Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Subgraph ismorphism in dynamic graphs

Scientific Students' Association Report

Author:

László Sári
Kinga Csilla Lauter

Advisor:

Dr. Luca Szegletes

2021

Contents

Kivonat

A gráf- és a részgráf-izomorfia vizsgálatának számos alkalmazási területe van kezdve a szociális hálózatok analíizisétől az áramkörök tervezéséig. A gráfok csúcspontokból és csúcspontokat összekötő élekből állnak. A részgráf-izomorfia azt jelenti, hogy G1 és G2 gráfokat nézve van-e olyan G2' részgráfja G2-nek, amelyik izomorf G1-gyel, azaz létezik egy-egyértelmű megfeleltetés (bijekció) G2' és G1 csúcsai és élei között. Ez a probléma NP-teljes, vagyis nem ismerünk polinomiális idejű algoritmust a megoldására. A problémára azonban számos state-of-the-art megoldás létezik, melyek különböző módokon próbálják minimalizálni az adott eljárás lépésszámát.

A valóságban léteznek olyan gráfok is, melyek időben változhatnak. Folyamatosan új csúcsok, élek keletkezhetnek és törlődhetnek. A dolgozat célja a részgráf-izomorfia vizsgálata ilyen dinamikusan változó gráfok esetén.

Abstract

Graph- and subgraph-isomorphism is used for a wide range of problems such as social network analysis and circuit board design. A graph consists of vertices and edges that connect two distinct vertices. Subgraph-isomorphism means that given G1 and G2 graphs, a G2' subgraph of G2 exists such that it is isomorphic with G1. In other words, there exists a bijection between the vertices of G2' and G1. This is a well-known NP-hard problem. Several state-of-the-art algorithms exist that optimize their workings in different ways in order to accelerate their performance.

In reality, however, not all graphs are static in time. Some graphs are chaning constantly; new vertices and edges are created and deleted on a regular basis. The goal of this work is the analysis of subgraph-isomorphism in case of dynamic graphs.

Chapter 1

Introduction

Subgraph isomorphism is an NP-hard problem which has many application areas. It is called substructure search in cheminformatics and it is used to find similar molecular compounds based on their structural formula. In bioinformatics, complex biological systems are decomposed into several different networks, such as protein-protein interaction, metabolic interaction or hormone signaling networks, which are represented as graphs. Analyzing and understanding these large networks requires finding certain topological patterns, i.e. subgraph isomorphism.

Graph pattern matching is also a core concept of social network analysis. Such graphs tend to be extremely large with millions of vertices and billions of edges in the real world. Although subgraph isomorphism has been extensively studied in the past, there was a renewed interest in the topic recently, which yielded some notable results. The newer algorithms significantly out-perform the previous state-of-the-art solutions (Ullman's algorithm [12] [6], VF2 [7]), sometimes even in order of magnitudes. This made it possible to query subgraph isomorphisms in such large graphs. However social networks are not static. In practice, they are frequently updated with typically small changes like adding or removing edges. Despite the changes being small, they will still have an impact on the matches. This means that the matches have to be re-computed from scratch on every update, which is highly infeasible even with the newer and faster subgraph isomorphism algorithms. To minimize unnecessary re-computations, incremental algorithms can be used, that compute the changes in matches based on the changes in the search graph.

[9] discusses in depth several types of incremental graph pattern matching algorithms. However the authors' topic of interest in [8] and [9] is incremental graph pattern matching with (bounded) graph simulation. A graph G matches a pattern q via graph simulation if there exists a binary relation $S \subseteq V_q \times V_G$ such that

1. for each $u \in V_q$, there exists $v \in V_G$ such that $(u, v) \in S$;
2. for each $(u, v) \in S$,
   (a) $\mathcal{L}(u) = \mathcal{L}(v)$, and
   (b) for each edge $(u, u') \in E_q$, there exists a non-empty path $\rho = v \rightsquigarrow v'$ in G such that $(u', v') \in S$ and the length of ρ is less than the maximum allowed length defined on the given edge in q.

Graph simulation is less strict about the topology of its results than graph isomorphism. This can be beneficial if we want to express loose connections in our query patterns, and

on top of that, pattern matching with graph simulation can be done in $\mathcal{O}(n^3)$. However if we do require strict matches, only subgraph isomorphism can come into play. Although the authors provided an incremental algorithm for subgraph isomorphism, it was more of a demonstration that even in an NP-hard case, computing matches incrementally (which is also NP-hard) can out-perform a fast solution, VF2. The approach introduced there does not take full advantage on previous computations, and it was also not evaluated in much detail. Note that this was not the main focus of the paper.

In this work, we investigate how two state-of-the-art algorithms (VF2++, DAF) can be converted into their incremental version. First, we give an introduction how the two algorithms work. Then we describe a method to make them incremental. Finally, we evaluate the results both in terms of complexity and practical measurements.

1.1 Background

Definition 1 (Graph). A graph is a pair $G = (V, E)$, where V is a set of vertices, E is a set of paired vertices that denotes the undirected edges of the graph. ▪

Definition 2 (Labelling). $\mathcal{L} : V \to K$, is a vertex labelling function which maps vertices into arbitrary sets whose elements are the labels of the given node. Two vertices, u, v are equivalent if $\mathcal{L}(u) = \mathcal{L}(v)$. ▪

Definition 3 (Isomorphism). G_1 and G_2 are isomorphic if a bijection exists between V_1 and V_2 such that two vertices are neighbours in G_1 if and only if their respective pairs in G_2 are neighbours, neighbouring vertex pairs have the same number of edges between each other, and a vertex and its pair have the same labels. ▪

Definition 4 (Subgraph). G_1 is a subgraph of G_2 if $V_1 \subseteq V_2$, $E_1 \subseteq E_2$ and two vertices are neighbours in G_1 only if they are neighbours in G_2. ▪

Definition 5 (Induced subgraph). If E_1 consists of those edges from E_2 whose both vertices are in V_1, and E_1 contains all these edges, then G_1 is an induced subgraph of G_2. ▪

Definition 6 (Subgraph isomorphism). G_1 is subgraph isomorphic to G_2 if G_1 is isomorphic to any subgraphs of G_2. ▪

Definition 7 (Induced subgraph isomorphism). G_1 is induced subgraph isomorphic to G_2 if G_1 is isomorphic to any induced subgraphs of G_2. Throughout this paper, we refer to induced subgraph isomorphism with the term of subgraph isomorphism. In this paper, we call G_1 as query graph q and G_2 as data graph G. $M(q, G)$ denotes the set of mappings found by an arbitrary subgraph isomorphism algorithm. ▪

Definition 8 (Mapping). An injection $m : D \to V_G$ is called a (partial) mapping, where $D \subseteq V_q$. ▪

Notation 1. *m_q and m_G denotes the domain and the range of m respectively.*

Definition 9 (Coverage). A mapping m covers a node $u \in V_q \cup V_G$ if $u \in m_q \cup m_G$. ▪

Definition 10 (Whole mapping). A mapping m is a whole mapping if it covers all the nodes of V_q. ▪

This paper concentrates on the induced subgraph isomorphism problem in dynamic graphs, i.e. graphs that change with time. After finding the initial matches given a query graph q and a data graph G, keep the set of matches up to date in response to small updates ΔG on G without recomputing all matches from scratch. ΔG can be one of the following operations:

- add a new node to G,
- remove an existing node from G along with all its associated edges,
- add an edge between two nodes of G and
- remove an existing edge between two nodes of G.

1.2 Algorithms

This section gives a brief overview on how the two algorithms of interest (DAF and VF2++) work.

1.2.1 VF2

Since VF2++ is an extension over VF2, first, we describe how VF2 works. It is a recursive algorithm where each state of the matching process can be associated with a partial mapping m. VF2 starts with an empty mapping and it gradually extends it until a whole mapping is reached. For the current mapping m, it calculates a candidate set of (u, v) pairs to be included in m. It iterates over the (u, v) elements of the candidate set, and if $\mathcal{F}(m, (u, v))$ is feasible then it recursively tries to extend m', where $\mathcal{F}$ is the feasibility function and m' is a partial mapping obtained by adding (u, v) to m.

Definition 11. Let m be a mapping. $Cons(m, (u, v))$ is a logical consistency function which is true if and only if m satisfies the requirements of induced subgraph isomorphism considering q_m and G_m, where q_m and G_m are the subgraphs of q and G induced by m_q and m_G respectively. $Cons$ is used to verify that the consistency of m also holds after extended with (u, v). ▪

Definition 12. Let m be a mapping. $Cut(m)$ is a logical cutting function which is false if there exists a sequence of extensions of m for which the resulting mapping is whole and it satisfies the requirements of induced subgraph isomorphism. Cut is used to determine if the current partial mapping is not contained in any whole mapping, thus trying to extend it would be useless. ▪

Definition 13. The feasibility function $\mathcal{F}$ is defined as follows:

$\mathcal{F}(m, (u, v)) = Cons(m, (u, v)) \wedge \neg Cut(m)$. ▪

The feasibility function ensures that the algorithm considers only (u, v) candidates such that m extended with (u, v) remains consistent, and it eliminates the need of processing partial mappings for which it can be proven that they cannot be extended to a whole mapping.

Let $T_q(m) := u \in V_q \; m_q : \exists u' \in m_q : (u, u') \in E_q$, and
$T_G(m) := v \in V_G \; m_G : \exists v' \in m_G : (v, v') \in E_G$.

Algorithm 1.1: VF2 algorithm

```
1  Procedure vf2(m)
2      if m covers V_q then
4          Output(m)
5      else
7          P_m ← the candidate set of pairs for extending m
8          foreach (u, v) ∈ P_m do
9              if F(m, (u, v)) then
11                 vf2 (extend(m, (u, v)))
12         end
13     end
```

The candidate set for extending m is P_m. P_m consists of the pairs of uncovered neighbors of covered nodes. If there exists no such pair, P_m contains all uncovered nodes. Formally,

$$P_m = \begin{cases} T_q(m) \times T_G(m), & \text{if } T_q(m) \neq \emptyset \text{ and } T_G(m) \neq \emptyset \\ (V_q \setminus m_q) \times (V_G \setminus m_G), & \text{otherwise.} \end{cases}$$

1.2.2 VF2++

VF2++ [11] was published by Alpár Jüttner and Péter Madarasi in 2018. The algorithm makes performance improvements compared to VF2 by calculating a matching order, in which the vertices of q are processed in a partial mapping, and by applying a more efficient cutting function.

The order of the nodes of q to be matched have a huge impact on the number of visited states. In case of VF2, the lack of strictly defined matching order can result in evaluating an unnecessarily larger number of states that could have been pruned if the matching order was different. By choosing a proper matching order, one can eliminate such non-fruitful computations earlier.

Example 1. *The following example was taken from the original VF2++ paper. Let q be a query graph that cannot be mapped to G, and $u \in V_q$. Let $q' := V_q \cup \{u'_1, u'_2, \cdots, u'_k\}, E_q \cup \{(u, u'_1), (u'_1, u'_2), \cdots, (u'_{k-1}, u'_k)\}$, i.e. q' is the same graph as q which was extended with a k long path, which is disjoint from q and one of its starting nodes is connected to $u \in V_q$.*

If the first k vertices in the matching order were the nodes of the newly added path, VF2 would iterate through all possible k long paths in G, only to realize that no partial mappings can be extended to q'.

However, if the algorithm started the matching process with vertices from q, it would not have matched any nodes from the path.

The first step of VF2++ is determining the matching order in which the algorithm will process the vertices of the query graph q. During the order's computation, VF2++ takes into account the structure of q and its labeling. First, it chooses a node with the least common label and with the largest degree. This node will be the root of a tree, which will be used to determine the final matching order $\mathcal{O}$.

$$root = r | r \in \arg\max_{deg}(\arg\min_{label_{\mathcal{O}}}(V_q \setminus \mathcal{O})),$$

where $label_{\mathcal{O}}(n) := |v \in V_G : \mathcal{L}(n) = \mathcal{L}(v)| - |u \in \mathcal{O} : \mathcal{L}(n) = \mathcal{L}(u)|$, i.e. it computes the difference between the number of vertices in G with the label of n and the number of vertices in $\mathcal{O}$ with the label of n. Next, the algorithm computes a tree T by traversing q in BFS (Breadth first search) order from the previously calculated root, and it processes each level of T the following way. Let $V_{q,d}$ denote the set of vertices of T in depth d. The process selects the vertices with the largest connectivity respect to $\mathcal{O}$ denoted by *conn*, i.e. those nodes whose number of neighbors that are already in $\mathcal{O}$ is the largest. Then from these nodes, it selects the ones with the largest degree, then those with the rarest label.

$$o = u | u \in \arg\min_{label_{\mathcal{O}}}(\arg\max_{deg}(\arg\max_{conn}(V_{q,d}))),$$

The selected node o is appended to $\mathcal{O}$, and it is removed from $V_{q,d}$. This continues until $V_{q,d}$ has no more elements. Algorithm 1.2 and 1.3 describe the matching order procedures on a high level.

Algorithm 1.2: VF2++ order

```
1  Procedure vf2pp_order()
3      O := ∅
4      while V_q \ O ≠ ∅ do
6          r ∈ arg max_deg(arg min_label_O(V_q \ O))
8          T ← BFS(r)
9          foreach d = 0, 1, ··· depth(T) do
11             V_{q,d} := nodes of the d-th level
13             process(V_q, d)
14         end
15     end
```

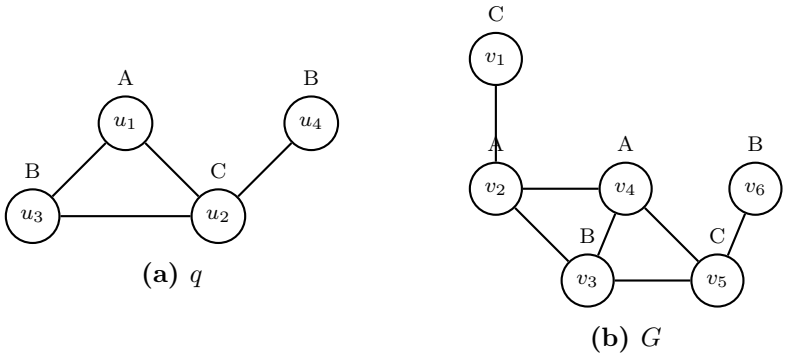
Algorithm 1.3: VF2++ process the d-th level of T

```
1  Procedure process(V_{q,d})
2      while V_{q,d} ≠ ∅ do
4          o ∈ arg min_label_O(arg max_deg(arg max_conn(V_{q,d})))
6          V_{q,d} ← V_{q,d} \ o
8          O.append(o)
9      end
```

Example 2. *Let q and G be the graphs from figure 1.1.*

We want to compute the matching order of q. To select the root node for our BFS tree T, we need the least frequent labels. In this example, all labels in G are assigned twice, which means that our root candidates are still u_1, u_2, u_3, u_4. From these vertices, the vertex with the largest degree is u_2, thus it will be the root of T. The resulting tree from traversing q in BFS order can be seen in figure 1.2.

Processing T begins with the 0-th level. On this level, the node with the largest connectivity, with the largest degree and with the most infrequent label is u_2 because it is the only vertex. u_2 is appended to the order $\mathcal{O}$ and since there is no other nodes left, we move on to the next level. Here, all vertices have the same connectivity because all nodes are adjacent to u_2. The nodes with the largest degree are u_1 and u_3, both of them have a degree of



(a) q **(b)** G

Figure 1.1: Graphs for VF2++ order example

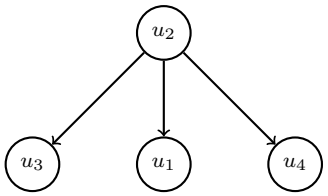


Figure 1.2: BFS tree of q from u_2

2. Neither of the vertices share the same label as the only vertex in the order, thus their label's frequencies are to be considered. In this case both of them are equally frequent with a frequency of 2. We have to choose one from them, let u_1 be the new node to add to $\mathcal{O}$. Since there are two more remaining vertices on the level, we stay on the 1st level. This time, the node with the largest connectivity is u_3 because it is adjacent to u_1 and u_2, while u_4 is only adjacent to u_2. u_3 is added to $\mathcal{O}$. The only vertex left on the first level is u_4, thus it is finally appended to the order. All $u \in V_q$ is added to the order and the procedure stops. The final order is the following:

$$\mathcal{O} = \{u_2, u_1, u_3, u_4\}$$

VF2++'s additional changes to the original include a refined way of determining the candidates of $u \in V_q$ and applying cutting rules on them. In this version, $u \in V_q$'s candidates will be $v \in V_G$ that are in the neighborhood of m_G, are not covered yet and are consistent with m.

$$P_m(u) = v \in V_G | \neg covered_m(v) \wedge \forall u' \in V_q : (u, u') \in E_q \wedge u' \in m_q \iff (v, m(u')) \in E_G$$

VF2++ also introduces a new cutting rule which verifies that for a given candidate pair (u, v), v in G has at least as many neighbors with the appropriate labels as u in q.

Summarizing the two extensions made over VF2, the final VF2++ algorithm is defined in algorithm 1.4.

Example 3. *Let us continue our example started in example 2. As a reminder, the matching order is $\mathcal{O} = \{u_2, u_1, u_3, u_4\}$. Now we have to find all mappings of q in G. VF2++ starts with an empty mapping. The next vertex to match according to $\mathcal{O}$ is u_2. Since it is the first node in the mapping, its candidates will be the set of all nodes $v_j \in V_G$. u_2 has three neighbors. Two with label B and one with label A. Now we can check the feasibility of our candidates.*

Algorithm 1.4: VF2++ algorithm

```
1  Procedure vf2pp(q, G)
3      O ← vf2pp_order()
5      match(∅, 0)
6  Procedure match(m, depth)
7      if m covers V_q then
9          Output(m)
10     else
12         u ← O[d]
14         P_m(u) =
             v ∈ V_G | ¬covered_m(v) ∧ ∀u' ∈ V_q : (u, u') ∈ E_q ∧ u' ∈ m_q ⟺ (v, m(u')) ∈ E_G
15         foreach (u, v) ∈ P_m do
16             if F(m, (u, v)) then
18                 match (extend(m, (u, v)), d + 1)
19         end
20     end
```

- v_1 *has one neighbor with label* A*, thus* (u_2, v_1) *is infeasible.*
- v_2 *has three neighbors, one with label* A*, one with label* B *and one with label* C*. It is short on neighbors with label* B*, thus* (u_2, v_2) *is also infeasible.*
- v_3 *has two neighbors with label* A *and two with label* C*, which means* (u_2, v_3) *is once again infeasible.*
- v_4 *has also not enough neighbors with label* B*, which makes* (u_2, v_4) *infeasible.*
- v_5 *has one neighbor with label* A *and two neighbors with label* B *which means* (u_2, v_5) *is feasible, thus we extend our empty mapping with this pair.*

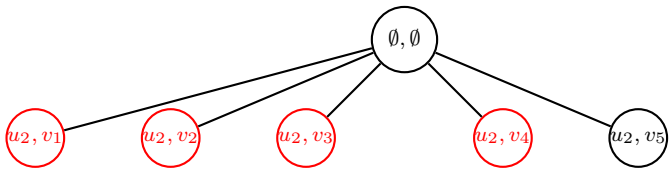


Figure 1.3: First level of a VF2++ search tree

The next vertex in the order is u_1 *which has one neighbor that is in the current mapping:* u_2*.* u_1*'s candidates are the vertices* $v_j \in V_G$ *that are adjacent to* $m(u_2) = v_5$*. The candidates are* v_3, v_4, v_5*. Now we apply our cutting rules.* v_3 *is not feasable because it has no neighbors with label* B*.* v_4 *is feasable because it has all the necessary neighbors with the right labels. We extend the tree with* (u_1, v_4)*. Figure 1.4 shows the second level of the search tree.*

The thrid vertex is u_3*. We select its neighbor in* q*, which is* u_1*. Now we consider* u_1*'s pair* v_4*, and we select its uncovered neighbors in* G*. These are the vertices* v_2 *and* v_3*. We check if these vertices are adjacent to* u_2*'s pair* v_5*.* v_2 *is not, thus the only candidate is* v_5*.* v_5 *also passes the cutting test, and we extend the mapping with* (u_3, v_3)*. Figure 1.5 shows the search tree with three levels.*

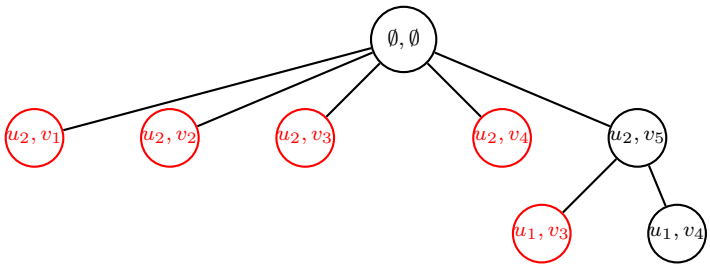


Figure 1.4: Second level of a VF2++ search tree

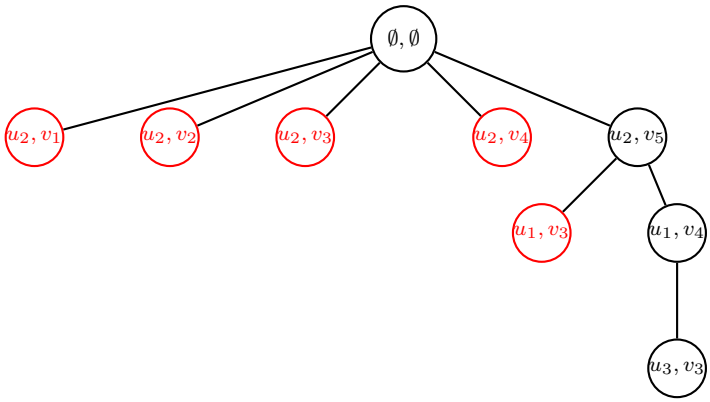


Figure 1.5: Third level of a VF2++ search tree

The last vertex is u_4. Its only neighbor in q is u_2. u_2's pair is v_5. We select the its uncovered neighbors in G which is v_6. It also passes the feasability test, thus we extend the current mapping with (u_4, v_6), which is a whole mapping.

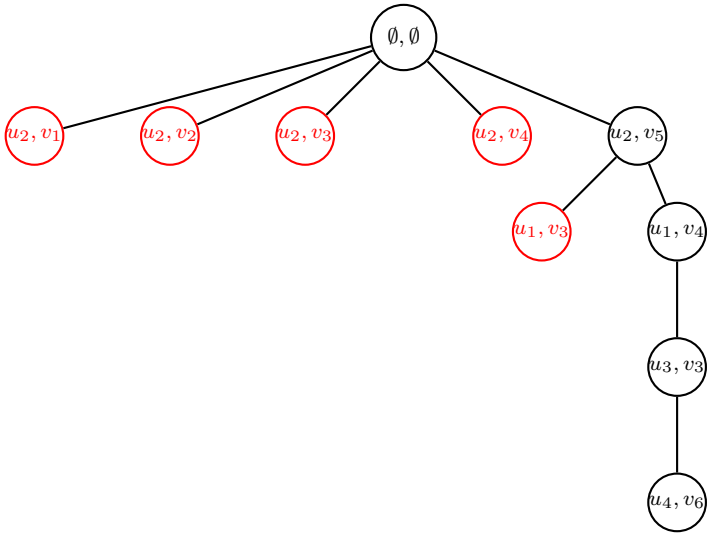


Figure 1.6: Fourth level of a VF2++ search tree

Now we backtrack in the search tree. v_3 has no other candidates. We check u_1's last candidate, v_6, which is infeasible. The last candidate to check is (u_2, v_6) which is also infeasible. The final search tree can be found in figure 1.7.

The mapping that we found via VF2++ is the following: $(u_1, v_4), (u_2, v_5), (u_3, v_3), (u_4, v_6)$.

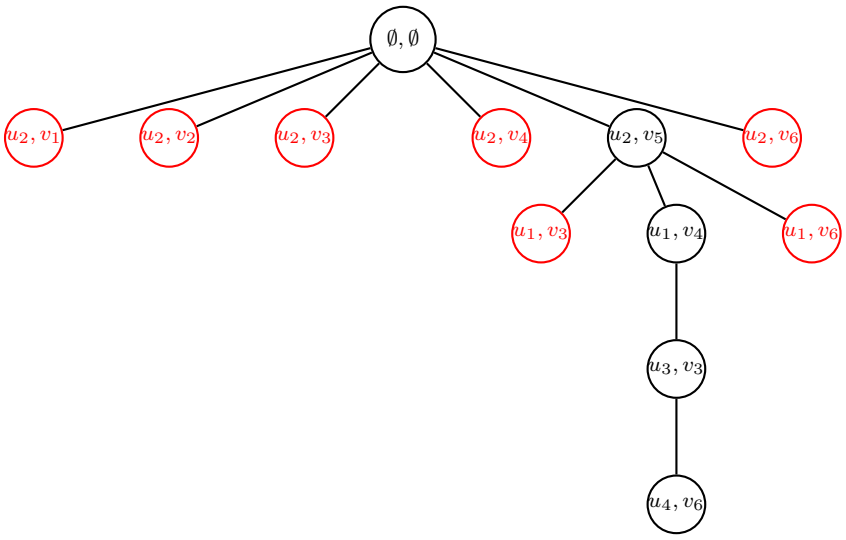


Figure 1.7: Complete VF2++ search tree

1.2.3 DAF

DAF was originally introduced in [10]. The algorithm converts the input graphs q and G into its own special data structure which is computed by (D)dynamic programming. It also uses an (A)adaptive matching order and (F)failing sets, hence the name, DAF.

As a first step, DAF creates its own data structures from q and G, called *candidate space* (CS). Then DAF searches mappings in CS with a backtracking algorithm in which the matching order is adaptive. Furthermore, it uses failing sets in order to prune the parts of the search space for which it can be proven that they contain no whole mappings. Algorithm 1.5 shows a high level description of DAF.

Algorithm 1.5: DAF

Input: q, G
Output: all mappings
2 $q_D \leftarrow build\_dag(q, G)$
4 $CS \leftarrow build\_cs(q, q_D, G)$
6 $M \leftarrow \emptyset$
8 $backtrack(q, q_D, CS, M)$

1.2.3.1 Build DAG

DAF creates a DAG (directed acyclic graph) q_D from q. q_D will be used for computing CS, and later for computing the matching order adaptively. For each node $u \in V_q$, let the initial candidate set be:

$$C_{ini}(u) = v | v \in V_G \wedge deg_G(v) \geq deg_q(u) \wedge \mathcal{L}_q(u) = \mathcal{L}_G(v)$$

, i.e. those $v \in V_G$ vertices with the same label as u in q whose degree is also greater than or equal to the degree of u. Now that the initial candidates are computed, we define the root of q_d and the first vertex to be matched as

$$r \in \arg\min \frac{|C_{ini}(u)|}{deg_q(u)},$$

where $u \in V_q$. The less this quotient is the less candidates u will have and since it has a large degree, it will have more topological constraints then others, thus the algorithm can prune more branches.

Starting from r, DAF traverses q in BFS order and it directs all edges in q_D from upper levels to lower levels. I.e. given an undirected edge $e = (u, v) \in E_q$, the resulting edge in q_D will have a direction of $u \to v$, if $depth(u, BFS_q(r)) \geq depth(v, BFS_q(r))$, otherwise the direction will be $u \leftarrow v$. Nodes on the same level of the BFS tree are sorted by how infrequent their labels are and by their degrees in descending order. The direction of an edge between nodes on the same level is determined by previously described order. After q_D is computed, it is used to create another DAG q_D^{-1} which is the same as q_D but the direction of its edges are reversed.

The next step is to build the candidate space structure CS.

1.2.3.2 Build CS

Given a query graph q and a data graph G, a CS structure built from q and G contains candidates $C(u)$ for all $u \in V_q$ such that

1. $\forall u \in V_q : \exists$ a candidate set $C(u) \subseteq C_{ini}(u)$, and
2. there is an edge between $v \in C(u)$ and $v' \in C(u')$ if and only if $(u, u') \in E_q$ and $(v, v') \in E_G$.

Initially all $C(u)$ in CS is set to $C_{ini}(u)$. The initial candidate sets can be further refined to exclude unnecessary elements with the help of q_D and q_D^{-1}. The refinement procedure alternates between the two DAGs in each of its execution. The current DAG is denoted by q', and initially it is set to q_D^{-1}. The procedure refines all $C(u)$ into $C'(u)$ with dynamic programming.

$v \in C'(u)$ if and only if $v \in C(u)$ and there exists a v_c vertex which is a neighbor of v, and $v_c \in C(u_c)$ for all u_c children of u in q'. I.e. v remains in the candidate set if it is connected to at least one candidate from all children of u in q'. The procedure traverses q' in a reversed order, thus nodes will be processed only after all their children has been processed. According to the authors' empirical study, CS cannot be further refined after three steps usually.

During refinement, only the candidate sets are maintained, the edges are not. They are added to the CS structure only after the refinement procedure has ended. The edges are stored in adjacency lists $N^u_{u_c}(v)$ for each $v \in C(u)$, and for each $(u, u_c) \in E_{q_D}$. $N^u_{u_c}(v)$ contains a list of vertices v_c adjacent to v in G such that $v_c \in C(u_c)$.

1.2.3.3 Backtracking and adaptive matching order

Finally, DAF searches all mappings of q in CS which corresponds searching mappings in G. A vertex $u \in V_{q_D}$ is extendable respected to partial mapping m if all predecessors of u in q_D are covered by m. In each state of the algorithm, it determines the set of extendable

vertices. For each extendable vertex, it calculates the set of extendable candidates $C_m(u)$ respected to partial mapping m. Let $p_1, \cdots, p_k$ be the predecessors of an extendable vertex u in q_D. The extendable candidate set of u is defined as the candidates that are adjacent to the pairs of all neighbors of u in q. Formally,

$$C_m(u) = \cap_{i=1}^{k} N_u^{p_i}(m(p_i))$$

The matching order is determined based on the extendable candidate sets. The authors describe two methods.

- Candidate-size order: the next vertex to be matched is an extendable vertex u where $|C_m(u)|$ is minimal.
- Path-size order: the next selected extendable vertex u is where $w_m(u)$ is minimal. $w_m(u)$ is an estimate for the weight of a path mapping. Since we used the candidate size order in our experiments, we don't define $w_m(u)$ in much more detail here.

The matching order is adaptive because it depends on the current partial mapping m. After an extendable vertex u has been selected, DAF extends the mapping with (u, v_c) for each candidate v_c in $C_m(u)$, and it backtracks after traversing the whole subtree.

1.2.3.4 Failing sets

DAF uses failing sets to find and prune branches that are unnecessary to traverse because they cannot contain any whole mappings. The search tree is traversed in DFS (depth first search) order with a backtracking algorithm. It is possible however, that a partial mapping m extended with (u, v) will not be wholly extendable because of other conflicting pairs in m which were added higher up in the search tree, thus matching u with other v' vertices will not result in any whole mappings either. Failing sets help to find these branches. A failing set is denoted by F_m where m is a partial mapping corresponding to a path in the search tree. Failing sets are computed from bottom-up. Let $anc(u)$ and $succ(u)$ denote the ancestors and successors of u in q, respectively. A leaf can belong to one of three classes.

- Conflict class: If (u, v) is a leaf, and v is already covered then (u, v) belongs to the conflict class, and it is denoted by $(u, v)!$. In case of a conflict class, $F_m = anc(u) \cup anc(m^{-1}(v))$.
- Empty class: If u has no extendable candidates, $(u, \emptyset)$ belongs to the empty class. $F_m = anc(u)$.
- Mapping class: If the mapping belonging to the leaf is a whole mapping, then it belongs to the mapping class. $F_m = \emptyset$.

Failing sets for internal, non-leaf vertices are defined by the failing sets of their children. Let (u_n, v_i) be the children of the current node (u, v), and let m_i denote the partial mappings of the children and let the current mapping of (u, v) be m. F_m is computed the way described in algorithm 1.6.

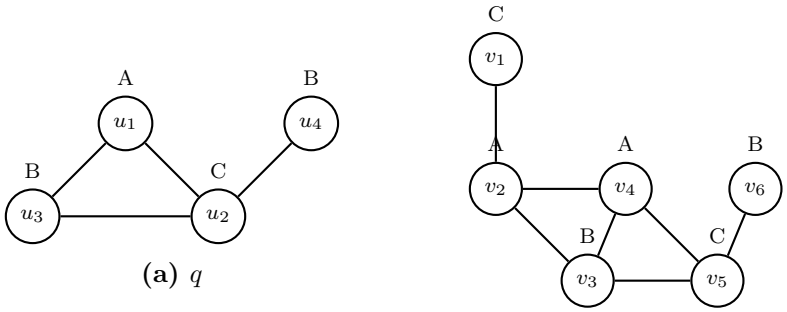
If the algorithm is at a node (u, v) in the search tree whose $F_m \neq \emptyset$ and $u \notin F_m$, then it means that it does not matter, which candidate is matched to u, no whole mappings will

Algorithm 1.6: Calculate the failing set of an internal node

if $\exists$*child node* m_i *such that* $F_{m_i} = \emptyset$ **then**
  $F_m = \emptyset$
else
  if $\exists$*child node* m_i *such that* $u_n \notin F_{m_i}$ **then**
    $F_m = F_{m_i}$
  else
    $F_m = \cup_{i=1}^{k} F_{m_i}$
  end
end

be found, thus all siblings of (u, v) are redundant which means that these branches can be pruned.

Example 4. *In the following example, we follow through an execution of DAF. Figure 1.8 shows the graphs we will use during the execution.*



(a) q **(b)** G

Figure 1.8: Graphs for DAF execution example

First, we compute the initial C_{ini} *values.*

$$C_{ini}(u_1) = \{v_2, v_4\} \rightarrow \frac{|C_{ini(u_1)}|}{deg_q(u_1)} = \frac{2}{2} = 1$$

$$C_{ini}(u_2) = \{v_5\} \rightarrow \frac{|C_{ini(u_2)}|}{deg_q(u_2)} = \frac{1}{3} = 0.\dot{3}$$

$$C_{ini}(u_3) = \{v_3\} \rightarrow \frac{|C_{ini(u_3)}|}{deg_q(u_3)} = \frac{1}{2} = 0.5$$

$$C_{ini}(u_4) = \{v_3, v_6\} \rightarrow \frac{|C_{ini(u_4)}|}{deg_q(u_4)} = \frac{2}{1} = 2$$

The smallest number is $0.\dot{3}$*, thus the root will be* u_2*. Starting from* u_2 *in* q*, we get the following* q_D *and* q_D^{-1} *graphs.*

The next step is to create the initial CS structure and then reduce it with the help of q_D *and* q_D^{-1}*. Figure 1.10 shows the initial CS structure.*

We use $q' = q_D^{-1}$ *for our first refinement. The first vertex will be* u_2 *because it has no children, thus* $C'(u_2) = \{v_5\}$ *The second vertex to be processed is* u_1 *because all its children (*u_2*) has been processed already.* v_2 *is not connected with any of the candidates*

(a) q_D **(b)** q_D^{-1}

Figure 1.9: DAGs created from q

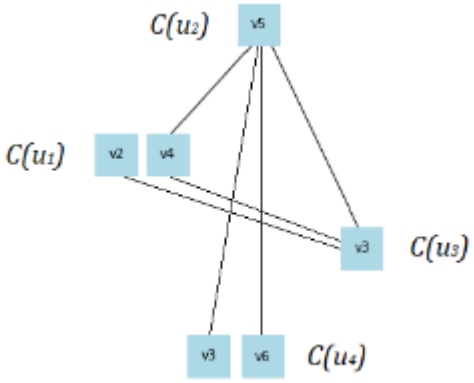


Figure 1.10: Initial CS structure

of u_2. Because of this, v_2 will no longer be part of u_1's candidates: $C'(u_1) = \{v_4\}$. The next vertex is u_4, because its children u_2 has been processed. Its new candidates are $C'(u_4) = \{v_3, v_6\}$. The last vertex is u_3, for which $C'(u_3) = \{v_3\}$. Figure 1.11 shows the CS structure after the first refinement.

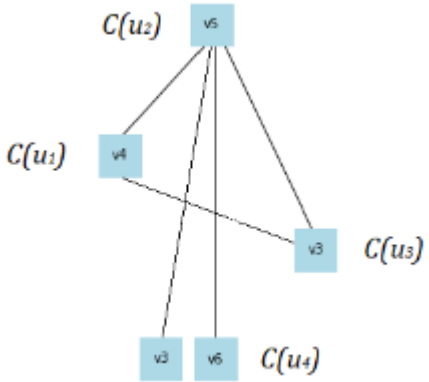


Figure 1.11: CS structure after the first refinement

Now $q' = q_D$, however it is not possible to refine CS any further. Next, we find all mappings in CS. The first elements of the search tree is u_2 and its candidate (u_2, v_5). The set of extendable vertices is $\{u_1, u_4\}$ because their only parent u_2 in q_D has been processed. We compute the appropriate candidates:

$$C_m(u_1) = \{v_4\}$$

$$C_m(u_4) = \{v_3, v_6\}$$

Because of the candidate-size order, we extend the search tree with u_1 *and its candidate* (u_1, v_4)*. At this point,* u_3 *becomes extendable because its parents were procesed. Its candidates are* $C_m(u_3) = \{v_3\}$*, and since it is the only extendable vertex at the moment, we extend the search tree with* (u_3, v_3)*. The only vertex left is* u_4*.* $C_m(u_4) = \{v_3, v_6\}$*. The search tree is extended with* (u_4, v_3)*!* (v_3 *is already covered, hence the exclamation mark) and* (u_4, v_6)*. The branch containing* (u_4, v_6) *corresponds to a whole mapping. The failing sets of all nodes on this branch will be empty. The failing set of* (u_4, v_3)*!* *is* $F_m = \{u_3, u_4\}$*. In this case, failing sets have no effect on the algorithm. Figure 1.12 shows the search tree with failing sets produced by DAF.*

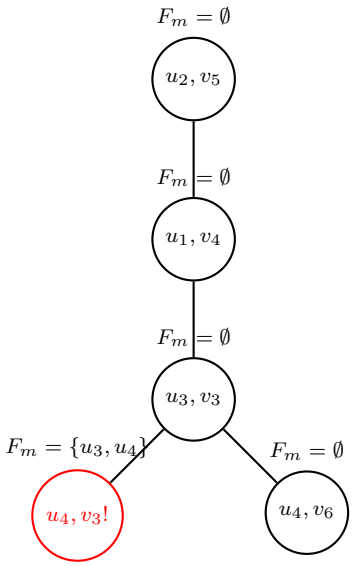


Figure 1.12: Search tree with failing sets created by DAF

We got as a result that q *is isomorphic to one of* G*'s subgraph and the mapping is the following:* $(u_1, v_4), (u_2, v_5), (u_3, v_3), (u_4, v_6)$*.*

Chapter 2

Incremental algorithms

This chapter describes two incremental algorithms for finding subgraph isomorphisms in dynamic graphs. First we introduce the approach described by Wenfei Fan in [9]. Then we present our generic method which can be applied to DAF and VF2++ respectively.

2.1 Locality based method

Let d denote the diameter of the query graph q which corresponds to the length of the longest shortest path in q. Let Δe denote the addition or deletion of edge $e = (v, v')$ in G. Moreover let $V(d, e)$ be the set of vertices in G that are within a distance d from v or v'. Finally, let $G(d, e)$ be the subgraph of G induced by $V(d, e)$, and $\Delta G(d, e)$ be the subgraph of ΔG induced by $V(d, e)$, where $\Delta G = (V, E \setminus \{e\})$ or $\Delta G = (V, E \cup \{e\})$ depending on the operation of Δe. With these notations, we can define the locality property of subgraph isomorphism. For any given changes of Δe in G, the changes ΔM in the set of mappings $M(q, G)$ is the difference between $M(q, G(d, e))$ and $M(q, \Delta G(d, e))$, i.e. the difference between the mappings found in the dth neighborhood of the original graph and in the neighborhood's modified version. This is a trivial property because new mappings can appear or become obsolete only in the immediate vicinity of the affected edge e. Mappings further away from e than d are not affected by Δe because neither vertices of e are reachable from there. Regardless of that a new edge was added or an old one was removed, mappings can both appear and become obsolete in the updated graph. The mappings that have to be added to the existing set is $M(q, \Delta G(d, e)) \setminus M(q, G(d, e))$, while the mappings that have to be removed are $M(q, G(d, e)) \setminus M(q, \Delta G(d, e))$.

The algorithm uses this locality property as follows. Given a query graph q and a data graph G, compute the initial set of mappings M with a classic subgraph isomorphism algorithm, e.g. VF2++. Then, for each update Δe, (1) find the diameter d of q, (2) extract the subgraph $\Delta G(d, e)$ from G, (3) compute $M(q, \Delta G(e, d))$, (4) then update M as described above.

Instead of re-computing all mappings in G, this method works on a subset of G. Although a regular subgraph isomorphism is running in the background, reducing the size of the input data graph can make this variant faster. The effectiveness depends on the query graph and the data graph. If $G(d, e)$ remains small compared to G, there will be a massive speed up. However if $G(d, e) \sim G$, for example in case of small world networks, the algorithm performs the same as a regular subgraph isomorphism algorithm.

Algorithm 2.1: Locality algorithm

Input: $q, G, \Delta e, M = VF2++(q, G)$

```
1 d ← diameter(q);
2 G_{d,e} ← G(d, e);
3 ΔM ← VF2++(q, G_{d,e});
4 M = M ∪ M(q, ΔG(d, e)) \ M(q, G(d, e));
5 M = M \ M(q, G(d, e)) \ M(q, ΔG(d, e));
6 return M
```

2.2 Search tree based method

This chapter describes our method of making the previously introduced algorithms incremental. One way or another, both algorithms traverse a search tree eventually. In both cases the search tree is purely abstract, it has no physical manifestation in the memory. It is only a concept for depicting the recursive paths an algorithm traverses during its matching process, allowing us to analyze the search space of a given solution.

2.2.1 Search tree

We propose to store the traversed search tree while running a subgraph isomorphism algorithm initially, and make use of it in future updates. $T_{q,G}$ is a search tree of query graph q and data graph G, where paths from the root to a leaf correspond to (partial) mappings, and a node contains an (u, v) pair where $u \in V_q, v \in V_G$ denotes a single vertex mapping. The root r of $T_{q,G}$ contains an empty mapping. A root-to-leaf path, whose length $l = |V_q|$ is a complete mapping because it maps all nodes of q. The rest of the root-to-leaf paths correspond to partial mappings.

In a typical scenario, at the end of an execution of a subgraph isomorphism algorithm, the search tree contains some complete mappings and orders of magnitude more partial mappings that could not be extended any further. These partial mappings play a crucial role in the incremental version. One can think about them as potential partially precalculated mappings. The reason, why a partial mapping could not be extended is because the algorithm ran out of valid candidates in the area of the mapped nodes' neighborhood in G. This is caused by a conflict between the topology of q and the topology of the mapped nodes and their neighborhood. I.e. there was no more vertices left such that the mapping obtained by extending the current partial mapping with it would remain a partial subgraph isomorphism. These conflicts could be resolved when a new edge is added to or an old one is deleted from G. At this point, having these partial mappings in memory can speed up the process of removing matches that became obsolete and finding new ones that were impossible in the past.

In the upcoming sections, we define an incremental algorithm for each type of graph modifications.

2.2.2 Node operations

First, we describe incremental node operations, i.e. vertex insertion and deletion.

2.2.2.1 Node deletion

Deleting a node v_d from G can only reduce the original number of mappings. Mappings that did not contain v_d are unaffected, while those that contained v_d have to be removed since they no longer can map to v_d. Now we only have to think through that no new mappings could arise. Indeed, that cannot be the case because the only edges that were removed are edges belonging to v_d which means only partial mappings which contain v_d are affected. However since v_d itself is also deleted, these mappings become obsolete, which means that no new mappings can be found.

Since there are no new mappings to be found, the only work to do in case of a node deletion is pruning the search tree $T_{q,G}$. More specifically, cut off every sub-branch that starts with a node (u, v) where $v = v_d$. In worst case scenario this would mean traversing the whole search space. We already saw however, that any given change can only affect mappings whose vertices are in the dth neighborhood $N_d(G, v_d)$ of v_d. Thus we can skip any branches that map a query graph vertex to a node $v \notin N_d(G, v_d)$.

Algorithm 2.2: Delete node incrementally

```
   Input: q, G, T_{q,G}, v_d
 2 d ← diameter(q)
 4 N_d(G, v_d) ← dth neighborhood of v_d in G
 5 Procedure delete_node(node, v_d)
 6     foreach child ∈ node.children do
 8         (u, v) ← child.mapping
 9         if v = v_d then
11             node.remove(child)
12         else if v ∈ N_d(G, v_d) then
14             delete_node(child, v_d)
15         end
16     end
```

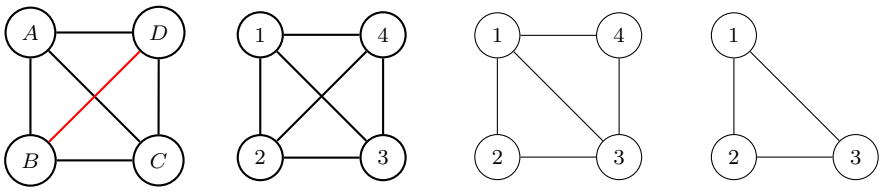
2.2.2.2 Node insertion

Inserting a new node has no effects on mappings because at the time of insertion, it cannot be part of any mappings because it has no connections. Thus inserting a new node needs no special care.

2.2.3 Edge operations

In case of induced subgraph isomorphism, both inserting and deleting an edge can result in loosing and gaining mappings. Removing an edge e makes mappings where nodes of e were part of the mapping no longer valid. At the same time, in case of partial mappings where e made it impossible to further extend, removing it may cause new mappings to appear. For example in 2.1, as a result of removing the BD edge from G, $M(q_1, G')$ will loose all of its mappings, while $M(q_2, G')$ will find only new mappings.

The same goes for edge insertion. If we were to insert the BD edge into G then $M(q_1, G')$ would find new mappings, while $M(q_2, G')$ should remove the half of them. It is clear that these can happen simultaneously, as well.



(a) G **(b)** q_1 **(c)** q_2 **(d)** q_2

Figure 2.1: Example how mappings can appear and disappear in both cases of edge deletion and insertion.

2.2.3.1 Edge deletion

When an edge $e = (v_1, v_2)$ is removed from G, first, we have to prune the search tree to remove mappings that are no longer valid and we have to somehow find the new mappings. We have to prune all branches that correspond to a partial mapping which both v_1 and v_2 are part of, and there is an edge between $m^{-1}(v_1)$ and $m^{-1}(v_2)$ in q. We find all paths that correspond to such partial mappings, and prune them down from the first point where v_1 and v_2 both appeared in the path. 2.3 shows the pruning algorithm when an edge was deleted.

Algorithm 2.3: Prune search tree on edge deletion

```
1  Procedure prune(node, v1, v2)
3      (u, v) = node.mapping
4      if v1 not marked as found ∧ v = v1 then
6          mark v1 as found
7      else if v2 not marked as found ∧ v = v2 then
9          mark v2 as found
10     end
11     if v1 is marked ∧ v2 is marked then
12         if (m^-1(v1), m^-1(v2)) ∈ E_q then
14             return true
15         end
16     end
18     node.children = {child ∈ node.children : ¬prune(child, v1, v2)}
```

Fig 2.2 shows an example partial search tree of G and q_3 from Fig. 2.1. The red nodes represent the pruned sub-branches as a result of removing the BD edge from G.

Finding new mappings is somewhat harder because we have to extend the search tree instead of just pruning it. Because of the locality property of the incremental subgraph isomorphism problem, we know for sure that any new mappings that will appear have to contain either v_1 or v_2 or both. Thus our job is reduced to look for partial mappings that contain either of those two vertices. There are two kinds of partial mappings to consider. Typically, the search tree will contain some partial mappings that contain one of these two vertices but definitely not all of them. As a first step, we add these partial mappings to the search tree. We traverse the tree and check if a node has a child that maps the next vertex $u \in V_q$ to v_i. If it does not, and v_i was not covered already in the mapping, and it causes no conflicts between the topology of the mapping and q then we create a new child under the current node with $m(u) = v_i$. Now that we have all the partial mappings containing v_1 or v_2 - note that not all of them are expanded yet - we find all those leaves whose path

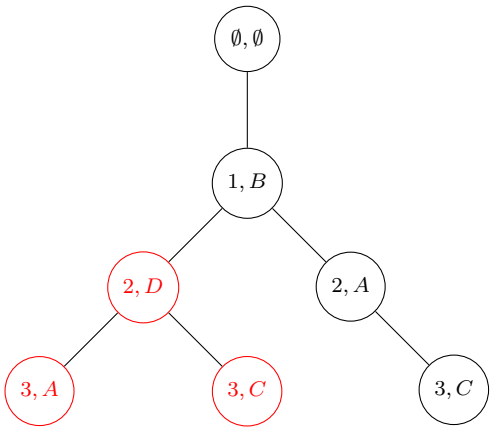


Figure 2.2: Pruning of a partial search tree of G and q_3 from Fig. 2.1.

from the root corresponds to a partial mapping that contains v_1 or v_2 and we execute our choice of subgraph isomorphism algorithm from that point. Naturally, these two steps can be combined into one traversal. If we hit a node where (1) we can create a new child or (2) a node is a leaf and its path contains v_1 or v_2, and it is not a complete mapping, we execute our subgraph isomorphism algorithm. We can speed up the algorithm by only considering those branches that contain vertices such that $\forall v \in m : v \in N_d(G, v_1) \cup N_d(G, v_2)$, i.e. branches that contain only mapped vertices such that they are in the dth neighborhood of e. algorithm 2.4 formally describes the algorithm of finding new mappings after an edge has been deleted, and algorithm 2.5 shows the incremental algorithm for subgraph isomorphism in case of edge deletion.

Algorithm 2.4: Find new mappings incrementally

```
1  Procedure find_mappings(node, forced_candidates)
2      if node.depth = |V_q| ∨ node.mapping.v ∉ N_d(G, (v_1, v_2)) then
4          return
6      u ← node.extendable_vertex_of_children
7      foreach v_c ∈ forced_candidates do
8          if ∄child : child.mapping = (u, v_c) then
9              if v_c is not covered ∧ v_c is good candidate then
11                 child = node.add_child(u, v_c)
13                 ISO(child)
14     end
15     if node is leaf ∧ (v_1 ∈ m ∨ v_2 ∈ m) then
17         ISO(node)
18     else
19         foreach child ∈ node.children do
21             find_mappings (child, forced_candidates)
22         end
23     end
```

2.2.3.2 Edge insertion

Inserting a new edge into G will have a really similar effect on the mappings as and edge deletion. Mappings can both appear and disappear from the original mapping set. In fact,

Algorithm 2.5: Delete edge incrementally

Input: $q, G, T_{q,G}, e$
$d \leftarrow \text{diameter}(q)$
$(v_1, v_2) = e$
$N_d(G, (v_1, v_2)) \leftarrow d$th neighborhood of v_1 and v_2 in G
`prune` (root($T_{q,G}$), v_1, v_2)
`find_mappings` (root($T_{q,G}$), $\{v_1, v_2\}$)

the algorithm described above is almost applicable to this problem out of the box. It is clear that finding new mappings will be the same in this case, as well. We have to take care about how we prune the search tree before that. Indeed, removing nodes that pass the test $(m^{-1}(v_1), m^{-1}(v_2)) \in E_q$ makes no sense because we just added that new edge. We can make this condition generic however, so that both edge operations can use the same predicate. Instead of separating the two cases, we will simply check that an edge between v_1 and v_2 in G exists if and only if an edge between $m^{-1}(v_1)$ and $m^{-1}(v_2)$ in q exists. With these modifications, the final pruning algorithm can be found in algorithm 2.6.

Algorithm 2.6: Prune search tree on edge operation

Procedure `prune`(*node,* v_1, v_2)
  (u, v) = node.mapping
  if v_1 *not marked as found* $\wedge$ $v = v_1$ **then**
    mark v_1 as found
  else if v_2 *not marked as found* $\wedge$ $v = v_2$ **then**
    mark v_2 as found
  end
  if v_1 *is marked* $\wedge$ v_2 *is marked* **then**
    is_edge_in_q $\leftarrow (m^{-1}(v_1), m^{-1}(v_2)) \in E_q$
    is_edge_in_G $\leftarrow (v_1, v_2) \in E_G$
    if *is_edge_in_q* $\neq$ *is_edge_in_G* **then**
      return *true*
    end
  end
  node.children $= \{child \in node.children : \neg\texttt{prune}(child, v_1, v_2)\}$

Chapter 3

Evaluation

In this chapter, we evaluate our version of incremental induced subgraph isomorphism algorithms.

3.0.1 Implementation details

All algorithms were written in Python. Although Python is not necessarily the right tool for implementing high performance algorithms, the main goal of this work was to verify if we can make significant improvements compared to running VF2++/DAF from scratch by computing mappings incrementally. To evaluate the correctness of our work, two already existing induced subgraph isomorphism implementations were used: networkx's (Python graph library) isomorphism module and boost c++ library's vf2_subgraph_iso module[1]. Both modules implement VF2. We tried to compare the performance against the original VF2++ implementation [4] however it did not produce the same results as the selected baseline implementations. We also wanted to compare our results to the original DAF implementation, however no sources for the project were available at time of writing, only a set of pre-compiled binaries [2] which we did not run in the end.

3.0.2 Datasets

We used multiple datasets in our experiments. The first dataset was the Graph Challange[3] dataset which was published by Massachusetts Institute of Technology (MIT) and Amazon Web Services for the challange. The dataset consists of several large graphs.

- as: network of autonomous systems.
- ca-GrQc: collaboration network of general relativity researchers on arxiv.org.
- ca-HepTh: collaboration network of high energz physics researchers on arxiv.org.
- oregon1: peering infromation of autonomous systems in project Route Views of University of Oregon.

The other dataset contains multiple generated graphs, namely scale-free networks generated by the Barabasi-Albert model[5].

3.0.3 Measurements

For each of the graphs, the following operations were made. We ran an initial VF2++ and DAF on the given graph G with query graph defined in figure 3.1. Then we randomly deleted and inserted nodes and edges into G, and we measured how much time does it take to incrementally compute the new mapping set. After the graph modifications we ran our baseline measurement on the resulting graph G', which was networkx's VF2 implementation and we measured the time it took for the algorithm to finish. We also verified that the number of ismorphisms found by all algorithms were the same.

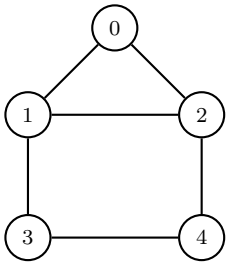


Figure 3.1: Query graph for our experiments

The first step of our incremental algorithm is to run VF2++ and DAF from scratch. DAF seems to struggle with more complex queries. Note that this might be caused by our inefficient implementation. We tried to reach out for the authors of DAF to discuss this issue, however we did not get any answers. As expected, VF2++ is generally faster than VF2. More importantly, the results verify our proposed algorithm in case of VF2++. It is significantly faster to update a graph and incrementally search changes in the mappings than running a subgraph ismorphism algorithm from scratch. This applies to all kinds of graphs, even where it takes all three variants a significant amount of time to find mappings, the incremental version still finishes at least one order of magnitudes earlier.

Figure 3.2 shows the results of our measurements on the ca graphs from Graph Challange. ca-GrQc is special, this was the only graph where VF2 could beat both initial runs of VF2++ and DAF. It took 20s for VF2 to find all initial 868012 mappings, while it took 36s and 232s for VF2++ and DAF respectively. However even in this case, the incremental version will outperform the original in the long run, because adding/removing nodes and edges is so much faster. In this case, incremental VF2++ removed nodes in 0.62s and it removed/inserted edges in 7.48s on average. These values were 0.166s and 178s for DAF. The edges to be deleted/inserted were selected randomly.

Algorithm	Initial	Node deletion	Edge deletion
VF2++	35.7s	0.62s	7.48s
DAF	232s	0.166s	178s
networkx.VF2	20s	N/A	N/A

Table 3.1: Table showing results of ca-GrQc

The oregon graphs from Graph Challange were the largest and this were the graphs where the incremental version really shined. To find the initial 3,568,766 mappings in oregon_010331, it took 1595s, 7106s, and 4896s for VF2++, DAF and VF2 respectively. In case of node deletion and edge insertion/deletion, incremental VF2++ was able to update the set of mappings in 37.6s and 179.5s. Incremental DAF was able to delete nodes in 2.51s but it took almost as much time (4133s) to update the mappings in case of

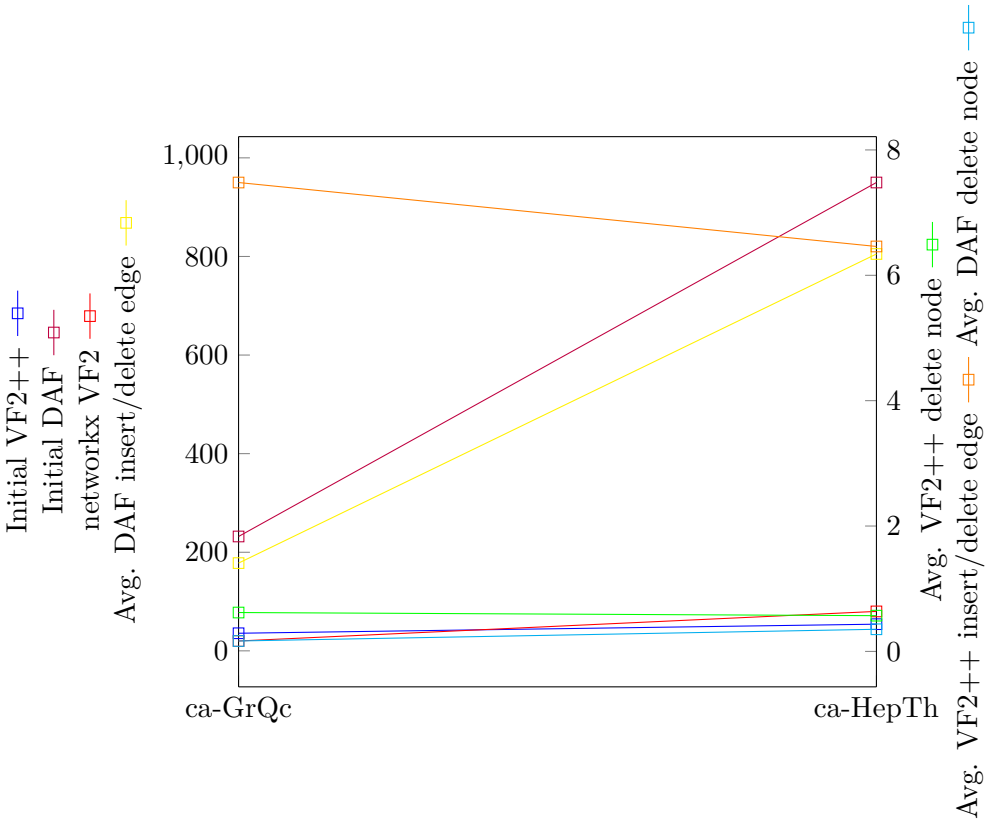


Figure 3.2: Results of ca graphs from Graph Challange

edge insertions as its initial run. There are missing DAF results for the other two oregon graphs. This is because it did not finish in 10,000s.

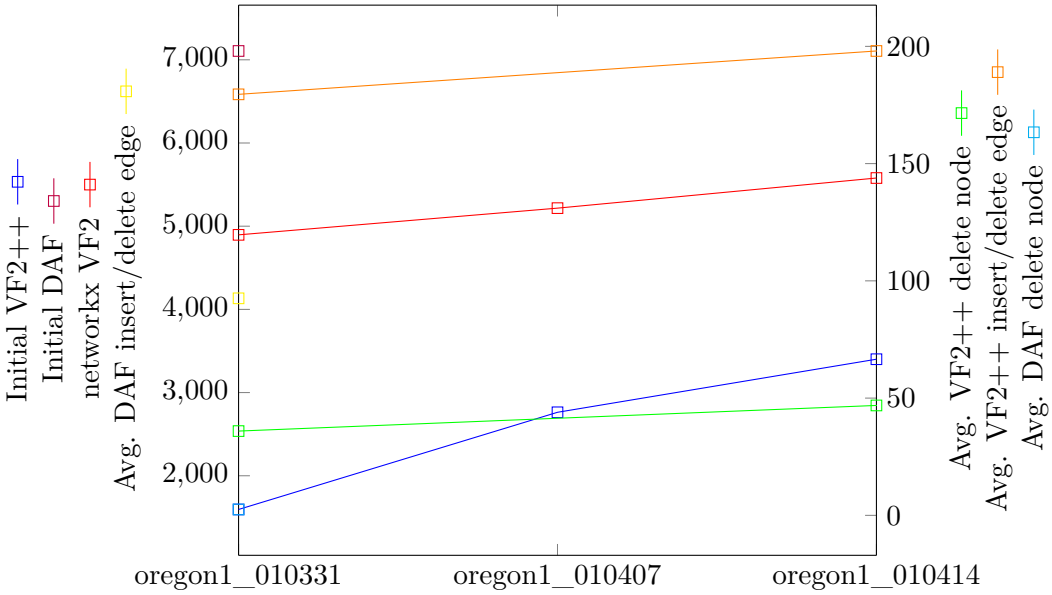


Figure 3.3: Results of oregon graphs from Graph Challange

Incremental VF2++ was also the fastest in case of scale-free graphs. We can see that it took roughly 1/10-th of VF2's time for VF2++ to finish its initial run, and it took roughly

Algorithm	Initial	Node deletion	Edge deletion
VF2++	1595s	35.94s	179.54s
DAF	7106s	2.51s	4133s
networkx.VF2	4896s	N/A	N/A

Table 3.2: Table showing results of oregon-010331

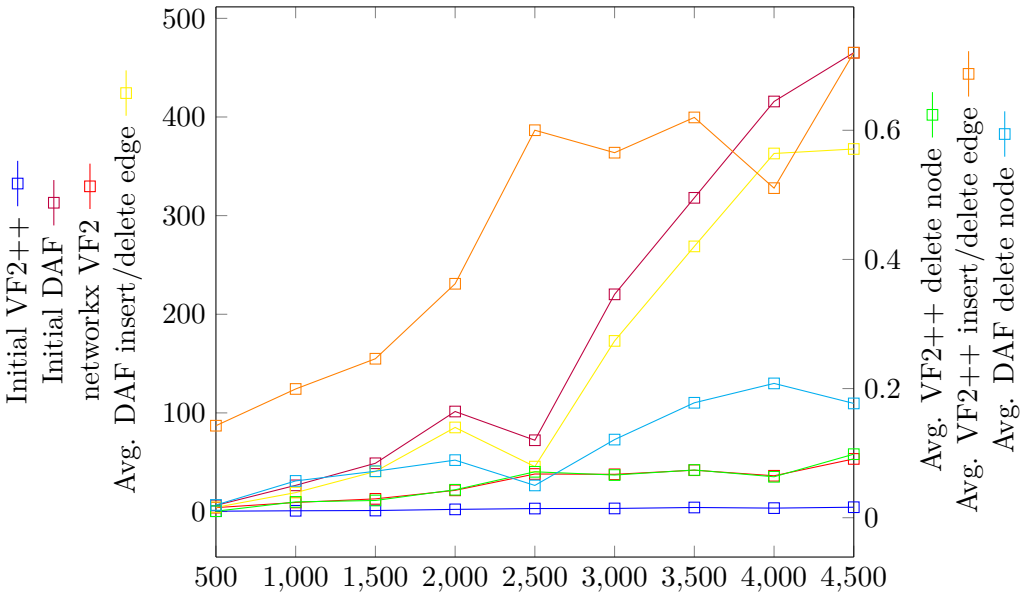


Figure 3.4: Results of scale-free graphs generated by Barabasi-model

1/10-th of the time of VF2++'s initial run for the incremental algorithm to find the new set of mappings in a modified graph. DAF performed similarly poorly as in previous cases.

Algorithm	Initial	Node deletion	Edge deletion
VF2++	4.38s	0.0984s	0.7201s
DAF	465.15s	0.177s	367.56s
networkx.VF2	53.33s	N/A	N/A

Table 3.3: Table showing results of a scale-free graph with 4500 nodes and 13500 edges

Chapter 4

Future work

We implemented our algorithms in Python to verify that our approach could work in reality. As stated earlier, Python in itself is not suited for high performance algorithms. Future works can include implementing these incremental algorithms in a language which allows more fine-tuned memory handling, e.g. C++.

Since both VF2++ and DAF work on tree structures, both algorithms can be naturally implemented paralelly. This also applies for our modified incremental algorithm. Future research can involve parallelizing these algorithms.

Another area of future work would be using neural networks for detecting isomorphisms in large graphs. The use cases may not fully align with our current work, because statistical approaches can not guarantee to find all subgraph ismorphisms. However the performance gain might out-weight the issue of missing mappings.

Chapter 5

Conclusion

In this work, we introduced a novel incremental algorithm for finding induced subgraph isomorphisms in large dynamic graphs. The algorithm can be used in cases when there are frequent changes to the data graph. We implemented a version based on VF2++ and DAF. In these use-cases the proposed procedure based of VF2++ out-performs any induced subgraph isomorphism solution run from scratch.

Bibliography

[1] C++ boost vf2. https://www.boost.org/doc/libs/master/libs/graph/doc/vf2_sub_graph_iso.html.

[2] Daf binaries. https://github.com/SNUCSE-CTA/DAF.

[3] Graph challange datasets. http://graphchallenge.mit.edu/data-sets.

[4] Lemon graph library. https://lemon.cs.elte.hu/trac/lemon/log/lemon/lemon/vf2pp.h?rev=1433.

[5] Albert-Laszlo Barabasi and Reka Albert. Albert, r.: Emergence of scaling in random networks. science 286, 509-512. *Science (New York, N.Y.)*, 286:509–12, 11 1999. DOI: 10.1126/science.286.5439.509.

[6] Uros Cibej and Jurij Mihelič. Improvements to ullmann's algorithm for the subgraph isomorphism problem. *International Journal of Pattern Recognition and Artificial Intelligence*, 29:150705201143007, 07 2015. DOI: 10.1142/S0218001415500251.

[7] Luigi Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26:1367 – 1372, 11 2004. DOI: 10.1109/TPAMI.2004.75.

[8] Wenfei Fan. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, page 8–21, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307918. DOI: 10.1145/2274576.2274578. URL https://doi.org/10.1145/2274576.2274578.

[9] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 925–936, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306614. DOI: 10.1145/1989323.1989420. URL https://doi.org/10.1145/1989323.1989420.

[10] Myoungji Han, Hyunjoong Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. *Proceedings of the 2019 International Conference on Management of Data*, 2019.

[11] Alpár Jüttner and Péter Madarasi. Vf2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242, 03 2018. DOI: 10.1016/j.dam.2018.02.018.

[12] Julian R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23:31 – 42, 1976.