32nd International Workshop on Unification (UNIF 2018), Informal Proceedings
Oxford, 7th July, 2018

Proximity-Based Generalization*

Temur Kutsia and Cleo Pau

RISC, Johannes Kepler University Linz, Austria
{kutsia,ipau}@risc.jku.at

Abstract

We study anti-unification under the condition that some mismatches in the names of function symbols is tolerated. The mismatches are expressed by proximity relations, which are reflexive, symmetric, but not transitive fuzzy binary relations. Their crisp version corresponds to undirected graphs. Computing all maximal clique partitions in them is needed to compute least general generalizations with respect to proximity relations. We report about our progress in developing both all-clique-partitions and anti-unification algorithms.

1 Introduction

In this paper we study anti-unification under the condition that some mismatches between function symbol names is tolerated. Names which are 'close to each other' should not be distinguished. Such a treatment of symbols is adequate in situations where one has to manage imprecise information, for example, for detecting clones in copied and slightly modified software code.

The relation of 'being close' is not transitive: For instance, if one considers two cities being close to each other if the distance between them is not more than 200 km, then Salzburg is close to Linz (133 km) and Linz is close to Vienna (185 km), but Salzburg is not close to Vienna (300 km). Nontransitivity has to be dealt in a special way. Proximity relations (reflexive symmetric fuzzy binary relations) characterize the notion of 'being close' numerically. They become crisp once we fix the threshold from which on the distance between the objects can be called 'close'.

We consider a first-order language where function symbols are in a proximity relation with each other. To compute generalizations in this language, we assume that the threshold (so called λ-cut) is fixed and we know which symbols are close to each other and which are not. The obtained crisp relation can be represented as an undirected graph, and symbols that belong to a complete subgraph (a clique) of it should be considered 'close' to each other. But the same symbol might belong to two or more cliques, and we need to choose one of them to which the symbol is assigned. Hence, we essentially need to partition the graph into maximal cliques, treat the symbols in the same clique as being the same, and apply the first-order anti-unification algorithm to compute generalizations.

This approach leads to two problems. First, while one can easily compute one maximal vertex-clique partition of a graph [7, 4], computing all maximal ones is more involved. But we need them to compute least general generalizations. We were not able to find an appropriate algorithm in the literature and, therefore, decided to design and implement one ourselves. It is optimal in the sense that each maximal

[*] Supported by Austrian Science Fund (FWF) under the project P 28789-N32

clique partition is computed only once, and generating and discarding false answers is avoided. We briefly describe the algorithm in this paper.

The second problem is related to the anti-unification algorithm. Computing all maximal clique partitions of the given proximity relation at the beginning and then switching to the standard anti-unification would be an overkill, because many non-minimal answers will be computed, or the same answers might be returned many times. Therefore, it is more reasonable to incorporate the clique partitioning procedure into the anti-unification algorithm and perform partitions only on demand. The algorithm described in this paper works in this way.

In related work, Aït-Kaci and Pasi [1] studied anti-unification with respect to similarity relations, which differ from proximity relations by being transitive and, in this way, are more specific. On the other hand, they allow arity mismatch between symbols, which we do not consider in this paper. Unification with proximity relations has been studied in [6], from which we take the basic definitions.

This work is still in progress. We are currently in the process of a detailed investigation of the properties of presented algorithms, and implementing anti-unification. The latter, after implementation, will be included into our online open-source library of unification and anti-unification algorithms [3].

2 Terms, Substitutions, Proximity Relations

We consider first-order terms defined as usual: $t := x \mid f(t_1,\dots,t_n)$, where x is a variable and f is an n-ary function symbol with $n \geq 0$. We use the letters f,g,h,a,b,c,d and e for function symbols, x,y,z,u,v and w for variables, and s,t,l, and r for terms.

For a term t, its *set of positions* $pos(t)$ is a set of sequences of positive integers defined as follows: If t is a variable, then $pos(t) = \{\epsilon\}$, where ϵ is the empty sequence; If $t = f(s_1,\dots,s_n)$, then $pos(t) = \{\epsilon\} \cup \bigcup_{i=1}^{n}\{i.p \mid p \in pos(s_i)\}$. By $t[p]$ we denote the symbol in t at position p. The notation $t|_p$ denotes the subterm of t at position p.

A *substitution* is a mapping from variables to terms, which is the identity almost everywhere. We will use the traditional finite set representation of substitutions. The lower case Greek letters are used to denote substitutions, with the exception of the identity substitution for which we write Id. The related notions such as substitution application, term instance, substitution composition, etc. are defined in the usual way, see, e.g. [2].

A binary *fuzzy relation* on a set S is a fuzzy subset on $S \times S$, that is, a mapping from $S \times S$ to the real interval $[0,1]$. If $\mathcal{R}$ is a fuzzy relation on S and λ is a number $0 \leq \lambda \leq 1$, then the λ-*cut* of $\mathcal{R}$ on S, denoted $\mathcal{R}_\lambda$, is an ordinary (crisp) relation on S defined as $\mathcal{R}_\lambda := \{(x,y) \mid \mathcal{R}(x,y) \geq \lambda\}$.

A fuzzy relation $\mathcal{R}$ on a set S is called a *proximity relation* on S iff it is reflexive, i.e., $\mathcal{R}(x,x) = 1$ for all $x \in S$, and symmetric, i.e., $\mathcal{R}(x,y) = \mathcal{R}(y,x)$ for all $x,y \in S$. A special class of proximity relations are *similarity relations,* which are *transitive* proximity relations: $\mathcal{R}(x,z) \geq \min(\mathcal{R}(x,y), \mathcal{R}(y,z))$ for all $x,y,z \in S$.

A cut value of a proximity relation $\mathcal{R}$ on S is a number λ such that $\mathcal{R}(x,y) = \lambda$ for some $x,y \in S$. The set of cut values of $\mathcal{R}$ are called approximation levels of $\mathcal{R}$.

Given a proximity relation $\mathcal{R}$ on a set S and $\lambda \in [0,1]$, a *proximity block of level* λ (or, shortly, a λ-*block*) is a subset B of S such that the restriction of $\mathcal{R}_\lambda$ to B is a total relation, and B is maximal with this property.

Below we consider proximity relations defined on the set of variables and function symbols and assume that no variable is close to any function symbol and to any other variable. Hence, variables always belong to singleton blocks of level 1. Also, function symbols of different arities are not close to each other, i.e., each block consists of symbols of the same arity.

The notion of proximity is defined for terms. The intuition behind it, according to [6], is based on the following idea: two terms t_1 and t_2 are λ-approximate when they have the same set of positions;

their symbols, in their corresponding positions, belong to the same λ-block; and a certain symbol is always assigned to the same λ-block (throughout a computation). The following definition formalizes this intuition:

Definition 1. Given a proximity relation $\mathcal{R}$ on $\mathcal{F}$ and $\lambda \in [0,1]$, two terms t and s are λ-*approximate* (or λ-*close*) with respect to $\mathcal{R}$, written $t \approx_{\mathcal{R},\lambda} s$, if the following conditions hold:

1. $pos(t) = pos(s)$, i.e, the terms have exactly the same positions, hence, the same structure.
2. For all positions $p \in pos(t)$, $t[p]$ and $s[p]$ belong to the same λ-block of the relation $\mathcal{R}$.
3. For all positions $p, p' \in pos(t)$ with $p \neq p'$,
   (a) If $t[p] = t[p']$, then $s[p]$ and $s[p']$ belong to the same λ-block of $\mathcal{R}$.
   (b) If $s[p] = s[p']$, then $t[p]$ and $t[p']$ belong to the same λ-block of $\mathcal{R}$.

When $\mathcal{R}$ is clear from the context, we will just write $t \approx_\lambda s$. The relation $\preceq_{\mathcal{R},\lambda}$ modifies its classical counterpart, the instantiation quasi-ordering $\preceq$ (see, e.g., [2]) by replacing equality with $\approx_{\mathcal{R},\lambda}$: A term t is more general than s at level λ with respect to $\mathcal{R}$, written $t \preceq_{\mathcal{R},\lambda} s$, iff there exists a substitution φ such that $t\varphi \approx_{\mathcal{R},\lambda} s$.

The strict part of the $\preceq_{\mathcal{R},\lambda}$ relation is denoted by $\prec_{\mathcal{R},\lambda}$. The fact that $\approx_{\mathcal{R},\lambda}$ is not a transitive relation implies that $\preceq_{\mathcal{R},\lambda}$ is not a quasi-ordering.

Given a proximity relation $\mathcal{R}$ and a cut λ, a term r is a common $(\lambda,\mathcal{R})$-generalization of t and s iff it is more general than both t and s at level λ with respect to $\mathcal{R}$. It is a least general common $(\lambda,\mathcal{R})$-generalization ($(\lambda,\mathcal{R})$-lgg) of t and s iff it is a common $(\lambda,\mathcal{R})$-generalization of t and s and there is no term l such that $l \prec_{\mathcal{R},\lambda} r$ holds.

Below we assume that λ is fixed and, hence, consider crisp version of proximity relations. Such a relation can be represented as undirected graph, whose maximal cliques (maximal complete subgraphs) are counterparts to blocks. The goal is to design an algorithm which computes $\mathcal{R}$-lggs for a given pair of terms and the proximity relation $\mathcal{R}$. We do not distinguish between $\mathcal{R}$ and the graph that represents it.

Example 1. If (a,b) and (a,c) both belong to $\mathcal{R}$ but (b,c) does not, then $f(a,a)$ and $f(b,b)$ are close to each other, but $f(a,a)$ and $f(b,c)$ are not. The latter pair has two $\mathcal{R}$-lggs: $f(a,x)$ and $f(x,a)$.

A *clique* in an undirected graph $G = (V,E)$ is a set of vertices $W \subseteq V$ such that for each pair of vertices in W there is an edge in E. A clique is *maximal* if it is not a proper subset of another clique. A *clique partition* of a graph G is a set of its cliques $\{C_1,\dots,C_n\}$ such that $\bigcup_{i=1}^n C_i = V$ and $C_i \cap C_j = \emptyset$ for all $1 \le i,j \le n$, $i \neq j$.

Let $S_1 = \{C_1,\dots,C_n\}$ and $S_2 = \{D_1,\dots,D_m\}$ be two sets of cliques of the same graph. We say that S_1 is *subsumed* by S_2, written $S_1 \sqsubseteq S_2$, iff for all $1 \le i \le n$ there exists $1 \le j \le m$ such that $C_i \subseteq D_j$. If S_1 and S_2 are, in particular, partitions, then we also say that S_1 is a *subpartition* of S_2 if $S_1 \sqsubseteq S_2$. A clique partition of a graph is *maximal* if it is not properly subsumed by any other partition of the graph. A graph may have several maximal clique partitions. We will use them in the anti-unification algorithm in Sect. 3. In Sect. 4 we discuss an algorithm that computes all maximal clique partitions in a graph.

3 The Proximity-Based Anti-Unification Algorithm

Our anti-unification algorithm works on tuples $A; C; S; \mathcal{R}; G$, called configurations. Here A, C, and S are sets of anti-unification triples (AUTs, constructions of the form $x : s \triangleq t$, meaning that x is a variable that generalizes s and t), $\mathcal{R}$ is a crisp version of a proximity relation, and G is a term. The rules transform configurations into configurations. Intuitively, the problem set A contains AUTs that have not been solved yet, the set C contains AUTs of the form $x : a \triangleq b$, where a and b are constants such that $(a,b) \in \mathcal{R}$ and the AUTs are not solved yet. The store S contains the already solved AUTs, $\mathcal{R}$ is the proximity relation which gets more and more refined during computation by identifying symbols that belong to the same clique in some partition of $\mathcal{R}$, and G is the generalization which becomes more and more specific as the algorithm progresses by applying the rules.

Dec: **Decomposition**

$$\{x_1 : f_1(s^1_1,\dots,s^1_{k_1}) \triangleq g_1(t^1_1,\dots,t^1_{k_1}),\dots,x_n : f_n(s^n_1,\dots,s^n_{k_n}) \triangleq g_n(t^n_1,\dots,t^n_{k_n})\};\ C;\ S;\ \mathcal{R};\ G \Longrightarrow$$
$$\{y^j_1 : s^j_1 \triangleq t^j_1,\dots,y^j_{k_j} : s^j_{k_j} \triangleq t^j_{k_j} \mid 1 \le j \le m\};\ C;$$
$$\{x_j : f_j(s^j_1,\dots,s^j_{k_j}) \triangleq g_j(t^j_1,\dots,t^j_{k_j}) \mid m+1 \le j \le n\} \cup S;\ \mathcal{R}';\ G\vartheta,$$

where (a) $k_i > 0$ and $(f_i,g_i) \in \mathcal{R}$ for all $1 \le i \le n$; (b) there exist a maximal vertex-clique partition P of the subrelation $\mathcal{Q} = \{(f_1,g_1),\dots,(f_n,g_n)\} \subseteq \mathcal{R}$ and the index $1 \le m \le n$ such that for each (f_j,g_j), $1 \le j \le m$, there is a clique $Cl \in P$ with $f_j, g_j \in Cl$, and for no (f_j,g_j), $m+1 \le j \le n$ there is such a clique; (c) $\mathcal{R}'$ is obtained from $\mathcal{R}$ by replacing the subrelation $\mathcal{Q}$ by its partition P; (d) $\vartheta = \{x_j \mapsto f_j(y^j_1,\dots,y^j_{k_j}) \mid 1 \le j \le m\}$.

Sol: **Solve**

$$\{x : f(\tilde{s}) \triangleq g(\tilde{t})\} \uplus A;\ C;\ S;\ \mathcal{R};\ G \Longrightarrow A;\ C; \{x : f(\tilde{s}) \triangleq g(\tilde{t})\} \cup S;\ \mathcal{R};\ G, \qquad \text{if } (f,g) \notin \mathcal{R}.$$

Post: **Postpone**

$$\{x : a \triangleq b\} \uplus A;\ C;\ S;\ \mathcal{R};\ G \Longrightarrow A;\ \{x : a \triangleq b\} \cup C; S;\ \mathcal{R};\ G, \qquad \text{if } (a,b) \in \mathcal{R}.$$

Gen-Con: **Generalize Constants**

$$\varnothing; \{x_1 : a_1 \triangleq b_1,\dots,x_n : a_n \triangleq b_n\};\ S;\ \mathcal{R};\ G \Longrightarrow \varnothing;\ \varnothing;\ \{x_j : a_j \triangleq b_j \mid m+1 \le j \le n\} \cup S;\ \mathcal{R}';\ G\vartheta,$$

where (a) $(a_i,b_i) \in \mathcal{R}$ for all $1 \le i \le n$; (b) there exist a maximal vertex-clique partition P of the subrelation $\mathcal{Q} = \{(a_1,b_1),\dots,(a_n,b_n)\} \subseteq \mathcal{R}$ and the index $1 \le m \le n$ such that for each (a_j,b_j), $1 \le j \le m$ there is a clique $Cl \in P$ with $a_j, b_j \in Cl$, and for no (a_j,b_j), $m+1 \le j \le n$ there is such a clique; (c) $\mathcal{R}'$ is obtained from $\mathcal{R}$ by replacing $\mathcal{Q}$ by its partition P; (d) $\vartheta = \{x_i \mapsto a_i \mid 1 \le i \le m\}$.

To anti-unify two terms s and t with respect to the proximity relation $\mathcal{R}$, we create the initial tuple $\{x : s \triangleq t\}; \varnothing; \varnothing; \mathcal{R}; x$ and apply the rules in all ways as long as possible. In the search space, branching is caused by all possible maximal clique partitions in Dec and Gen-Con. Generalizations in successful branches form the computed result. We call this algorithm $\mathsf{PR\text{-}AU_{lin}}$. The subscript lin indicates that it computes linear generalizations (i.e., those in which each generalization variable appears at most once).

Theorem 1. $\mathsf{PR\text{-}AU_{lin}}$ terminates and computes a minimal complete set of linear generalizations.

Proof sketch. Termination follows form the fact that Gen-Con can be applied only once and the other rules strictly reduce the number of symbols in A. In computed answers, no generalization variable appears more than once, because there is no rule that would merge them. Hence, computed generalizations are linear. They are also lggs among linear generalizations, because (a) the algorithm decomposes the terms as much as possible, and (b) it maximizes the number of nonvariable subterms appearing in generalizations, which is done with the help of clique partitions of subrelations (not of the entire relation!) at each decomposition and constant generalization steps. All linear lggs are computed, because branching at Dec and Gen-Con rules explores all maximal clique partitions. □

Example 2. For terms $f(g_1(g_2(a)), g_2(a), a)$ and $f(g_2(g_3(b)), g_3(c), b)$ and the relation $\mathcal{R}$ given in the form of maximal clique set (not a partition) $\{\{f\}, \{g_1, g_2\}, \{g_2, g_3\}, \{a, b\}, \{b, c\}\}$, the algorithm $\mathsf{PR\text{-}AU_{lin}}$ returns two $\mathcal{R}$-lggs: $f(g_1(z_1), y_2, a)$ and $f(y_1, g(y_2), a)$ and misses the nonlinear one $f(g_1(y_2), y_2, y_3)$. We illustrate now how the algorithm works:

$$\begin{aligned}
&\{x : f(g_1(g_2(a)), g_2(a), a) \triangleq f(g_2(g_3(b)), g_3(c), b)\};\ \varnothing;\ \varnothing;\ \mathcal{R};\ x \Longrightarrow_{\mathsf{Dec}}\\
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c),\ y_3 : a \triangleq b\};\ \varnothing;\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Post}}\\
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3).
\end{aligned}$$

At this stage, the subrelation $\{(g_1, g_2), (g_2, g_3)\}$ of $\mathcal{R}$ can be partitioned in two ways, which gives two new relations $\mathcal{R}_1 = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a, b\}, \{b, c\}\}$ and $\mathcal{R}_2 = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a, b\}, \{b, c\}\}$. Therefore, we can use the Dec rule and proceed in two different ways:

Alternative 1. Proceeding by $\mathcal{R}_1$.

$$\begin{aligned}
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_1 : g_2(a) \triangleq g_3(b)\};\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_{11};\ f(g_1(z_1), y_2, a).
\end{aligned}$$

where $\mathcal{R}_{11} = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a, b\}, \{c\}\}$. Note that if we required in the condition of the Gen-Con rule to partition the relation itself (instead of its subrelation), we would get also $\mathcal{R}_{12} = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a\}, \{b, c\}\}$, which would lead to another successful branch

$$\begin{aligned}
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c), z_1 : g_2(a) \triangleq g_3(b), y_3 : a \triangleq b\};\ \mathcal{R}_{12};\ f(g_1(z_1), y_2, y_3).
\end{aligned}$$

However, the computed generalization is not an lgg, since it is more general than the previous one.

Alternative 2. Proceeding by $\mathcal{R}_2$.

$$\begin{aligned}
&\{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)),\ y_2 : g_2(a) \triangleq g_3(c)\};\ \{y_3 : a \triangleq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_2 : a \triangleq c\};\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b))\};\ \mathcal{R}_2;\ f(y_1, g(z_2), y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)), z_2 : a \triangleq c\};\ \mathcal{R}_2;\ f(y_1, g(y_2), y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c)\};\ \mathcal{R}_{21};\ f(y_1, g(y_2), a),
\end{aligned}$$

where $\mathcal{R}_{21} = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a, b\}, \{c\}\}$. Again, if we were allowed to partition the whole $\mathcal{R}_2$ in Gen-Con, we would get another partition $\mathcal{R}_{22} = \{\{f\}, \{g_1\}, \{g_2, g_3\}, \{a\}, \{b, c\}\}$, which would give the following successful branch:

$$\begin{aligned}
&\varnothing;\ \{y_3 : a \triangleq b\};\ \{y_1 : g_1(g_2(a)) \triangleq g_2(g_3(b)), z_2 : a \triangleq c\};\ \mathcal{R}_2;\ f(y_1, g(y_2), y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \triangleq g_3(c), y_3 : a \triangleq b\};\ \mathcal{R}_{22};\ f(y_1, g(y_2), y_3).
\end{aligned}$$

However, the solution obtained in this branch is more general than the previous one.

As the next step, we extend the algorithm to add a rule for merging variables. It uses a partial function $\mathit{refine}(\{s_1 \approx t_1, \ldots, s_n \approx t_n\}, \mathcal{R})$, which is supposed to refine the given relation $\mathcal{R}$ into a new relation $\mathcal{R}'$ so that $s_i \approx_{\mathcal{R}'} t_i$, $1 \le i \le n$, if such a refinement of $\mathcal{R}$ exists.

Mer: **Merge**

$A;\ C;\ \{x : s_1 \doteq t_1,\ y : s_2 \doteq t_2\} \uplus S;\ \mathcal{R};\ G \Longrightarrow A;\ C;\ \{x : s_1 \doteq t_1\} \cup S;\ \mathcal{R}';\ G\{y \mapsto x\}$,
where $\mathcal{R}' = \mathit{refine}(\{s_1 \approx s_2, t_1 \approx t_2\}, \mathcal{R})$.

The obtained algorithm is denoted by PR-AU. The function *refine* is defined as follows:

$$\mathit{refine}(\varnothing, \mathcal{R}) = \mathcal{R}. \qquad \mathit{refine}(\{t \approx t\} \uplus E, \mathcal{R}) = \mathit{refine}(E, \mathcal{R}).$$
$$\mathit{refine}(\{f(s_1, \dots, s_n) \approx g(t_1, \dots, t_n)\} \uplus E, \mathcal{R}) = \mathit{refine}(\{s_1 \approx t_1, \dots, s_n \approx t_n\} \cup E, \mathcal{R}'),$$

if $(f, g) \in \mathcal{R}$ and $\mathcal{R}' = \mathcal{R} \setminus (S_1 \cup S_2)$, where $S_1 = \{(f, h) \mid (g, h) \notin \mathcal{R}\} \cup \{(h, f) \mid (g, h) \notin \mathcal{R}\}$ and $S_2 = \{(g, h) \mid (f, h) \notin \mathcal{R}\} \cup \{(h, g) \mid (f, h) \notin \mathcal{R}\}$. Otherwise, *refine* is not defined.

From the specification of *refine* it follows that Mer is correct. It is also an alternative to the rules above, meaning that it would introduce additional branching, because of which PR-AU might recompute the same solution on different branches. For instance, in Example 2, when we proceed by $\mathcal{R}_1$, we get two branches that compute the same result (recall that $\mathcal{R}_{11} = \{\{f\}, \{g_1, g_2\}, \{g_3\}, \{a, b\}, \{c\}\}$):

$$\begin{aligned}
&\{y_1 : g_1(g_2(a)) \doteq g_2(g_3(b)),\ y_2 : g_2(a) \doteq g_3(c)\};\ \{y_3 : a \doteq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_1 : g_2(a) \doteq g_3(b)\};\ \{y_3 : a \doteq b\};\ \{y_2 : g_2(a) \doteq g_3(c)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\varnothing;\ \{y_3 : a \doteq b\};\ \{y_2 : g_2(a) \doteq g_3(c), z_1 : g_2(a) \doteq g_3(b)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \doteq g_3(c), z_1 : g_2(a) \doteq g_3(b)\};\ \mathcal{R}_{11};\ f(g_1(z_1), y_2, a) \Longrightarrow_{\mathsf{Mer}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \doteq g_3(c)\};\ \mathcal{R}_{11};\ f(g_1(y_2), y_2, a)
\end{aligned}$$

and

$$\begin{aligned}
&\{y_1 : g_1(g_2(a)) \doteq g_2(g_3(b)),\ y_2 : g_2(a) \doteq g_3(c)\};\ \{y_3 : a \doteq b\};\ \varnothing;\ \mathcal{R};\ f(y_1, y_2, y_3) \Longrightarrow_{\mathsf{Dec}}\\
&\{z_1 : g_2(a) \doteq g_3(b)\};\ \{y_3 : a \doteq b\};\ \{y_2 : g_2(a) \doteq g_3(c)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Sol}}\\
&\varnothing;\ \{y_3 : a \doteq b\};\ \{y_2 : g_2(a) \doteq g_3(c), z_1 : g_2(a) \doteq g_3(b)\};\ \mathcal{R}_1;\ f(g_1(z_1), y_2, y_3) \Longrightarrow_{\mathsf{Mer}}\\
&\varnothing;\ \{y_3 : a \doteq b\};\ \{y_2 : g_2(a) \doteq g_3(c)\};\ \mathcal{R}_1;\ f(g_1(y_2), y_2, y_3) \Longrightarrow_{\mathsf{Gen\text{-}Con}}\\
&\varnothing;\ \varnothing;\ \{y_2 : g_2(a) \doteq g_3(c)\};\ \mathcal{R}_{11};\ f(g_1(y_2), y_2, a).
\end{aligned}$$

However, we can not postpone Mer till the end, after A and C get empty (as it is usually done in anti-unification algorithms), because in this case we will miss solutions, as the example below shows.

Example 3. Let $\mathcal{R} = \{\{f\}, \{h_1\}, \{h_2\}, \{a\}, \{g_0, g_1\}, \{g_0, g_2\}\}$, $s = f(g_0(a), h_1(g_0(a)), h_1(g_0(a)))$, and $t = f(g_1(a), h_2(g_2(a)), h_2(g_0(a)))$. Then PR-AU computes two solutions: $f(g_0(a), y_2, y_3)$ and $f(y_1, y_2, y_2)$. The first one is obtained by applying Sol before Mer, and the second one in the other way around. However, if Mer is applied only at the very end, then the first solution is not computed.

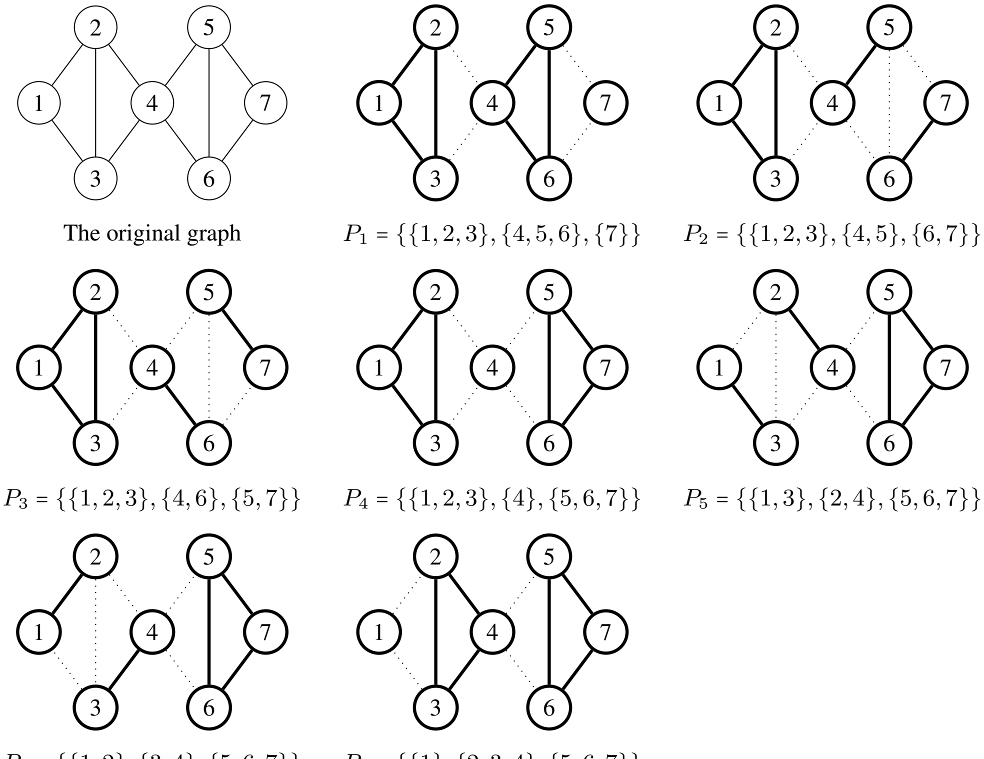
Merging variables can significantly increase the size of the computed set of generalizations:

Example 4. Let the arity of f be $n + 2$, $\mathcal{R} = \{\{f\}, \{h_1\}, \{h_2\}, \{a\}, \{g_0, g_1\}, \dots, \{g_0, g_n\}\}$ and

$$\begin{aligned}
s &= f(g_0(a),\ h_1(g_0(a)), \dots,\ h_1(g_0(a)),\ h_1(g_0(a))),\\
t &= f(g_1(a),\ h_2(g_2(a)), \dots,\ h_2(g_n(a)),\ h_2(g_0(a))).
\end{aligned}$$

$\mathsf{PR\text{-}AU_{lin}}$ computes only one generalization: $f(g_0(a), y_2, \dots, y_n, y_{n+1}$. With PR-AU, we have, in addition, $n - 1$ other generalizations: $f(y_1, y_2, \dots, y_n, y_2), \dots, f(y_1, y_2, \dots, y_n, y_n)$.

Theorem 2. PR-AU computes a minimal complete set of generalizations.



The original graph

$P_1 = \{\{1,2,3\},\{4,5,6\},\{7\}\}$

$P_2 = \{\{1,2,3\},\{4,5\},\{6,7\}\}$

$P_3 = \{\{1,2,3\},\{4,6\},\{5,7\}\}$

$P_4 = \{\{1,2,3\},\{4\},\{5,6,7\}\}$

$P_5 = \{\{1,3\},\{2,4\},\{5,6,7\}\}$

$P_5 = \{\{1,2\},\{3,4\},\{5,6,7\}\}$

$P_6 = \{\{1\},\{2,3,4\},\{5,6,7\}\}$

Figure 1: All maximal clique partitions of a graph.

4 Computing All Maximal Clique Partitions in a Graph

The anti-unification algorithm in the previous section relies on the computation of all maximal clique partitions in an undirected graph. We describe the corresponding algorithm here.

First, we compute all maximal cliques of the given graph and give each of them a name. All maximal cliques can be computed, e.g., by Bron-Kerbosch algorithm [5]. For the graph in Fig. 1, there are four of them: $C_1 = \{1,2,3\}, C_2 = \{2,3,4\}, C_3 = \{4,5,6\}, C_4 = \{5,6,7\}$. These cliques will get revised during computation by removing elements from them. At the end, we report those which are not empty.

After computing the initial cliques, we collect all shared vertices and indicate among which cliques they are shared. In the graph in Fig. 1, the shared vertices are 2, 3, 4, 5, and 6. We have $2 \in C_1 \cap C_2$, $3 \in C_1 \cap C_2$, $4 \in C_2 \cap C_3$, $5 \in C_3 \cap C_4$, and $6 \in C_3 \cap C_4$.

Our goal is to compute each solution exactly once. At the end, it can happen that some cliques consist of shared vertices only. However, such cliques can have any of the names of the original cliques they originate from. For instance, the node 4 alone can form a clique either as C_2 or C_3, giving two identical partitions which differ only by the clique names:

$$\begin{array}{llll} C_1 = \{1,2,3\}, & C_2 = \{4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\}, \\ C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4\}, & C_4 = \{5,6,7\}. \end{array}$$

We want to avoid such duplicates. Therefore, for such alternatives we choose one single clique to which they belong in this configuration, and forbid the others. For the example graph in Fig. 1, we can allow the vertices 2 and/or 3 to form a clique as C_2, the vertex 4 to form a clique as C_3, and the vertices 5 and/or 6 to form a clique as C_4. (Note that allowing does not necessarily mean that we will get result cliques of that form.) Thus, the forbidden configurations are $C_1 \neq \{2\}, C_1 \neq \{3\}, C_1 \neq \{2,3\}, C_2 \neq \{4\}, C_3 \neq \{5\}, C_3 \neq \{6\}, C_3 \neq \{5,6\}$.

Starting from the initial set of cliques, our algorithm All-Maximal-Clique-Partitions performs the following steps:

1. Compute the set of shared vertices and the forbidden configurations.
2. If the set of shared vertices is empty, return the current set of cliques and stop.
3. Select a shared vertex and nondeterministically assign it to one of the cliques it belongs to. Remove the vertex from the other cliques and from the set of shared vertices.
4. For each pair of cliques C_i, C_j, where $C_i \subseteq C_j$, make C_i empty and adjust the set of shared elements. In addition, if C_i was the chosen clique for the shared elements, remove those elements from the forbidden list of C_j.
5. If the union of two nonempty cliques is a subset of an original clique, or if a forbidden configuration arises, stop the development of this branch with failure. Otherwise go to step 2.

Checking for the subset relations is needed to avoid computing cliques which are not maximal. For instance, the partition $C_1 = \{1,2\}$, $C_2 = \{3\}$, $C_3 = \{4\}$, $C_4 = \{5,6,7\}$ should be rejected because $\{1,2\} \cup \{3\}$ is a subset of the original C_1 clique. Step 5 helps to detect such situations early.

The partitions shown in Fig. 1 correspond to the following final values of cliques, computed by the All-Maximal-Clique-Partitions algorithm:

$$\begin{array}{lllll}
P_1: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,5,6\}, & C_4 = \{7\} \\
P_2: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,5\}, & C_4 = \{6,7\} \\
P_3: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4,6\}, & C_4 = \{5,7\} \\
P_4: & C_1 = \{1,2,3\}, & C_2 = \varnothing, & C_3 = \{4\}, & C_4 = \{5,6,7\} \\
P_5: & C_1 = \{1,2\}, & C_2 = \{2,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\} \\
P_6: & C_1 = \{1,2\}, & C_2 = \{3,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\} \\
P_7: & C_1 = \{1\}, & C_2 = \{2,3,4\}, & C_3 = \varnothing, & C_4 = \{5,6,7\}.
\end{array}$$

5 Conclusion

We designed and implemented an algorithm to compute all maximal vertex-clique partitions in an undirected graph and used it in the computation of proximity-based least general generalizations. The next steps are to study the properties of both algorithms in detail and to implement the one for anti-unification. A more remote goal is to study applicability of proximity-based anti-unification in program analysis and clone detection.

References

[1] H. Aït-Kaci and G. Pasi. Lattice operations on terms with fuzzy signatures. *CoRR*, abs/1709.00964, 2017.

[2] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[3] A. Baumgartner and T. Kutsia. A library of anti-unification algorithms. In E. Fermé and J. Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557. Springer, 2014.

[4] J. Bhasker and T. Samad. The clique-partitioning problem. *Computers and Mathematics with Applications*, 22(6):1–11, 1991.

[5] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[6] P. J. Iranzo and C. Rubio-Manzano. Proximity-based unification theory. *Fuzzy Sets and Systems*, 262:21–43, 2015.

[7] C.-J. Tseng. *Automated synthesis of data paths in digital systems*. PhD thesis, Dept. of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburg, PA, 1984.