Subgraph Isomorphism

G1 is subgraph isomorphic to G2，denoted as G1 ⊆ G2:

iff there is an injective function f : V(G1) →V(G2), such that ∀(u, v) ∈ E(G1), (f(u),f(v)) ∈ E(G2)

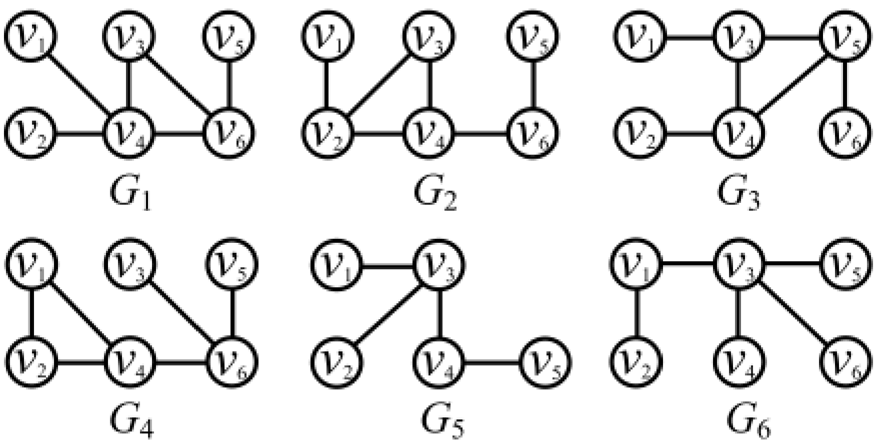
If G1 is subgraph isomorphic to G2,

G1 is called a ***subgraph*** of G2, G2 is called a ***supergraph*** of G1.

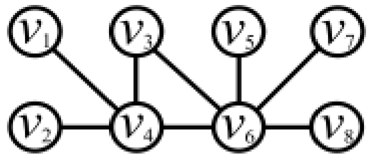
Problem Statement

- a graph database D = {G_1,G_2,..., G_n}，each Gi ∈D is called a data graph
- a graph Q called ***query-graph***
- ***supergraph search*** : find all graphs in D, that are subgraph isomorphic to Q
- Answer set ***A(Q)***= {G_i|G_i ∈D,G_i ⊆ Q}

Example



(a) Database Graphs



(b) Query Graph

Answer set A(Q)= $\{G_1, G_5, G_6\}$

EXISTING SOLUTIONS

大多数现有的超图检索解决方案遵循修剪-验证框架，该框架根据修建阶段的特征修建错误答案（不符合特征的图数据），并在验证阶段对剩余图执行子图同构测试。

- 依靠频繁的子图挖掘算法来生成特征，开销昂贵，并且不能生成大的特征
- 验证阶段的开销也很昂贵
- 以固定的顺序处理特征而不考虑他们与查询图之间的关系

contribution

- 提出了DGTree的索引结构。可以从图数据中直接选择特征图，而不用依靠频繁子图挖掘算法。并且在构建DGTree的过程中，考虑到了修剪和结构共享的能力。
- 基于特征树，提出了一种新的查询算法。设计了一个评分函数，评估特征图的共享成本和修剪能力，以便可以避免处理无用的特征。
- 提出了改进优化。

DGTree: A Full-structure Index

Match

a graph P with nodes $\{u_1, u_2, \ldots, u_{|V(P)|}\}$

a data-graph G

a match f of P in G is a mapping from V(P) to V(G) such that the following two conditions hold:

- (**Conflict-free**) : for any pair of nodes $u_i \in V(P)$ and $u_j \in V(P)$ ($u_i \mathrel{!=} u_j$), $f(u_i) \mathrel{!=} f(u_j)$
- (**Structure-preserved**): For any edge $(u_i,u_j) \in E(P)$, $(f(u_i),f(u_j)) \in E(G)$

$f =[v_1,v_2,...,v_{|V(P)|}]$ to denote the match f

$f(u_i)= v_i$, for any $1 \leqslant i \leqslant |V(P)|$.

If $f(u_i)= v_i$,wehave $f-1(v_i)= u_i$

DGTree Structure

each tree-node g :

g.children	The set of child tree-nodes of g. A leaf tree-node is a tree-node g with $\lvert g.\text{children}\rvert = 0$.
g.graph	The feature-graph of g. The set of nodes in the feature-graph is represented by $\{1, 2, \ldots, \lvert V(g.\text{graph})\rvert\}$.
g.grow-edge	The edge added to g.graph from the feature-graph of the parent tree-node of g.
g.edge-type	The type of g.grow-edge, which is CLOSE if no new node is created in g.graph and OPEN if a new node is created in g.graph after adding g.grow-edge.
$g.\mathcal{S}$	The set of data-graphs containing g.graph, i.e., $g.\mathcal{S} = \{G \mid G \in \mathcal{D}, g.\text{graph} \subseteq G\}$. If g is a leaf tree-node, $g.\mathcal{S}$ contains the data-graph equaling to g.graph, and thus the union of $g.\mathcal{S}$ for all leaf tree-nodes g is $\mathcal{D}$.
$g.\mathcal{M}(G_i)$	The set of matches of g.graph in G_i for each $G_i \in g.\mathcal{S}$.
$g.\mathcal{S}^*$	$g.\mathcal{S}^* \subseteq g.\mathcal{S}$. If g is the root tree-node, we have $g.\mathcal{S}^* = \mathcal{D}$. If g is a leaf tree-node, we have $\lvert g.\mathcal{S}^*\rvert = 1$. For a non-leaf tree-node g, we have: (1) for any $g_i \in g.\text{children}$ and $g_j \in g.\text{children}$ ($g_i \neq g_j$), $g_i.\mathcal{S}^* \cap g_j.\mathcal{S}^* = \emptyset$; and (2) $\bigcup_{g_i \in g.\text{children}} g_i.\mathcal{S}^* = g.\mathcal{S}^*$. In other words, $g_i.\mathcal{S}^*$ for all $g_i \in g.\text{children}$ form a disjoint cover of $g.\mathcal{S}^*$.
g.score	The score of g.graph, which is used to select the best edge to grow.

Algorithm 1: DGTreeConstruct(database $\mathcal{D} = \{G_1, \ldots, G_n\}$)

$g_r \leftarrow$ a new tree-node;
$g_r.\mathsf{graph} \leftarrow$ a single-edge graph;
$g_r.\mathcal{S} \leftarrow \mathcal{D}$; $g_r.\mathcal{S}^* \leftarrow \mathcal{D}$; $g_r.\mathsf{grow\text{-}edge} = \emptyset$;
for $G_i \in \mathcal{D}$ **and** $(v, v') \in E(G_i)$ **do**
    $g_r.\mathcal{M}(G_i) \leftarrow g_r.\mathcal{M}(G_i) \cup \{[v, v'], [v', v]\}$;
TreeGrow(g_r);
return g_r;

Procedure TreeGrow(tree-node g)
$\mathcal{H} \leftarrow$ CandidateFeature(g);
$\mathcal{C} \leftarrow g.\mathcal{S}^*$;
while $\mathcal{C} \neq \emptyset$ **do**
    $g^+ \leftarrow$ BestFeature($\mathcal{H}, \mathcal{C}$);
    if $|g^+.\mathcal{S}^*| > 1$ **then**
        $g^+.\mathsf{graph} \leftarrow$ a graph by adding $g.\mathsf{grow\text{-}edge}$ in $g.\mathsf{graph}$;
        TreeGrow(g^+);
    else $g^+.\mathsf{graph} \leftarrow$ the graph in $g.\mathcal{S}^*$; $g^+.\mathcal{S} \leftarrow g^+.\mathcal{S}^*$;
    $g.\mathsf{children} \leftarrow g.\mathsf{children} \cup \{g^+\}$;
    $\mathcal{C} \leftarrow \mathcal{C} \setminus g^+.\mathcal{S}^*$;

Algorithm 2: CandidateFeature(tree-node g)

$\mathcal{H} \leftarrow \emptyset$;
for data-graph $G \in g.\mathcal{S}^*$ **and** match $f \in g.\mathcal{M}(G)$ **do**
  for $u_i \leftarrow 1$ **to** $|f|$ **and** $v \in \mathsf{Nbr}(f(u_i), G)$ **do**
    if $v \in f$ **then** $u_j \leftarrow f^{-1}(v)$; $t \leftarrow \mathsf{CLOSE}$;
    else $u_j \leftarrow |f| + 1$; $t \leftarrow \mathsf{OPEN}$;
    if $u_j > u_i$ **and** $(u_i, u_j) \notin E(g)$ **then**
      $g^+ \leftarrow \mathcal{H}.\mathsf{Find}((u_i, u_j))$;
      if $g^+ = \emptyset$ **then**
        $g^+ \leftarrow$ a new tree-node;
        $g^+.\mathsf{grow\text{-}edge} \leftarrow (u_i, u_j)$;
        $g^+.\mathcal{S}^* \leftarrow \{G\}$; $g^+.\mathsf{score} \leftarrow 0$;
        $g^+.\mathsf{edge\text{-}type} \leftarrow t$;
        $\mathcal{H}.\mathsf{Push}(g^+)$;
      else $g^+.\mathcal{S}^* \leftarrow g^+.\mathcal{S}^* \cup \{G\}$;

for data-graph $G \in g.\mathcal{S}$ **and** match $f \in g.\mathcal{M}(G)$ **do**
  for $u_i \leftarrow 1$ **to** $|f|$ **and** $v \in \mathsf{Nbr}(f(u_i), G)$ **do**
    if $v \in f$ **then** $u_j \leftarrow f^{-1}(v)$;
    else $u_j \leftarrow |f| + 1$;
    if $u_j > u_i$ **and** $(u_i, u_j) \notin E(g)$ **then**
      $g^+ \leftarrow \mathcal{H}.\mathsf{Find}((u_i, u_j))$;
      if $g^+ \neq \emptyset$ **then**
        $g^+.\mathcal{S} \leftarrow g^+.\mathcal{S} \cup \{G\}$;
        if $g^+.\mathsf{edge\text{-}type} = \mathsf{OPEN}$ **then**
          $g^+.\mathcal{M}(G) \leftarrow g^+.\mathcal{M}(G) \cup \{[f, v]\}$;
        else $g^+.\mathcal{M}(G) \leftarrow g^+.\mathcal{M}(G) \cup \{f\}$;

for $g^+ \in \mathcal{H}$ **do**
  compute $g^+.\mathsf{score}$; $\mathcal{H}.\mathsf{Update}(g^+)$;

return $\mathcal{H}$;

Algorithm 3: BestFeature(heap $\mathcal{H}$, uncovered graphs $\mathcal{C}$)

$g^+ \leftarrow \mathcal{H}.\mathsf{Pop}()$;
while $g^+.\mathcal{S}^* \nsubseteq \mathcal{C}$ **do**
  $g^+.\mathcal{S}^* \leftarrow g^+.\mathcal{S}^* \cap \mathcal{C}$;
  if $g^+.\mathcal{S}^* \neq \emptyset$ **then** {compute $g^+.\mathsf{score}$; $\mathcal{H}.\mathsf{Push}(g^+)$};
  $g^+ \leftarrow \mathcal{H}.\mathsf{Pop}()$;

return g^+;

Score

- 为了在查询处理中分担计算成本，应使选择的g包含g.graph的数据图的数量最大化，也就是最大化|g.S|,但这个会生成冗余的特征图，因此选择最大化|g.S*|

$$g.score_1 = |g.S^*|$$

- 如果一个数据图中的一个特征图平均匹配数很小，那么它不太可能包含在一个查询图中，因此这样一个特征图的剪枝能力很强

$$\text{平均匹配数} = \sum_{G \in g.S} \frac{|g.M(G)|}{|g.S|}$$

$$g.score_2 = \frac{g.score_1}{\text{平均匹配数}} = \frac{|g.S^*| \times |g.S|}{\sum_{G \in g.S} |g.M(G)|}$$

Example

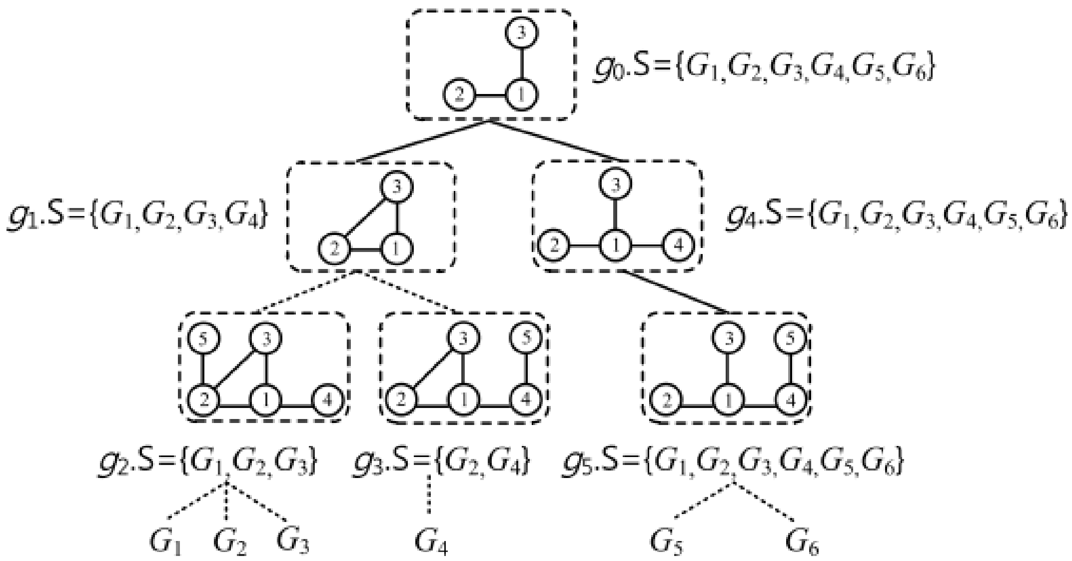


Fig. 2: An Example of DGTree

Query-dependent Supergraph Search

Algorithm 4: SuperGraphSearch(query Q, DGTree with root g_r)

$\mathcal{C} \leftarrow \{G | G \in g_r.\mathcal{S}, |E(G)| \leq |E(Q)|, |V(G)| \leq |V(Q)|\}$;
$\mathcal{H} \leftarrow \emptyset$; $\mathcal{A}(Q) \leftarrow \emptyset$;
$q \leftarrow$ a new entry;
q.tree-node $\leftarrow g_r$; $q.\mathcal{S}^* \leftarrow \mathcal{C}$; $q.\mathcal{M}(Q) \leftarrow \emptyset$;
for $(v, v') \in E(Q)$ **do** $q.\mathcal{M}(Q) \leftarrow q.\mathcal{M}(Q) \cup \{[v, v'], [v', v]\}$;
compute q.score; $\mathcal{H}$.Push(q);
while $\mathcal{C} \neq \emptyset$ **do**
  $q \leftarrow$ BestFeature($\mathcal{H}, \mathcal{C}$); $g \leftarrow q$.tree-node;
  for $g^+ \in g$.children **do**
    if $|g^+$.children$| = 0$ **then**
      search a match f of g^+.graph by extending $q.\mathcal{M}(Q)$;
      if $f \neq \emptyset$ **then** $\mathcal{A}(Q) \leftarrow \mathcal{A}(Q) \cup g^+.\mathcal{S}$;
      $\mathcal{C} \leftarrow \mathcal{C} \setminus g^+.\mathcal{S}$;
    else FeatureExpansion($Q, q, g^+, \mathcal{H}, \mathcal{C}$);
return $\mathcal{A}(Q)$;

Procedure BestFeature(heap $\mathcal{H}$, candidate data-graph set $\mathcal{C}$)
$q \leftarrow \mathcal{H}$.Pop();
while $q.\mathcal{S}^* \nsubseteq \mathcal{C}$ **do**
  $q.\mathcal{S}^* \leftarrow q.\mathcal{S}^* \cap \mathcal{C}$;
  if $q.\mathcal{S}^* \neq \emptyset$ **then** {compute q.score; $\mathcal{H}$.Push(q)};
  $q \leftarrow \mathcal{H}$.Pop();
return q;

Algorithm 5: FeatureExpansion(query-graph Q, entry q, tree-node g^+, heap $\mathcal{H}$, candidate data-graph set $\mathcal{C}$)

$q^+ \leftarrow$ a new entry;
q^+.tree-node $\leftarrow g^+$; $q^+.\mathcal{S}^* \leftarrow g^+.\mathcal{S} \cap \mathcal{C}$; $q^+.\mathcal{M}(Q) \leftarrow \emptyset$;
$(u_i, u_j) \leftarrow g^+$.grow-edge;
for match $f \in q.\mathcal{M}(Q)$ **do**
  if g^+.edge-type = OPEN **then**
    for $v \in$ Nbr$(f(u_i), Q)$ **do**
      if $v \notin f$ **then** $q^+.\mathcal{M}(Q) \leftarrow q^+.\mathcal{M}(Q) \cup \{[f, v]\}$;
  else if $(f(u_i), f(u_j)) \in E(Q)$ **then**
    $q^+.\mathcal{M}(Q) \leftarrow q^+.\mathcal{M}(Q) \cup \{f\}$;
if $q^+.\mathcal{M}(Q) \neq \emptyset$ **then**
  compute q^+.score; $\mathcal{H}$.Push(q^+);
else $\mathcal{C} \leftarrow \mathcal{C} \setminus q^+.\mathcal{S}^*$;

Query-dependent Feature Score

- 如果q的特征图包含在大量的候选数据图中，则可以最大限度地分担q的处理成本 ,所以最大化|q.S*|

$$q.score_1 = |q.S^*|$$

- 在Q中具有少量匹配的特征图将具有较高的裁剪能力

$$q.score_2 = \frac{1}{q.M(Q)}$$

$$q.score = q.score_1 \times q.score_2$$