Querying graphs

Davide Mottin, Anton Tsitsulin
Hasso Plattner Institute

Graph Mining course Winter Semester 2017
Lecture road

- Containment queries
- Similarity queries (not covered)
- Reachability queries
Graph Isomorphism

Given two graphs, $G_1: \langle V_1, E_1, l_1 \rangle$, $G_2: \langle V_2, E_2, l_2 \rangle$ $G_1$ is isomorphic to $G_2$ iff exists a bijective function $f: V_1 \rightarrow V_2$ s.t.:

1. For each $v_1 \in V_1$, $l(v_1) = l(f(v_1))$
2. $(v_1, u_1) \in E_1$ iff $(f(v_1), f(u_1)) \in E_2$
A graph $Q: (V_Q, E_Q, l_Q)$ is subgraph isomorphic to a graph $G: (V, E, l)$ if exists a subgraph $G' \subseteq G$, isomorphic to $Q$. 

Subgraph Isomorphism
Which queries in a graph?

- Find the friends of a friend who are interested in pop-music
- Find all the German presidents that have been elected twice
- Find all the molecules that contain a particular compound
- Find the 10 cities that have a female mayor with the largest population
- Find a place with more historical monuments that is closest to Berlin
- ...

GRAPH MINING WS 2017
Why querying a graph

- A graph exposes relationships among objects
- Can represent complex facts with a very clear structure
- Can encode different objects (atoms, proteins, individuals, places, ...) with different characteristics

They have:
- An intuitive representation (not quite like text)
- A more structured representation (not quite like relational)
Challenges in querying a graph

- No fixed schema (i.e., no rule for the structure)
- Hard to find information in a graph
- Not a single a query language (SPARQL, Gremlin, Cypher, ...)
- Many different queries (reaching a point, best neighbors, several options)
- The user is usually imprecise and impatient in the requirements
Different queries

- **Containment queries**
  - Ask if a structure/pattern is contained in a graph
  - Retrieve **all graphs** from a graph (database), such that they **contain** a given query graph.
    - Find all the molecules containing a specific compound
    - Find who is the president of Germany and his vice-president

- **Similarity queries**
  - Retrieve **all graphs** from a graph (database), such that they are **similar** to the query graph (exact and approximate).
    - Find the other presidents of Germany apart Angela Merkel

*Not in the lecture*
Containment queries

Query (a graph)

Graph

Solution

- The user asks all the occurrences of a particular structure
- Recursively match structures from the query to the graph
- Return all the substructures of that kind
The “simplest” kind of queries query

- Use **subgraph isomorphism** to find matches of the exact structure
  - Give me all the presidents of united states married with an actress

![Diagram showing relationships between President, USA, Married, Profession, and Actor]

- SPARQL queries (but also Gremlin, MQL) allows you to specify such query
  - SELECT ?x,?y
    WHERE {
      ?x president "USA".
      ?x married ?y.
      ?y profession "Actor".
    }
Subgraph isomorphism problem

Computational Problems

• The subgraph problem is **NP**-complete.
• There is a very simple reduction:
  • CLIQUE $\leq_p$ Subgraph Isomorphism
• However, for many specific types of practical problems (even with ‘big’ inputs), algorithms do answer fast.
Ullmann’s algorithm


- Although old, this algorithm is still very popular and having good results in practice.
Permutation matrix

- Use the notion of a permutation matrix.
  - Any permutation matrix is equivalent to an isomorphic correspondence.

\[ A_H = \begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix} \]

Isomorphic Mapping

\[ F = 1_{H_2} - 1_H \\
2_{H_2} - 3_H \\
3_{H_2} - 2_H \\
4_{H_2} - 4_H \]

Permutation Matrix

\[ P = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]

\[ F \sim P \]
Permutation matrix

- Two graphs $H$ and $H_2$ are isomorphic with a correspondence $F \Leftrightarrow A_H$ is similar to $A_{H_2}$, and the similarity matrix is $F \sim P$.

\[ F = \begin{align*}
1_{H_2} &- 1_H \\
2_{H_2} &- 3_H \\
3_{H_2} &- 2_H \\
4_{H_2} &- 4_H
\end{align*} \quad P = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}

Isomorphic Correspondence $\sim$

Permutation Matrix

Isomorphism criterion:

\[ A_{H_2} = P A_H P^T \]

iff $H_2$ is isomorphic to $H$, with a correspondence $F \sim P$.
Skeptical?

\[ P^T \text{ moves the row and columns of } A_H \text{ using the mapping } (1 \ 2 \ 3 \ 4) \]

1: Moves only the rows

\[ P A_H = \begin{bmatrix}
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix} \]

2: Moves only the columns

\[ P A_H P^T = \begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0
\end{bmatrix} = A_{H^2} \]
Subgraph isomorphism

- In a similar fashion (without proof) we have an algebraic criterion for a subgraph isomorphism.

\[
P = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\]

\[
A_H = \begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\]

Isomorphic Correspondence Permutation Matrix

\[
F = 1_G - 1_H
\]

\[
2_G - 3_H
\]

\[
3_G - 2_H
\]

\[
4_G - \phi
\]

Subgraph isomorphism criterion:

\[
A_G = PA_H P^T
\]

iff G is subgraph isomorphic to H, with a correspondence F rectangular M'.
The Ullmann Algorithm

**Input**: G, H graphs

**Output**: is G subgraph isomorphic to H?

- Search for a *permutation matrix* $P^*$ of size $|V_G| \times |V_H|$ that satisfies the subgraph isomorphism criterion.

- Enumerate over candidate permutation matrices of the same size, denoting a candidate by $P$, from a set of candidates that satisfies: $(\text{The set of all } P^*) \subseteq (\text{The set of all } P)$ [since the wanted permutations are less than all the permutations]
  - Check the isomorphism criterion over each candidate. If a candidate satisfies the criterion, we will return ‘yes’. If we would not find such a candidate, we will return ‘no’.
The Ullmann Algorithm

- Construction of another matrix $P^{(0)}$ with the same size of the $P$ matrices:

\[
p^{(0)}_{i,j} = \begin{cases} 
1 & \text{if } \deg(V_{G_i}) \leq \deg(V_{H_j}) \\
0 & \text{otherwise}
\end{cases}
\]

- Generation of all $P$ matrices by setting to 0 all but one 1 in each row of $P^{(0)}$
- A subgraph isomorphism has been found if $M = PA_HP^T$ implies $(a_{G_{i,j}} = 1) \implies (m_{i,j} = 1)$

**Pruning rule**

\[
A_H = \begin{bmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\]

\[
A_G = \begin{bmatrix}
0 & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 0
\end{bmatrix}
\]

\[
P^0 = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0
\end{bmatrix}
\]

$P$ generates only valid candidate permutations
Example: The Ullman Algorithm

Invalid permutation. Why?

Root - \( P^{(0)} \)

Inner Nodes P

Leaves P

\[ M = P A_H P^T \]

\[
\begin{bmatrix}
0 & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 0
\end{bmatrix}
\]

compared with \( A_g = \)

\[
\begin{bmatrix}
0 & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 0
\end{bmatrix}
\]
The Ullman Algorithm

Ullmann’s refinement rule for pruning the search tree

- **Observation:**
  - If a vertex of $G$, $v_G$, corresponds to a vertex of $H$, $v_H$, then for each adjacent vertex of $v_G$ in $G$, denoted $v^A_G$, there must be a vertex in $H$, denoted $v^A_H$, in $H$ that holds:
    - $A.$ $v^A_H$ is adjacent to $v_H$ in $H$
    - $B.$ $v^A_H$ corresponds to $v^A_G$

Formally:

\[ \forall x: 1 \leq x \leq |V_G| \left( (a_G(i,x) = 1) \Rightarrow \exists y: 1 \leq y \leq |V_H| (p(x,y) \ast a_H(y,j) = 1) \right) \]
VF2 algorithm

- Another approach to subgraph isomorphism is starting from a node in the pattern to match and recursively check if the neighbors are candidate for an isomorphism.
- Stop when the algorithm has matched all the nodes in the pattern.
- Use some rules to early prune nodes that cannot be part of an isomorphism.
- **Drawbacks:** require several recursive calls.
VF2 algorithm

- If P covers all nodes in Q
  - output P
- else
  - compute the set S(P) of all candidate pairs for inclusion in P
- For each pair p = (u, v) in S(P)
  - if p passes feasibility check
    - P’ ← P ∪ {p}
    - Call the algorithm recursively on P’
- Restore data structures

VF2: Feasibility check

For each pair \((u,v)\) in \(P\)

- The predecessors are already mapped and included in \(P\)
- Their successors can be mapped:
  - Check the degrees
  - Check the labels (if labeled graphs)
  - ...
- Conditions on cardinalities of predecessors and successors to ensure correctness and expandability
Is this enough?

- Ullmann and VF2 perform redundant computations and check
- They don’t have a proper strategy to select the next node/edge to match in the query
  - They blindly traverse the query and try to find correspondences in the graph
- Can we store some information in advance to filter non-isomorphic structures earlier?
Popular subgraph isomorphism algorithms

- **QuickSI (based on tree-features):**

- **GADDI (based on neighboring discriminative substructure):**

- **GraphQL (based on neighborhood signature):**

- **SPath (based on shortest paths):**

For a comparison among algorithms

Filter and verification algorithms

**Input:** a set of graphs $D = \{G_1, ... G_n\}$, and a query $Q$

1. **Filter phase**
   Reduce the search space removing non matching structures/nodes in advance

   **Candidates:** a subset $D' \subseteq D$ of possible matches of $Q$

2. **Verification**
   Check if there are isomorphic subgraphs

   **Output:** a subset $D'' \subseteq D'$ of graphs containing $Q$
QuickSI

- QuickSI is an algorithm that represents the query as a special spanning tree (QI-sequence)
- Has a smart verification phase that avoids redundant computation
  - Traverse the query in a specific order and iteratively match nodes in the graph
- Works for set of small graphs but can be easily adapted to large graphs

Ql-sequences

- Store the signature of an enriched spanning tree in terms of *spanning entries* and *extra entries*

- A **spanning entry** $T_i$ represents a node in the spanning tree containing:
  1. The node of the query to which it is mapped
  2. The label of the node (if labeled)
  3. The parent of the node (since it is a spanning tree)

- An **extra entry** $R_{ij}$ represents an extra information
  1. The degree of the node $i$
  2. An extra edge (not appearing in the spanning tree) among $i$ and another entry vertex $k$, indicated as [edge:$k$] only if the edge is from $T_i$ to $T_k$ and $k < i$.
  3. **NOTE:** $j$ represents an incremental index starting from 1 to the entries relative to node $i$
QI-sequences = Spanning tree + extra edges and degrees

Theorem
Given two graphs $G'$ and $G$, if the two corresponding QI-Sequences are identical, than $G'$ and $G$ are isomorphic.
Another sequence

There exists multiple QI-sequences for the same query. Why?

**Answer:** Depending to the starting vertex and the traversal strategy you obtain different spanning trees.

Let’s start from node 4!

<table>
<thead>
<tr>
<th>Type</th>
<th>[parent, label]</th>
<th>Mapped node</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>[0, C]</td>
<td>v4</td>
</tr>
<tr>
<td>T2</td>
<td>[1, C]</td>
<td>v5</td>
</tr>
<tr>
<td>T3</td>
<td>[1, C]</td>
<td>v3</td>
</tr>
<tr>
<td>T4</td>
<td>[2, C]</td>
<td>v6</td>
</tr>
<tr>
<td>T5</td>
<td>[3, C]</td>
<td>v7</td>
</tr>
<tr>
<td>T6</td>
<td>[5, C]</td>
<td>v2</td>
</tr>
<tr>
<td>R61</td>
<td>[deg:3]</td>
<td></td>
</tr>
<tr>
<td>R62</td>
<td>[edge:3]</td>
<td></td>
</tr>
<tr>
<td>T7</td>
<td>[6, N]</td>
<td>v1</td>
</tr>
</tbody>
</table>
Find a better QI-sequence

- How can we select a good QI-sequence for Q?
  - Ideally we want to reduce the number of candidates in advance

- **Idea:** let us select first edges/nodes with low frequency in the hope of reducing the number of checks.
  - Associate to each node and edge in the query the average number of time the node label or the edge (node label co-occurrence) appear in the set of graphs $D$

  $$
  \phi(v) = \frac{|\{ f | f(v) \in V(G) \land G \in D \} |}{|\{ g | f(v) \in V(G) \land G \in D \} |}, \quad \phi(e) = \frac{|\{ f | f(e) \in E(G) \land G \in D \} |}{|\{ g | f(e) \in E(G) \land G \in D \} |}
  $$

  - Return the minimum spanning tree considering the values computed before for nodes and edges

```
$G_1$

H - C - H
  |   |
  H   H

$G_2$

H - C - H
  |   |
  N   H

Q

1.1

2.5

3.0

```

<table>
<thead>
<tr>
<th>Vertices</th>
<th>$\phi(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>5/2 = 2.5</td>
</tr>
<tr>
<td>H</td>
<td>2/2 = 1</td>
</tr>
<tr>
<td>N</td>
<td>1/1 = 1</td>
</tr>
<tr>
<td>C</td>
<td>1/1 = 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edges</th>
<th>$\phi(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(H,C)</td>
<td>4/2 = 2</td>
</tr>
<tr>
<td>(H,N)</td>
<td>1/1 = 1</td>
</tr>
<tr>
<td>(C,N)</td>
<td>1/1 = 1</td>
</tr>
</tbody>
</table>
QuickSI algorithm

Input
• The QI-sequence of the query Q
• A graph G

Initialization:
• Set d=1

Pick the node $T_d$ in the query QI-sequence

Find all candidates of $n_d$ in the graph G

Refine the candidates using pruning rules

d=d+1
QuickSI algorithm

- **Input:** QI-sequence $SEQ_q$ of a query $q$, a graph $g$
- **Output:** Is $Q$ a subgraph of $G$?

```plaintext
1  if $d > \beta$ then
2     return True;
3  $T := T_d \in SEQ_q$;
4  $V := \emptyset$;
5  if $d = 1$ then
6     $V := \{v | v \in V(g), l(v) = T.l$ and $F_v = 0\};$
7  else
8     $V := \{v | v \in V(g), (v, H_T.p) \in E(g), l(v) = T.l$ and $F_v = 0\};$
9  for each vertex $v \in V$ do
10     for each restriction $R_{dj} \in SEQ_q$ do
11        goto line-9 if $R_{dj}$ is not satisfied;
12     $H_d := v$;
13     $F_v := 1$;
14     if QuickSI$(SEQ, g, H, F, d + 1)$ then
15        return True; /* Terminate */
16     $F_v := 0$;
17  return False;
```

- $d$ represents the depth (i.e., the number of nodes already checked)
- Check if the edges/nodes of $G$ sequentially match edges in the QI-sequence
- Check if the vertex fulfills the restriction $R_{dj}$ (minimum degree – extra edge)
- $H_d$ contains the candidate nodes for the query node $d$
- $F$ stores the nodes in the graph already checked and to check $F_i = 1$ if the node $i$ has been already considered
QuickSI - Filtering with index

- **Idea:** The query is decomposed into features before being checked

- Features are chosen among frequent subgraphs (will see what it means)

- Checking if a graph contains a feature is **NP-complete**

- **Idea:** if two features share a common subgraph don’t check twice!
  - Solution: store the features in an index (a prefix tree) using the QI-sequence
QuickSI: Swift-index

Graph database (set of graphs)

Index

- Features are selected using **frequency**:
  - **In a large graph**: the number of times the structure is contained in the graph
  - **In a set of graphs**: the number of graphs that contain that structure
Lecture road

Containment queries

Similarity queries

Reachability queries
Reachability Queries

- **The problem**: Given two vertices $u$ and $v$ in a directed graph $G$, is there a path from $u$ to $v$?
- In symbols we want an answer to $u \leadsto v$?

Can 11 be reached by 1? Yes
Can 9 be reached by 3? No
Applications

- XML
- Biological networks
- Ontology
- Knowledge representation (Lattice operation)
- Object programming (Class relationship)
- Distributed systems (Reachable states)
Converting to a DAG

**Theorem**
A reachability query can be answered over a corresponding directed acyclic graph (DAG) of the graph based on strongly connected components.

**Definition (Strongly connected component [SCC])**
A strongly connected component of a graph $G$ is a subgraph $G'$ of $G$ such that for each pair of nodes $u, v$ both $u \leadsto v$ and $v \leadsto u$ are true.

To convert a graph into a DAG, we need to:
1. Identify all the strongly connected components $C_1, C_2, ...$
2. Replace all the connected components $C_i$ with a node $v$
3. Connect all the nodes adjacent to the component with node $v$ instead, preserving the direction.

**Questions:**
1. Why do we obtain a DAG?
2. Why the reachability is preserved?
Two main approaches

1. Traverse the graph on-demand performing a BFS or DFS traversal
   - **Problem:** high query cost $\Rightarrow O(|V| + |E|)$ (why? Think about traversing the whole graph!)

2. Construct the transitive closure TC in advance and check the pair $u, v$ exists in the closure, i.e., $(u, v) \in TC$?
   - The transitive closure contains all the pairs of reachable nodes
   - **Problem:** the complete transitive closure is $O(|V| \times |V|) = O(|V|^2)$ in space

How can we have a trade-off between query time and space consumption?
## The different approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Query time</th>
<th>Index construction time</th>
<th>Index size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transitive Closure [1]</td>
<td>O(1)</td>
<td>O(nm)</td>
<td>O(n²)</td>
</tr>
<tr>
<td>Tree+SSPI [2]</td>
<td>O(m - n)</td>
<td>O(m + n)</td>
<td>O(m + n)</td>
</tr>
<tr>
<td>GRIPP [3]</td>
<td>O(m - n)</td>
<td>O(m + n)</td>
<td>O(m + n)</td>
</tr>
<tr>
<td>Dual-Labeling [4]</td>
<td>O(1)</td>
<td>O(n+m+t³)</td>
<td>O(n + t²)</td>
</tr>
<tr>
<td>Tree Cover [5]</td>
<td>O(log n)</td>
<td>O(nm)</td>
<td>O(n²)</td>
</tr>
<tr>
<td>Chain Cover [6]</td>
<td>O(log k)</td>
<td>O(n² + kn√k)</td>
<td>O(nk)</td>
</tr>
<tr>
<td>Path-Tree Cover [7]</td>
<td>O(log² k’)</td>
<td>O(mk’) or O(nm)</td>
<td>O(nk’)</td>
</tr>
<tr>
<td>2-Hop Cover [8]</td>
<td>O(√m)</td>
<td>O(n³</td>
<td>TC</td>
</tr>
<tr>
<td>3-Hop Cover [9]</td>
<td>O(log n + k)</td>
<td>O(kn²</td>
<td>Con(G)</td>
</tr>
</tbody>
</table>
Some intuition on advanced methods

- Use spanning trees instead of DAGs
- Compute a code of a signature for each node
- Complement the spanning tree with the missing edges

Not in the exam
Let $G_S: \langle V, E \rangle$ a tree. The **single interval tree coding scheme (SIT)** assigns a node $u \in V$ a code which is an interval $[u_{\text{start}}, u_{\text{end}}]$, $u_{\text{start}} < u_{\text{end}}$ that represents the preorder and postorder value of a DFS search.

- The preorder and postorder is the first and the last time the node is considered in a DFS, respectively.

<table>
<thead>
<tr>
<th>Node</th>
<th>Start</th>
<th>End</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>0</td>
<td>21</td>
<td>tree</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>20</td>
<td>tree</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>7</td>
<td>tree</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>4</td>
<td>tree</td>
</tr>
<tr>
<td>F</td>
<td>5</td>
<td>6</td>
<td>tree</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>9</td>
<td>tree</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
<td>19</td>
<td>tree</td>
</tr>
<tr>
<td>G</td>
<td>11</td>
<td>14</td>
<td>tree</td>
</tr>
<tr>
<td>B'</td>
<td>12</td>
<td>13</td>
<td>non-tree</td>
</tr>
<tr>
<td>H</td>
<td>15</td>
<td>18</td>
<td>tree</td>
</tr>
<tr>
<td>A'</td>
<td>16</td>
<td>17</td>
<td>non-tree</td>
</tr>
</tbody>
</table>

G is reachable from r just checking the SIT scheme
- $r_{\text{start}} = 0, r_{\text{end}} = 21$
- $G_{\text{start}} = 11, G_{\text{end}} = 14$
- $0 < 11, 14 < 21$ ✔

What about $D \sim B$?
The code does not match but they are reachable.

Check recursively on the tree following non-tree nodes.
Any questions?
References


References


- Fan, W., Li, J., Ma, S., Wang, H. and Wu, Y. Graph homomorphism revisited for graph matching. PVLDB, 2010
