Efficient model transformations for novices

Sten Vercammen

University of Antwerp

Abstract

This paper will give an overview, and understandable explanation with examples, of known techniques for pattern matching and solving the sub-graph homomorphism problem. For clarity reasons, the presented techniques do not include performance adaptation, but will list some possible extensions. It is intended as a guideline, even for novices, and provides an in-depth look at the workings behind various techniques for efficient pattern matching.

Keywords: search plans, constraint satisfaction problem, Ullmann, VF2, pattern matching

1. Introduction

While pattern matching is not limited to the scope of model driven engineering, it affects model driven engineering quite drastically. In model driven engineering one does not code the software, one models it. Which allows for more efficient software development. This can easily be seen when you consider that changing one association in a model can affect thousands of lines of code. Changing the association of a model might take a few seconds, but rewriting thousands of lines of code will take much, much longer. An other advantage of model driven engineering is that part of the complexity will be hidden. Which makes it possible for domain experts to easily verify (or even model) the models, as they do not need to have knowledge of programming (code).

In model driven engineering models get developed and these models must confirm to some meta-models. Furthermore, dynamic semantics are necessary. These models can be represented as graphs. This allows for defining some kind of execution, in the form of so-called transformation rules. The rules consist of two parts, a so-called left hand side (LHS) (the pattern we want to find) and a so-called right hand side (RHS) (the transformed pattern). A transformation rule is executed on a so called host graph. These graph represents the model. Executing a transformation rule means searching for an occurrence of the pattern from the LHS in the host graph and adapting the occurrence so that it matches the pattern from the RHS. A LHS can contain one or multiple NAC's, these patterns will prevent the transformation rule from executing if they are found in the graph. For clarity reasons we will assume that no transformation rules will have a NAC.

Email address: Sten.Vercammen@student.uantwerpen.be (Sten Vercammen)

While adapting the found occurrence to the pattern of the RHS, or the so-called replacement operation, is not computationally expensive, matching the graph pattern is the central efficiency problem for the execution of these replacement rules. In general, a naive algorithm for executing a graph replacement operation step (a transformation rule) has the time complexity $\mathcal{O}(P * N^L)$ where P is the number of inspected productions, N is the number of vertices in the current work graph, and L is the maximum size of a left-hand side of the system [1].

This paper will explain, in-depth, some known techniques for pattern matching and solving the sub-graph homomorphism problem. For coherence between this paper and example we provide an easy to understand implementation for all algorithms[1]. To keep the presented techniques understandable, they will not include performance adaptation. We will however list some possible extensions to further improve their execution time.

2. Graph Pattern Matching

For models implemented as graphs, pattern matching, and in particular the sub-graph homomorphism problem, is NP-complete [2]. Trough the use of heuristics, various exponential-time worst case algorithm can reduce their average-time complexity. These approaches can be divided clearly into two categories: search plans[2] and constraint satisfaction problems (CSP).

In the next sections, we start by explaining and listens the problems of a naive implementation. Then we will explaining both categories of graph pattern matching and explain some of their known (and most efficient) algorithms.

2.1. Running Example

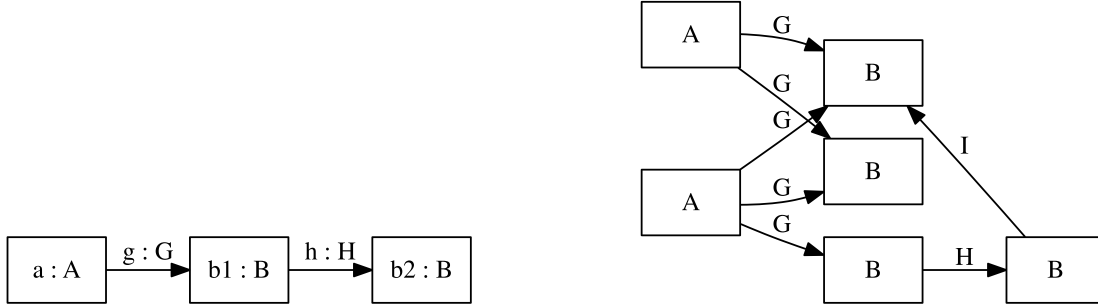
Throughout this paper we will use the same example when showing the working of the different algorithms. Consider a host graph H, on which we want to perform a Graph Transformation (GT) rule $L \rightsquigarrow R$. The GT rule exists of a graph pattern P, the LHS L, and a replacement graph, the RHS R. Executing a GT rule is done by first matching an occurrence of the LHS in the host graph (H), then changing the matched occurrence to the RHS of the GT rule. Figure 1a and 1b will respectively be used as our example of the pattern and host graph.

3. Naive implementation

A naive implementation would select a pattern vertex, then match it to a graph vertex of the same type. It would then try to match all edges of the pattern vertex onto to the matched graph vertex. The algorithm would then, for each of those pattern edges their

[1]Which can be found on:
`http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/projects/Sten.Vercammen`

[2]Note: all subsections of section 4 are heavily based upon [3]



(a) A simple pattern graph P.

(b) A simple host graph H.

Figure 1: Running example

source or target (depending if it was an incoming or outgoing edge), try to match a graph vertex onto it. And then match its edges and so on.

This is very inefficient and has an exponential runtime. The *matchNaive* function in the *patternMatching.py* file implements this behaviour[1].

4. Search Plans

In short, a search plan defines the traversal order for the vertices of the model to check if a pattern can be matched (i.e., an occurrence). A valid search plan P is defined as a well-formed sequence $P = \langle o_1, ..., o_k \rangle$ of primitive matching operations.

4.1. Primitive Matching Operations

A primitive matching operation is an atomic search action that deals with exactly one pattern element. There are five distinct kinds of primitive matching operations:

lkp(x): A *lookup* operation that binds the pattern element x to a corresponding host graph element.

in(v, e): An *incoming edge* operation that binds the pattern edge e, an incoming edge of the already bound pattern vertex v, to a corresponding edge of the host graph.

out(v, e): An *outgoing edge* operation that works analogously to **in**(v, e), but e is now an *outgoing* pattern edge of v.

src(e): A *get source* operation that binds the source vertex of an already bound pattern edge e to the source vertex of the current host graph image of e.

tgt(e): A *get target* operation that works analogously to **src**(e), but instead of working with the source vertex, it works with the *target* vertex.

The operation $o := \mathbf{lkp}(v)$, with pattern vertex v will be successfully executed if there is at least one unbound vertex w in H such that v and w have compatible types. If this is the case, o will bind v to w. Otherwise the execution is not successful, and fails.

Matching a primitive operation is implemented by the *matchOP* function under the *matchSP* function in the *patternMatching.py* file[1].

4.2. Example Search Plans

A search plan $P = \langle o_1, ..., o_k \rangle$ is valid if:

1. Every element of the pattern graph is bound exactly once.
2. If an operation o_i requires that a pattern element is already bound, the element must be bound by one of its preceding operations $o_1, ..., o_{i-1}$

P_1 shows an invalid search plan:

$$P_1 := \langle \mathbf{out}(a, g), \mathbf{lkp}(\mathit{b1}), \mathbf{tgt}(g), \mathbf{lkp}(h) \rangle$$

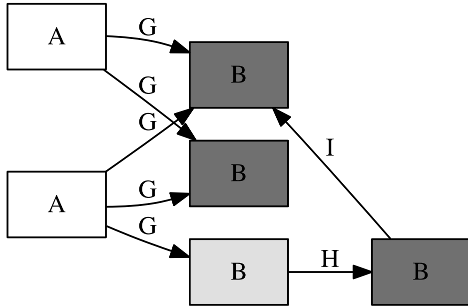
It is invalid for several reasons. Firstly, the operation $\mathbf{out}(a, g)$ is not valid, as it requires that the pattern vertex a is already matched (which can not be as it is the first operation). Secondly, the second and third operation both want to bind $b1$, which is not allowed (***lkp**(b1)* binds *b1* explicitly and $\mathbf{tgt}(g)$ binds *b1* implicitly, as it binds the target vertex of an already bound pattern edge g which, in this case is *b1*). And thirdly, as every element of the pattern graph must be bound exactly once, there must be exactly as many operations in the search plan as there are elements in the pattern graph. Because there are five elements in the pattern graph, and only four operations in the search plan, for this reason also, the search plan is invalid.

Keeping the previous in mind, following two search plans are considered valid.

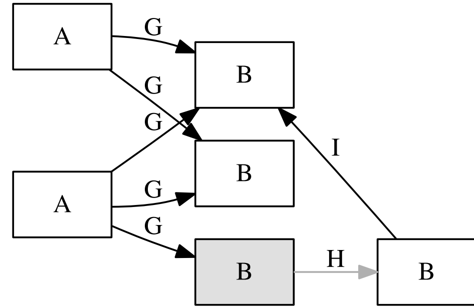
$$P_2 := \langle \mathbf{lkp}(\mathit{b1}), \mathbf{out}(\mathit{b1}, h), \mathbf{in}(\mathit{b1}, g), \mathbf{tgt}(h), \mathbf{scr}(g) \rangle$$
$$P_3 := \langle \mathbf{lkp}(h), \mathbf{tgt}(h), \mathbf{src}(h), \mathbf{in}(\mathit{b1},g), \mathbf{src}(g)) \rangle$$

In our running example (with the host graph from figure 1b) there is only one occurrence of the pattern P (pattern from figure 1a). If we execute P_2, the first operation ($o_1 := \mathbf{lkp}(\mathit{b1})$) will perform a lookup of the type B. There are four alternatives (see figure 2a). If we choose a “wrong” vertex we will not be able to find the pattern and we will have to use backtracking to continue our search. Assuming we do choose the right vertex (in figure 2a, the lightest highlighted one), we will be able to match our second operation (see figure 2b). The second operation binds an outgoing edge from *b1* of type H. The third operation binds an incoming edge to *b1* of type G (see figure 2c). The fourth operation binds a target vertex from the previously bound edge h (see figure 2d). The fifth and last operation binds a source vertex from the previously bound edge g (see figure 2e).

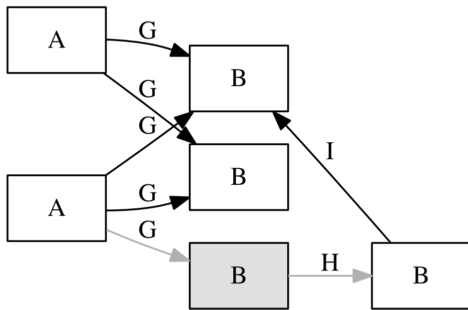
As you might have noticed by now, P_2 is not a bad solution in terms of speed, as in the worst case, it processes eight elements (if in the first step we first execute all other type B vertices, before choosing the correct one like in figure 2a). P_3 however is the optimal solution, as it always processes the minimum amount of elements (in this case, five). As there are many possible search plans, one way of finding the “best” possible search plan is through the use of a cost model.



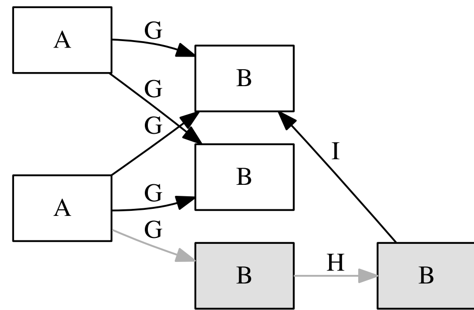
(a) o1, four alternative highlighted matches of type B, we choose one (the lightest)



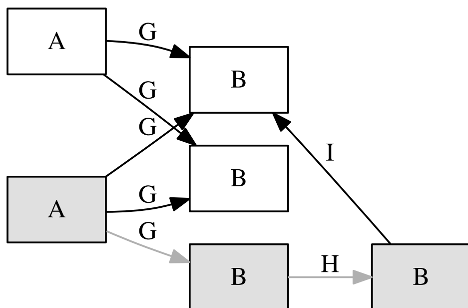
(b) o2, outgoing edge h match from previously matched *b1*



(c) o3, incoming edge g match from previously matched *b1*



(d) o4, match target vertex from a previously matched vertex



(e) o5, match source vertex from a previously matched vertex

Figure 2: Execution of search plan P_2

4.3. Cost Model

A cost model assigns a cost to every matching operation, and consequently to every search plan. This makes search plan generation an optimization problem and allows the generation of matching strategies at runtime depending on the current host graph. The example in section 4.2 showed us that an operation that resulted in choice, might cause the total number of processed elements to go up. As we want to process as least elements as possible, operations that result in choice are deemed more costly.

We can assign a cost for each primitive matching operation:

Operation	Cost	Explanation
$\mathbf{lkp}(x)$	$\lvert\text{typeof}(x) \text{ in } H\rvert$	The more elements of type x the host graph H contains, the more choices the operation creates.
$\mathbf{in}(v, e)$, $\mathbf{out}(v, e)$	AVG(possibilities)	The example $\mathbf{out}(a, g)$ will correspond into two or three possibilities for binding g, depending on the binding of a. We therefore assign the average of the possibilities to the cost function. (Note: average can be geometric mean or arithmetic mean, etc)
$\mathbf{src}(e)$, $\mathbf{tgt}(e)$	1	As each edge has exactly one source and one target vertex, these operations do not create a choice.

Calculating the average possibilities for the $\mathbf{in}(v, e)$, $\mathbf{out}(v, e)$ operations requires keeping some statistics. These can be kept, and updated, by storing a counter for each edge type, for each vertex type. (So we save multiple edge type counters, one for each vertex type.) We do this for both the incoming and outgoing edges. An implementation of this can be found in the *searchGraph.py* file[1]. It extends a normal graph, and overloads the adding of the edges for updating the counters.

The total cost of a plan will be assigned as:

Plan	Cost	Description
$P = \langle o_1, ..., o_k \rangle$	$\sum_{i=1}^{k} \prod_{j=1}^{i} c_j$	For search plan $P = \langle o_1, ..., o_k \rangle$, if $c(o_i) > 0$, with $1 \leq i \leq k$, then the execution of o_i might cause backtracking. This means that the remaining part $o_{i+1}, ..., o_k$ might be executed up to $c(o_i) > 0$ times. So $c(P) = c_1 + c_1 c_2 + ... + c_1 c_2 ... c_k$

The cost of these primitive matching operations, and consequently the cost of the search plan, depends on the current host graph. An analysis on the host graph to calculate the cost is linear to the number of elements in the host graph.

4.4. Generating a Plan Graph

Before generating an "optimal" search plan, we will first need to explain the so called plan graph, as it is used for calculating the "optimal" search plan. A plan graph is generated from a pattern graph and will look quite similar to it (e.g. figure 3 is the corresponding plan graph of the pattern from figure 1a).

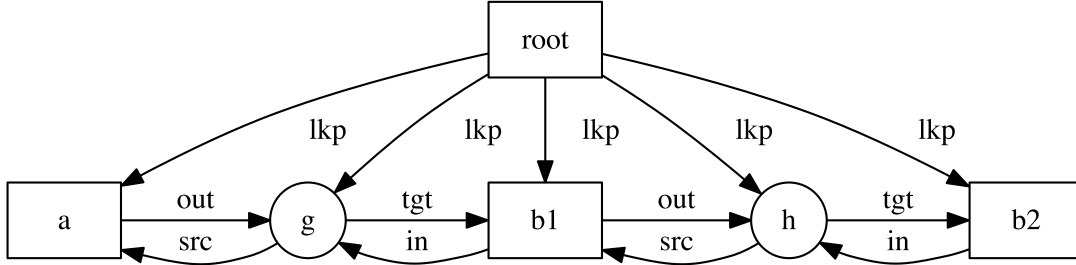


Figure 3: Plan graph of the simple pattern graph in figure 1a

Creating a plan graph $\widetilde{L}$ from a pattern graph L follows following rules:

1. For every edge and every vertex in L, create exactly one vertex representing the pattern element and label it with the same name as the pattern element.
2. Create a *root* vertex in $\widetilde{L}$.
3. For each element x in $\widetilde{L}$ that is not the *root* vertex, create an edge from the *root* vertex to x and label it with **lkp**.
4. For every element x in $\widetilde{L}$ that represents an edge e in L;
   - create an edge labelled with **tgt**, leading from x to the vertex in $\widetilde{L}$ representing the target vertex of e in L, and a reverted edge labelled **in**.
   - create an edge labelled with **src**, leading from x to the vertex in $\widetilde{L}$ representing the source vertex of e in L, and a reverted edge labelled **out**.

The names of the edges in the pattern graph represent the primitive matching operations.

An implementation for generating the searchplan can be found in the *planGraph.py* file[1]. It contains a function for updating the cost of the edges so that we can regenerate an optimal search plan every time we change the host graph.

4.5. Logarithmized Cost Function

The most significant term of cost of the plan $P = \langle o_1, ..., o_k \rangle$ is $c_1 c_2 ... c_k$. As we want to minimize the cost, we can look for the minimal cost of the most significant term. As the cost of all operations in P appears exactly once in $c_1 c_2 ... c_k$ and each $c_i > 1$ (if we assume that we can match the pattern), we can rewrite it to a *minimal operation selection* $S :=\{o_1, ..., o_2\}$. The cost of $S = c_1 c_2 ... c_k$. As all individual costs of the operations are larger than one, we can perform our search for a minimal operation selection equivalently with a logarithmized cost $(\ln(c_1 ... c_k) = \ln(c_1) + ... + \ln(c_k))$. This has the benefit that we no longer need to minimize a product, but that we can minimize a sum, which is computationally much faster.

We use the logarithmized cost is build in into the *planGraph.py* file[1].

4.6. Generating a Search Plan

The mapping between the possible operation selections and the set of *directed spanning trees* (DST) of the plan graph is a one-on-one mapping. The corresponding DST of a minimal operation selection is a *minimum* directed spanning tree (MDST) according to logarithmized cost. Therefore, finding a minimal operation selection corresponds to finding a MDST in the plan graph, which can be solved in polynomial time by the Edmonds algorithm [4].

Adding the cost for each primitive pattern operation from our host graph (figure 1b) to the plan graph (figure 3) results in figure 4. The bold drawn edges represent the MDST. The according mimimal operation selection is:

$$S := \{\mathbf{lkp}(h), \mathbf{tgt}(h), \mathbf{src}(h), \mathbf{in}(\mathit{b1}, g), \mathbf{src}(g)\}$$

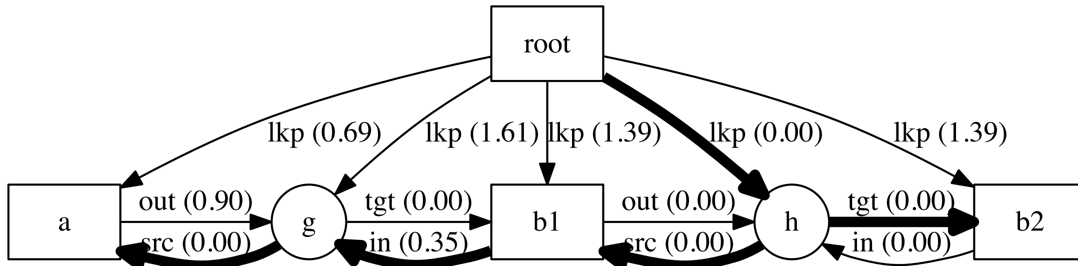


Figure 4: The plan graph from figure 3 with logarithmized costs induced by the host graph from figure 1b. The bold drawn edges mark an MDST

Edmonds algorithm will repeatedly select edges, minimal edges are preferred, for the MDST. When more than one edge concerned has minimal cost[3], the kind of primitive operation will determine the preference. We favour $\mathbf{src}(e)$ and $\mathbf{tgt}(e)$ operations over $\mathbf{in}(v, e)$ and $\mathbf{out}(v, e)$ operations (as $\mathbf{src}(e)$ and $\mathbf{tgt}(e)$ are almost cost free). We favour $\mathbf{in}(v, e)$ and $\mathbf{out}(v, e)$ operations over $\mathbf{lkp}(x)$ operations. The reason behind this is that lookup operations do not take the contiguity of pattern and host graph into account, which could raise needless choices during the graph pattern matching.

We have not explained Edmonds algorithm in this paper, as we considered it to be out of scope, but it is implemented in the *planGraph.py* file under the *Edmonds* function [1]. There are a lot of comments to help you understand the algorithm, so if you are interested in the workings, definitely take a look at it.

Matching all primitive operation is done in a recursive manner (implemented by the *matchAllOP* function). We do note that recursion in python has a very limited depth. If you want to use it for bigger graphs we advice to rewrite it to eliminate recursion.

[3]The logarithmized cost values are floats, so we need to to weaken them before we can compare them (e.g. round to 0.001).

4.7. Operation Ordering

Building a valid search plan from the minimal operation selection S, can be done by simply traversing the MDST starting from the root vertex. While traversing, we successively emit the operations represented by each edge. This generates all operation sequences that are valid search plans.

Finding the "best" search plan is done by traversing the plan graph in a best-first manner, preferring edges of minimal cost. The reason behind this is that: If we look at the plan cost: $c(P) = c_1 + c_1c_2 + ... + c_1c_2...c_k$, we can see that the earlier the operation occurs, the more impact on the overall cost it has. Placing the cheap operations as early as possible and the expensive ones as late as possibly will result in a smaller cost.

We implemented a SortedContainer class to aid the traversal of the MDST in the *planGraph.py* file [1].

4.8. Alternative Techniques

As we have seen in section 4.3, a cost model assigns a cost to every matching operation, and consequently to every search plan. This makes search plan generation an optimization problem and allows the generation of matching strategies at runtime depending on the current host graph. This also means that variations of the search plan technique often just adapt the heuristic. There are numerous of techniques that propose a heuristic, graph based algorithm for efficient pattern matching [5], [6]. These heuristics can take into account given typing information with respect to the meta-model elements and/or the cardinality constraints defined in the meta-model. More complex model-specific optimization steps where adaptive search plans are generated also exist. [7]. In the latter, the optimal search plan will be selected from previously generated search plans at run-time, based on statistical data that is collected from the current instance model under transformation.

Other optimizations for search plans are [8] (but not limited to): indexing on type and/or by storing reverse associations, caching, pivoting and overlapped pattern matching.

5. Constraint Satisfaction Problems (CSP)

In this section, we will briefly explain what a CSP is, before explaining some efficient and well-known algorithms.

5.1. CSP Definition

A *constraint satisfaction problem* (CSP) is defined [9] by the following triplet $\langle X, D, C\rangle$. Where $X = \{X_1, X_2, ..., X_n\}$ is an ordered set of n variables. $D = \{D_1, ..., D_n\}$ where each D_i is a finite domain of possible values for each variable X_i. $C = \{C_1, ..., C_m\}$ is a set of *constraints* among variables. Each constraint consists of a pair $\langle\{j_1, ..., j_r\}, R_{j_1,...,j_r}\rangle$, or in short $\langle t_j, R_j\rangle$, where $t_j \subset X$ is a subset of r variables and R_j a r-ary relation on the corresponding subset domains D_j. $R_{j_1,...,j_r}$ is the constraint on the ordered set of variables $\{j_1, ..., j_r\}$, which is a subset of $D_{j_1 \times} ... \times D_{j_r}$ and only contains the *allowed* combinations of values for the variables $j_1, ..., j_r$.

An evaluation v on the constraint $\langle t_j, R_j \rangle$, a function from t_j to R_j, is satisfied if the values from t_j satisfy the relation R_j.

An evaluation is *complete* if it includes every variable in X. A (possibly incomplete) evaluation is *consistent* if it does not violate any constraints it is involved with. We call an evaluation a *solution* for the constraint satisfaction problem if it is both consistent and complete.

5.2. Use Cases for CSP

Constraint satisfaction problems can be used for finding one solution, finding all solutions, and for finding the best solution, given some preference criteria. While one might think we are interested in finding the best solution for graph matching, this is not the case. We want to find *a* solution as fast as possible, but we only need to find one solution. An example of the best solution would be finding the shortest route for the travelling salesmen problem [10].

5.3. In General

For solving constraint satisfaction problems, most algorithms start with an empty (trivially consistent) assignment. They attempt to extend it by adding one variable at a time. The variable can only be added successfully if, after adding it, the assignment is still consistent. If no variable can be added while keeping the assignment consistent, the algorithm will backtrack and change its previous decision. The general idea behind this is no different than the one used for search plans. The algorithm will stop when a total assignment has been computed, or when the whole search space has been unsuccessfully traversed. During the search, a variable can be one of following three types: a past variable if it is assigned, a future variables if it is unassigned, or a current variable if it is under consideration.

Look-ahead algorithms are considered to be the best choice for solving non-trivial CPS problems. These algorithms remove some future variables each time a current variable is assigned. This effectively limits the search space. The remaining future variables are often called the *feasible* variables.

5.4. Graph pattern matching as a CSP

Graph pattern matching can be described as a constraint satisfaction problem [11] by the following triplet: $\langle X, D, C \rangle$. The elements of the pattern graph (used in the LHS of a GT rule) represent the variables (X). The elements of the model (the host graph) denote the domain (D) and typing (each variable has its own domain). While the links (edges) and attributes define the set of constraints (C). As mentioned in section 5.3, the CSP algorithms will use backtracking [12] for finding an occurrence (an isomorphic sub-graph) of the pattern graph in the host graph. Most of these algorithms will explore the search space in a depth-first order (as this tends to require less memory than breath first algorithms which, in general, are better suited for finding the shortest path). In the next sections we will explain the Ullmann [13] [4] and VF2 [15] [5] algorithm. These are some of the most efficient

[4]Note: section 5.6 and 5.6.3 are heavily based upon [14]

[5]Note: section 5.7 and 5.7.3 are heavily based upon [14]

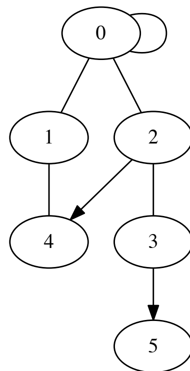
and well-known algorithms for solving the sub-graph isomorphism problem, as a constraint satisfaction problem.

5.5. Adjacency Matrices

In this section we briefly explain adjacency matrices, if you are familiar with it, you can skip this section.

The obvious way of representing a graph G is via its ordered pair, where $G = (V, E)$. V is the set of all vertices and E the set of all edges. A (finite) graph can also be represented with an adjacency matrix. For a graph with n vertices, this results in a $n \times n$ dimensional square matrix A. Where $A[i, j] = 1$ if there is an edge from vertex i to vertex j, otherwise it is 0. In the special case were i is equal to j and there is an edge from vertex i to vertex j (the so-called self-loop) $A[i, j] = 2$. If the graph is undirected, the adjacency matrix is symmetric.

Figure 5a shows an example of a graph with undirected edges, directed edges and a self loop. Figure 5b represents its adjacency matrix. If we call this matrix M, then we see that $M[0, 0]$ is 2. This is because vertex "0" has a self loop. $M[0, 1]$ and $M[1, 0]$ are both 1 because there is an undirected edge between these two vertexes. $M[3, 5]$ is 1, but $M[5, 3]$ is 0 because there is a directed edge from vertex "3" to vertex "5", but not from "5" to "3".



(a) Example graph

x	0	1	2	3	4	5
0	2	1	1	0	0	0
1	1	0	0	0	1	0
2	1	0	0	1	1	0
3	0	0	1	0	0	1
4	0	1	0	0	0	0
5	0	0	0	0	0	0

(b) Adjacency matrix M

Figure 5: Example graph with his adjacency matrix

Creating the adjacency matrix is implemented in the *createAdjacencyMatrixMap* function in the *patternGraph.py* file[1]. This also returns an array of the vertices to save the order in which the matrix is created.

5.6. Ullmann

Given two graphs $H = (V_H, E_H)$ (the host graph) and $P = (V_P, E_P)$ (the pattern graph), the Ullmann algorithm tests whether P is a sub-graph of H. We would not have explained

adjacency matrices if we did not use them, so we denote $\mathbf{H}$ and $\mathbf{P}$ as the adjacency matrices of H and P (which we will use later in the algorithm). In the first step of the Ullmann algorithm a $|V_P| \times |V_H|$ binary matrix $\mathbf{M^*}$ is created, such that:

$$\mathbf{M^*}[v, w] = \begin{cases} 1, & \text{if } \deg(v) \leq \deg(w) \text{ for } v \in V_P \text{ and } w \in V_H \\ 0, & \text{otherwise.} \end{cases}$$

Where $\deg : V \to \mathbb{N}^+$ is a function mapping a vertex to its degree: the amount of incident edges it is connected to. $\mathbf{M^*}$ represents all possible vertex candidates of V_H that are isomorphic to vertices of V_P. If $\mathbf{M^*}[v, w] = 1$, the vertex $v \in V_P$ is isomorphic to vertex $w \in V_H$.

Creating the adjacency matrix is implemented in the *createM_star* function under the *matchUllmann* function in the *patternGraph.py* file[1].

In the second step, the algorithm tries to find an isomorphic mapping for the vertices of P to H. This mapping is represented by a matrix $\mathbf{M}$ if, and only if $\mathbf{M}(\mathbf{MH})^T = \mathbf{P}^T$. Each row of the matrix $\mathbf{M}$ must have exactly one 1 element and each column may have at most one 1 element.

Starting from $\mathbf{M^*}$, the algorithm will thus search for a valid $\mathbf{M}$, by iterating over each possible vertex candidate (in a depth first order). At each step a vertex from V_H, a row of $\mathbf{M}$ is assigned one of the matches (in decreasing order of degree[6], implemented in the *createDecreasingOrder* function under the *matchUllmann* function in the *patternGraph.py* file[1]), by setting its column to 1 and the rest to 0. This search happens in a depth-first manner, but it is optimized with a *refinement procedure* (see section 5.6.1) that takes the neighbouring vertices into account.

Whenever this refining of $\mathbf{M}$ results in a row without a 1 element, the algorithm backtracks and the next potential match is tried. Otherwise (the row has one 1 element), the algorithm continues with the next row of $\mathbf{M}$. The algorithm will end if either a complete match is found or when all possible matches have been tried.

5.6.1. Refinement Procedure

A vertex v of V_P may only match, a vertex w of V_H represented by a 1 in $M[i, j]$, if all its neighbours can be matched. If at least one of its neighbours does not match, it will not consider w a valid match for v and set the corresponding element of the matrix to 0 ($M[i, j]$), effectively reducing the search space. This refinement procedure verifies all possible matchings (candidates) for all vertices in the pattern. If at least one candidate is eliminated (put to 0), it will rerun the refinement procedure. At first this might not seem like a big improvement, but eliminating one candidate in the a round, might result in eliminating multiple candidates in the next round.

This refinement procedure is implemented in the *refineM* function under the *matchUllmann* function in the *patternGraph.py* file[1].

[6] for efficiency, to fail as fast as possible, it turns out that the more edges a vertex has, the sooner it will fail in matching the pattern

5.6.2. Implicit Isomorphism

It is important to note that we do not need to explicitly check whether the found subgraph of H, represented in $\mathbf{M}$, is isomorphic to P. We do not need the calculate if $\mathbf{M}(\mathbf{MH})^T = \mathbf{P}^T$. This is implicitly calculated by the refinement procedure described in section 5.6.1.

If you do not want the Ullmann optimazations, and thus only do a depth first search, and check the isomorphism explicitly for each possible match, you can do so by using the commented code in the *propConnected* function under the *matchUllmann* function in the *patternGraph.py* file[1].

5.6.3. Ullmann Efficiency and Extension Options

The more sparse $\mathbf{M}^*$ initially is, the faster the algorithm will be. The unmodified Ullmann algorithm does not take into consideration the type of the vertexes, their attributes, labels etc. Extending the algorithm so that generating $\mathbf{M}^*$ does take the type of the vertexes etc into consideration can drastically increase the sparsity of $\mathbf{M}^*$, effectively speeding up the algorithm. Some approaches also extend the deg function mapping to incorporate more sophisticated feasibility tests (which will in turn also result in a more sparsely $\mathbf{M}^*$).

5.6.4. Ullmann Example Without Refinement

In this section we will run the Ullmann algorithm on our running example (our host graph H in figure 1b and our pattern graph P in figure 1a). We first explain it without the use of the refinement procedure. Effectively running a normal depth first search and checking only on the end if the found match is isomorphic to the pattern. We will explain it with the refinement procedure in the next section.

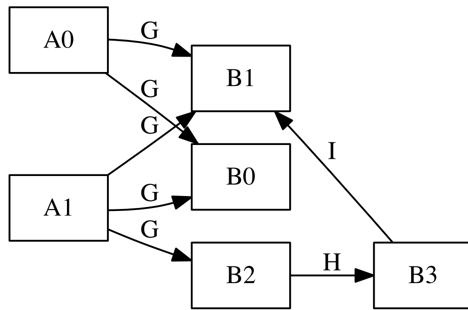
The example has directed edges and we will take this into account. In an undirected graph, the incident edges do not have a direction, so we can count them all. In a directed graph we can associate a vertex with its in-degree and its out-degree, where the in-degree represents the number of edges which target are to the vertex and the out-degree represents the number of edges which source is the vertex. Which one we choose in not important (we can even choose both), as long as we are consequent in our choice. In the example, we will use the out-degree.

In figure 6a we relabelled the vertices, this allows for more clarity (as now the vertices are unique) when building our adjacency matrix $\mathbf{H}$ in figure 6b and in our $|V_P| \times |V_H|$ binary matrix $\mathbf{M}^*$ in figure 8. Even though the labels of the vertices are extended with a number, we still mean that the type of the vertex is unaltered (so the type is without the number).

Ullmann's algorithm requires the use of the adjacent matrices of the host graph H and pattern graph P, these can respectively be found in figure 6b and 7. The $|V_P| \times |V_H|$ binary matrix $\mathbf{M}^*$ is created in figure 8. As we choose the out-degree, $\mathbf{M}^*[v, w]$ will result in a one if the out-degree of $v \leq$ out-degree w, with $v \in V_P$ and $w \in V_H$. As vertex $a \in V_P$ has out-degree one and vertex $A0 \in V_H$ has out-degree two, $\mathbf{M}^*[a, A0] = 1$. As vertex $a \in V_P$ has out-degree one and vertex $B0 \in V_H$ has out-degree zero, $\mathbf{M}^*[a, B0] = 0$. We continue until all elements of the matrix $\mathbf{M}^*$ are filled.

In the next step, Ullmann will try to find a match. Figure 9 represents the execution of the Ullmann algorithm on our example host graph H for matching pattern P. It will start

with $\mathbf{M}^*$, match a vertex from V_H, assign one of the matches with one (in decreasing order of degree), so $A1$ will be our first match, as it has a degree of three (all other elements of that row will become zero). As Ullmann's algorithm searches depth first, the next match is from the second column. Unfortunately this match is not compatible so the algorithm will backtrack. As we did not use the refinement procedure we will explicitly need to check if the pattern graph is isomorph to the found occurrence. Once an match is found (in the figure, a lowest matrix block without the darkest colours), Ullmann verifies if the found match is isomorphic. $\mathbf{P}$ is isomorphic if and only if $\forall i, j, \mathbf{P}[i,j] = 1 : \mathbf{M}(\mathbf{MH})^T[j,i] = 1$.



(a) Relabelled host graph

x	A0	A1	B0	B1	B2	B3
A0	0	0	1	1	0	0
A1	0	0	1	1	1	0
B0	0	0	0	0	0	0
B1	0	0	0	0	0	0
B2	0	0	0	0	0	1
B3	0	0	0	1	0	0

(b) Adjacency matrix **H**

Figure 6: Relabelled host graph with his adjacency matrix

x	a	b1	b2
a	0	1	0
b1	0	0	1
b2	0	0	0

Figure 7: Adjacency matrix **P**

x	A0	A1	B0	B1	B2	B3
a	1	1	0	0	1	1
b1	1	1	0	0	1	1
b2	1	1	0	0	1	1

Figure 8: $|V_P| \times |V_H|$ binary matrix $\mathbf{M}^*$

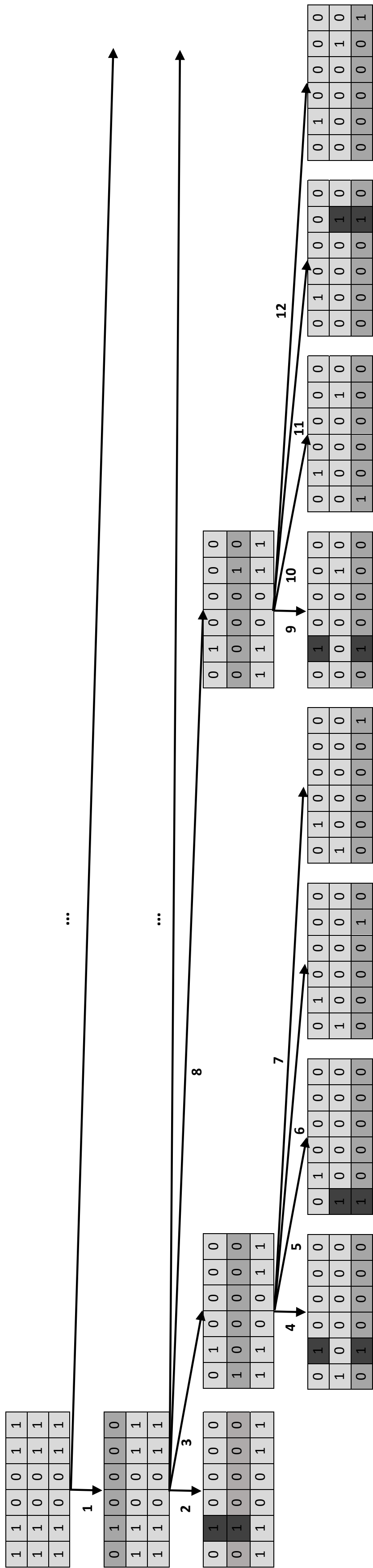


Figure 9: Ullmann execution without the refinement procedure

Our first match is after step 5. We now can check whether our found match is isomorphic.

$$\begin{aligned}
\mathbf{M}(\mathbf{MH})^T &= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \left(\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \right)^T \\
&= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \left(\begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \right)^T \\
&= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}
\end{aligned}$$

$\mathbf{P}$ is isomorphic if and only if $\forall i, j, \mathbf{P}[i, j] = 1 : \mathbf{M}(\mathbf{MH})^T[j, i] = 1$. Which in this case, it is not. Therefore the algorithm will continue searching. The second and third match will also fail this check, but the fourth match (after step 12) will succeed:

$$
\begin{aligned}
\mathbf{M}(\mathbf{MH})^T &= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \left(\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \right)^T \\
&= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \left(\begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \right)^T \\
&= \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}
\end{aligned}
$$

$\mathbf{P}$ is isomorphic if and only if $\forall i, j, \mathbf{P}[i, j] = 1 : \mathbf{M}(\mathbf{MH})^T[j, i] = 1$. Which in this case, it is. The algorithm stops searching as it found a match. The arrows with the three dots in figure 9 are just to show how the algorithm would continue (backtracking) if in step 12 no match was found.

5.6.5. Ullmann Example With Refinement

After each step, the refinement procedure will prune the (until then considered valid) possible mappings who's neighbours can not be matched. In figure 10 we can see that after the third time (arrow with 3 on) the refinement procedure turns the selected node for matching to a 0 (represented by the white 0 and darkest background colour), creating a 0 row. This refinement procedure prunes the search tree, and the matrices that are in the boxed figure will not be executed. Instead the algorithm no directly follows arrow 8.

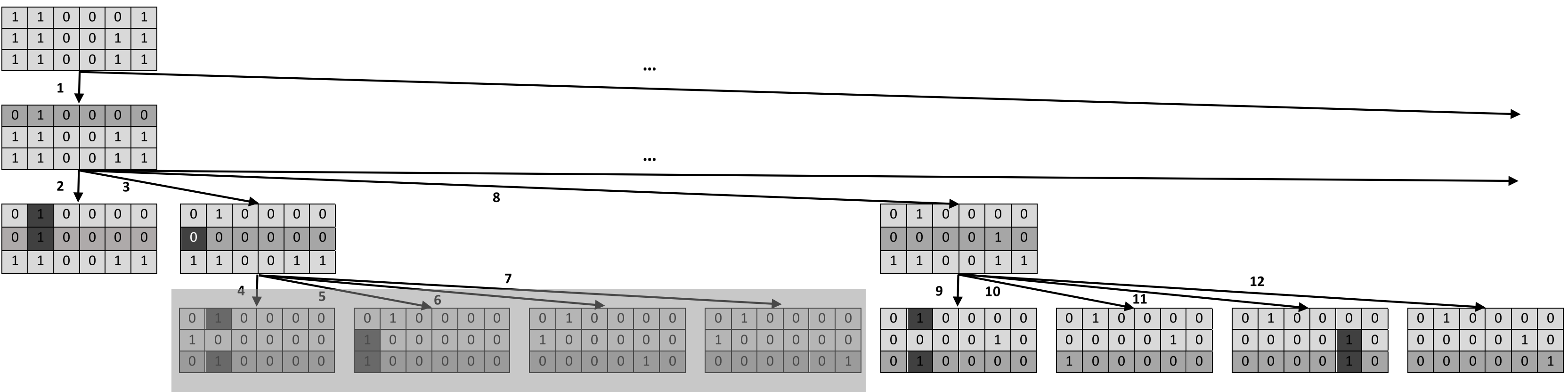


Figure 10: Ullmann execution with the refinement procedure

5.7. VF2

The VF2 algorithm operates in a similar fashion as the Ullmann algorithm. It also constructs the search tree, traversing the host graph depth-first and backtracks when the current-state does not succeed the feasibility test. It also performs pruning on the search space during the matching process.

Given two graphs $G = (V_G, E_G)$ (the host graph) and $H = (V_H, E_H)$ (the pattern graph), we denote $M : V_H \to V_G$ as the isomorphic vertex mapping. $M(c)$ holds the set of current matches (v_H, v_G) at the search state s (the dashed lines in figure 11, which links the black vertices).

$M_H(s)$ and $M_G(s)$ respectively represent the vertices of V_H and V_G contained in $M(s)$ (respectively the black vertices in H and G in figure 11). $N_H^{in}(s)$ is the set of vertices adjacent to $M_H(s)$ along the incoming edges and $N_H^{out}(s)$ the set of vertices adjacent to $M_H(s)$ along the outgoing vertices. $N_H(s) = N_H^{in}(s) \cup N_H^{out}(s)$ and corresponds to the highlighted vertices in figure 11. $\overline{V}_H = V_H - M_H(s) - N_H(s)$ represents the vertices not connected to the current mapping (these correspond to the white vertices in figure 11). $N_G(s)$ and $\overline{V}_G(s)$ are defined analogously.

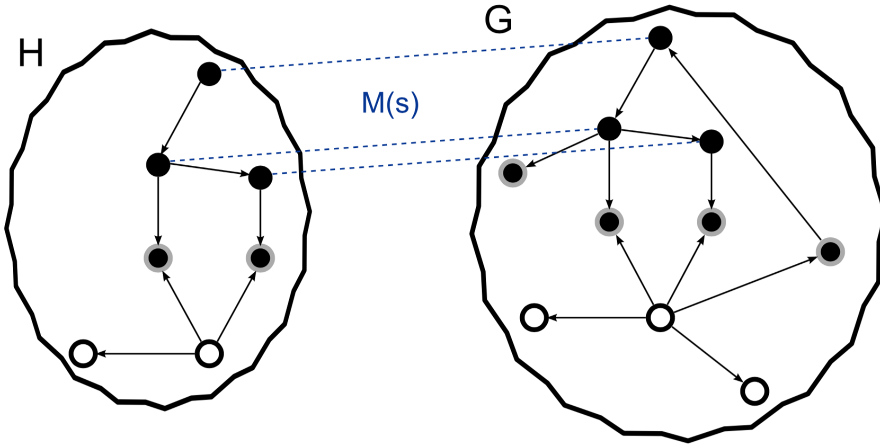


Figure 11: Partial set from the pruning technique of VF2 [7]

Each step of the depth-first search, the algorithm will choose a candidate pair $p = (v_p, w_p)$ from the ordered list $P(s)$, which contains all candidate pairs. The order implemented by VF2 gives priority to the vertices in N_H^{out} and N_G^{out}, then in N_H^{in} and N_G^{in} and lastly (only if for unconnected graphs) in $\overline{V}_H(s)$ and $\overline{V}_G(s)$. The *feasibility test* (see section 5.7.1) will decide if the search-state gets augmented by the chosen candidate pair p. When the feasibility test fails, the algorithm backtracks to the previous state s and tries another candidate. When $M(s)$ covers all the vertices of H, a match is found and the search stops, otherwise the search will continue until all the candidate pairs of $P(s)$ have been tried.

[7]Image from [14] http://www.cs.mcgill.ca/∼esyria/publications/dissertation.pdf

The preferred order is implemented by first executing the *preferred* function for the vertices connected with outgoing edges, then via the incoming edges and finally by calling the *leastPreferred* function, which represent the not connected edges. (The functions can be found under the *matchVF2* function in the *patternMatching.py* file[1].)

5.7.1. Feasibility test

The test runs on $s' = s \cup p$ tests the following three criteria in the mentioned order:

1. check if the new mapping $M(s')$ is a valid isomorphism (by checking if the edges between v_p, its adjacent vertices in $M_H(s')$ and the edges between wp_p and its adjacent vertices in $M_G(s')$ correspond,
2. check if the number of external edges between $M_H(s')$ and $N_H(s')$ is less than or equal to the number of external edges between $M_G(s')$ and $N_G(s')$,
3. check if the number of external edges between $N_H(s')$ and $\overline{V}_H(s')$ is less than or equal to the number of external edges between $N_G(s')$ and $\overline{V}_G(s')$.

Doing this allows VF2 to reduce the search space and ensures that no incompatibilities will occur in the future search steps.

The feasibility test is implemented in the feasibilityTest function under the *Ullmann* function in the *patternGraph.py* file[1]. We do take a small coding short-cut by reusing the naive implementation to check if there is a valid mapping between the matched vertices and the adjacent vertices.

5.7.2. Metaphorical

The algorithm effectively matches one vertex. It then stepwise expands the match with vertices of the highest priority (first directly connected via outgoing edges, to the already matched sub-graph, then directly connected via incoming edges and lastly the not, to the current mapping, connected vertices). The feasibility test is an optimization that tries to detect as soon as possible that the current search branch will not result in a valid match.

5.7.3. VF2 Efficiency and Extension Options

Experimental results [16] show that for larger graphs VF2 preforms better than Ullmann. The time complexity of VF2 in the best case is $\mathcal{O}(N^2)$, as $N = |V_H| + |V_G|$ search state will be visited. In the worst case there are $N!$ search states, resulting in a time complexity of $\mathcal{O}(N!N)$. For both cases, VF2 is a linear order of magnitude more efficient than Ullmann. Another advantage of VF2 is that its spatial complexity is linear, while Ullmann's is cubic (because of the adjacency matrices).

The main difference between Ullmann and VF2 is within the backtracking step. While Ullmann only compares pairs of adjacent vertices, VF2 compares the vertex with its neighbourhood. Furthermore, Ullmann's $\mathbf{M^*}$ matrix verifies if the pair of vertices, in the match, is semantically compatible, while VF2's feasibility test ensures a correct structure of the match.

As VF2 and Ullmann focus on different optimizations, one can combine [17] the two algorithms for lowering the time complexity.

6. Ullmann vs VF2

6.1. Depth First Search

Ullmann first build a $\mathbf{M}^*$ matrix that stores the possible mappings from vertices from the pattern to vertices from the graph. As it incorporated the degree of the vertices, vertices from the graph that have less outgoing and/or incoming edges than the vertex from the pattern are excluded from its possible mapping and will not be used to try and map it. (This does not take into account the semantic attributes of the edges.)

VF2 on the other hand does not build such a matrix but verifies this when trying to create a mapping from the pattern vertex to the graph vertex. This does take into account the semantic attributes of the edges.

6.2. Matching order

In the matching phase, Ullmann sorts the vertices based on its amount of edges, creating a total order. It preference the vertex with the most edges. The reason behind this is that the vertices with more edges will fail sooner, thus resulting in increased efficiency.

VF2 preference is not so specific, it prefers the adjacent vertices of the currently matched vertices connected via outgoing edges, then the ones that are connected via incoming edges and finally the not connected vertices[8].

6.3. Pruning the search space

Ullmann's refinement procedure test whether all remaining (currently viewed as) possible mappings (from $\mathbf{M}^*$) their neighbours can be correctly matched. If not, it removes it from $\mathbf{M}^*$. This means that Ullmann's refinement procedure not only can stop the current branch, but also limit future possible mappings.

Contrary to Ullmann, VF2 can only stop the current branch and backtrack.

7. Generating and visualizing graphs, patterns and found occurences

The implementation includes a *GraphGenerator* function in the *generator.py* file that can generate random graphs and random patterns for that graph. It is called in the *main.py* file. The *printGraph* function in the *graphToDot.py* file saves your current host graph and pattern to ".dot" files. Which can be visualized by GraphViz[9].

If you call the *printGraph* function with the matched vertices and edges, it will highlight the occurrence in the visualization.

[8] Combining the two might lead to even greater efficiency.

[9] `http://graphviz.org`

8. Conclusion

In this paper we have seen the two main different categories for efficient model transformations (for pattern matching and solving the sub-graph homomorphism problem): search plans and constraint satisfaction problems (CSP's). We first explained search plans and presented the workings of such an algorithm. We also listed some possible extensions for improving its performance. We then explained the definition of CSP's and their use cases. Lastly we presented the workings of well-known and some of the most efficient algorithms for pattern matching (and solving the sub-graph homomorphism problem) as CSP's, namely Ullmann and VF2. In both cases we again listed some possible extensions for improving their performance.

References

[1] H. Bunke, T. Glauser, T.-H. Tran, An efficient implementation of graph grammars based on the rete matching algorithm, in: Graph Grammars and Their Application to Computer Science, Springer, 1991, pp. 174–189.

[2] M. Kurt, Graph algorithms and np-completeness (1984).

[3] G. V. Batz, M. Kroll, R. Geiß, A first experimental evaluation of search plan driven graph pattern matching, in: Applications of Graph Transformations with Industrial Relevance, Springer, 2008, pp. 471–486.

[4] J. Edmonds, Optimum branchings, Journal of Research of the National Bureau of Standards B 71 (4) (1967) 233–240.

[5] A. Zündorf, Graph pattern matching in progres, in: Graph Grammars and Their Application to Computer Science, Springer, 1996, pp. 454–468.

[6] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. Szalkowski, Grgen: A fast spo-based graph rewriting tool, in: Graph Transformations, Springer, 2006, pp. 383–397.

[7] G. Varró, K. Friedl, D. Varró, Adaptive graph pattern matching for model transformations using model-sensitive search plans, Electronic Notes in Theoretical Computer Science 152 (2006) 191–205.

[8] C. Â. G. Gomes, A framework for efficient model transformations.

[9] J. Larrosa, G. Valiente, Graph pattern matching using constraint satisfaction, in: Proc. Joint APPLIGRAPH/GETGRATS Worksh. Graph Transformation Systems, 2000, pp. 189–196.

[10] O. Angelsmark, Constructing algorithms for constraint satisfaction and related problems: Methods and applications.

[11] M. Rudolf, Utilizing constraint satisfaction techniques for efficient graph pattern matching, in: Theory and Application of Graph Transformations, Springer, 2000, pp. 238–251.

[12] E. B. Krissinel, K. Henrick, Common subgraph isomorphism detection by backtracking search, Software: Practice and Experience 34 (6) (2004) 591–607.

[13] J. R. Ullmann, An algorithm for subgraph isomorphism, Journal of the ACM (JACM) 23 (1) (1976) 31–42.

[14] E. Syriani, A multi-paradigm foundation for model transformation language engineering, Ph.D. thesis, McGill University (2011).

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

[16] P. Foggia, C. Sansone, M. Vento, A database of graphs for isomorphism and sub-graph isomorphism benchmarking, in: Proc. of the 3rd IAPR TC-15 International Workshop on Graph-based Representations, 2001, pp. 176–187.

[17] M. Provost, Himesis: A hierarchical subgraph matching kernel for model driven development, in: Masters Abstracts International, Vol. 45, 2006.