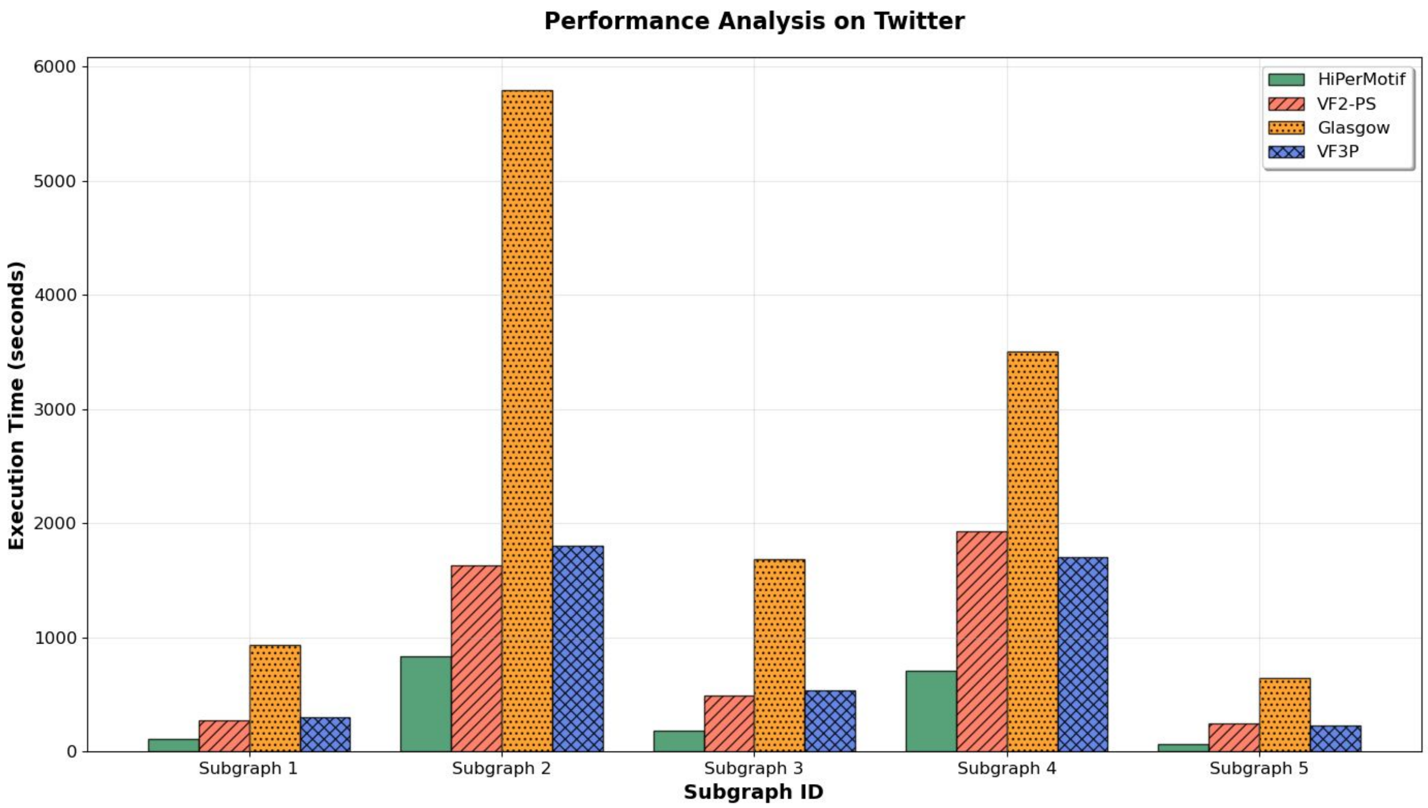
- **High Memory Overhead**

  Tracking numerous partial states leads to significant memory consumption in traditional approaches.

- **Limited Parallelization**

  Early search stages lack effective parallelization, reducing algorithm scalability on large graphs.

Twitter


Performance Analysis on Twitter
Execution Time (seconds)
0
1000
2000
3000
4000
5000
6000
HiPerMotif
VF2-PS
Glasgow
VF3P
Subgraph 1
Subgraph 2
Subgraph 3
Subgraph 4
Subgraph 5
Subgraph ID

VF2-PS

Algorithm 2 Parallel *VF2-PS* algorithm that generates the mappings of vertices u from the host graph that are mapped to vertices v from the subgraph.

Input: A state $S_{current}$ with the current mapping information for a given recursive depth d.

Output: Mappings M of all host graph and subgraph pairs that induce a monomorphism.

$M = list(int)$ ▷ Parallel-safe list.
if $d == n_2$ **then** ▷ n_2 is the size of the subgraph.
    for $v \in S_{current}.core$ **do**
        $M.pushBack(v)$
    end for
    return M
end if
$candidates = getCandidatePairs(S_{current})$
for all $(u, v) \in candidates$ **do**
    if $isFeasible(u, v, S_{current})$ **then**
        $S_{clone} = S_{current}.clone()$
        $addToTinTout(u, v, S_{clone})$
        $M_{new} = VF2(S_{clone}, d + 1)$
        for $m \in M_{new}$ **do**
            $M.pushBack(m)$
        end for
    end if
end for all
return M

The **optimal point** to spawn the tasks is immediately after generating candidates

we leverage the highly efficient and dynamic parallelization capabilities of Chapel, which automatically generates parallel tasks and assigns them to available threads based on the current system load

Changes:

- States
- Fast start
- Early Termination
- getCandidatePairs
- getUnmappedNodes
- Support of Properties
- ONLY count
- ONLY time

NJIT New Jersey Institute of Technology

Algorithms

Algorithm 1 Structural Reordering

procedure $\texttt{StructuralReorder}(\texttt{src}, \texttt{dst})$
  Compute $\deg(v)$ for all $v \in V$
  $\mathcal{R} \leftarrow \varnothing$
  $v^* \leftarrow \underset{v \in V}{arg\,max}\,\sigma(v)$
  $\textsc{Swap}(v^*, \pi(1))$
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{v^*\}$
  while $|\mathcal{R}| < |V|$ **do**
    $u \leftarrow \pi(|\mathcal{R}|)$
    Update indeg, outdeg, totaldeg
    $N^+(u) \leftarrow \{w \notin \mathcal{R} \mid (u \rightarrow w) \in E\}$
    if $N^+(u) \neq \varnothing$ **then**
      $w^* \leftarrow \underset{w \in N^+(u)}{arg\,max}\,\sigma(w)$
    else
      $w^* \leftarrow \underset{w \in V \setminus R}{arg\,max}\,\sigma(w)$
    end if
    $\textsc{Swap}(w^*, \pi(|\mathcal{R}|+1))$
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{w^*\}$
  end while
  return $(\texttt{src}, \texttt{dst}, \pi)$
end procedure

Algorithm 2 Vertex Validator

procedure $\mathrm{VV}(G_1, G_2)$
  $\texttt{vertexFlag} = [a_1, \dots, a_n]$
  $T^0_{\text{in}} \leftarrow |N^{\text{in}}_{G_1}(0)|$, $T^0_{\text{out}} \leftarrow |N^{\text{out}}_{G_1}(0)|$
  for all $v \in V(G_2)$ **do**
    $T^v_{\text{in}} \leftarrow |N^{\text{in}}_{G_2}(v)|$, $T^v_{\text{out}} \leftarrow |N^{\text{out}}_{G_2}(v)|$
    if $\texttt{checkAttributes}(v, 0) \wedge T^v_{\text{in}} \geq T^0_{\text{in}} \wedge T^v_{\text{out}} \geq T^0_{\text{out}}$ **then**
      $\texttt{vertexFlag}[v] \leftarrow \texttt{true}$
    end if
  end for
  return $\texttt{vertexFlag}$
end procedure

Algorithms

Algorithm 3 Edge Validator

procedure EV(u, v, s)
  $T^{u,v}_{\text{in/out}} \leftarrow N^{\pm}_{G_2}(u, v)$; $T^{0,1}_{\text{in/out}} \leftarrow N^{\pm}_{G_1}(0, 1)$
  if $\neg\texttt{match}(v, 1)$ **then return** `false`
  end if
  $e_1 \leftarrow \texttt{getEdgeId}(u, v)$; $e^r_1 \leftarrow \texttt{getEdgeId}(v, u)$
  $e_2 \leftarrow \texttt{getEdgeId}(0, 1)$; $e^r_2 \leftarrow \texttt{getEdgeId}(1, 0)$
  if $\neg\texttt{match}(e_1, e_2) \vee (e^r_2 \neq -1 \wedge e^r_1 = -1)$ **then**
    return `false`
  end if
  if $e^r_1 \neq -1 \wedge e^r_2 \neq -1 \wedge \neg\texttt{checkAttributes}(e^r_1, e^r_2)$ **then**
    return `false`
  end if
  if $|T^v_{\text{in}}| < |T^1_{\text{in}}| \vee |T^v_{\text{out}}| < |T^1_{\text{out}}|$ **then return** `false`
  end if
  $N_{u,v} \leftarrow T^u_{\text{in}} \cup T^u_{\text{out}} \cup T^v_{\text{in}} \cup T^v_{\text{out}}$
  $N_{0,1} \leftarrow T^0_{\text{in}} \cup T^0_{\text{out}} \cup T^1_{\text{in}} \cup T^1_{\text{out}}$
  if $|N_u \cap N_v| < |N_0 \cap N_1|$ **then return** `false`
  end if
  $s.T^{G_2}_{\text{in/out}} \leftarrow T^u_{\text{in/out}} \cup T^v_{\text{in/out}} \setminus \{u, v\}$
  $s.T^{G_1}_{\text{in/out}} \leftarrow T^0_{\text{in/out}} \cup T^1_{\text{in/out}} \setminus \{0, 1\}$
  $s.\text{depth} \leftarrow s.\text{depth} + 2$
  $s.\text{core}[0] \leftarrow u$; $s.\text{core}[1] \leftarrow v$
  return `true`
end procedure

Algorithm 4 HiPerMotif Algorithm

procedure HiPerMotif(G_1, G_2)
  $M \leftarrow$ new list(int)
  for all $e \in E_2$ **do** ▷ Parallel over edges
    $u \leftarrow \text{src}(e)$, $v \leftarrow \text{dst}(e)$
    if $\texttt{vertexFlag}[u] \wedge u \neq v$ **then**
      $s \leftarrow$ new State$(|V_2|, |V_1|)$
      if EV(u, v, s) **then**
        $M_{\text{new}} \leftarrow \texttt{VF2-PS}(s, 2)$
        $M \leftarrow M \cup M_{\text{new}}$
      end if
    end if
  end for
  return M
end procedure

NJIT New Jersey Institute of Technology