# Mining Tree Patterns with Partially Injective Homomorphisms

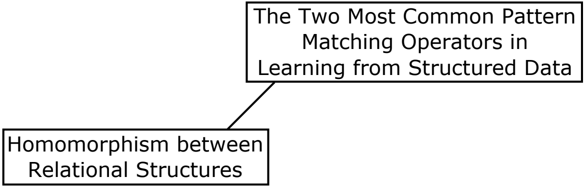
Till Hendrik Schulz, Tamás Horváth, Pascal Welke, and Stefan Wrobel

University of Bonn

# Main Idea

The Two Most Common Pattern Matching Operators in Learning from Structured Data

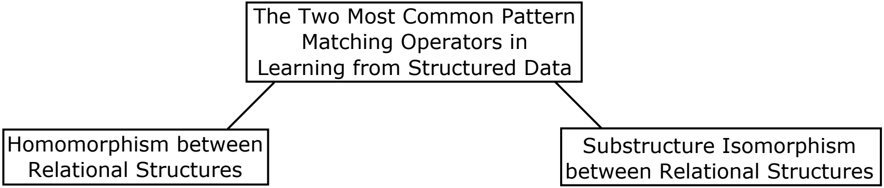
# Main Idea



The Two Most Common Pattern Matching Operators in Learning from Structured Data

Homomorphism between Relational Structures

# Main Idea

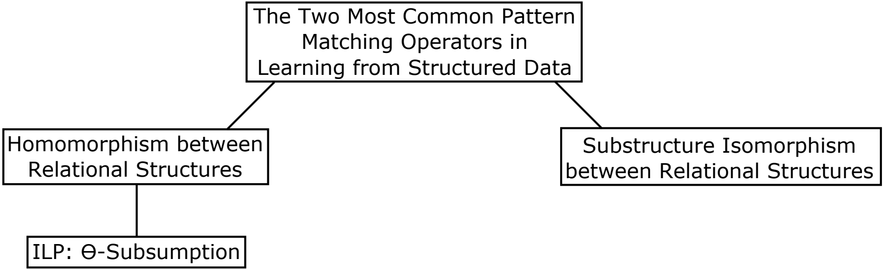


The Two Most Common Pattern Matching Operators in Learning from Structured Data

Homomorphism between Relational Structures

Substructure Isomorphism between Relational Structures

# Main Idea



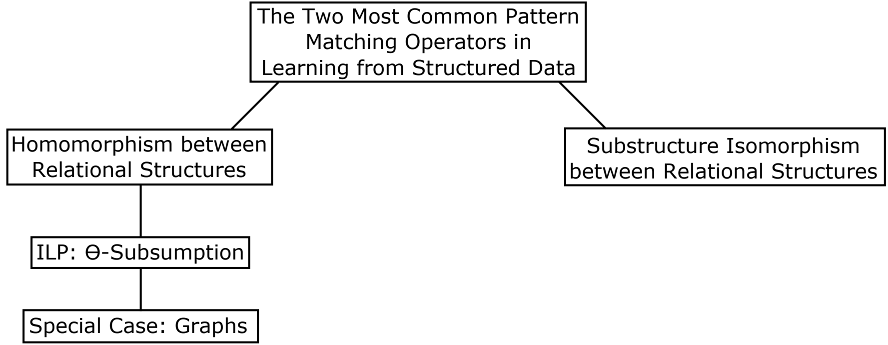
The Two Most Common Pattern Matching Operators in Learning from Structured Data

Homomorphism between Relational Structures

ILP: Θ-Subsumption

Substructure Isomorphism between Relational Structures

# Main Idea



The Two Most Common Pattern Matching Operators in Learning from Structured Data

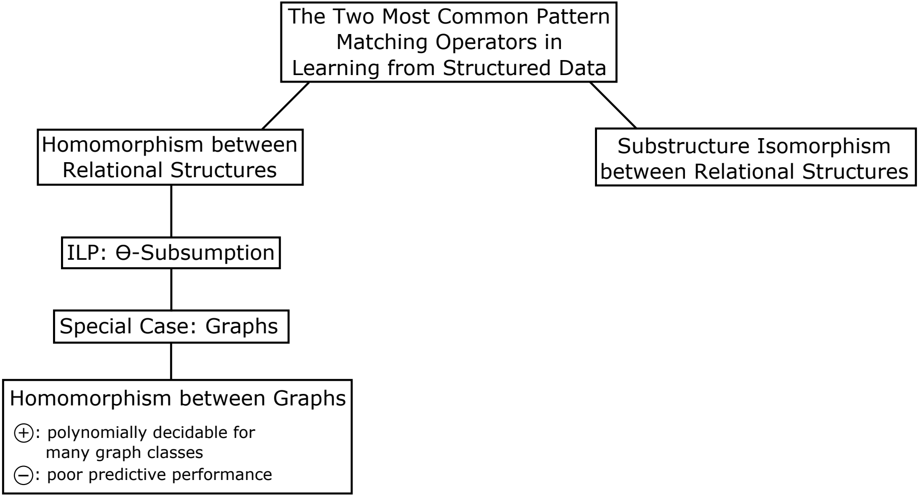
Homomorphism between Relational Structures

ILP: Θ-Subsumption

Special Case: Graphs

Substructure Isomorphism between Relational Structures

# Main Idea



The Two Most Common Pattern Matching Operators in Learning from Structured Data

Homomorphism between Relational Structures

Substructure Isomorphism between Relational Structures

ILP: Θ-Subsumption

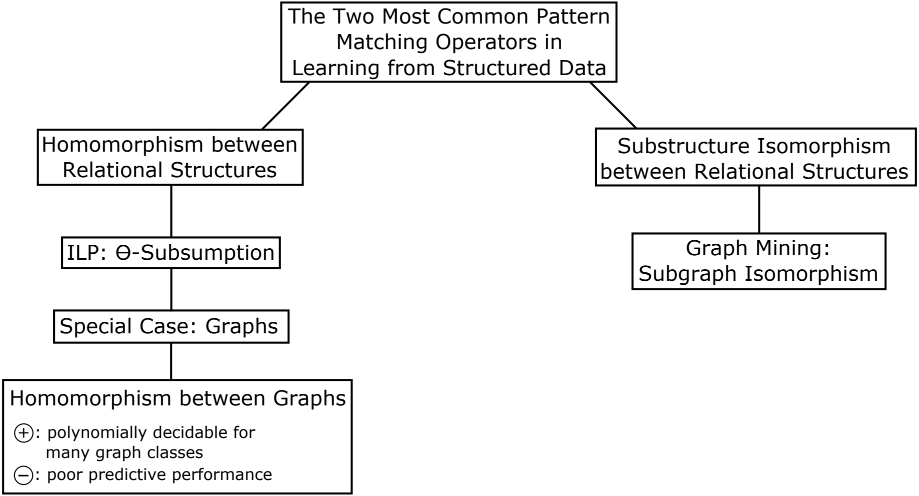
Special Case: Graphs

Homomorphism between Graphs

⊕: polynomially decidable for many graph classes

⊖: poor predictive performance

# Main Idea



The Two Most Common Pattern Matching Operators in Learning from Structured Data

Homomorphism between Relational Structures

ILP: Θ-Subsumption

Special Case: Graphs

**Homomorphism between Graphs**

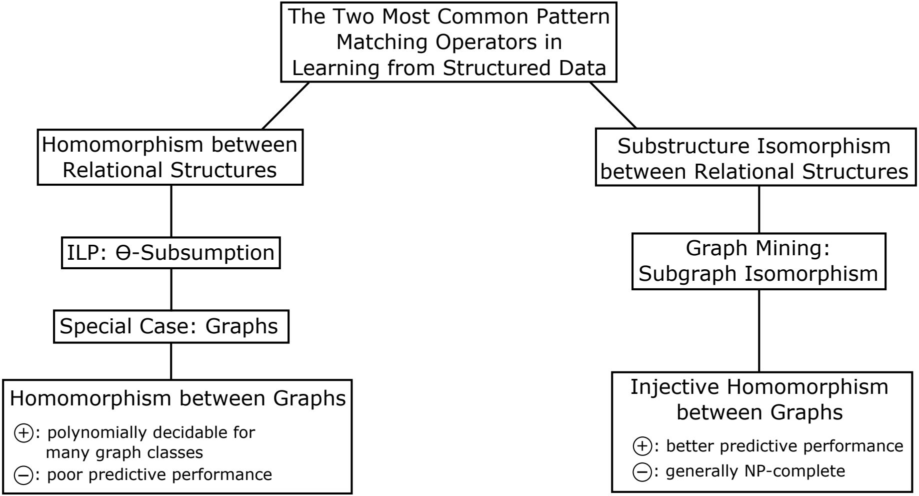
⊕: polynomially decidable for many graph classes

⊖: poor predictive performance

Substructure Isomorphism between Relational Structures

Graph Mining: Subgraph Isomorphism

# Main Idea



The Two Most Common Pattern Matching Operators in Learning from Structured Data

Homomorphism between Relational Structures

ILP: Θ-Subsumption

Special Case: Graphs

**Homomorphism between Graphs**

⊕: polynomially decidable for many graph classes

⊖: poor predictive performance

Substructure Isomorphism between Relational Structures

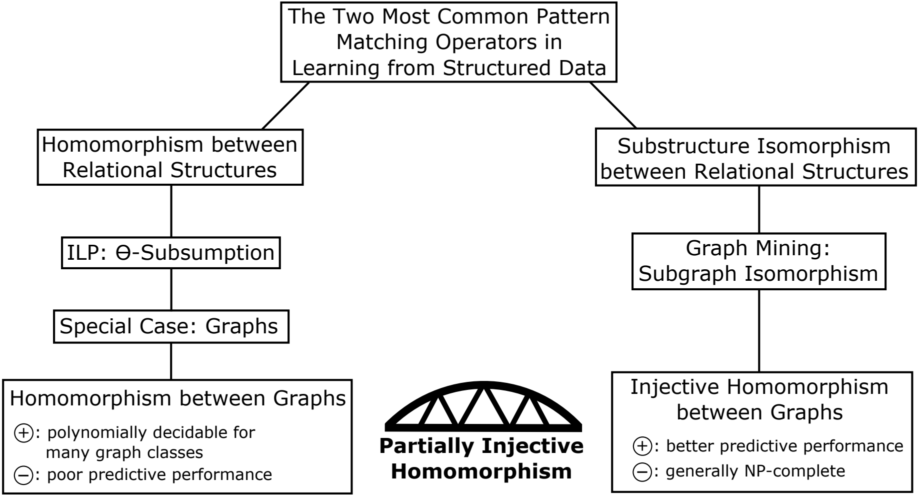
Graph Mining: Subgraph Isomorphism

**Injective Homomorphism between Graphs**

⊕: better predictive performance

⊖: generally NP-complete

# Main Idea



The Two Most Common Pattern Matching Operators in Learning from Structured Data

Homomorphism between Relational Structures

ILP: Θ-Subsumption

Special Case: Graphs

Homomorphism between Graphs

⊕: polynomially decidable for many graph classes

⊖: poor predictive performance

**Partially Injective Homomorphism**

Substructure Isomorphism between Relational Structures

Graph Mining: Subgraph Isomorphism

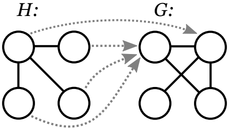
Injective Homomorphism between Graphs

⊕: better predictive performance

⊖: generally NP-complete

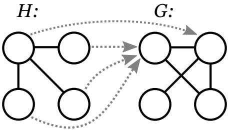
# Injective Homomorphism

Graph homomorphism:

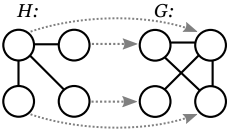


# Injective Homomorphism

Graph homomorphism:

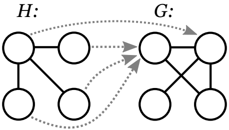


Subgraph isomorphism can be reduced to homomorphism:

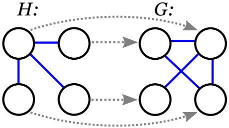


# Injective Homomorphism

Graph homomorphism:



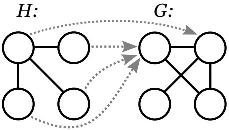
Subgraph isomorphism can be reduced to homomorphism:



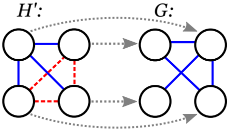
- Color all edges of $H$ and $G$ in blue (original edges)

# Injective Homomorphism

Graph homomorphism:



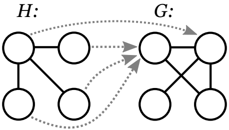
Subgraph isomorphism can be reduced to homomorphism:



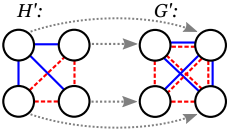
- Color all edges of $H$ and $G$ in blue (original edges)
- Add to $H$ red edges between all *unconnected* vertex pairs (constraint edges)

# Injective Homomorphism

Graph homomorphism:



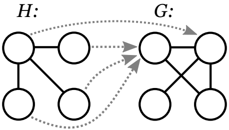
Subgraph isomorphism can be reduced to homomorphism:



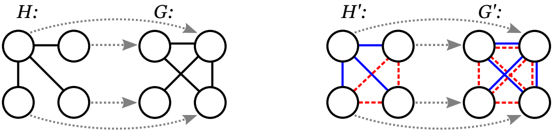
- Color all edges of $H$ and $G$ in blue (original edges)
- Add to $H$ red edges between all *unconnected* vertex pairs (constraint edges)
- Connect *all* vertex pairs in $G$ by a red edge

# Injective Homomorphism

Graph homomorphism:



Subgraph isomorphism can be reduced to homomorphism:

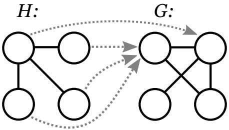


- Color all edges of $H$ and $G$ in blue (original edges)
- Add to $H$ red edges between all *unconnected* vertex pairs (constraint edges)
- Connect *all* vertex pairs in $G$ by a red edge

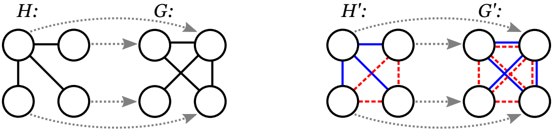
Red edges in $H'$ enforce vertices to be mapped to distinct vertices in $G'$

# Injective Homomorphism

Graph homomorphism:



Subgraph isomorphism can be reduced to homomorphism:



- Color all edges of $H$ and $G$ in blue (original edges)
- Add to $H$ red edges between all *unconnected* vertex pairs (constraint edges)
- Connect *all* vertex pairs in $G$ by a red edge

Red edges in $H'$ enforce vertices to be mapped to distinct vertices in $G'$

$\Rightarrow$ **$H$ is subgraph isomorphic to $G$ iff there exists a homomorphism from $H'$ into $G'$**

# Partially Injective Homomorphism

Partially injective homomorphism requires injectivity constraints for only a subset of vertex pairs in the pattern

# Partially Injective Homomorphism

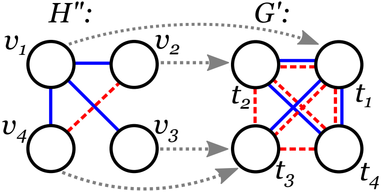
Partially injective homomorphism requires injectivity constraints for only a subset of vertex pairs in the pattern

- i.e. add to $H$ only a selection of red edges

# Partially Injective Homomorphism

Partially injective homomorphism requires injectivity constraints for only a subset of vertex pairs in the pattern

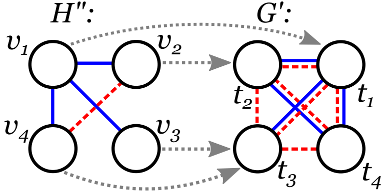
- i.e. add to $H$ only a selection of red edges



# Partially Injective Homomorphism

Partially injective homomorphism requires injectivity constraints for only a subset of vertex pairs in the pattern

- i.e. add to $H$ only a selection of red edges

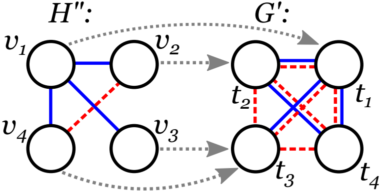


- $v_2$ and $v_4$ of $H$" must be mapped to distinct vertices in $G'$

# Partially Injective Homomorphism

Partially injective homomorphism requires injectivity constraints for only a subset of vertex pairs in the pattern

- i.e. add to $H$ only a selection of red edges



- $v_2$ and $v_4$ of $H''$ must be mapped to distinct vertices in $G'$

☺ **Partially injective homomorphism can be decided in polynomial time if the pattern graph (blue + red edges) has bounded tree width.**

# Our approach

Generate *frequent trees* w.r.t. partially injective homomorphism

# Our approach

Generate *frequent trees* w.r.t. partially injective homomorphism

- i.e. blue edges form a tree

# Our approach

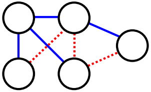
Generate *frequent trees* w.r.t. partially injective homomorphism

- i.e. blue edges form a tree
- and blue + red edges form a graph of bounded tree-width

# Our approach

Generate *frequent trees* w.r.t. partially injective homomorphism

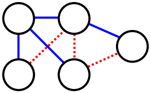
- i.e. blue edges form a tree
- and blue + red edges form a graph of bounded tree-width



# Our approach

Generate *frequent trees* w.r.t. partially injective homomorphism

- i.e. blue edges form a tree
- and blue + red edges form a graph of bounded tree-width

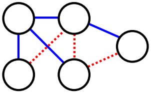


Mining algorithm: levelwise search

# Our approach

Generate *frequent trees* w.r.t. partially injective homomorphism

- i.e. blue edges form a tree
- and blue + red edges form a graph of bounded tree-width



Mining algorithm: levelwise search

Technical issues: only a subset of patterns is kept

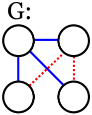
# Refinement Operator

The refinement operator utilizes the algorithmic definition of $k$-trees

# Refinement Operator

The refinement operator utilizes the algorithmic definition of *k*-trees

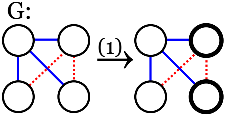
**Refinement step**:



# Refinement Operator

The refinement operator utilizes the algorithmic definition of *k*-trees

**Refinement step**:

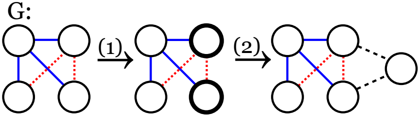


(1) select a 2-clique

# Refinement Operator

The refinement operator utilizes the algorithmic definition of $k$-trees

**Refinement step**:



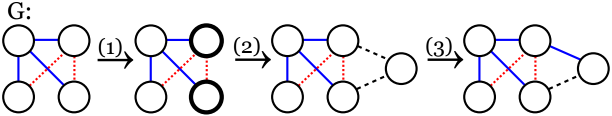
(1) select a 2-clique

(2) add a vertex and connect it to the 2-clique

# Refinement Operator

The refinement operator utilizes the algorithmic definition of *k*-trees

**Refinement step**:



(1) select a 2-clique

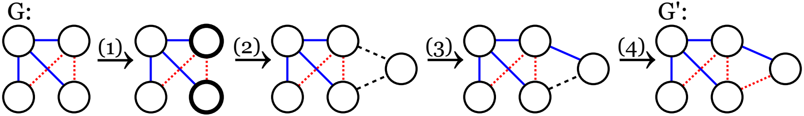
(2) add a vertex and connect it to the 2-clique

(3) color one edge blue

# Refinement Operator

The refinement operator utilizes the algorithmic definition of *k*-trees

**Refinement step**:



(1) select a 2-clique

(2) add a vertex and connect it to the 2-clique

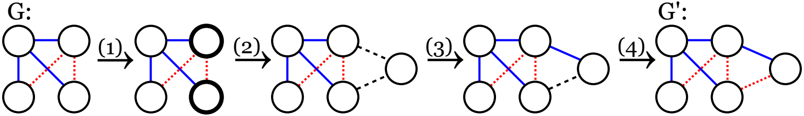
(3) color one edge blue

(4) color all others red

# Refinement Operator

The refinement operator utilizes the algorithmic definition of $k$-trees

**Refinement step**:



(1) select a 2-clique

(2) add a vertex and connect it to the 2-clique

(3) color one edge blue

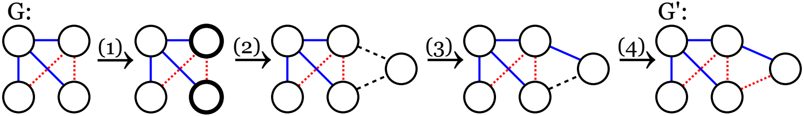
(4) color all others red

$G'$ is a refinement of $G$. Both graphs have tree-width 2.

# Refinement Operator

The refinement operator utilizes the algorithmic definition of $k$-trees

**Refinement step**:



(1) select a 2-clique

(2) add a vertex and connect it to the 2-clique

(3) color one edge blue

(4) color all others red

$G'$ is a refinement of $G$. Both graphs have tree-width 2.

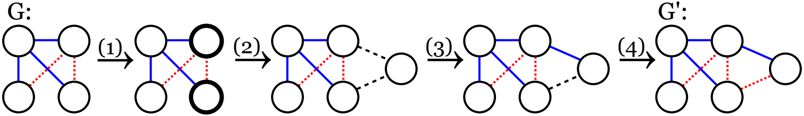
Properties:

- graphs are *maximally* constrained (i.e. adding another red edge increases the tree-width)

# Refinement Operator

The refinement operator utilizes the algorithmic definition of $k$-trees

**Refinement step**:



(1) select a 2-clique

(2) add a vertex and connect it to the 2-clique

(3) color one edge blue

(4) color all others red

$G'$ is a refinement of $G$. Both graphs have tree-width 2.

Properties:

- graphs are *maximally* constrained (i.e. adding another red edge increases the tree-width)
- the embedding decision problem is guaranteed to lie in $P$

# Experiments

| Dataset | Frequent Patterns | $\lvert E\rvert = 4$ | $\lvert E\rvert = 5$ | $\lvert E\rvert = 6$ | $\lvert E\rvert = 7$ | $\lvert E\rvert = 8$ |
|---|---|---|---|---|---|---|
| NCI1 | s.g.i. graphs | 79.30 | 84.36 | 86.48 | 87.17 | 87.18 |
| | s.g.i. trees | 79.30 | 84.10 | 86.16 | 86.83 | 86.92 |
| | p.i.h. trees ($k = 2$) | 78.94 | 83.18 | 85.02 | 85.41 | 86.07 |
| | p.i.h. trees ($k = 3$) | 80.51 | 84.53 | 85.83 | 86.64 | 86.68 |
| | p.i.h. trees ($k = 4$) | 79.30 | 84.66 | 86.02 | 86.39 | 86.11 |
| NCI109 | s.g.i. graphs | 80.04 | 84.64 | 86.50 | 87.04 | 87.29 |
| | s.g.i. trees | 80.04 | 84.51 | 86.39 | 86.79 | 87.09 |
| | p.i.h. trees ($k = 2$) | 79.17 | 83.42 | 85.27 | 85.31 | 85.41 |
| | p.i.h. trees ($k = 3$) | 81.21 | 85.08 | 85.97 | 86.34 | 86.63 |
| | p.i.h. trees ($k = 4$) | 80.04 | 85.18 | 85.89 | 86.27 | 86.28 |
| PTC | s.g.i. graphs | 63.46 | 66.42 | 70.73 | 73.98 | 73.11 |
| | s.g.i. trees | 63.46 | 66.42 | 70.77 | 74.03 | 73.18 |
| | p.i.h. trees ($k = 2$) | 64.12 | 65.54 | 67.45 | 69.60 | 70.66 |
| | p.i.h. trees ($k = 3$) | 63.39 | 67.15 | 71.31 | 73.02 | 72.93 |
| | p.i.h. trees ($k = 4$) | 63.50 | 67.67 | 72.56 | 72.68 | 72.14 |
| MUTAG | s.g.i. graphs | 72.71 | 85.28 | 89.27 | 89.89 | 90.29 |
| | s.g.i. trees | 72.71 | 83.05 | 87.23 | 88.50 | 89.49 |
| | p.i.h. trees ($k = 2$) | 59.02 | 65.60 | 77.99 | 69.92 | 74.69 |
| | p.i.h. trees ($k = 3$) | 63.03 | 82.95 | 86.88 | 88.41 | 89.05 |
| | p.i.h. trees ($k = 4$) | 71.42 | 82.44 | 85.31 | 88.53 | 88.93 |

Table: Prediction measures stated as AUC values in % ($k$: tree-width; $s.g.i.$: subgraph isomorphism; $p.i.h.$: partially injective homomorphism)

# Experiments

| Dataset | Frequent Patterns | $\|E\| = 4$ | $\|E\| = 5$ | $\|E\| = 6$ | $\|E\| = 7$ | $\|E\| = 8$ |
|---|---|---|---|---|---|---|
| NCI1 | s.g.i. graphs | 79.30 | 84.36 | 86.48 | 87.17 | 87.18 |
| | s.g.i. trees | 79.30 | 84.10 | 86.16 | 86.83 | 86.92 |
| | p.i.h. trees ($k = 2$) | 78.94 | 83.18 | 85.02 | 85.41 | 86.07 |
| | p.i.h. trees ($k = 3$) | 80.51 | 84.53 | 85.83 | 86.64 | 86.68 |
| | p.i.h. trees ($k = 4$) | 79.30 | 84.66 | 86.02 | 86.39 | 86.11 |

Table: Prediction measures stated as AUC values in % ($k$: tree-width; $s.g.i.$: subgraph isomorphism; $p.i.h.$: partially injective homomorphism)

# Conclusion

Partially Injective Homomorphisms:

- can be decided in polynomial time (for this work)

# Conclusion

Partially Injective Homomorphisms:

- can be decided in polynomial time (for this work)
- achieve prediction performance close to traditional frequent subgraphs

# Conclusion

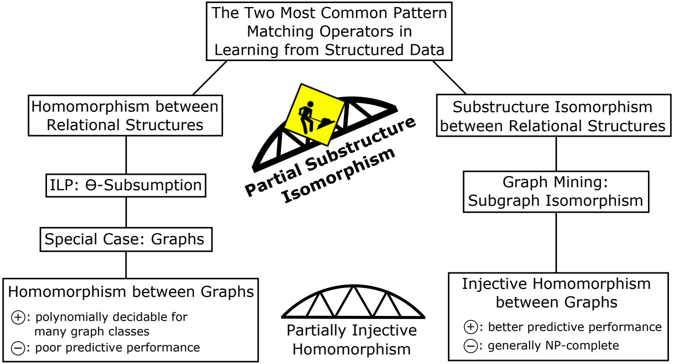
Partially Injective Homomorphisms:

- can be decided in polynomial time (for this work)
- achieve prediction performance close to traditional frequent subgraphs
- can be generalized to first-order logic:

# Conclusion

Partially Injective Homomorphisms:

- can be decided in polynomial time (for this work)
- achieve prediction performance close to traditional frequent subgraphs
- can be generalized to first-order logic:



The Two Most Common Pattern Matching Operators in Learning from Structured Data

Homomorphism between Relational Structures

ILP: Θ-Subsumption

Special Case: Graphs

Homomorphism between Graphs

⊕: polynomially decidable for many graph classes

⊖: poor predictive performance

Substructure Isomorphism between Relational Structures

Graph Mining: Subgraph Isomorphism

Injective Homomorphism between Graphs

⊕: better predictive performance

⊖: generally NP-complete

Partial Substructure Isomorphism

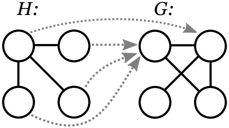
Partially Injective Homomorphism

# Appendix

# Homomorphism & Subgraph Isomorphism

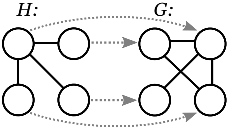
A **homomorphism** from a graph $H$ (the *pattern*) into a graph $G$ (the *text*) is a mapping $\varphi : V(H) \to V(G)$ that preserves the edges (i.e., $uv \in E(H)$ implies $\varphi(u)\varphi(v) \in E(G)$ for all $u, v \in V(H)$).

Example:



A **subgraph isomorphism** from $H$ to $G$ is an *injective* homomorphism $\psi : V(H) \to V(G)$ (i.e. $\psi$ is a homomorphism from $H$ into $G$ and for all $u, v \in V(H)$ with $u \neq v$ holds $\psi(u) \neq \psi(v)$).

Example:



# Tree-width

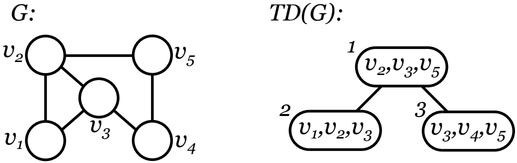
A **tree decomposition** of a graph $G = (V, E)$ is a pair $TD(G) = (T, X)$ where

- $T = (I, F)$ is an unordered tree,
- $X = \{bag(i) : i \in I\}$ is a family of subsets of $V$, s.t.
  - (i) $\bigcup_{i \in I} bag(i) = V$
  - (ii) for every $\{u, v\} \in E$ there is an $i \in I$ with $\{u, v\} \subseteq bag(i)$
  - (iii) for every $v \in V$ the set of nodes $\{i | v \in bag(i)\}$ forms a subtree of $T$

The **width** of $TD(G)$ is $max_i|bag(i)| - 1$

The **tree-width** of $G$ is the minimum width over all tree decompositions of $G$

Example:



$TD(G)$ has a width of 2 which is also the tree-width of $G$.

# k-trees

Algorithmic definition of **k-trees**:

(i) A clique of $k+1$ vertices is a $k$-tree and

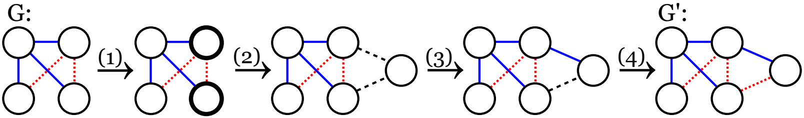
(ii) given a $k$-tree $T_k$ with $n$ vertices, a $k$-tree with $n+1$ vertices is obtained from $T_k$ by adding a new vertex $v$ to $T_k$ and connecting $v$ to all vertices of a $k$-clique of $T_k$.

Properties:

- A $k$-tree has tree-width $k$
- Adding an edge to a $k - tree$ results in a graph of tree-width $k+1$.

# Refinement Operator

**Refinement step**:



(1) select a 2-clique

(2) add a vertex and connect it to the 2-clique

(3) color one edge blue

(4) color all others red

$G'$ is a refinement of $G$. Both graphs are k-trees with $k = 2$.

Properties:

- graphs are *maximally* constrained (i.e. adding another red edge increases the tree-width)
- the embedding decision problem is guaranteed to lie in $P$

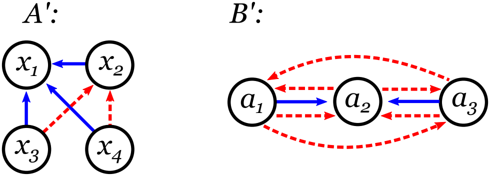
# Partial Substructure Isomorphism

Partially Injective Homomorphisms can be generalized to first-order logic:

Let $A$ and $B$ be function-free first-order clauses. A **partial substructure isomorphism** from $A$ to $B$ satisfying the injectivity constraints in $\mathcal{C} \subseteq [\mathrm{Var}(A)]^2$ is a substitution $\theta$ such that $A\theta \subseteq B$ and for all $xy \in \mathcal{C}$, $x\theta \neq y\theta$.

Example:

$$A = \{P(x_2, x_1), P(x_3, x_1), P(x_4, x_1)\}$$
$$B = \{P(a_1, a_2), P(a_3, a_2)\}$$
$$\mathcal{C} = \{x_3x_2, x_4x_2\}$$



There is a partial substructure isomorphism from $A$ to $B$ w.r.t. constraints $\mathcal{C}$ iff $A'$ subsumes $B'$.