Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Elektronikus terelők

Scientific Students' Association Report

Author:

Jakab Gipsz
Második Szerző

Advisor:

dr. Egy Konzulens
Kettő Konzulens

2014

Contents

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

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 , 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.

ullman, vf2

[] discusses in depth several types of incremental graph pattern matching algorithms. However the authors' topic of interest in [] 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 $S \subseteq V_q \times V$ such that

cite https://www.pure.ed.ac.uk/ws/portalfiles/portal/17894104/Fan_Li_ET_AL_2011_Incremental_Graph_Pattern_Matching.pdf

1. for each $u \in V_q$, there exists $v \in V$ 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. $M(q, G)$ denotes the set of mappings found by an arbitrary subgraph isomorphism algorithm. ▪

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

Definition 9. m_q and m_G denotes the domain and the range of m respectively. ▪

Definition 10. A mapping m covers a node $u \in V_q \cup V_G$ if $u \in m_q \cup m_G$. ▪

Definition 11. 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,
- 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.

describe mappings in a section above this one

Definition 12. 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 13. 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 14. 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 $T_G(m) := v \in V_G \; m_G : \exists v' \in m_G : (v, v') \in E_G$.

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,

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

Algorithm 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
```

1.2.2 VF2++

VF2++ 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.

cite

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 lead to evaluating states that cannot be wholly extended, only to realize this deep down on the search tree. By choosing a proper matching order, one can eliminate such unnecessary computations earlier.

example

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 $\mathcal{O}$, 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 15 and 9 describe the matching order procedures on a high level.

Algorithm 2: VF2++ order

Procedure `vf2_order()`
  $\mathcal{O} := \emptyset$
  while $V_q \setminus \mathcal{O} \neq \emptyset$ **do**
    $r \in \arg\max_{deg}(\arg\min_{label_{\mathcal{O}}}(V_q \setminus \mathcal{O}))$
    $T \leftarrow \text{BFS}(r)$
    foreach $d = 0, 1, \cdots depth(T)$ **do**
      $V_{q,d} :=$ nodes of the d-th level
      `process`(V_q, d)
    end
  end

Algorithm 3: VF2++ process the d-th level of T

Procedure `process`$(V_{q,d})$
  while $V_{q,d} \neq \emptyset$ **do**
    $o \in \arg\min_{label_{\mathcal{O}}}(\arg\max_{deg}(\arg\max_{conn}(V_{q,d})))$
    $V_{q,d} \leftarrow V_{q,d} \setminus o$
    $\mathcal{O}$.append(o)
  end

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, $u \in q$'s candidates will be $v \in G$ that are in the neighborhood of m_G, are not covered yet and are consistent with m.

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

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 Alg. 20. example

1.2.3 DAF

DAF was originally introduced in . 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. cite

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 8 shows a high level description of DAF.

Algorithm 4: VF2++ algorithm

Procedure `vf2pp`(q, G)
  $\mathcal{O} \leftarrow$ `order()`
  `match`$(\emptyset, 0)$
Procedure `match`$(m, depth)$
  if m *covers* V_q **then**
    Output(m)
  else
    $u \leftarrow \mathcal{O}[d]$
    $P_m(u) =$
     $v \in G | \neg covered_m(v) \wedge \forall u' \in q : (u, u') \in E_q \wedge u' \in m_q \iff (v, m(u')) \in E_G$
    foreach $(u, v) \in P_m$ **do**
      if $\mathcal{F}(m, (u, v))$ **then**
        `match` (extend$(m, (u, v))$, $d + 1$)
    end
  end

Algorithm 5: DAF

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

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 $u \in q$, 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 $v \in G$ 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 lesser this quotient is the lesser 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 $u \rightarrow v$, 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 $u \in q$ 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 $u \in q_D$ 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. 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. A leaf can belong to one of three classes. Let $anc(u)$ and $succ(u)$ denote the ancestors and successors of u in q, respectively.

- 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, while let the current mapping of (u, v) be m. F_m is computed the way described in algorithm 12.

Algorithm 6: Calculate the failing set of an internal node

```
if ∃child node m_i such that F_{m_i} = ∅ then
    F_m = ∅
else
    if ∃child node m_i such that u_n ∉ F_{m_i} then
        F_m = F_{m_i}
    else
        F_m = ∪_{i=1}^{k} F_{m_i}
    end
end
```

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

example

Chapter 2

Incremental algorithms

This chapter describes two incremental algorithms for finding subgraph isomorphisms in dynamic graphs. First we introduce the approach proposed in []. 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 7: Locality algorithm

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

$d \leftarrow diameter(q)$;
$G_{d,e} \leftarrow G(d, e)$;
$\Delta M \leftarrow VF2++(q, G_{d,e})$;
$M = M \cup M(q, \Delta G(d, e)) \setminus M(q, G(d, e))$;
$M = M \setminus M(q, G(d, e)) \setminus M(q, \Delta G(d, e))$;
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 deletion

Deleting a node from 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 8: Delete node incrementally

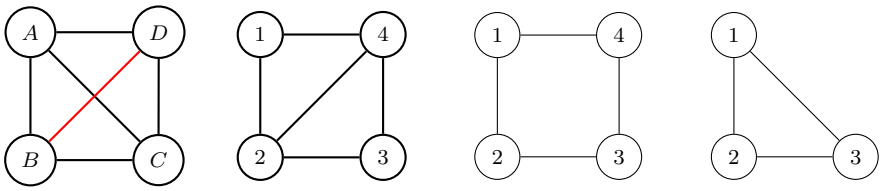
```
   Input: q, G, T_{q,G}, v_d
 2 d ← diameter(q)
 4 N_d(G, v_d) ← dth neighborhood of v_d in G
 5 Procedure delete_node(node, v_d)
 6     foreach child ∈ node.children do
 8         (u, v) ← child.mapping
 9         if v = v_d then
11             node.remove(child)
12         else if v ∈ N_d(G, v_d)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

2.2.3 Node insertion

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

2.2.4 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 half of its mappings, while $M(q_2, G')$ will double the amount of its mappings.



(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.

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.

2.2.4.1 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. 9 shows the pruning algorithm when an edge was deleted.

Algorithm 9: 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.

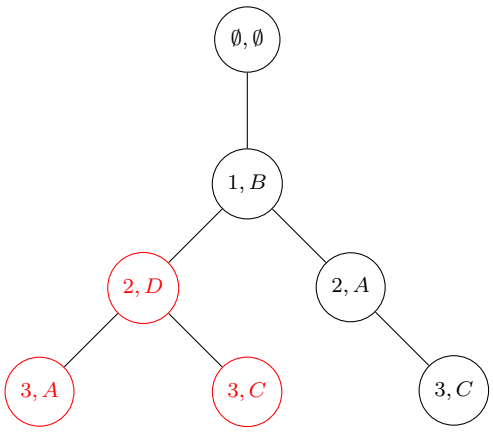


Figure 2.2: Pruning of a partial search tree of G and q_3 from Fig. 2.1.

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 $u \in q$ 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 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. Alg. 10 formally describes the algorithm of finding new mappings after an edge has been deleted, and Alg. 11 shows the incremental algorithm for subgraph isomorphism in case of edge deletion.

Algorithm 10: 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
```

Algorithm 11: Delete edge incrementally

```
   Input: q, G, T_{q,G}, e
2  d ← diameter(q)
4  (v_1, v_2) = e
6  N_d(G, (v_1, v_2)) ← dth neighborhood of v_1 and v_2 in G
8  prune (root(T_{q,G}), v_1, v_2)
10 find_mappings (root(T_{q,G}), {v_1, v_2})
```

2.2.4.2 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, 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 Alg. 12.

Algorithm 12: Prune search tree on edge operation

```
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
13         is_edge_in_q ← (m^-1(v1), m^-1(v2)) ∈ E_q
15         is_edge_in_G ← (v1, v2) ∈ E_G
16         if is_edge_in_q ≠ is_edge_in_G then
18             return true
19         end
20     end
22     node.children = {child ∈ node.children : ¬prune(child, v1, v2)}
```

Chapter 3

Implementation

All algorithms were written in Python. 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. Both modules implement VF2.

3.1 DAF

While implementing DAF, we chose the candidate-size matching order because it was the faster option to evaluate.

Acknowledgements

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Bibliography

Appendix

A.1 A TeXstudio felülete

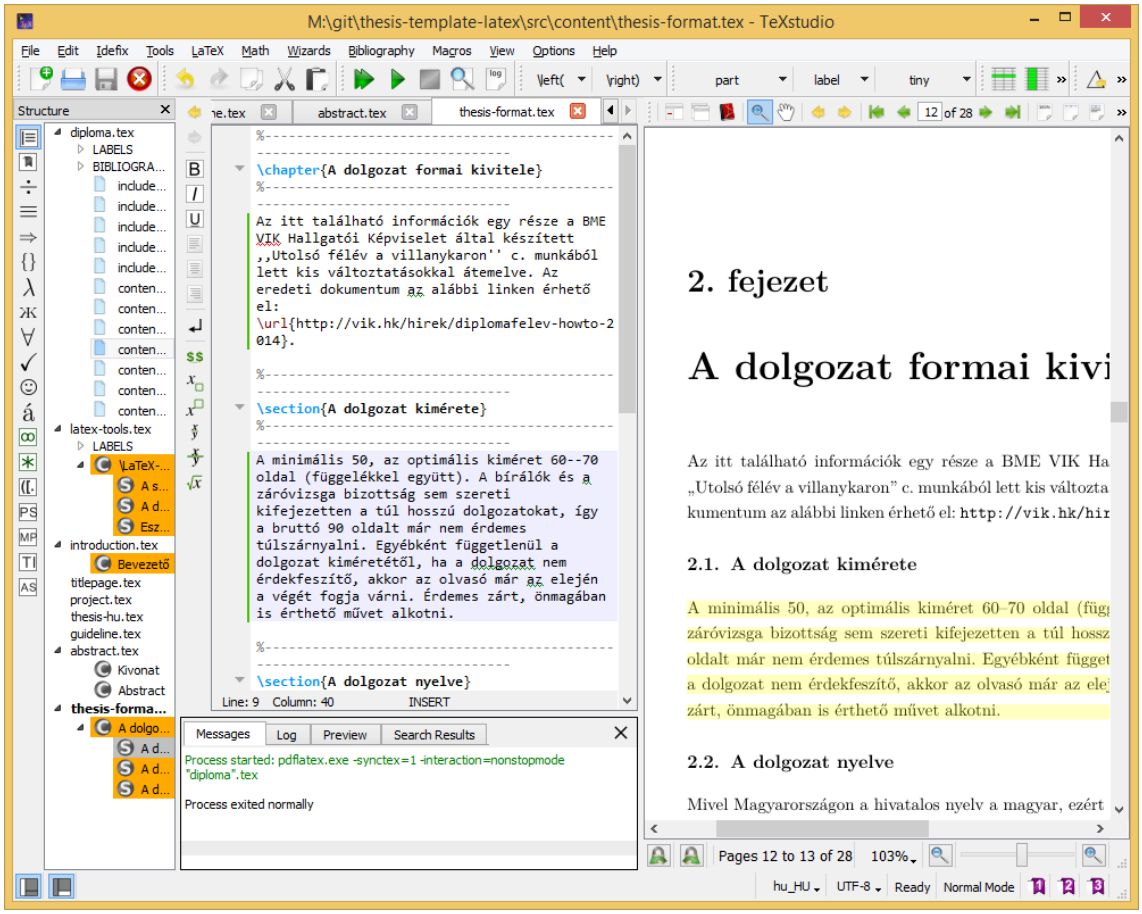


Figure A.1.1: A TeXstudio LaTeX-szerkesztő.

A.2 Válasz az „Élet, a világmindenség, meg minden” kérdésére

A Pitagorasz-tételből levezetve

$$c^2 = a^2 + b^2 = 42. \tag{A.2.1}$$

A Faraday-indukciós törvényből levezetve

$$\operatorname{rot} E = -\frac{dB}{dt} \qquad \longrightarrow \qquad U_i = \oint_{\mathbf{L}} \mathbf{Edl} = -\frac{d}{dt} \int_{A} \mathbf{Bda} = 42. \tag{A.2.2}$$