An Optimization Technique for Subgraph Matching Strategies

Gernot Veit Batz

batz@ipd.info.uni-karlsruhe.de
Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik, Universität Karlsruhe (TH), Germany

16. September 2006

Internal Report 2006-7
ISSN 1432-7864

Abstract

Subgraph matching (aka graph pattern matching or the subgraph isomorphism problem) is NP-complete. But in practice subgraph matching should be performed in reasonable time if possible.
In this work a heuristically optimizing approach to subgraph matching on labeled graphs is described. It relies on the fact that the runtime of the matching process can vary significantly for different matching strategies. The finding of a good matching strategy is stated as an optimization problem which is solved heuristically. The cost model for the possible matching strategies takes the structure of the present host graph into account. The necessary information can be obtained by an analysis of the host graph.

1 Introduction

Given a labeled *pattern graph* G and a labeled *host graph* H the question is, whether there is an occurrence of G in H. This task, which is often referred to as *subgraph matching*, *graph pattern matching*, or the *subgraph isomorphism problem*, is known to be NP-complete (see problem GT48 in Garey and Johnson [9]). But for many problem instances it is possible to find a solution in reasonable time. This in turn may depend on the order in which the elements of the pattern graph are searched for. In other words for given G and H there are different matching strategies and in several cases it makes a significant difference which of them is used.

In this paper the finding of good matching strategies is understood as an optimization problem which is solved heuristically. For this reason matching strategies are represented as *search plans* (section 3). According to the runtime raised by different search plans (section 4) a cost model is defined (section 5). Then a heuristic method for the generation of preferably good search plans is given (section 7). The cost model assigns a cost to all the search plans possible for G. This cost in turn depends on the current host graph H. If there is not enough domain specific knowledge available, the required information can be obtained by an analysis of H (section 6).

2 Foundations

2.1 Labeled Directed Multigraphs

In this work graphs are *labeled directed multigraphs*. A multigraph allows multiple edges of equal direction between two nodes. Moreover all nodes and edges have labels assigned. The *node labels* are taken from the finite alphabet Σ_V and the *edge labels* from the finite alphabet Σ_E. A graph G is 6-tupel

$$G = (V_G, E_G, src_G, tgt_G, \ell_{V_G}, \ell_{E_G})$$

with the functions

$$\begin{array}{ll} src_G \colon E_G \to V_G & \ell_{V_G} \colon V_G \to \Sigma_V \\ tgt_G \colon E_G \to V_G & \ell_{E_G} \colon E_G \to \Sigma_E. \end{array}$$

The notion of multigraphs requires edges to be first class citizens. For this reason the functions src_G and tgt_G are needed, which assign a *source* and a *targt node* to every edge. The functions ℓ_{V_G} and ℓ_{E_G} assign the labels to the nodes and edges of G. For $v \in V_G$ and $e \in E_G$ the two sets

$$\begin{array}{lcl} inc_{V_G}(v) & := & \{e \in E_G \mid src_G(e) = v \vee tgt_G(e) = v\} \ \subseteq \ E_G \\ inc_{E_G}(e) & := & \{src_G(e), tgt_G(e)\} \ \subseteq \ V_G \end{array}$$

denote the edges (or nodes) incident to a given node (or edge). For a graph G a *subgraph* $G' \subseteq G$ is a graph

$$G' = \left(V_{G'}, E_{G'}, src_G|_{E_{G'}}, tgt_G|_{E_{G'}}, \ell_{V_G}|_{V_{G'}}, \ell_{E_G}|_{E_{G'}}\right)$$

with $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$. The *empty graph* O is the graph with empty node and edge set, i.e., $V_O = E_O = \emptyset$. By $|G| := |V_G| + |E_G|$ the number of all elements of a graph G is denoted.

A sequence of edges $e_1, \ldots, e_n$ of G is called a *path* in G if $tgt_G(e_i) = src_G(e_{i+1})$ holds for $1 \leq i < n$. We say the path *leads* from $src_G(e_1)$ to $tgt_G(e_n)$. Let $v_0 \in V_G$

a node of G. If there is a path in G leading from v_0 to v for all other nodes $v \in V_G \backslash \{v_0\}$, then v_0 is called a *root* of G.

Let v_0 a root of G and $n := |E_G|$. Then a sequence of edges $e_1, \ldots, e_n$ is called a *traversal* of G with root v_0 if $\{e_1, \ldots, e_n\} = E_G$ and $src_G(e_i) \in \bigcup_{j=1}^{i-1} inc_{E_G}(e_j) \cup \{v_0\}$ holds for $1 \leq i \leq n$.

2.2 Graph Homomorphisms

The fact that a graph G occurs in another graph H is expressed by *graph homomorphisms*. A graph homomorphisms $\varphi \colon G \to H$ from G to H is a pair of functions $\varphi = (\varphi_V \colon V_G \to V_H, \varphi_E \colon E_G \to E_H)$. Additionally the following two conditions must hold:

1. $src_H \circ \varphi_E = \varphi_V \circ src_G$ and $tgt_H \circ \varphi_E = \varphi_V \circ tgt_G$
2. $\ell_{V_G} = \ell_{V_H} \circ \varphi_V$ and $\ell_{E_G} = \ell_{E_H} \circ \varphi_E$

The first condition demands that the shape of the graph G must be preserved, the second that the labeling of the graph G must be preserved. The maps φ_V and φ_E are called the *node* and *edge map* of φ. The indices V and E are omitted if possible.

By $img(\varphi)$ we denote the subgraph of H, that is formed by the node set $img(\varphi_V) \subseteq V_H$ and the edge set $img(\varphi_E) \subseteq E_H$. We call $img(\varphi) \subseteq H$ the *image* of φ. Accordingly, by $dom(\varphi)$ we denote the graph that is formed by $dom(\varphi_V) \subseteq V_G$ and $dom(\varphi_E) \subseteq E_G$. It is called the *domain* of φ.

A *partial graph homomorphism* $\gamma \colon G \nrightarrow H$ is a graph homomorphism with partial node and edge maps. In this case for $dom(\gamma) \subseteq G$ the condition $dom(\gamma) \neq G$ holds. Accordingly a graph homomorphism with total node and edge maps is called a *total* graph homomorphism. For a total graph homomorphism $\varphi \colon G \to H$ it is $dom(\varphi) = G$. By $\omega_{G,H}$ we denote the partial graph homomorphism from G to H with $dom(\omega_{G,H}) = img(\omega_{G,H}) = O$. It is called the *empty* partial graph homomorphism from G to H.

2.3 Subgraph Matching

Given two graphs G and H. The task of finding a graph homomorphism $\varphi \colon G \to H$, such that the node and edge maps of φ are both injective, is called *subgraph matching*, *graph pattern matching*, or the *subgraph isomorphism problem*. This problem is known to be NP-complete[1] [9]. The term of subgraph matching or graph pattern matching can also refer to the problem of finding a *non-injective* graph homomorphism. The optimization technique described here works in both cases.

[1]For fixed G the time complexity of subgraph matching is polynomial, i.e., $O(|H|^{|G|})$. But a runtime of, for example, $O(|H|^{10})$ with $|G| = 10$ is still not feasible.

In the context of subgraph matching G is called the *pattern graph* and H the *host graph*. A total graph homomorphism $\varphi\colon G \to H$ is called a *match*.

2.4 The Minimum Spanning Arborescence Problem

An *arborescence with root* v_0 is a subgraph $T \subseteq G$ such that there is a unique path in T from v_0 to v for all other nodes $v \in V_T \backslash \{v_0\}$. If $V_T = V_G$ holds, then T is called a *spanning* arborescence (SA for short). Let $cost\colon E_G \to \mathbb{R}_{\geq 0}$ a nonnegative cost function for the edges of G. Then a *minimum spanning arborescence* (MSA) is a spanning arborescence $T \subseteq G$ that has minimal total cost

$$\sum_{e \in E_T} cost(e).$$

An MSA can be found in polynomial time by an algorithm which has been independently proposed by Edmonds as well as by Chu and Liu [6, 3]. Gabow et al. proposed a variant of this algorithm, which only needs $O(|E_G| + |V_G| \cdot \log |V_G|)$ time [8]. An MSA can be understood as a kind of minimum spanning tree for directed graphs. However, the well known greedy algorithms of Prim and Kruskal as well as similar algorithms do not work in case of directed graphs.

2.5 Arithmetic and Geometric Mean

Given some quantities $X_1, \ldots, X_n$ which are summarized to $Y := X_1 + \cdots + X_n$ one wants to know, which quantity X leads to the same result

$$Y \;=\; \underbrace{X + \cdots + X}_{n\text{-times}}$$

if n-times added to itself. This question is answered by the *arithmetic mean*. The arithmetic mean yields an *average summand*. Accordingly it is defined as $X := \frac{1}{n} \sum_{i=1}^{n} X_i$.

Now given some factors $X_1, \ldots, X_n > 0$ and their product $Y := X_1 \cdots X_n$ one wants to know, which factor $X > 0$ leads to the same result

$$Y \;=\; \underbrace{X \cdots X}_{n\text{-times}}$$

if n-times *multiplied* with itself. This second question is answered by the *geometric mean*. The geometric mean yields an *average factor*. Accordingly it is defined as $X := (\prod_{i=1}^{n} X_i)^{1/n}$. The difference between the arithmetic and the geometric mean is explained very nicely in an online question corner of the University of Toronto [1], which also inspired the explanation here.

3 Matching Strategies

A matching strategy for a given pattern graph G is represented by a so called *search plan*, which is defined as a sequence of *primitive matching operations* $P = \langle o_1, \ldots o_k \rangle$. A primitive operation o_i represents the matching of a single node or edge of the pattern graph G to an appropriate node or edge of the host graph H. The whole search plan represents the stepwise construction of one (or all) matches $\varphi\colon G \to H$ starting from $\omega_{G,H}$. A partly constructed match, which is in fact a partial graph homomorphism $\gamma\colon G \rightharpoonup H$, is called a *candidate*.

3.1 Kinds of primitive Operations

There are two kinds of primitive matching operations: *lookup* operations and *extension* operations.

Definition 1 [Primitive Matching Operations] Let G a pattern graph and $v \in V_G$, $e \in E_G$. Then $\mathsf{lkp}(v)$ and $\mathsf{lkp}(e)$ are called *lookup operations* and $\mathsf{ext}(v, e)$ is called an *extension operation*. In case of an extension operation the condition $v \in inc_{E_G}(e)$ must hold. The lookup and extension operations possible for the pattern graph G form the set $Ops(G)$. The elements of $Ops(G)$ are called the *primitive matching operations* of G. □

A lookup operation represents the query for a host graph element which is not necessarily connected with the already matched subgraph $img(\gamma)$. More precisely a lookup operation $\mathsf{lkp}(x)$ with $x \in V_G \cup E_G$ represents the expansion of a candidate γ by *any* node (or edge) of the host graph which can be matched by the given pattern node (or edge) x. In case $x \in V_G$ this means that an appropriate node $w \in V_H$ of the host graph must fulfill $\ell_{V_H}(w) = \ell_{V_G}(x)$. In case $x \in E_G$ this means that an appropriate edge $f \in E_H$ of the host graph must fulfill $\ell_{E_H}(f) = \ell_{E_G}(x)$, $\ell_{V_H}(src_H(f)) = \ell_{V_G}(src_G(x))$, and $\ell_{V_H}(tgt_H(f)) = \ell_{V_G}(tgt_G(x))$. The element x needs not to be connected to $dom(\gamma)$. Hence, lookup operations enable the matching of non-connected pattern graphs.

An extension operation represents a local search coming from an already matched node. That means an operation $\mathsf{ext}(v, e)$ represents the expansion of a candidate γ along a given pattern edge e coming from a pattern node v with $v \in dom(\gamma)$. As $v \in inc_{E_G}(e)$ holds, an appropriate edge $f \in E_H$ must be incident to the image of v under the current candidate, i.e., $f \in inc_{V_H}(\gamma(v))$.

Of course the edge f must fulfill the same requirements as in case of an edge lookup $\mathsf{lkp}(e)$, i.e., $\ell_{E_H}(f) = \ell_{E_G}(e)$, $\ell_{V_H}(src_H(f)) = \ell_{V_G}(src_G(e))$ and $\ell_{V_H}(tgt_H(f)) = \ell_{V_G}(tgt_G(e))$ must hold. Furthermore f and e must have the same direction, which means that f must be incoming (or outgoing) at the host graph node $\gamma(v)$ if and only if e is incoming (or outgoing) at the pattern node v. If the other node incident to e (this is the node v' with $inc_{E_G}(e) = \{v, v'\}$)

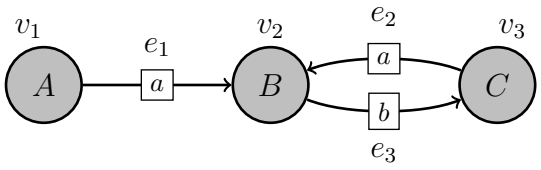

Figure 1: A simple pattern graph G_1.

is also already matched, i.e., $v' \in dom(\gamma)$, then f must additionally fulfill $inc_{E_H}(f) = \{\gamma(v), \gamma(v')\}$.

A pattern edge $e \in E_G$ cannot be matched without matching the two pattern nodes incident to e. More precisely, after the execution of an operation $\mathsf{lkp}(e)$ or $\mathsf{ext}(v, e)$ the condition $inc_{E_G}(e) \subseteq dom(\gamma)$ holds. Thus, the matching of a node $v \notin dom(\gamma)$ can not only happen explicitly by an operation $\mathsf{lkp}(v)$, but also implicitly by an operation $\mathsf{lkp}(e)$ or $\mathsf{ext}(v', e)$ with $inc_{E_G}(e) = \{v', v\}$.

If injective matching is required (see section 2.3), some extra conditions must be fulfilled. This is because an injective graph homomorphism must not match a host graph element twice. More precisely, a match $\varphi\colon G \to H$ with $x, y \in V_G \cup E_G$, $\varphi(x) = \varphi(y)$, and $x \neq y$ is not allowed. For lookup operations $\mathsf{lkp}(v)$ with $v \in V_G$ this means that an appropriate host graph node $w \in V_H$ must fulfill $w \notin img(\gamma)$ for a current candidate γ. Accordingly for operations $\mathsf{lkp}(e)$ and $\mathsf{ext}(v, e)$ an appropriate edge $f \in E_H$ must fulfill $f \notin img(\gamma)$. Additionally the condition $w' \neq w$ with $inc_{E_H}(f) = \{w, w'\}$ must be checked if $v \neq v'$ with $inc_{E_G}(e) = \{v, v'\}$. This ensures that e is matched to a loop edge f only if it is a loop edge itself.

3.2 Valid Search Plans

In general there are many possible search plans for a given pattern graph G. For example consider the simple pattern graph G_1 shown in figure 1. Amongst others for G_1 the following search plans are possible:

$$\begin{aligned} P_1 &= \langle \mathsf{lkp}(v_3), \mathsf{ext}(v_3, e_3), \mathsf{lkp}(v_1), \mathsf{ext}(v_2, e_2), \mathsf{ext}(v_2, e_1) \rangle \\ P_2 &= \langle \mathsf{lkp}(e_2), \mathsf{lkp}(v_1), \mathsf{ext}(v_2, e_1), \mathsf{lkp}(e_3) \rangle \end{aligned}$$

However, some operation sequences do not form admissible search plans. Consider, for example, the following invalid operation sequence:

$$\langle \mathsf{ext}(v_2, e1), \mathsf{lkp}(e_2), \mathsf{lkp}(v_3) \rangle$$

It contains three errors: Firstly, each valid search plan must start with a lookup operation. This is necessary, because an extension operation $\mathsf{ext}(v, e)$ requires that the node v has already been matched. However, this is not the case if no

matching has been performed yet. Secondly, the operation $\mathsf{lkp}(v_3)$ demands the matching of a node that has already been matched implicitly by the preceding operation $\mathsf{lkp}(e_2)$. This is not really wrong but needless. Thirdly, there is no operation present dealing with edge e_3. But a valid search plan must cover *all* nodes and edges of a pattern graph G.

To characterize the nodes and edges covered by a given operation sequence, the following two functions are defined: Firstly they are defined for single operations, where $v \in V_G$ and $e \in E_G$:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{nodes}[\![\mathsf{lkp}(v)]\!] & := & \{v\} & \mathit{edges}[\![\mathsf{lkp}(v)]\!] & := & \emptyset \\ \mathit{nodes}[\![\mathsf{lkp}(e)]\!] & := & \mathit{inc}_{E_G}(e) & \mathit{edges}[\![\mathsf{lkp}(e)]\!] & := & \{e\} \\ \mathit{nodes}[\![\mathsf{ext}(v,e)]\!] & := & \mathit{inc}_{E_G}(e) & \mathit{edges}[\![\mathsf{ext}(v,e)]\!] & := & \{e\} \end{array}$$

Then the definition is lifted to complete operation sequences:

$$\begin{aligned} \mathit{nodes}[\![\langle o_1, \ldots, o_k \rangle]\!] &:= \bigcup_{j=1}^{k} \mathit{nodes}[\![o_j]\!] \\ \mathit{edges}[\![\langle o_1, \ldots, o_k \rangle]\!] &:= \bigcup_{j=1}^{k} \mathit{edges}[\![o_j]\!] \end{aligned}$$

Now the notion of a valid search plan can be defined in terms of the above *nodes*- and *edges*-functions. In this way it gets apparent, that one can determine statically[2] whether a given operation sequence forms a valid search plan or not.

Definition 2 [Valid Search Plans] Given a pattern graph G and a sequence of primitive matching operations $P = \langle o_1, \ldots o_k \rangle$ with $o_i \in \mathit{Ops}(G)$ for $1 \leq i \leq k$. If the conditions

1. If $o_i = \mathsf{ext}(v, e)$, then node v has already been matched and edge e has not. I.e., $v \in \mathit{nodes}[\![\langle o_1, \ldots, o_{i-1} \rangle]\!]$ and $e \notin \mathit{edges}[\![\langle o_1, \ldots, o_{i-1} \rangle]\!]$.

2. There are no needless lookup operations present. I.e., if $o_i = \mathsf{lkp}(x)$ then $x \notin \mathit{nodes}[\![\langle o_1, \ldots, o_{i-1} \rangle]\!] \cup \mathit{edges}[\![\langle o_1, \ldots, o_{i-1} \rangle]\!]$.

3. The whole pattern graph is covered. I.e., $\mathit{nodes}[\![P]\!] = V_G$ and $\mathit{edges}[\![P]\!] = E_G$.

hold, then P is called a *(valid) search plan* for the pattern graph G. The set of all valid search plans possible for G is denoted by $\mathit{Plans}(G)$. □

Condition 1 implies, that each valid search plans must begin with a lookup operation.

[2]This means it can be decided whether a search plan is valid without executing it.

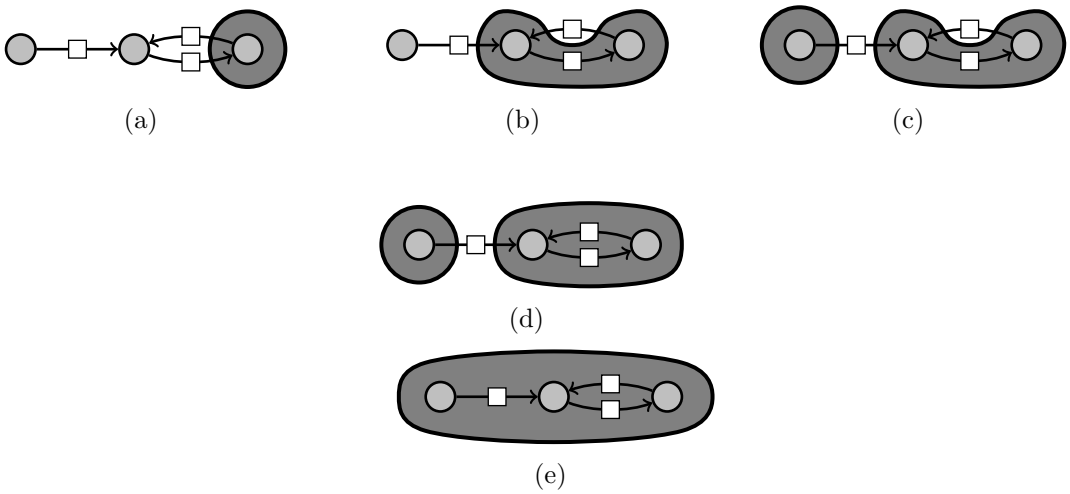

Figure 2: On the execution of the search plan P_1 for the pattern graph G_1 the subgraph $dom(\gamma) \subseteq G_1$ grows step by step until G_1 is covered completely.

3.3 Execution of Search Plans

The execution of a search plan $P = \langle o_1, \dots o_k \rangle$ consists in consecutively performing the primitive matching operations $o_1, \dots, o_k$. While P is executed, the subgraph $dom(\gamma) \subseteq G$ is growing step by step. This is depicted in figure 2 for the example pattern graph G_1 (see figure 1) and the example search plan P_1.

If an operation o_i is executed, then it might be possible to expand a candidate γ in multiple ways. If a lookup operation is performed (i.e., $o_i = \mathsf{lkp}(x)$), a new candidate can be created for *any* appropriate host graph node (or edge). If an extension operation is performed (i.e., $o_i = \mathsf{ext}(v, e)$), a new candidate can be created for each appropriate host graph edge that is *incident* to $\gamma(v)$. If one of these two situation occurs, then we say that the candidate γ *splits* into several new candidates.

Consider the example pattern graph G_1 shown in figure 1 and the example host graph H_1 shown in figure 3. Obviously H_1 contains exactly one occurrence of G_1. Assume that the operation $\mathsf{lkp}(v_1)$ is performed with the empty partial graph morphism $\gamma := \omega_{G_1,H_1}$ as current candidate. Since H_1 contains two nodes with label A, the candidate splits into two new candidates each matching a single node (see figure 4(a)).

In the following, equally labeled edges which connect equally labeled nodes in equal direction are referred to as *isomorphic*[3]. Now, if the operation $\mathsf{ext}(v_1, e_1)$ is performed for a candidate γ with $dom(\gamma_V) = \{v_1\}$, $dom(\gamma_E) = \emptyset$, and $\gamma(v_1) = w_1$, then four new candidates can be created. This is because there are four edges incident to the node w_1 which are appropriate for the edge e_1 (see figure 4(b)).

[3]This term has been adopted from Dörr [5].

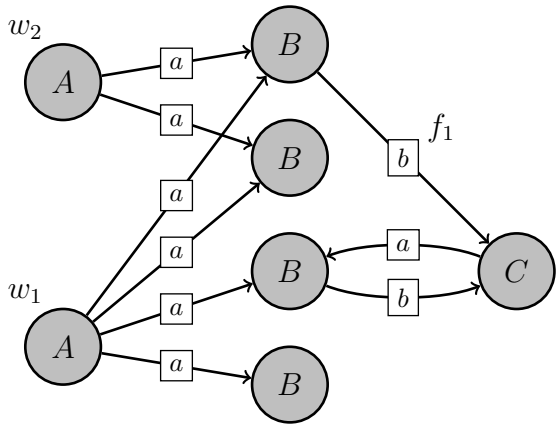

Figure 3: A simple host graph H_1.

If a whole search plan $P = \langle o_1, \dots o_k \rangle$ is executed, each primitive operation o_i may cause splitting of candidates. To find all (or one) matches $\varphi\colon G \to H$, it may be necessary to process all candidates. However, one does not need to do this all at once; that is, all candidates can be treated one after another. More precisely, if a candidate splits in several new candidates, the execution of P can be continued only for one of these candidates. In this case the other candidates can be treated later, which can be carried out by backtracking.

4 Time Complexity

The runtime needed for the execution of a search plan depends on host graph H. However, for a fixed host graph H the runtime may vary significantly for the different search plans.

4.1 The Runtime of a Primitive Matching Operation

The runtime raised by a search plan depends—among other things—on the runtime needed by the primitive matching operations.

The execution of lookup operations can be accelerated by storing all nodes and edges in different lists according to their label. If there is exactly one list for each node or edge label $\xi \in \Sigma_V \cup \Sigma_E$, which contains all nodes (or edges) with label ξ, then a node (or edge) with label ξ can be provided directly by simply accessing the according list. In this case the execution of an operation $\mathsf{lkp}(x)$ takes only $O(1)$ time per matched host graph element.

If all incident edges of a node are stored in a list associated with that node, then the execution of an operation $\mathsf{ext}(v, e)$ only requires the scanning of those edges of H, that are incident to the node $\gamma(v)$. In the worst case this takes still

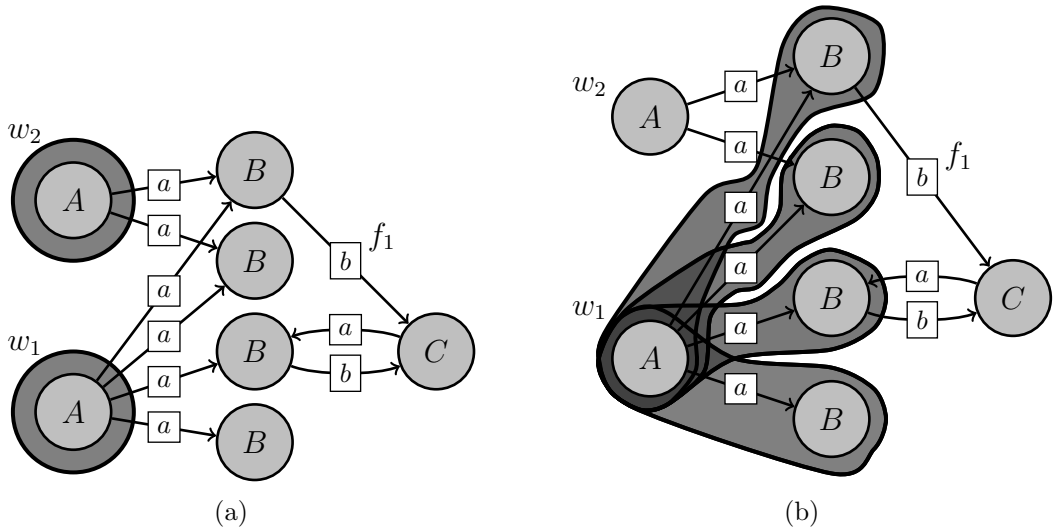

Figure 4: Splitting of candidates in the host graph H_1. The highlighted subgraphs $img(\gamma) \subseteq H_1$ denote the images of candidates γ created by splitting.

$O(|E_H|)$ time. But for host graphs being sparse in the sense, that every node has $O(1)$ incident edges, the execution of an operation $\mathsf{ext}(v, e)$ takes $O(1)$ time per matched edge this way.

Imagine, for example, a host graph node with many incoming and few outgoing edges. If the edges of that node are scanned while looking for an *outgoing* edge, then most of the edges are unnecessarily processed. So, if the incoming and outgoing edges of a node are stored in two *separate* lists, even more needless work can be avoided.

4.2 The Runtime of a Search Plan

On the execution of a search plan $P = \langle o_1, \ldots, o_k \rangle$ splitting of candidates may occur for any operation o_i. The number of candidates which can be newly created from a previous candidate γ is called the *splitting factor* of o_i for γ:

- If $o_i = \mathsf{lkp}(x)$ with $x \in V_G \cup E_G$, the splitting factor equals the number of elements that are appropriate for the pattern element x.
- If $o_i = \mathsf{ext}(v, e)$, the splitting factor is the number of isomorphic edges incident to $\gamma(v)$, that are appropriate for the pattern edge e.

As some elements might lead to non-injective mapping, the splitting factors may be smaller in case of injective matching.

If splitting occurs for a significant ratio of the operations and candidates, this leads to an exponential growth of the set of possible candidates. Even if

every node or edge can be matched in $O(1)$ time (see section 4.1), this leads to a runtime of $O(|G| \cdot s_{\max}^{|G|})$ where $s_{\max}$ is the maximal occurring splitting factor. If the matching of edges needs $O(|E_H|)$ time each, which is the worst case, the matching process requires even $O(|G| \cdot s_{\max}^{|G|+1})$ time. However, if no splitting arises (i.e., $s_{\max} = 1$) and all nodes and edges can be matched in $O(1)$ time, the time complexity is linear, that is $O(|G|)$. If splitting can be avoided, but all edges require $O(|E_H|)$ time each, then the time complexity is $O(|E_H| \cdot |G|)$.

Again consider the example graphs G_1 and H_1 shown in the figures 1 and 3. Assume that the following search plan is executed:

$$P_3 = \langle \mathsf{lkp}(v_1), \mathsf{ext}(v_1, e_1), \mathsf{lkp}(v_3), \mathsf{ext}(v_3, e_3), \mathsf{ext}(v_2, e_2) \rangle.$$

This search plan causes an intense splitting of candidates: Starting with the empty candidate $\gamma = \omega_{G_1,H_1}$ the execution of the first operation $\mathsf{lkp}(v_1)$ causes a splitting of γ with factor two. So, two new candidates γ_1 and γ_2 are created, each matching v_1 to w_1 or w_2, respectively. Now the operation $\mathsf{ext}(v_1, e_1)$ is performed: In case of γ_1 this leads to a splitting with factor four, in case of γ_2 to a splitting with factor two. Altogether there are six candidates. Then the execution of the fourth operation $\mathsf{ext}(v_3, e_3)$ reduces the number of possible candidates to two. After execution of the last operation only one match remains. Now assume that another search plan is performed:

$$P_4 = \langle \mathsf{lkp}(v_3), \mathsf{ext}(v_3, e_2), \mathsf{ext}(v_2, e_1), \mathsf{ext}(v_2, e_3) \rangle$$

Search plan P_4 in contrast to search plan P_3 causes no splitting at all. Furthermore all scanned lists of incoming or outgoing edges all contain one or two edges. Altogether the execution of P_4 requires only 5 steps. The execution of P_3 requires 11 steps by contrast.

In case of the lookup operation $\mathsf{lkp}(v_1)$ splitting arises, because there is more than one node with label A present in H. But in case of the extension operations the crucial point is, that P_4 follows the edges e_3 and e_1 in the *opposite direction* as P_3 does. Obviously, the direction an edge is followed can determine, whether a candidate splits or not.

5 A Cost Model

Under the assumption that all graph elements can be matched in time $O(1)$ (see section 4.1) subgraph matching in linear time is identical to the avoidance of splitting (see section 4.2).

In case of lookup operations this means, that only nodes and edges with seldom labels should be matched in this way. In case of extension operations that splitting should be avoided by choosing the right direction (see section 4.2). However, this is not always possible. In some host graphs all labels might appear

similar often. Or a host graph edge might have a bunch of isomorphic edges on *both* incident nodes. This means that splitting will occur *always*. If this is the case, one can only choose a node or edge with a *comparatively* seldom label or—in case of extension operations—the direction which causes *less* splitting.

To achieve the generation of search plans with a low *overall* amount of splitting a cost model is defined which assigns a cost to all possible search plans (section 5.3). However, the splitting caused by a search plan $P = \langle o_1, \ldots, o_k \rangle$ depends on the splitting caused by the primitive operations o_i. So, costs are assigned to the primitive operations first (section 5.1 and 5.2).

5.1 The Cost of a Lookup Operation

In case of non-injective matching the splitting factor s of an operation $o = \mathsf{lkp}(x)$ for a current candidate γ is equal to the number of appropriate host graph elements. In case of injective matching some host graph elements may be dropped. So, with the sets

$$\begin{aligned} V_\zeta &:= \{v \in V_H \mid \ell_{V_H}(v) = \zeta\} \\ E_\sigma &:= \{e \in E_H \mid \ell_{E_H}(e) = \sigma\} \end{aligned}$$

which gather all graph elements with a certain label $\zeta \in \Sigma_V$ or $\sigma \in \Sigma_E$ respectively the condition

$$s \leq \begin{cases} \left|V_{\ell_{V_G}(x)}\right| & \text{if } x \in V_G \\ \left|E_{\ell_{E_G}(x)}\right| & \text{if } x \in E_G \end{cases}$$

holds. So, a lookup operation $\mathsf{lkp}(x)$ gets the cost

$$c(\mathsf{lkp}(x)) := \max\left\{1, \left|X_{\ell_{X_G}(x)}\right|\right\}$$

assigned, where $X = V$ for $x \in V_G$ or $X = E$ for $x \in E_G$, respectively. This cost is independent from a current candidate γ and it is $s \leq c(\mathsf{lkp}(x))$ for *all* possible candidates γ.

5.2 The Cost of an Extension Operation

In the following we refer to all bunches of isomorphic edges that have the same direction and that are incident to the same node, as *instances* of *V-structures*. This useful terminology has originally been invented by Dörr [5].

Definition 3 [V-Structures and their Instances] Given a host graph H. Then a 4-tupel $vs = (\zeta, \sigma, \xi, d) \in \Sigma_V \times \Sigma_E \times \Sigma_V \times \{\mathsf{in}, \mathsf{out}\}$ is called a *V-structure*. Now consider a subgraph $I \subseteq H$ consisting of a node $w \in V_H$ and n nodes $w_1, \ldots, w_n \in V_H$ as well as n edges $f_1, \ldots, f_n \in E_H$ with $inc_{E_H}(f_i) = \{w, w_i\}$ for $1 \leq i \leq n$. If the conditions

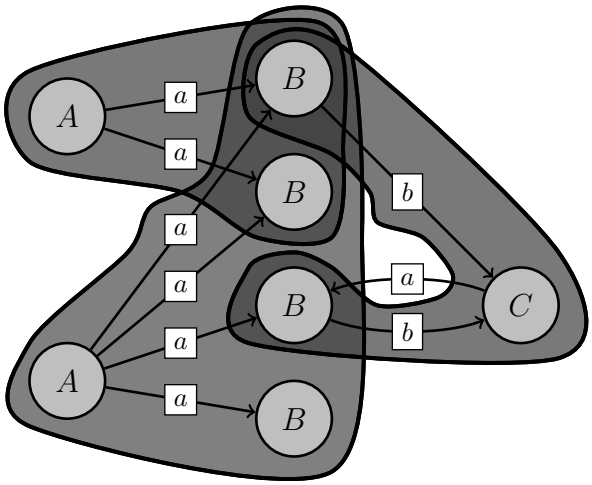

Figure 5: The host graph H_1 from figure 3 with highlighted V-instances.

1. $\ell_{V_H}(w) = \zeta$
2. $\ell_{E_H}(f_i) = \sigma$ for $1 \leq i \leq n$
3. $\ell_{V_H}(w_i) = \xi$ for $1 \leq i \leq n$
4. $d = \mathsf{out}$ implies $src_{E_H}(f_i) = w$ and $d = \mathsf{in}$ implies $tgt_{E_H}(f_i) = w$ for $1 \leq i \leq n$

hold, then I is called an *n-fold instance* of vs with *root node* w. □

In this paper instances of V-Structures are often referred to as *V-Instances* for short. Additionally we also say that an n-fold V-instance has *multiplicity* n. Figure 5 shows the host graph H_1 from figure 3 with highlighted V-instances. There are three V-instances present: A fourfold and a twofold instance of the V-structure (A, a, B, out) and a twofold instance of the V-structure (C, b, B, in).

If an operation $\mathsf{ext}(v, e)$ is performed, the V-instances present in a host graph can cause the splitting of one or more candidates. However, this can only happen if the operation and a V-instance apply to each other:

Definition 4 [Critical V-Structures] Given a V-structure $vs = (\zeta, \sigma, \xi, d)$ and an extension operation $o = \mathsf{ext}(v, e)$. If the conditions

1. $\ell_{V_G}(v) = \zeta$
2. $\ell_{E_G}(e) = \sigma$
3. $\ell_{V_G}(v') = \xi$ with $inc_{E_G}(e) = \{v, v'\}$

4. $d = \mathsf{out}$ implies $src_G(e) = v$ and $d = \mathsf{in}$ implies $tgt_G(e) = v$

hold, then vs is called *critical* for o. For a lookup operation a V-structures is never critical. □

To extension operations $\mathsf{ext}(v, e)$ we assign an *average* splitting factor as cost: Let $vs = (\zeta, \sigma, \xi, d)$ a V-structure, such that vs is critical for $o = \mathsf{ext}(v, e)$. Assume H contains exactly m instances $I_1, \ldots, I_m \subseteq H$ of vs with according multiplicities $n_1, \ldots, n_m$. Then $\mathsf{ext}(v, e)$ gets the cost

$$c(\mathsf{ext}(v, e)) := \max \left\{ 1, \left(\prod_{j=1}^{m} n_j \right)^{1/|V_\zeta|} \right\}$$

assigned. In this way the multiplicities of the instances of vs, which are present in H, are multiplicatively accumulated. Thereafter the result is normalized by the number of potential root nodes by extracting the $|V_\zeta|$-th root. Hence, the cost of an extension operation is determined according to the geometric mean (see section 2.5). The idea behind this is that the multiplicity of a V-instance potentially causes a *growth* of the candidate set during the matching process.

5.3 The Cost of a Search Plan

Having defined the costs for the primitive operations in $Ops(G)$ it is possible to define the cost $c(P)$ for a whole search plan $P = \langle o_1, \ldots, o_k \rangle$. This is done by the following formula:

$$c(P) := c_1 + c_1c_2 + c_1c_2c_3 + \cdots + c_1c_2c_3 \cdots c_k \tag{1}$$

with $c_i := c(o_i)$. With the assumption that each operation takes $O(1)$ time per matched host graph element (see section 4.1) the essential idea of this formula is to overestimate the average[4] runtime needed by the execution of P.

To understand this, consider what happens on the execution of P: The operation o_1 is a lookup operation. Moreover, before o_1 is performed, no element of the host graph is matched. So the execution of o_1 causes the matching of $O(c_1)$ nodes or edges (in case o_1 is an edge lookup there are also the implicitly matched nodes, but such nodes only cause a constant additional complexity for each edge). Furthermore, the initial empty candidate $\gamma = \omega_{G,H}$ splits into up to c_1 new candidates.

Now for up to c_1 candidates the operation o_2 is performed. If o_2 is a lookup operation, at most $O(c_1c_2)$ host graph elements are matched and the set of candidates grows to a number of up to c_1c_2. If o_2 is an extension operation, essentially the same happens. The only difference is that c_2 is no more an upper bound for

[4] Average in the sense of the geometric mean.

the splitting factor, but an *average* splitting factor in the sense of the geometric mean (see section 5.2). Furthermore, if no appropriate host graph edge is present for a candidate γ, then γ is removed from the candidate set. This means that c_2 *overestimates* the actual average splitting factor.

So, continuing this, one gets $O(c(P))$ as an overestimation of the average number of host graph elements matched on an execution of P—that, at least, is the idea. However, this is only a heuristics. Nevertheless, if $c_i = 1$ holds for an $i \in \{1, \ldots, k\}$, we can be sure that the operation o_i causes no splitting for any candidate γ. So, if $c_i = 1$ holds for *every* $i \in \{1, \ldots, k\}$ (which is equivalent to $c(P) = k$), we know that P causes no splitting at all. In this case an execution of P raises a runtime of $O(|G| \cdot |E_H|)$. If all nodes of H have $O(1)$ incident edges, the runtime is even linear, that is $O(|G|)$ (see section 4.2).

6 Analysis of the Host Graph

The assignment of costs to the primitive matching operations in $Ops(G)$ requires knowledge about the host graph H. On the one hand this may arise from the application domain. In this case it is not necessary to consider the present host graph. On the other hand, if the structure of the host graphs is not a priori known, an analysis of the host graph is required.

The analysis of the host graph has two objectives: Firstly, for each label $\xi \in \Sigma_V \cup \Sigma_E$ the number of graph elements with label ξ must be determined. Secondly, the multiplicities of all present instances must be accumulated for all V-structures $vs \in \Sigma_V \times \Sigma_E \times \Sigma_V \times \{\mathsf{in}, \mathsf{out}\}$ (see section 5.2).

Algorithm 5 [V-Structure Analysis] Given a host graph H. Then the following algorithm computes the values $p_\xi := |V_\xi|$ and $q_\sigma := |E_\sigma|$ for all $\xi \in \Sigma_V$ and $\sigma \in \Sigma_E$ as well as the product

$$r_{vs} := \prod_{j=1}^{m_{vs}} n_j^{vs}$$

of the multiplicities $n_1^{vs}, \ldots, n_{m_{vs}}^{vs}$ of all present instances for each V-structure $vs \in \Sigma_V \times \Sigma_E \times \Sigma_V \times \{\mathsf{in}, \mathsf{out}\}$.

A1 *Count all nodes with a certain label*
For each $\xi \in \Sigma_V$ initialize a counter $p_\xi := 0$. Then iterate over all nodes $w \in V_H$ and increment the according counter $p_{\ell_{V_H}(w)}$ for each w.

A2 *Count all edges with a certain label*
For each $\sigma \in \Sigma_E$ initialize a counter $q_\sigma := 0$. Then iterate over all edges $f \in V_H$ and increment the according counter $q_{\ell_{E_H}(f)}$ for each f.

A3 *Accumulate the multiplicities of all present V-instances*
For each $vs = (\zeta, \sigma, \xi, d) \in \Sigma_V \times \Sigma_E \times \Sigma_V \times \{\mathsf{in}, \mathsf{out}\}$ initialize variable $r_{vs} := 1$ and a counter $u_{\sigma,\xi,d} := 0$. Now for all nodes $w \in V_H$ perform the steps (a) and (b):

(a) Consider all incident edges $f \in inc_{V_H}(w)$. Let $\xi_f := \ell_{V_H}(w')$, where w' is the node at the other end of f, that is $inc_{E_H}(f) = \{w, w'\}$. Furthermore let $\sigma_f := \ell_{E_H}(f)$ and d_f the direction of f, which means that $d_f = \mathsf{in}$ if f is incoming on w and $d_f = \mathsf{out}$ if f is outgoing on w. Now increment the counter u_{σ_f,ξ_f,d_f}.

(b) Consider all incident edges $f \in inc_{V_H}(w)$ again with ξ_f, σ_f, and d_f as in step (a). Now set

$$r_{(\ell_{V_H}(w),\sigma_f,\xi_f,d_f)} := r_{(\ell_{V_H}(w),\sigma_f,\xi_f,d_f)} \cdot \begin{cases} u_{\sigma_f,\xi_f,d_f} & \text{if } u_{\sigma_f,\xi_f,d_f} \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

and directly after that $u_{\sigma_f,\xi_f,d_f} := 0$. The latter ensures that the multiplicity of each present V-instance is included only once. □

Step **A1** and **A2** need together $O(|H| + |\Sigma_V| + |\Sigma_E|)$ time. The time complexity of step **A3** is $O(|H| + |\Sigma_E| \cdot |\Sigma_V|^2 + |\Sigma_E|)$. If $|\Sigma_V|, |\Sigma_E| \in O(1)$ holds, the analysis of H needs altogether $O(|H|)$. If the p-, q-, and r-values are stored in a table, this requires $O(|\Sigma_E| \cdot |\Sigma_V|^2 + |\Sigma_E|)$ memory.

7 Heuristic Optimization

Since the runtime of the subgraph matching for fixed G and H can vary significantly for different search plans (see section 4.2), the key idea of fast search plan driven subgraph matching is to find a preferably good search plan. However, as subgraph matching is an NP-complete problem a feasible search plan does not always exist[5].

According to the cost model defined in section 5 we are looking for a search plan P with minimal cost $c(P)$, that is for a $P \in Plans(G)$ with

$$c(P) \;=\; \min_{P' \in Plans(G)} \big(c(P')\big).$$

As we do not know an algorithm, which solves this optimization problem efficiently, we use a heuristics which consists of two consecutive steps: the *operation selection* and the *operation ordering*[6]. The resulting search plan is not necessary optimal of course.

[5]Under the assumption that P ≠ NP holds.

[6]These can be compared with the tasks of instruction selection and instruction ordering performed in compiler backends.

In the first step, namely the operation selection, the largest term of $c(P)$ is minimized. This is the product $c_1c_2c_3\cdots c_k$ (see formula (1) in section 5.3). As this product includes the costs of *all* operations occurring in P, this corresponds to the selection of a cheapest set $S \subseteq \mathit{Ops}(G)$ such that the operations in S can form a valid search plan for G. This kind of subset is characterized by the following definition:

Definition 6 [Admissible Operation Selection] A subset $S \subseteq \mathit{Ops}(G)$ is called an *admissible operation selection* for G if a search plan $P = \langle o_1, \ldots, o_k\rangle \in \mathit{Plans}(G)$ exists with $S = \{o_1, \ldots, o_k\}$. The set of all admissible operation selections for a pattern graph G is denoted as $\mathit{Sel}(G)$. □

Let $c(S) := \prod_{o\in S} c(o)$ for $S \in \mathit{Sel}(G)$. Then the task of operation selection is to find an $S \in \mathit{Sel}(G)$ with

$$c(S) \;=\; \min_{S'\in \mathit{Sel}(G)} \big(c(S')\big). \tag{2}$$

As the set $\mathit{Sel}(G)$ has a quite complicated structure this is not a trivial task.

In the second step, namely the operation ordering, an order for the selected operations is computed, such that the cheaper operations occur preferably early and the more expensive operations preferably late. This is done because of the fact, that a cost has more impact on the value $c(P)$ the earlier the corresponding operation occurs in P (see formula (1) in section 5.3). So, expensive operations should be executed as late as possible.

7.1 The Plan Graph

The tasks of operation selection and operation ordering are solved by means of a so called *plan graph* $\tilde{G}$. The plan graph is a directed labeled graph $\tilde{G}$ which is generated from the pattern graph G in a specific way. The node labels of $\tilde{G}$ are taken from the set $V_G \cup E_G \cup \{\bot\}$ instead of Σ_V. This is because $\tilde{G}$ reflects the structure of the according pattern graph G. The edge labels are also taken from a set different from Σ_E, namely the set $\{\circledcirc, \otimes, \circledast, \bot\}$. The symbols $\circledcirc$, $\otimes$, and $\circledast$ represent different kinds of primitive matching operations. The symbol $\bot$ indicates edges which are only present for technical reason. Each edge has a cost given by the cost function $\mathit{cost}\colon E_{\tilde{G}} \to \mathbb{R}_{\geq 1}$.

Each node of $\tilde{G}$ (except for a special root node) represents an element of the pattern graph G and has the according element as node label. Each edge of $\tilde{G}$ (apart from edges labeled with $\bot$) represents a primitive matching operation $o \in \mathit{Ops}(G)$. In fact there is exactly one edge in $\tilde{G}$ for every operation in $\mathit{Ops}(G)$. The edges which come from the root node are labeled with $\circledast$ and represent the lookup operations. The other edges (except those labeled with $\bot$) represent extension operations: The ones labeled with $\circledcirc$ represent extension operations $\mathsf{ext}(v, e)$ that follow e *with* its direction. The ones labeled with $\otimes$ represent extension

operations $\mathsf{ext}(v, e)$ that follow e *against* its direction. The costs assigned to the edges represent the costs of the possible primitive operations according to the cost model defined in section 5.

Definition 7 [Plan Graph] Given a pattern graph G. Then the according *plan graph* $\tilde{G}$ is defined as follows:

1. $\tilde{G}$ has a root node $w_{\text{root}} \in V_{\tilde{G}}$ with label $\ell_{V_{\tilde{G}}}(w_{\text{root}}) = \perp$.
2. For each node or edge $x \in V_G \cup E_G$ the plan graph has a node $\tilde{x} \in V_{\tilde{G}}$ with label $\ell_{V_{\tilde{G}}}(\tilde{x}) = x$.
3. For each $x \in V_G \cup E_G$ there is an edge $f \in E_{\tilde{G}}$ from w_{root} to $\tilde{x}$ with label $\circledast$.
4. For each edge $e \in E_G$ with $v_s := src_G(e)$ and $v_t := tgt_G(e)$ the plan graph has four further edges:
   - An edge f_1 from $\tilde{v}_s$ to $\tilde{e}$ with label $\circledcirc$.
   - An edge f_2 from $\tilde{v}_t$ to $\tilde{e}$ with label $\otimes$.
   - Two edges f_3, f_4 both originating in $\tilde{e}$ and labeled with $\perp$. Furthermore $tgt_{\tilde{G}}(f_3) = \tilde{v}_s$ and $tgt_{\tilde{G}}(f_4) = \tilde{v}_t$ holds.
5. Let $f \in E_{\tilde{G}}$. Then the cost function is defined as

$$cost(f) := \begin{cases} c(\mathsf{lkp}(x)) & \text{if } \ell_{E_{\tilde{G}}}(f) = \circledast \text{ and } x = \ell_{V_{\tilde{G}}}(w) \\ c(\mathsf{ext}(v, e)) & \text{if } \ell_{E_{\tilde{G}}}(f) \in \{\circledcirc, \otimes\} \\ 1 & \text{otherwise} \end{cases}$$

   with $w := tgt_{\tilde{G}}(f)$, $v := \ell_{V_{\tilde{G}}}(src_{\tilde{G}}(f))$, and $e := \ell_{V_{\tilde{G}}}(w)$. The costs of the operations are as defined in section 5.1 and 5.2 □

Figure 6 shows the plan graph $\tilde{G}_1$ which belongs to the pattern graph G_1 from figure 1. The costs annotated to the edges of $\tilde{G}_1$ arise from the multiplicities of the V-instances present in the host graph H_1 (see figure 3). Edge f_1 represents the operation $\mathsf{lkp}(v_1)$. As there are two nodes with label A present in H_1, edge f_1 gets the cost 2. Edge f_2 represents the operation $\mathsf{ext}(v_1, e_1)$. So, f_2 is labeled with the symbol $\circledcirc$. As there are two instances of the V-Structure (A, a, B, out) and two nodes with the label A present in H_1, f_2 gets the cost $\sqrt{2 \cdot 4} \approx 2.83$ assigned. Finally there are seven edges with label A present in H_1. So, the cost of f_3 is 7.

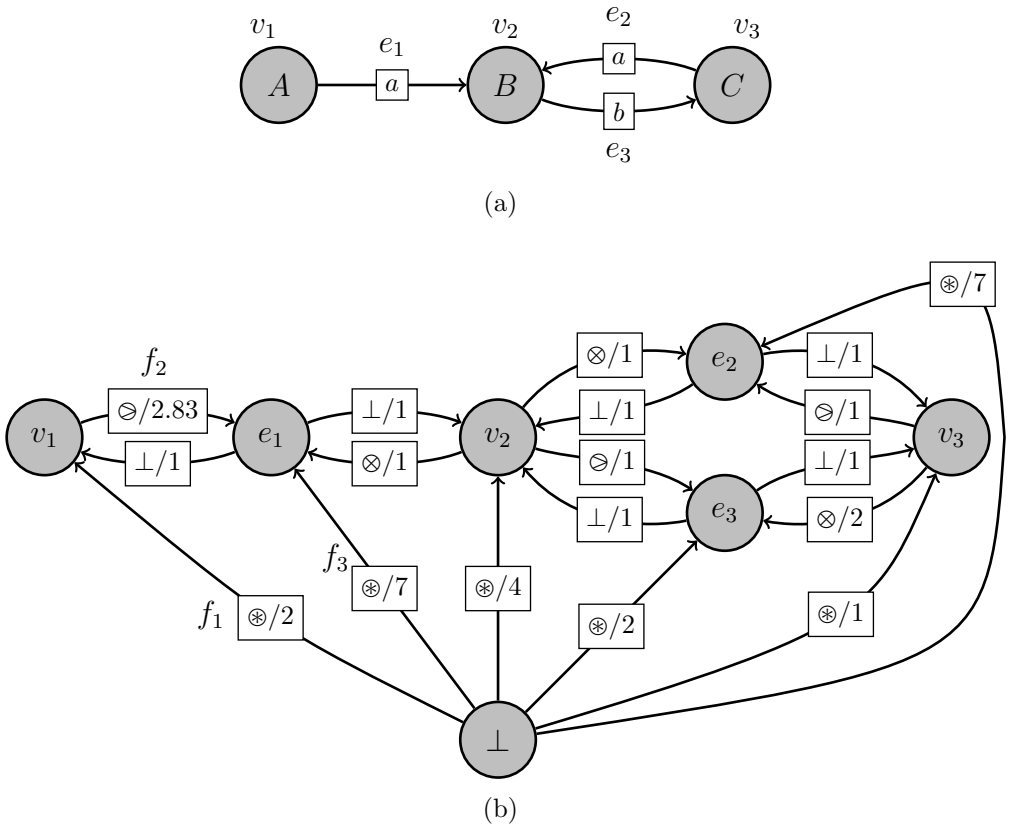

Figure 6: Subfigure (a) shows the pattern graph G_1 from figure 1. Subfigure (b) shows the according plan graph $\tilde{G}_1$.

7.2 Operation Selection

Each edge of $\tilde{G}$ which has a label other than $\perp$ corresponds to an operation in $Ops(G)$. Let $f \in E_{\tilde{G}}$. Then the function $op\colon E_{\tilde{G}} \to Ops(G) \cup \{\perp\}$ defined by

$$op(f) = \begin{cases} \mathsf{lkp}(x) & \text{if } \ell_{E_{\tilde{G}}}(f) = \circledast \\ \mathsf{ext}(v, e) & \text{if } \ell_{E_{\tilde{G}}}(f) \in \{\circledcirc, \otimes\} \\ \perp & \text{otherwise} \end{cases}$$

with $x := e := \ell_{V_{\tilde{G}}}(tgt_{\tilde{G}}(f))$ and $v := \ell_{V_{\tilde{G}}}(src_{\tilde{G}}(f))$ is a function which yields the primitive matching operation corresponding to an edge f of the plan graph. Restricted to the set of edges with a label other than $\perp$ the function op is even injective.

The problem of operation selection is to find an admissible operation selection $S \in Sel(G)$ that is also minimal in the sense of equation (2). The finding of an admissible (but not necessary minimal) selection is addressed by the following two lemmas, which relate the finding of an admissible selection to the finding of a spanning (but not necessary minimum) arborescence of $\tilde{G}$.

Lemma 8 [From SAs to Admissible Operation Selections] Given a pattern graph G and a spanning arborescence $T \subseteq \tilde{G}$ with root w_{root}. Then the set $S := op(E_T)\backslash\{\perp\}$ of primitive matching operations is an admissible operation selection for G. That is $S \in Sel(G)$ holds.

Proof. The condition $S \in Sel(G)$ holds if and only if there is a valid search plan $P = \langle o_1, \ldots, o_k \rangle \in Plans(G)$ with $\{o_1, \ldots o_k\} = S$. However, an appropriate P can be constructed from E_T as follows: Firstly, compute a traversal $f_1, \ldots, f_{k'}$ of T with root w_{root} (see section 2.1). Secondly, read the traversal from left to right and for each edge f_i with $\ell_{E_T}(f_i) \neq \perp$ emit the operation $op(f_i)$. If $\ell_{E_T}(f_i) = \perp$ holds, emit nothing. Thirdly, consider the operation sequence $P' = \langle o'_1, \ldots, o'_k \rangle$ with $k \leq k'$ constructed that way. If there are two operations $o'_i \in \{\mathsf{lkp}(e), \mathsf{ext}(v, e)\}$ and $o'_j = \mathsf{lkp}(v')$ in P' with $inc_{E_G}(e) = \{v, v'\}$ and $i < j$, then exchange the positions of o'_i and o'_j. Do this until no such operations are present anymore. The resulting operation sequence $P = \langle o_1, \ldots, o_k \rangle$ is a valid search plan, that is $P \in Plans(G)$. The proposition $\{o_1, \ldots, o_k\} = S$ holds by construction. □

The above lemma means that every spanning arborescence of the plan graph $\tilde{G}$ corresponds to an admissible operation selection for the pattern graph G. The following lemma represents the inverted statement that for every admissible operation selection there is an according spanning arborescence.

Lemma 9 [From Admissible Operation Selections to SAs] Given a pattern graph G and an admissible operation selection $S \in Sel(G)$. Then there exists a spanning arborescence $T \subseteq \tilde{G}$ such that $op(E_T)\backslash\{\perp\} = S$.

Proof. An appropriate spanning arborescence can be constructed as follows: For each operation $o \in S$ choose the corresponding edge $f_o \in E_{\tilde{G}}$ with $op(f_o) = o$. Having done this, consider all extension operations $o = \mathsf{ext}(v, e) \in S$. Let $f'_o \in E_{\tilde{G}}$ the unique edge with $\ell_{E_{\tilde{G}}}(f'_o) = \perp$, $src_{\tilde{G}}(f'_o) = tgt_{\tilde{G}}(f_o)$, and $tgt_{\tilde{G}}(f'_o) \neq src_{\tilde{G}}(f_o)$ each. If there is no edge $f \in E_{\tilde{G}}$ with $tgt_{\tilde{G}}(f) = tgt_{\tilde{G}}(f'_o)$ chosen yet, then also choose f'_o. Now let E the set of chosen edges. Then the subgraph $T \subseteq \tilde{G}$ that is induced by the edge set E is a spanning arborescence of $\tilde{G}$ with root w_{root}. Furthermore $op(E_T)\backslash\{\perp\} = S$ holds. □

According to lemma 8 and 9 the finding of an admissible operation selection is the same as the computation of a spanning arborescence. However, the operation selections found this way are not necessary minimal. For this reason a *minimum* spanning arborescence is computed. Thereby the only problem is, that the cost of an MSA T is computed by summation and the cost of an operation selection $S \in Sel(G)$ by a product. However, this can be settled easily: The log function is a strictly monotonic increasing transformation between $(\mathbb{R}_{\geq 1}, \cdot)$ and $(\mathbb{R}_{\geq 0}, +)$. Furthermore $\log(ab) = \log a + \log b$ for $a, b \in \mathbb{R}_{\geq 1}$ and $c(o) \geq 1$ for $o \in Ops(G)$ holds. Thus, if we compute an MSA of $T_{\min} \subseteq \tilde{G}$ with cost function $cost' \colon E_{\tilde{G}} \to$

$\mathbb{R}_{\geq 0}$ defined by $cost'(f) := \log(cost(f))$, then $T_{\min}$ is also an MSA of $\tilde{G}$ with cost function *cost* according to a total cost that is defined by the product

$$\prod_{f \in E_T} cost(f).$$

Such an MSA corresponds to a minimal admissible operation selection for the pattern graph G, as the following lemma shows:

Lemma 10 [MSAs and Minimal Operation Selections] Given an MSA $T \subseteq \tilde{G}$ according to the cost function $cost' := \log \circ cost$: Then the corresponding operation selection $S := op(E_T) \backslash \{\bot\} \in Sel(G)$ is minimal, that is $c(S) = \min_{S' \in Sel(G)}(c(S'))$ holds.

Proof. According to lemma 9 for each admissible operation selection $S' \in Sel(G)$ there exists an MSA $T' \subseteq \tilde{G}$ with $S' = op(E_{T'}) \backslash \{\bot\}$. Let $c(\bot) := 1$. Then it is

$$\log c(S') = \log \prod_{o \in S'} c(o) = \sum_{o \in S'} \log c(o) = \sum_{f \in E_{T'}} \log c(op(f)) =$$
$$= \sum_{f \in E_{T'}} \log(cost(f)) = \sum_{f \in E_{T'}} cost'(f).$$

The total cost $\sum_{f \in E_T} cost'(f)$ of T is minimal. As log is a strictly monotonic increasing transformation between $(\mathbb{R}_{\geq 1}, \cdot)$ and $(\mathbb{R}_{\geq 0}, +)$ this yields that $c(S)$ is minimal too. Thus, the condition $c(S) = \min_{S'' \in Sel(G)}(c(S''))$ holds. □

Figure 7 shows the plan graph $\tilde{G}_1$ from figure 6 with transformed costs $cost' = \log \circ cost$. The bold drawn arrows denote an MSA $T \subseteq \tilde{G}_1$. The total cost of T is 0, which is the logarithm of the total cost 1 of a multiplicatively computed MSA. The operation selection $S := op(E_T) \backslash \{\bot\}$ corresponding to T is the set

$$S = \Big\{ \mathsf{ext}(v_2, e_1), \mathsf{ext}(v_2, e_3), \mathsf{ext}(v_3, e_2), \mathsf{lkp}(v_3) \Big\}.$$

As the operations in S can be composed to a valid search plan, it is really an admissible operation selection. A possible search plan build of S is

$$\langle \mathsf{lkp}(v_3), \mathsf{ext}(v_3, e_2), \mathsf{ext}(v_2, e_3), \mathsf{ext}(v_2, e_1) \rangle,$$

for example. Furthermore S has the cost $c(S) = 1$. According to the cost model defined in section 5 no operation selection can have a cost smaller than 1. So S is a minimal admissible operation selection.

As stated in section 2.4 an MSA can be computed with the Edmonds/Chu-Liu algorithm in polynomial time [6, 3]. Alternatively a variant of this algorithm can be used, which only needs $O(|E_{\tilde{G}}| + |V_{\tilde{G}}| \cdot \log |V_{\tilde{G}}|)$ time. It has been proposed by Gabow et al. [8]. As $|\tilde{G}| \in O(|G|)$ holds and the construction of $\tilde{G}$ from G takes $O(|G|)$ time, the operation selection has an overall time complexity of $O(|G| \cdot \log |G|)$.

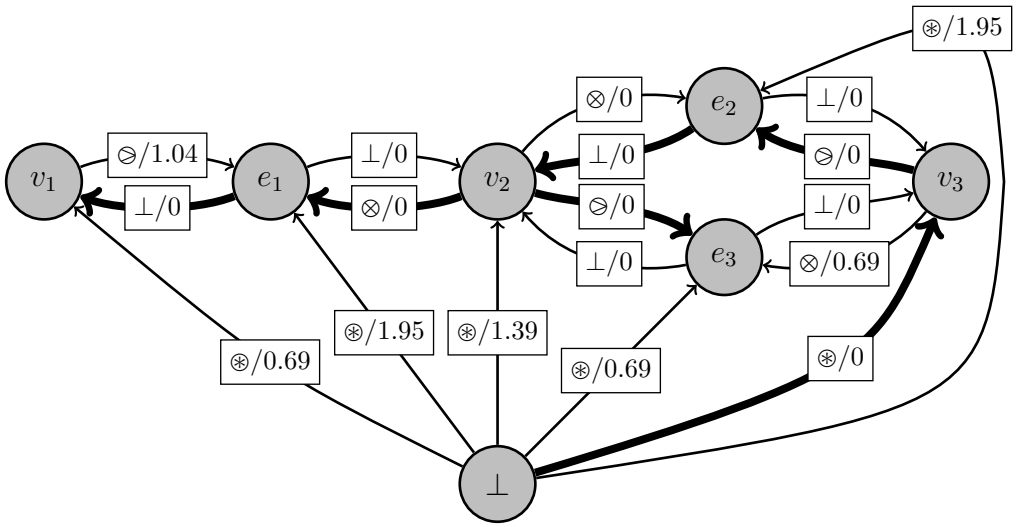

Figure 7: The plan graph $\tilde{G}_1$ with transformed cost function $cost' = \log \circ cost$. The bold drawn edges denote an MSA.

7.3 Operation Ordering

According to formula (1) a primitive operation has more impact to $c(P)$ the earlier the operation occurs in a search plan P. So having selected an admissible operation selection $S \in Sel(G)$ that is minimal in the sense of equation 2, we schedule the operations in S in a way such that the cheaper operations occur preferably early and the more expensive operations preferably late.

This is done by a simple best-first strategy: Like in the proof of lemma 8 a traversal of the MSA $T \subseteq \tilde{G}$ with root w_{root} is constructed. But this time the cheaper edges are preferred to the more expensive ones, which is different from the construction given in the proof. Having found such a best-first traversal $f_1, \ldots, f_k$ of T a search plan can be generated in a very simple way: Go through the traversal from left to right emitting the operation $op(f_i)$ each unless $op(f_i) = \bot$.

Admittedly, just as in the proof of lemma 8 this does not always yield a valid search plan. In fact it fails if and only if there are edges f_i, f_j with $i < j$ present in the traversal such that $op(f_i) \in \{\mathsf{lkp}(e), \mathsf{ext}(v', e)\}$, $op(f_j) = \mathsf{lkp}(v)$, and $inc_{E_G}(e) = \{v, v'\}$. As this demands the lookup of an already matched node v the emitted search plan is invalid then.

However, this requires only a slight modification to the best-first strategy constructing the traversal of T: Any time an edge f_i with $op(f_i) \in \{\mathsf{lkp}(e), \mathsf{ext}(v, e)\}$ is appended to the hitherto traversal $f_1, \ldots, f_{i-1}$, the nodes $w \in V_{\tilde{G}}$ with $\ell_{V_{\tilde{G}}}(w) \in inc_{E_G}(e)$ are marked as *reached*. In the following steps an edge f with $op(f) = \mathsf{lkp}(v')$ is omitted if the node $w' := tgt_{\tilde{G}}(f)$ is marked as reached. With given MSA T this modified best-first construction can be performed in $O(|G| \cdot \log |G|)$

time. This can be achieved, for example, by using a Fibonacci heap.

Consider the MSA shown in figure 7. Then the above best-first strategy yields one of the two following search plans:

$$\begin{aligned} P_1 &= \langle \mathsf{lkp}(v_3), \mathsf{ext}(v_3, e_2), \mathsf{ext}(v_2, e_1), \mathsf{ext}(v_2, e_3) \rangle \\ P_2 &= \langle \mathsf{lkp}(v_3), \mathsf{ext}(v_3, e_2), \mathsf{ext}(v_2, e_3), \mathsf{ext}(v_2, e_1) \rangle \end{aligned}$$

Note that the situation shown in figure 7 requires no omitting of edges which represent a lookup operation.

7.4 Summary

Altogether the process of search plan generation consists in the following steps:

1. The assignment of costs to primitive matching operations requires information about the present host graph H. If not enough domain specific knowledge about the occurring host graphs is available, an analysis of H is required. This takes $O(|H|)$ time (see section 6).

2. From the given pattern graph G a plan graph $\tilde{G}$ is constructed, which is a labeled directed graph with edge costs. The generation of $\tilde{G}$ takes $O(|G|)$ time (see section 7.1).

3. Having constructed the plan graph $\tilde{G}$ an MSA $T \subseteq \tilde{G}$ is computed. If the algorithm proposed by Gabow et al. is used, this requires $O(|G| \cdot \log |G|)$ time. The resulting MSA T represents a minimal admissible operation selection S for G (see section 7.2).

4. Having found an MSA T a best-first traversal of T is computed. Then an according search plan is emitted. This can be done in $O(|G| \cdot \log |G|)$ time and corresponds with finding a best-first order of the operations in S. The result is a valid search plan P for the given pattern graph G (see section 7.3).

Let P a generated search plan. If $c(P) = k$ holds, this means that the search plan P raises no splitting at all for the current host graph H. In this case the execution of P takes $O(|G| \cdot |E_H|)$ time. If all nodes of H have $O(1)$ incident edges, this even reduces to $O(|G|)$, which is a linear runtime (see section 5.3).

8 Related Work

The optimization technique described in this work extends a method originally invented by Dörr [5]. Dörr already identified bunches of isomorphic edges present in the host graph as a cause of unfeasible time complexity. To conceptualize this

he invented the terminology of strong V-structures and their instances, which is also used in this work. However, Dörrs approach does not feature a cost model. This means, that only a linear time search plan can be generated or no search plan at all. In cases where Dörrs method yields a linear time search plan, the method described here yields a linear time search plan, too. So the cost model based optimization technique described here is a significant extension.

An older version of the cost model based approach has already been presented by Batz [2]. At this state lookup operations for edges were not included.

Accordingly the plan graphs as used there do not contain special nodes for the representation of pattern edges. Nevertheless, this older version has been implemented very successfully in a tool called GrGen [10]. GrGen is a generative programming system for rule based graph transformations[7], which includes the task of subgraph matching. To determine the costs of the primitive matching operations during the search plan generation GrGen uses the analysis of the host graph described in section 6.

The search plan based subgraph matching performed by GrGen works very well for a benchmark introduced by Varró [11, 20, 19]. Assuming that many application domains feature approximately sparse graphs with rich label alphabets, we hope that such good behavior will also show in practice. For the future it is desirable to implement the technique described in this paper at its full extend, maybe as an enhancement of the GrGen system. It is likely that this yields a further improvement to the runtime of the subgraph matching performed by GrGen.

Independently Varró et al. proposed a method, which is very similar to that older version of the optimization technique [21]. An equivalent of edge lookups is also not supported there. Search plans are defined directly as a traversal of the plan graph (or the search graph as it is called there). The generation of a search plan is also done by the computation of an MSA through the Edmonds/Chu-Liu algorithm with succeeding best-first enumeration.

A main difference consists in the cost model: Though the cost of a search plan is also computed by formula (1), Varró et al. compute the average splitting factors in terms of the *arithmetic* mean. Moreover, they allow primitive matching operations to have costs between zero and one. The total cost of the MSA is computed by summation and not by a product. This entire means, that the operation selection as done there does not only reduce the splitting factor but also includes the first-fail principle. But as a consequence the operation selection does no longer directly correspond to the minimization of the most significant term of formula (1) as it does here.

Zündorf also presented a search plan based approach to subgraph matching, namely in the context of the Progres graph rewriting tool [22]. His method

[7] GrGen implements the well founded SPO approach to graph rewriting [7] with slight restrictions and powerful extensions.

provides equivalents to the extension operations and node lookups described here. But additional features are also supported. Amongst others these are (1) the matching of paths of unbounded length, (2) so called *TestEdge* operations that check the existence of edges between already matched nodes, (3) matching with pre-assigned nodes, and (4) attributed nodes with indexing by attribute values. The latter allows the fast access to nodes via the values of a given attribute by an internal indexing structure.

To enable the rating of search plans Zündorf describes a quite sophisticated cost model. However, the planning strategy itself is a best-first method that works greedy except for the selection of expensive lookup operations. The costs of the primitive matching operations are derived from static knowledge and heuristic assumptions only. At this the static knowledge consists in so called cardinality assertion, which can be compared to the cardinalities known from UML class diagrams.

Lillqvist [14] describes a backtracking based algorithm and mentions the concept of search plans in the context of CORAL that is a framework for model driven engineering. However, he only says few about the planning itself. Nevertheless, his work contains a nice summary of some matching methods.

Ullmann presented a backtracking algorithm which enumerates possible node mappings while checking for required host graph edges and pruning the search space [18]. The pruning of the search space can significantly reduce the runtime needed by the matching process. This is done by the following two strategies: Firstly, an initial step rules out all node mappings which affect host graph nodes of to small degree. Secondly, during the enumeration of the remaining node mappings the so called *refinement procedure* is repeatedly invoked. According to the contiguity of the host graph the refinement procedure removes further node mappings from the search space. Each time the procedure is invoked the removal is done iteratively until a fixed point is reached, i.e., until no further node mapping can be precluded.

On his experiments Ullmann treats the nodes of the pattern graph in an order of decreasing degree. In this way it is likely, that the absence of required edges in the host graph is detected more early. This in turn has the consequence that the refinement procedure has more impact. Additionally, Ullmann suggests choosing the order, in which the pattern nodes are treated, according to the respective application context. However, this idea is similar to the concept of search planning.

With the VF2 algorithm [4] Cordella et al. also suggest a backtracking based technique, which also performs pruning of the search space during the matching process: Firstly, node mappings which do not conform locally to the contiguity of the pattern graph are ruled out. Secondly, some kind of locally restricted breadth-first search is performed. More precisely, it is checked, whether the neighborhood of a current potential extension of the node mapping contains enough nodes that are rightly connected to the already processed part of the pattern and the host

graph, respectively. Thirdly, it is also checked, whether that current potential extension provides appropriate node and edge labels.

The authors state that (according to the number of nodes) the VF2 algorithm has better asymptotic time and space complexity than the Ullmann algorithm: Linear versus cubic space complexity and a time complexity that is better by a linear factor.

Messmer and Bunke proposed a method which is somewhat similar to the RETE approach to pattern matching [15]. It is suited for the simultaneous matching of multiple pattern graphs to a single host graph. In a preprocessing step which can be performed offline a compact representation of the pattern graphs is generated. At this the essential idea is to represent common subgraphs of the pattern graphs only once. If n very similar pattern graphs are matched simultaneously, this can result in a speedup by a factor of up to n. If n totally different pattern graphs are matched, there is no speedup.

Rudolf [17] as well as Larrosa and Valiente [13] reduce subgraph matching to constraint satisfaction Problems (CSP). In this way all the techniques known in the area of CSPs get available to the area of subgraph matching. Moreover the problem of subgraph matching is decoupled from a fixed graph model.

However, to improve the time complexity by taking advantage from the structure of a respective graph implementation Rudolf invents the concept of *queries*. In doing so, possible properties of a graph data structure are accessible through an abstract concept. It turns out that the constraint graph of the CSP to that Rudolph reduces the subgraph matching problem is quite similar to the plan graph as defined here. The queries present in the constraint graph essentially correspond to the edges in the plan graph representing the primitive matching operations. This suggests that the concepts of queries and of primitive matching operations are related. Furthermore the concept of variable ordering used in the area of CSPs is essentially the same as the concept of search planning.

Larossa and Valiente consider four approaches within constraint satisfaction framework. Additionally they present the new nRF+ approach which (on the majority) outperforms the four others. The nRF+ algorithm prunes the search space while performing a backtracking based search. Larossa and Valiente state that Ullmanns algorithm is essentially the same thing as one of the four other approaches. According to this it turns out that the pruning procedure of the nRF+ algorithm is stronger than that of Ullmanns algorithm. Additional Larossa and Valiente suggest a benchmark for subgraph matching which is based upon the Stanford GraphBase (see Knuth [12]).

9 Conclusions

In this work a heuristic optimization technique for the generation of preferably good strategies for subgraph matching is described. Matching strategies are rep-

resented as search plans that are sequences of primitive matching operations. We support two kinds of primitive matching operations: lookup and extension operations. A lookup operation represents the search for an appropriate element regardless of its position in the host graph. This allows the matching of non-connected pattern graphs. On the other hand for seldom node or edge labels the runtime of subgraph matching can be significantly reduced this way. An extension operation represents a local search coming from an already matched node along an appropriate edge in the host graph.

To rate the different matching strategies, a cost model is defined, that assigns costs to primitive matching operations and search plans. According to this cost model a search plan is generated by a heuristic optimization. Although the resulting search plans are not guaranteed to be optimal, several problem instances which may occur in practice can be solved in reasonable time this way. An older version of this approach has been implemented very successfully in the generative graph rewriting tool GrGen.

The cost $c(P)$ assigned to a search plan P arises from the present host graph H. If the domain specific knowledge about the occurring host graphs is not sufficient, the needed information can be provided by the described analysis of H in $O(|H|)$ time. For a pattern graph G the actual search plan generation can be done by the heuristic optimization in $O(|G| \cdot \log |G|)$ time. The heuristic optimization works in two phases: the operation selection, which provides a cheapest selection of primitive matching operations, and the operation ordering, that assembles the selected operations to a valid search plan according to a best-first strategy. If the cost $c(P)$ assigned to a generated search plan P equals the number of operations of P, the runtime raised by P is linear (under the assumption that H is sparse in the sense that every node of H has $O(1)$ incident edges).

If the costs of the primitive matching operations are determined according to an analysis of the host graph, the generation of the search plan must be done just in time. This can be implemented by materializing the primitive matching operations and the search plans as data objects present at runtime. In this case a possible matching algorithm works like an interpreter that executes dynamically generated search plans. This is done by the mentioned graph rewrite tool GrGen. If the domain specific knowledge about the occurring host graphs is sufficient, the analysis can be omitted. In this case there is no need for a runtime representation of primitive matching operations and search plans. Instead the search plans can be generated statically. These statically generated search plans could be emitted as source code written in a common programming language.

The optimization technique described in this work is also suited for negative application conditions (NACs) that are common in the area of rule based graph transformations: The application of a graph rewrite rule requires that there is no occurrence of any NAC present in the host graph. To guaranty this, an exhaustive search of the search space belonging to each NAC must be performed. However, the optimality criterion for the search plan generation as described here is the

avoiding of splitting. This in turn helps to reduce the search space and to avoid exponential runtime behavior also for NACs, hence.

Acknowledgements. Thanks to all the students and researchers at the Chair of Prof. Goos at the Institut für Programmstrukturen und Datenorganisation in Karlsruhe. I appreciate the atmosphere there as very creative. Especially I want to thank Rubino Geiß, Sebastian Hack and Christoph Mallon for support and fruitful discussions. Christoph was a good discussion partner when I went in for the Edmonds/Chu-Liu algorithm. Rubino helped me a lot on the preparation of this report. Furthermore without him the lookup operations for edges would not have been included in the search plan based approach discussed here. Also I want to thank Moritz Kroll and Rubino for proofreading this work. Last of all I want to thank my lord and savoir Jesus Christ. All good things come through him.

References

[1] Applications of the geometric mean. http://www.math.toronto.edu/mathnet/questionCorner/geomean.html.

[2] Batz, G. V. Graphersetzung für eine Zwischendarstellung im Übersetzerbau (Diplomarbeit). Master's thesis, Universität Karlsruhe (TH), IPD Goos, 2005.

[3] Chu, Y., and Liu, T. On the shortest arborescence of a directed graph. *Science Sinica 14* (1965), 1396–1400.

[4] Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26*, 10 (2004), 1367–1372.

[5] Dörr, H. *Efficient Graph Rewriting and Its Implementation*, vol. 922 of *LNCS*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[6] Edmonds, J. Optimum Branchings. *Journal of Research of the National Bureau of Standards 71B* (1967), 233–240.

[7] Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., and Corradini, A. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In *[16]*, vol. 1. 1999, pp. 247–312.

[8] Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica 6*, 2 (1986), 109–122.

[9] Garey, M. R., and Johnson, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[10] Geiss, R. The GrGen homepage. http://www.info.uni-karlsruhe.de/software.php/id=7, April 2006.

[11] Geiss, R., Batz, G. V., Grund, D., Hack, S., and Szalkowski, A. M. GrGen: A Fast SPO-based Graph Rewriting Tool. In *Graph Transformations - ICGT 2006* (2006), A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, Eds., Lecture Notes in Computer Science, Springer, pp. 383 – 397. Natal, Brasilia.

[12] Knuth, D. E. *The Stanford GraphBase: a platform for combinatorial computing*. ACM Press, New York, NY, USA, 1993.

[13] Larrosa, J., and Valiente, G. Constraint satisfaction algorithms for graph pattern matching. *Mathematical. Structures in Comp. Sci. 12*, 4 (2002), 403–422.

[14] Lillqvist, T. Subgraph Matching in Model Driven Engineering. Master's thesis, Åbo Akademi University, Department of Information Technologies, Faculty of Technology, 2006.

[15] Messmer, B. T., and Bunke, H. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering 12*, 2 (2000), 307–323.

[16] Rozenberg, G., Ed. *Handbook on Graph Grammars and Computing by Graph Transformation 1 (Foundations)*. World Scientific, Singapore, 1997.

[17] Rudolf, M. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *TAGT'98: Selected papers from the 6th Intl. Workshop on Theory and Application of Graph Transformations* (1998), vol. 1764, LNCS, pp. 238–251.

[18] Ullmann, J. R. An algorithm for subgraph isomorphism. *J. ACM 23*, 1 (1976), 31–42.

[19] Varró, G. Graph transformation benchmarks page. http://www.cs.bme.hu/~gervarro/benchmark/2.0/, August 2005.

[20] Varró, G., Schürr, A., and Varró, D. Benchmarking for Graph Transformation. Tech. rep., Department of Computer Science and Information Theory, Budapest University of Technology and Economics, March 2005.

[21] Varró, G., Varró, D., and Friedl, K. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In *GraMoT 2005, International Workshop on Graph and Model Transformations* (2005), G. Karsai and G. Taentzer, Eds., ENTCS.

[22] Zündorf, A. Graph Pattern Matching in Progres. In *Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science* (1996), vol. 1073 of *LNCS*, Springer, pp. 454–468.