SAT-Based Algorithms for Regular Graph Pattern Matching

Miguel Terra-Neves[1], José Amaral[1], Alexandre Lemos[1], Rui Quintino[1], Pedro Resende[2], Antonio Alegria[1]

[1]OutSystems
[2]Zharta
miguel.neves@outsystems.com, jose.francisco.amaral@outsystems.com, alexandre.lemos@outsystems.com, rui.quintino@outsystems.com, pt.resende@icloud.com, antonio.alegria@outsystems.com

Abstract

Graph matching is a fundamental problem in pattern recognition, with many applications such as software analysis and computational biology. One well-known type of graph matching problem is graph isomorphism, which consists of deciding if two graphs are identical. Despite its usefulness, the properties that one may check using graph isomorphism are rather limited, since it only allows strict equality checks between two graphs. For example, it does not allow one to check complex structural properties such as if the target graph is an arbitrary length sequence followed by an arbitrary size loop.

We propose a generalization of graph isomorphism that allows one to check such properties through a declarative specification. This specification is given in the form of a Regular Graph Pattern (ReGaP), a special type of graph, inspired by regular expressions, that may contain wildcard nodes that represent arbitrary structures such as variable-sized sequences or subgraphs. We propose a SAT-based algorithm for checking if a target graph matches a given ReGaP. We also propose a preprocessing technique for improving the performance of the algorithm and evaluate it through an extensive experimental evaluation on benchmarks from the CodeSearchNet dataset.

1 Introduction

Pattern recognition is an important research area (Foggia, Percannella, and Vento 2014) due to its numerous applications ranging from detecting bad code patterns (Piotrowski and Madeyski 2020) and software analysis (Park et al. 2010; Singh et al. 2021; Zou et al. 2020) in general, to computational biology (Carletti, Foggia, and Vento 2013; Zaslavskiy, Bach, and Vert 2009). One fundamental problem in pattern recognition is graph matching (Livi and Rizzi 2013). Two common approaches are: (i) graph isomorphism (Cordella et al. 1999; Dahm et al. 2012; Ullmann 1976, 2010; Larrosa and Valiente 2002; Ullmann 2010; Zampelli, Deville, and Solnon 2010) and (ii) approximated graph matching (Bunke 1997; Raymond and Willett 2002; Sanfeliu and Fu 1983). The first consists of deciding if two graphs are identical, which can be too strict for some applications (Auwatanamongkol 2007; Conte et al. 2004). Approximated graph matching algorithms are less strict and normally employ some sort of distance metric to evaluate the graphs. Despite their usefulness, these approaches suffer from limitations. Neither of these allows one to check if a graph satisfies some specific complex structural properties, such as, for example, if some target graph contains two nested cycles since, given another similar reference graph that satisfies that property, one can increase their distance arbitrarily by adding extra nodes to, for example, the inner cycle.

Alternatively, regular-path queries (Cruz, Mendelzon, and Wood 1987) allow one to specify paths between nodes through a regular expression. Different formalisms for this type of query exist in the literature (Angles et al. 2018; Fan et al. 2012; Reutter, Romero, and Vardi 2017; Wang et al. 2020; Zhang et al. 2016; Libkin, Martens, and Vrgoc 2013). These are very expressive and useful, but do not allow one to check if a graph contains some complex subgraph structure. For example, it is not possible to specify a sequence of arbitrary nested loops with no external connections. Therefore, we propose Regular Graph Patterns (ReGaPs) as a generalization of graph isomorphism. The goal is to be able to define complex structural properties through a declarative specification in the form of a special graph. The proposed specification also borrows inspiration from regular expressions. This graph may contain special nodes, referred to as wildcards, representing arbitrary structures such as variable-sized sequences or subgraphs. These wildcards enable one to define compact representations of infinite sets of graphs.

The main contributions of this paper are three-fold: (i) a generalization of graph isomorphism matching in the form of ReGaP matching; (ii) a novel Boolean Satisfiability (SAT) encoding for the ReGaP matching problem; and (iii) a graph simplification technique for improving the performance of the SAT solver. The proposed solution is evaluated using control-flow graphs extracted from the Python code snippets in the CodeSearchNet dataset (Husain et al. 2020). The ReGaPs replicate the kind of bad code patterns that are integrated in the AI Mentor Studio (OutSystems 2023) code analysis engine for the OutSystems visual programming language. Note that, although the evaluation focuses on a specific use case, the concept and algorithm are generic, and thus may be applied in other contexts.

2 Background

In this section, we introduce the necessary background. We start with a brief introduction to graph isomorphism in Sec-

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

tion 2.1, followed by an explanation of SAT in Section 2.2.

2.1 Graph Isomorphism

Consider two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. G_1 and G_2 are isomorphic if and only if there exists a bijective mapping $f : V_1 \leftrightarrow V_2$ between the nodes of V_1 and V_2 such that for all $(u, v) \in E_1$, $(f(u), f(v)) \in E_2$ and vice-versa.

In an attributed graph $G = (V, E)$, each node has m attributes, denoted as $A_V = \{a_V^1, \ldots, a_V^m\}$. A node $v \in V$ is associated with an attribute vector $A_V(v) = \left[a_V^1(v), \ldots, a_V^m(v)\right]$, where $a_V^i(v)$ is the value of attribute a_V^i for node v. Similarly, each edge has n attributes, denoted as $A_E = \{a_E^1, \ldots, a_E^n\}$, and $A_E((u, v)) = \left[a_E^1((u, v)), \ldots, a_E^n((u, v))\right]$ is the attribute vector for edge (u, v). The definition of graph isomorphism must ensure consistency between all the attributes of a node/edge of G_1 and the respective equivalent in G_2, i.e. for all $u \in V_1$, $A_V(u) = A_V(f(u))$, and for all $(u, v) \in E_1$, $A_E((u, v)) = A_E((f(u), f(v)))$.

2.2 Boolean Satisfiability

Let X be a set of Boolean variables. A literal l is either a variable $x \in X$ or its negation $\overline{x}$. A clause c is a disjunction of literals $(l_1 \vee \cdots \vee l_k)$. A propositional logic formula F in Conjunctive Normal Form (CNF) is a conjunction of clauses $c_1 \wedge \cdots \wedge c_n$. A complete assignment $\alpha : X \rightarrow \{0, 1\}$ is a function that assigns a Boolean value to each variable in X. A literal x ($\overline{x}$) is satisfied by α if and only if $\alpha(x) = 1$ ($\alpha(x) = 0$). A clause c is satisfied by α if and only if at least one of its literals is satisfied. A CNF formula F is satisfied by α if and only if all of its clauses are satisfied. Given a CNF formula F, the SAT problem consists of deciding if there exists α which satisfies F. If so, then F is satisfiable and α is a model of F. Otherwise, F is unsatisfiable. Nowadays, most SAT solvers implement the conflict-driven clause learning algorithm (Audemard, Lagniez, and Simon 2013; Audemard and Simon 2009; Biere, Fleury, and Heisinger 2021; Liang et al. 2018; Marques-Silva and Sakallah 1996; Riveros 2021). Further details can be found in the literature (Biere et al. 2009).

3 Problem Definition

The ReGaP matching problem consists of determining if a given ReGaP $P = (V_P, E_P)$ matches some graph $G = (V, E)$. We assume that G is non-attributed for now. A ReGaP is a graph such that some of the nodes in V_P may be of a special type referred to as wildcard. We consider four wildcard types, inspired on regular expressions:

- **any-1+-sequence** (**any-0+-sequence**). Represents a directed path $v^1, \ldots, v^k$ of 1 (0) or more nodes such that, for each $i \in \{2..k\}$, the only edge in E towards v^i is (v^{i-1}, v^i). We use $W_P^{S+} \subseteq V_P$ ($W_P^{S*} \subseteq V_P$) to denote the set of all any-1+-sequence (any-0+-sequence) wildcards in V_P.
- **any-1+-subgraph** (**any-0+-subgraph**). Represents a subgraph of 1 (0) or more nodes. We use $W_P^{G+} \subseteq V_P$ ($W_P^{G*} \subseteq V_P$) to denote the set of all any-1+-subgraph (any-0+-subgraph) wildcards in V_P.

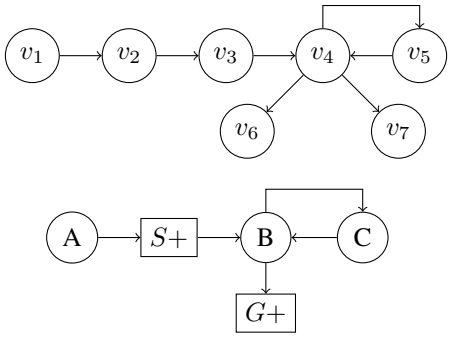

Figure 1: An example of a graph (top) and a ReGaP (bottom) that matches that graph. $S+$ represents an any-1+-sequence wildcard and $G+$ an any-1+-subgraph.

Example 1 *Figure 1 shows an example of a graph G and ReGaP P that matches G. P contains an any-1+-sequence wildcard $S+$ and an any-1+-subgraph wildcard $G+$.*

We use W_P^+ (W_P^*) to denote the set of all any-1+ (any-0+) wildcards, i.e. $W_P^+ = W_P^{S+} \cup W_P^{G+}$ ($W_P^* = W_P^{S*} \cup W_P^{G*}$), W_P^S (W_P^G) to denote the set of all sequence (subgraph) wildcards, i.e. $W_P^S = W_P^{S+} \cup W_P^{S*}$ ($W_P^G = W_P^{G+} \cup W_P^{G*}$), and W_P to denote the set of all wildcards, i.e. $W_P = W_P^S \cup W_P^G$.

The definition of matching between a ReGaP and a graph $G = (V, E)$ relies on a set of generalization rules depicted in Figure 2, which transform G into a generalized version G', i.e. G' is a ReGaP that matches G. For example, rule 1 replaces a non-wildcard $u \in V$ by an any-1+ wildcard, represented by the $+$ node. W represents any wildcard type, while A represents any node. The rules must be applied in order[1] (e.g. rule 1 cannot be applied after an instance of rule 2). By default, no constraint is imposed on the subgraph in the left-hand side of a rule and the respective connections to other nodes in V. The special anti-node $\times$ is used to specify such constraints. For example, the second anti-node in rule 2 dictates that no edge $(u', S) \in E$ may exist such that $u' \neq u$. Such anti-nodes prevent non-directed paths from being generalized into an any-1+-sequence. For example, consider a generalized graph of the form $u \rightarrow S+ \leftarrow v$. One cannot apply rule 2 on the edge $(u, S+)$ due to the aforementioned anti-node and the existence of the edge $(v, S+)$.

Definition 1 *Let $P = (V_P, E_P)$ be a ReGaP and G a graph. P is said to match G if and only if there exists a sequence of generalization rules transforming G into $G' = (V', E')$ such that there exists a bijective mapping $f : V_P \leftrightarrow V'$ that satisfies the following conditions:*

1. *for all $(u, v) \in E_P$, $(f(u), f(v)) \in E'$ and vice-versa.*
2. *for all $w \in W_P$, $f(w)$ is a wildcard of the same type.*

[1]Note that some rules are actually commutative, such as 7 and 8. However, others do need to follow the defined order, otherwise the definition would allow matches that do not make sense. For simplicity, a strict order is considered instead of a partial one.

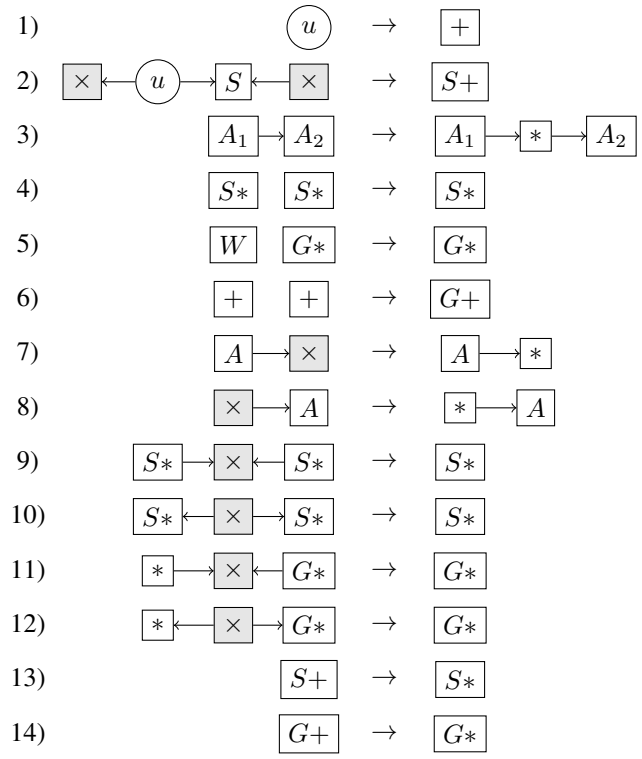

Figure 2: Rules for generalizing a graph G.

Example 2 *Consider the example from Figure 1. By applying rules 1 and 2 to replace v_2 and v_3 with $S+$ and rules 1 and 6 to replace v_6 and v_7 with $G+$ one obtains a generalized graph that satisfies the conditions of Definition 1.*

Regarding time complexity, consider a non-deterministic algorithm that guesses the sequence of rule applications and the certificate of isomorphism proving that the resulting generalized graph does match P. In the worst case, rule 1 is applied to each node in V and rule 2 to each edge in E. Similarly, rule 3 is applied at most $2V_PE$ times: 2 for both any-0+ wildcard types, and V_P because V_P may contain only any-0+ wildcards. The worst case number of rule applications is polynomial[2], and thus ReGaP matching is in NP.

Graph isomorphism is a special case of ReGaP matching. The introduction of wildcards enables the compact specification of infinite sets of graphs. ReGaPs can be further extended with new wildcard types, such as optional nodes/edges and sequences/subgraphs with size limitations.

4 ReGaP Matching Encoding

Our approach reduces ReGaP matching to an instance of SAT. First we detail the base encoding for the special case where V_P does not contain wildcards. The adaptations needed to support each wildcard type are explained in Sections 4.1 and 4.2. The full encoding relies on the implicit mapping that exists between the nodes on the left of a generalization rule and the respective generalized node on the right. For example, in rule 1, the node u is mapped to the any-1+ wildcard that is introduced in its place. In rule 2, the node u and the nodes mapped to S become mapped to the new $S+$.

The base encoding is an adaptation of an encoding for maximum common subgraph available in the literature (Feng et al. 2017; Terra-Neves et al. 2021). It solves the original problem by mapping the nodes and edges of G into those of P. For simplicity, some constraints are shown as at-most-1 constraints, i.e. of the form $\sum_i l_i \leq 1$, instead of clauses. Note that these can be converted to CNF by introducing the clause $(\neg l_i \vee \neg l_j)$ for each pair i, j such that $i \neq j$, or by using one of many CNF encodings available in the literature (Ansótegui and Manyà 2004; Chen 2010; Frisch et al. 2005; Klieber and Kwon 2007; Prestwich 2007).

The following sets of Boolean variables are considered:

- **Inclusion variables.** For each node $v_P \in V_P$, a variable o_{v_P} is introduced to encode if some node of V is mapped to v_P, i.e. if there exists a node $v \in V$ such that $f(v) = v_P$ (i.e. $o_{v_P} = 1$) or not (i.e. $o_{v_P} = 0$).
- **Mapping variables.** For each node pair $(v_P, v) \in V_P \times V$, a variable $m_{v_P,v}$ is used to encode if the node v is mapped to v_P. If $f(v) = v_P$, then $m_{v_P,v} = 1$, otherwise $m_{v_P,v} = 0$.
- **Control-flow variables.** These variables are the analogous of the inclusion variables for edges. For each edge $(u_P, v_P) \in E_P$, a variable c_{u_P,v_P} is used to encode if there exists an edge $(u, v) \in E$ mapped to (u_P, v_P). If so, then $c_{u_P,v_P} = 1$, otherwise $c_{u_P,v_P} = 0$. (u, v) is said to be mapped to (u_P, v_P) if both u and v are mapped to u_P and v_P respectively.

The SAT formula contains the following clauses:

- **Inclusion clauses.** Ensure consistency between the inclusion and the mapping variables, i.e. for each node $v_P \in V_P$, if $o_{v_P} = 1$, then at least one of the $m_{v_P,v}$ must also be set to 1 for some $v \in V$, and vice-versa.

$$\bigwedge_{v_P \in V_P} \left(o_{v_P} \leftrightarrow \bigvee_{v \in V} m_{v_P,v} \right). \tag{1}$$

- **One-to-one clauses.** Each node in V must be mapped to at most one node in V_P and vice-versa.

$$\bigwedge_{v_P \in V_P} \left(\sum_{v \in V} m_{v_P,v} \leq 1 \right) \wedge \bigwedge_{v \in V} \left(\sum_{v_P \in V_P} m_{v_P,v} \leq 1 \right). \tag{2}$$

- **Control-flow consistency clauses.** Each edge in E_P can only be mapped to edges that exist in E. More specifically, for each edge $(u_P, v_P) \in E_P$, if $(u, v) \notin E$, then either u is not mapped to u_P (i.e. $m_{u_P,u} = 0$), v is not mapped to v_P (i.e. $m_{v_P,v} = 0$), or no edge of E is mapped to (u_P, v_P) (i.e. $c_{u_P,v_P} = 0$).

$$\bigwedge_{(u_P,v_P) \in E_P} \bigwedge_{(u,v) \in (V \times V) \setminus E} (\overline{m_{u_P,u}} \vee \overline{m_{v_P,v}} \vee \overline{c_{u_P,v_P}}). \tag{3}$$

[2]Due to space limitations, the worst case for each rule is included in the extended version of the paper (Terra-Neves et al. 2023).

- **No spurious edge clauses.** If an edge $(u_P, v_P) \in E_P$ is mapped to some edge in E, then u_P and v_P must also be mapped to nodes of V.

$$\bigwedge_{(u_P,v_P)\in E_P} (c_{u_P,v_P} \rightarrow o_{u_P} \wedge o_{v_P}) . \tag{4}$$

- **Node isomorphism clauses.** All nodes in V must be mapped to a node in V_P and vice-versa.

$$\bigwedge_{v_P\in V_P} (o_{v_P}) \wedge \bigwedge_{v\in V} \left(\bigvee_{v_P\in V_P} m_{v_P,v} \right). \tag{5}$$

- **Edge isomorphism clauses.** All edges in E must be mapped to an edge in E_P and vice-versa. For each edge $(u, v) \in E$, we must ensure that, given a pair of nodes u_P, v_P of V_P such that $(u_P, v_P) \notin E_P$, then either u or v is not mapped to u_P or v_P respectively.

$$\bigwedge_{(u_P,v_P)\in E_P} (c_{u_P,v_P}) \wedge \bigwedge_{(u,v)\in E} \bigwedge_{(u_P,v_P)\in (V_P\times V_P)\setminus E_P} (\overline{m_{u_P,u}} \vee \overline{m_{v_P,v}}) . \tag{6}$$

4.1 Sequence Wildcards

In order to support any-0+-sequence wildcards, each $w \in W_P^{S*}$ must be expanded by replacing it with k non-wildcard nodes $w^1, \ldots, w^k$, as well as $k-1$ edges, one for each (w^i, w^{i+1}) such that $i \in \{1..k-1\}$. The choice of value for k is discussed at the end of this section. We use $V_P^{\text{exp}}(w) = \{w^1, \ldots, w^k\}$ to denote the set of non-wildcard nodes added to replace w, and $E_P^{\text{exp/mid}}(w) = \{(w^1, w^2), \ldots, (w^{k-1}, w^k)\}$ to denote the set of edges added between the nodes of $V_P^{\text{exp}}(w)$. Each edge $(u_P, w) \in E_P$ is replaced by the edge (u_P, w^1). We use $E_P^{\text{exp/in}}(w)$ to denote the set of edges added to replace each such (u_P, w). Similarly, each $(w, v_P) \in E_P$ is replaced by k edges (w^i, v_P), one for each $i \in \{1..k\}$. We use $E_P^{\text{exp/out}}(w, v_P)$ to denote the set of all new edges added to replace (w, v_P). Additionally, we use $S_P(w) \subset V_P$ to denote the set of successors of w, i.e. $S_P(w) = \{v_P \in V_P : (w, v_P) \in E_P\}$, and $E_P^{\text{exp/out}}(w)$ to denote the union of all sets $E_P^{\text{exp/out}}(w, v_P)$, i.e. $E_P^{\text{exp/out}}(w) = \bigcup_{v_P\in S_P(w)} E_P^{\text{exp/out}}(w, v_P)$.

Extra edges (u_P, v_P) are added from each predecessor u_P of w to each successor $v_P \in S_P(w)$. We use $E_P^{\text{exp/skip}}(u_P, w)$ to denote the set of edges added from u_P to the nodes in $S_P(w)$. Additionally, we use $B_P(w) \subset V_P$ to denote the set of predecessor nodes of w, i.e. $B_P(w) = \{u_P \in V_P : (u_P, w) \in E_p\}$, and $E_P^{\text{exp/skip}}(w, v_P)$ to denote the set of edges added from the nodes in $B_P(w)$ to v_P. We use $E_P^{\text{exp/skip}}(w)$ to denote the union of all such sets, i.e. $E_P^{\text{exp/skip}}(w) = \bigcup_{u_P\in B_P(w)} E_P^{\text{exp/skip}}(u_P, w)$. Lastly, we use $E_P^{\text{exp}}(w)$ to denote all edges added when replacing w, i.e. $E_P^{\text{exp}}(w) = E_P^{\text{exp/in}}(w) \cup E_P^{\text{exp/mid}}(w) \cup E_P^{\text{exp/skip}}(w) \cup E_P^{\text{exp/out}}(w)$.

Let $P^{\text{exp}} = (V_P^{\text{exp}}, E_P^{\text{exp}})$ denote the graph that results from the expansion. The encoding is built using P^{exp} instead of P, with the following changes:

- **Node isomorphism clauses.** For each wildcard $w \in W_P^{S*}$, the nodes in $V_P^{\text{exp}}(w)$ are optional and thus must be excluded from equation (5) as follows:

$$\bigwedge_{v_P\in V_P^{\text{exp}}\setminus\bigcup_{w\in V_P^{S*}} V_P^{\text{exp}}(w)} (o_{v_P}) \wedge \bigwedge_{v\in V} \left(\bigvee_{v_P\in V_P^{\text{exp}}} m_{v_P,v} \right). \tag{7}$$

- **Edge isomorphism clauses.** Analogously, the edges in $E_P^{\text{exp}}(w)$ must be excluded from equation (6) as follows:

$$\bigwedge_{(u_P,v_P)\in E_P^{\text{exp}}\setminus\bigcup_{w\in W_P^{S*}} E_P^{\text{exp}}(w)} (c_{u_P,v_P}) \wedge \bigwedge_{(u,v)\in E} \bigwedge_{(u_P,v_P)\in (V_P^{\text{exp}}\times V_P^{\text{exp}})\setminus E_P^{\text{exp}}} (\overline{m_{u_P,u}} \vee \overline{m_{v_P,v}}) . \tag{8}$$

However, extra clauses are necessary to ensure that an optional edge (u_P, v_P) is mapped to the edge $(u, v) \in E$ when u and v are mapped to u_P and v_P respectively.

$$\bigwedge_{(u_P,v_P)\in E_P^{\text{exp}}} \bigwedge_{(u,v)\in E} (m_{u_P,u} \wedge m_{v_P,v} \rightarrow c_{u_P,v_P}) . \tag{9}$$

- **Sequence clauses.** A sequence node w^i can be mapped to some node in V only if each of the sequence nodes that precede w^i have some node of V mapped to them.

$$\bigwedge_{w\in W_P^{S*}} \bigwedge_{w^i\in V_P^{\text{exp}}(w), i\geq 2} (o_{w^i} \rightarrow o_{w^{i-1}}) . \tag{10}$$

- **Incoming any-0+-sequence control-flow clauses.** If a node in V is mapped to a wildcard $w \in W_P^{S*}$, then each incoming edge of w must be mapped to an edge in E.

$$\bigwedge_{w\in W_P^{S*}} \bigwedge_{(u_P,v_P)\in E_P^{\text{exp/in}}(w)} (o_{w^1} \rightarrow c_{u_P,v_P}) . \tag{11}$$

- **Outgoing any-0+ control-flow clauses.** Analogous of equation (11) but for the outgoing edges of w.

$$\bigwedge_{w\in W_P^{S*}} \bigwedge_{v_P\in S_P(w)} \left(o_{w^1} \rightarrow \bigvee_{(u_P,v_P)\in E_P^{\text{exp/out}}(w,v_P)} c_{u_P,v_P} \right). \tag{12}$$

Note that, while w^1 must always be the first sequence node mapped to some node of V, the last such sequence node w^l can vary depending on the number l of nodes of V mapped to w. Only the outgoing edges of w^l can be mapped to some edge in E.

$$\bigwedge_{w\in W_P^{S*}} \bigwedge_{(w^i,v_P)\in E_P^{\text{exp/out}}(w), i\leq k-1} \left(o_{w^{i+1}} \rightarrow \overline{c_{w^i,v_P}}\right) . \tag{13}$$

- **Skip any-0+-sequence control-flow clauses.** For each wildcard $w \in W_P^{S*}$, if no node in V is mapped to w and w has at least one successor, then each predecessor u_P of w must have at least one of the edges that connect u_P to one of the successors of w mapped to some edge in E.

$$\bigwedge_{w\in W_P^{S*}, |S_P(w)|>0} \bigwedge_{u_P\in B_P(w)} \left(\overline{o_{w^1}} \rightarrow \bigvee_{(u_P,v_P)\in E_P^{\text{exp/skip}}(u_P,w)} c_{u_P,v_P} \right). \tag{14}$$

Analogous of equation (14) but for the successors.

$$\bigwedge_{w \in W_P^{S*}, |B_P(w)|>0} \bigwedge_{v_P \in S_P(w)} \left(\overline{o_{w^1}} \rightarrow \bigvee_{(u_P,v_P) \in E_P^{\text{exp/skip}}(w,v_P)} c_{u_P,v_P} \right). \tag{15}$$

Lastly, if a node in V is mapped to w, then the edges in E cannot be mapped to an edge in $E_P^{\text{exp/skip}}(w)$.

$$\bigwedge_{w \in W_P^{S*}} \bigwedge_{(u_P,v_P) \in E_P^{\text{exp/skip}}(w)} (o_{w^1} \rightarrow \overline{c_{u_P,v_P}}). \tag{16}$$

Note that the choice of k must ensure that P^{exp}, together with the aforementioned changes to the encoding, retains the same semantics as P. One (naive) solution is to set $k = |V|$. Moreover, for the sake of simplicity, the encoding, as described, assumes that P does not contain edges between wildcards. The expansion of such wildcards is actually an iterative process. Therefore, an edge $(w, w') \in E_P$ between a pair of wildcards $(w, w') \in W_P^{S*} \times W_P^{S*}$ ends up being replaced by k (w^i, w'^1) edges from each node $w^i \in V_P^{\text{exp}}(w)$ to the first non-wildcard node w'^1 introduced to replace w'. Additional edges must also be added from $w^1, \ldots, w^k$ and the predecessors of w to the successors of w'.

Given the above encoding, any-1+-sequence wildcards are supported by replacing each such $w \in W_P^{S+}$ with a non-wildcard node w^1, an any-0+-sequence w' and the edge (w^1, w'), and by setting the destination (source) of all incoming (outgoing) edges of w to w^1 (w').

4.2 Subgraph Wildcards

In order to support any-1+-subgraph wildcards, each $w \in W_P^{G+}$ is replaced by a copy of G, i.e. w is replaced by a w^v node for each $v \in V$ and a (w^u, w^v) edge for each $(u, v) \in E$. As in Section 4.1, $V_P^{\text{exp}}(w)$ ($E_P^{\text{exp/mid}}(w)$) denotes the set of node (edge) copies created to replace w. Each edge $(u_P, w) \in E_p$ is replaced by $|V|$ edges (u_P, w^v) from u_P to each $w^v \in V_P^{\text{exp}}(w)$. We use $E_P^{\text{exp/in}}(u_P, w)$ to denote the set of new edges that replace (u_P, w). Similarly, each edge $(w, v_P) \in E_P$ is replaced by $|V|$ edges (w^v, v_P) from each $w^v \in V_P^{\text{exp}}(w)$ to v_P. $E_P^{\text{exp/out}}(w, v_P)$ denotes the set of new edges that replace (w, v_P).

The encoding in Section 4.1 can be adapted to support any-1+-subgraph wildcards by replacing W_P^{S*} with $W_P^{S*} \cup W_P^{G+}$ in equation (7), equation (8) and equation (9), plus the following new clauses:

- **Any-1+-subgraph inclusion clauses.** At least one node in V must be mapped to each wildcard $w \in W_P^{G+}$.

$$\bigwedge_{w \in W_P^{G+}} \left(\bigvee_{v_P \in V_P^{\text{exp}}(w)} o_{v_P} \right). \tag{17}$$

- **Incoming any-1+-subgraph control-flow clauses.** Each incoming edge of $w \in W_P^{G+}$ must be mapped to some edge in E.

$$\bigwedge_{w \in W_P^{G+}} \bigwedge_{u_P \in B_P(w)} \left(\bigvee_{(u_P,v_P) \in E_P^{\text{exp/in}}(u_P,w)} c_{u_P,v_P} \right). \tag{18}$$

- **Outgoing any-1+ control-flow clauses.** Analogous of equation (18) for the outgoing edges.

$$\bigwedge_{w \in W_P^{G+}} \bigwedge_{v_P \in S_P(w)} \left(\bigvee_{(u_P,v_P) \in E_P^{\text{exp/out}}(w,v_P)} c_{u_P,v_P} \right). \tag{19}$$

If $w \in W_P^{G*}$ is an any-0+-subgraph wildcard, an extra set of edges from each predecessor $u_P \in B_P(w)$ to each successor $v_P \in S_P(w)$ must also be added as described in Section 4.1. $W_P^{S*} \cup W_P^{G+}$ must be replaced by W_P in equation (7), equation (8) and equation (9), and new clauses must be added encoding the incoming, outgoing and skip control-flow of w. Given that w is an any-0+ wildcard, the nodes in $V_P^{\text{exp}}(w)$ are optional, and thus the clauses in equation (17) do not apply. The full encoding is available in the extended version (Terra-Neves et al. 2023).

5 Attributed ReGaP Matching

In the attributed ReGaP matching problem, G is an attributed graph and P defines constraints over the attributes of the nodes/edges of G. There are 3 types of constraints:

- **Node constraints.** Each node $v_P \in V_P$ is assigned a node constraint ϕ_{v_P} over the attributes A_V of the nodes in V. Given a node $v \in V$, we use $\phi_{v_P}(v) = 1$ ($\phi_{v_p}(v) = 0$) to denote that v satisfies (does not satisfy) ϕ_{v_P}.
- **Edge constraints.** Analogously, an edge constraint $\phi_{(u_P,v_P)}$ is associated with each edge $(u_P, v_P) \in E_P$ and $\phi_{(u_P,v_P)}((u, v))$ denotes if the edge $(u, v) \in E$ satisfies $\phi_{(u_P,v_P)}$.
- **Node pair relation constraints.** A node pair relation constraint ψ_{u_P,v_P} is associated with each node pair $u_P, v_P \in V_p$ such that $u_P \neq v_P$. Note that the existence of a node pair relation constraint between u_P and v_P does not imply that the edge (u_P, v_P) exists.

The definition for the attributed problem must ensure that these constraints are satisfied. We assume that node and node pair relation constraints cannot be associated with wildcards.

Definition 2 *Let $P = (V_P, E_P)$ be a ReGaP and G an attributed graph. P is said to match G if and only if there exists a sequence of generalization rules transforming G into $G' = (V', E')$ such that there exists a bijective mapping $f : V_P \leftrightarrow V'$ that satisfies the following conditions:*

1. *for all $(u, v) \in E_P$, $(f(u), f(v)) \in E'$ and vice-versa.*
2. *for all $w \in W_P$, $f(w)$ is a wildcard of the same type.*
3. *for all $v \in V_P \setminus W_P$, $\phi_v(f(v)) = 1$.*
4. *for all $(u, v) \in E_P$, $\phi_{(u,v)}((f(u), f(v))) = 1$.*
5. *for all $u, v \in (V_P \setminus W_P) \times (V_P \setminus W_P)$ such that $u \neq v$, $\psi_{u,v}(f(u), f(v)) = 1$.*

5.1 Encoding

This section describes how to adapt the encoding in Section 4 for attributed matching. First, one must set the constraints for the extra nodes/edges added by wildcard expansion. Given a wildcard w of any type, the node constraint for each $w^i \in V_P^{\text{exp}}(w)$ is set to $\phi_{w^i}(v) \equiv 1$, i.e.

any node of V can be mapped to w^i. The same applies to the edge constraints for each edge in $E_P^{\text{exp/mid}}(w)$. Given a predecessor $u_P \in B_P(w)$, the edge constraint for each $(u_P, w^i) \in E_P^{\text{exp/in}}(u_P, w)$ is set to $\phi_{(u_P,w^i)}((u,v)) \equiv \phi_{(u_P,w)}((u,v))$. If w is an any-0+ wildcard, the constraints for the edges in $E_P^{\text{exp/skip}}(u_P, w)$ are set in the same way. Lastly, given a successor $v_P \in S_P(w)$, the edge constraint for each $(w^i, v_P) \in E_P^{\text{exp/out}}(w, v_P)$ is $\phi_{(w^i,v_P)}((u,v)) \equiv \phi_{(w,v_P)}((u,v))$.

The following additional clauses are necessary:

- **Node constraint consistency clauses.** The node constraints in P must be satisfied. More specifically, if a node $v \in V$ does not satisfy the node constraint of a node $v_P \in V_P$, then v cannot be mapped to v_P.

$$\bigwedge_{v_P \in V_P} \bigwedge_{v \in V, \phi_{v_P}(v)=0} (\overline{m_{v_P,v}}) . \tag{20}$$

- **Node pair relation constraint consistency clauses.** The node pair relation constraints in P must be satisfied. More specifically, for each pair of nodes u_P, v_P of V_P and u, v of V, if u is mapped to u_P and the pair u, v does not satisfy the node pair relation constraint of u_P, v_P, then v cannot be mapped to v_P.

$$\bigwedge_{\substack{(u_P,v_P) \in V_P \times V_P \\ u_P \neq v_P}} \bigwedge_{u \in V} \left(m_{u_P,u} \rightarrow \bigvee_{\substack{v \in V \\ u \neq v \wedge \psi_{u_P,v_P}(u,v)=1}} m_{v_P,v} \right). \tag{21}$$

Edge constraint satisfaction is ensured by changing the subscript of the inner conjunctions in equation (3) and equation (8) to exclude edges such that $\phi_{(u_P,v_P)}((u,v)) = 0$.

5.2 Node Merging

Wildcard expansion (see Sections 4.1 and 4.2) can have a severe impact in the size of the encoding, and thus the performance of the SAT solver. To mitigate this, we propose a sound and complete procedure that merges sequences of nodes in G that do not satisfy any node constraints in P, since such nodes can only be mapped to wildcards, regardless of the wildcard type, as long as P does not contain edges between wildcards.

Proposition 1 *Consider an attributed graph $G = (V, E)$ and a ReGaP $P = (V_P, E_P)$ with no edges between wildcards, i.e. $|W_P \cap \{u_P, v_P\}| \leq 1$ for all $(u_P, v_P) \in E_P$, and an edge $(u, v) \in E$ such that: for all $v_P \in V_P \setminus W_P$, $\phi_{v_P}(u) = \phi_{v_P}(v) = 0$; for all $(u', v) \in E$, $u' = u$; and, for all $(u, v') \in E$, $v' = v$. Let $G' = (V', E')$ be an attributed graph such that:*

- $V' = V \setminus \{u\}$; $A'_V(v') = A_V(v')$ *for all* $v' \in V'$;
- $E' = [E \setminus (\{(u,v)\} \cup \{(u',u) : (u',u) \in E\})] \cup \{(u',v) : (u',u) \in E\}$;
- $A'_E((u',v')) = A_E((u',v'))$ *for all* $(u',v') \in E \cap E'$;
- $A'_E((u',v)) = A_E((u',u))$ *for all* $(u',u) \in E$.

P matches G if and only if P matches G'.

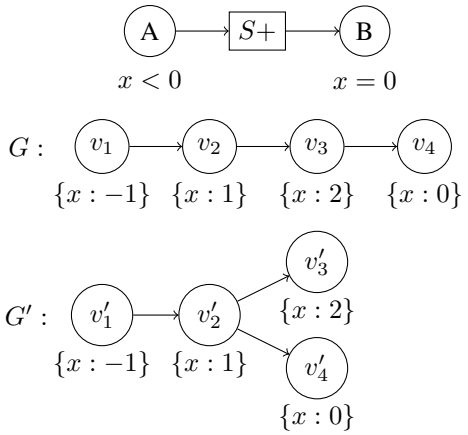

Figure 3: An example of an attributed ReGaP (top), an attributed graph G that matches the ReGaP and an attributed graph G' that does not match.

Example 3 *Consider the example ReGaP and attributed graphs in Figure 3. Node v_1 satisfies the node constraint of A, while v_4 satisfies the constraint of B. However, v_2 and v_3 do not satisfy either constraint. Therefore, we can safely merge v_2 and v_3 into a single node. On the other hand, v'_1 and v'_4 also satisfy the node constraints for A and B respectively, while v'_2 and v'_3 do not. However, v'_2 and v'_3 cannot be merged because there exists an edge from v'_2 to v'_4, thus v'_2 and v'_3 cannot be mapped to the same sequence wildcard.*

Due to space limitations, the proof is included in the extended version (Terra-Neves et al. 2023). Based on Proposition 1, if P does not contain edges between wildcard nodes, we apply a preprocessing step that repeatedly transforms G into G' until no more edges (u, v) exist in E satisfying the respective criteria.

6 Experimental Evaluation

This section evaluates the performance of the SAT-based approach for ReGaP matching. For the attributed graphs, we used a collection of control-flow graphs extracted from the Python code snippets in the CodeSearchNet dataset (Husain et al. 2020). The ReGaPs used in the evaluation replicate the kind of bad code patterns that are integrated in the AI Mentor Studio (OutSystems 2023) code analysis engine for the OutSystems visual programming language, which uses ReGaPs as a formalism for specifying such patterns. One concrete example of a bad performance pattern that occurs frequently in OutSystems is a database query Q1, followed by a loop that iterates the output of Q1 and performs another query Q2 with a filter by the current record of Q1. Typically, Q2 can be merged with Q1 through a join condition, resulting in just 1 query instead of $N + 1$, where N is the number of records returned by Q1.

The graph dataset and ReGaPs are publicly available[3], plus an executable that can be used to replicate the results presented in this evaluation. 13 ReGaPs are considered in

[3] https://github.com/MiguelTerraNeves/regap

this evaluation, containing up to 5 wildcards. We consider only the 72 812 graphs with at least 15 nodes because the ReGaP matcher was always able to solve the instances with smaller ones in under 7 seconds, resulting in a total of 946 556 instances. The maximum number of nodes is 1070, with a median of 21. Out of the 946 556 instances, 130 136 are know to be satisfiable, 791 426 are unsatisfiable, and 24 994 are unknown. Further statistics regarding the dataset and ReGaPs are available in the extended version (Terra-Neves et al. 2023).

SAT formulas were solved using PySAT (version 0.1.7.dev21) (Ignatiev, Morgado, and Marques-Silva 2018), configured to use the Glucose solver (version 4.1) (Audemard, Lagniez, and Simon 2013) with default settings. All experiments were run once with a timeout of 60 seconds[4] on an AWS m5a.24xlarge instance with 384 GB of RAM. Different experiments were split across 84 workers running in parallel, each running Glucose sequentially.

We aim to answer the following research questions regarding the performance of the proposed approach: **R1:** What is the impact of the node merging step? **R2:** What is the impact of the graph size? **R3:** What is the impact of the number of wildcards?

Recall from Section 1 that, although ReGaPs and regular-path queries are related, there exist structures expressible with ReGaPs that are not expressible using regular-path queries. Therefore, ReGaPs solve a fundamentally different problem, thus the lack of comparison with the state of the art in regular-path queries.

6.1 Impact of Node Merging

Table 1 shows the impact of node merging on the size of the graph and the SAT encoding. We only consider instances for which the ReGaP matcher did not timeout before the encoding was complete, both with and without node merging. Moreover, one of the ReGaPs used in this evaluation contains an edge between wildcards. The respective instances are also not considered since node merging is not applicable in this scenario (see Proposition 1). The reduction in the number of nodes is 15.4%, on average, which translates to a reduction, on average, of 25.7% less clauses in the SAT encoding. We observed that the overhead of node merging is at most 1 second for these instances.

Figure 4 compares the execution time, in seconds, with and without node merging. The ReGaP with the edge between wildcards is also not considered in this figure. Node merging has a significant impact on performance, being able to solve many more instances faster than the base encoding. For example, 42 498 more instances are solved in less than 10 seconds with node merging. 971 of these instances result in a timeout without node merging. The base encoding resulted in timeouts for 34 110 out of the 946 556 instances. With node merging, this value is reduced to 26 487. However, some instances are actually solved slower with node merging. In fact, node merging times out for 940 instances that are solved with the base encoding. We observed that

[4]The 60 seconds timeout is what is considered acceptable in the context of AI Mentor Studio.

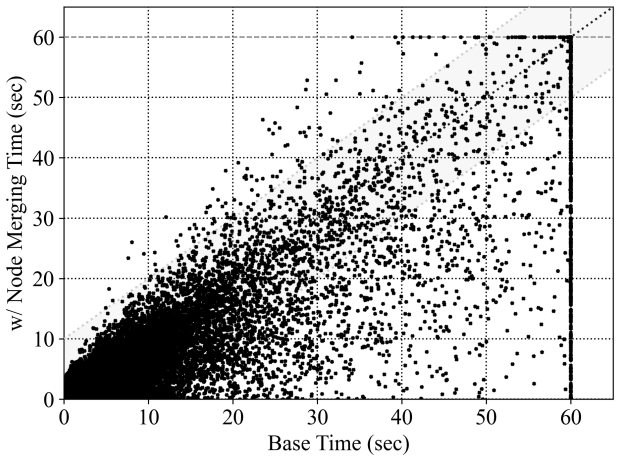

Figure 4: Execution time with node merging versus without.

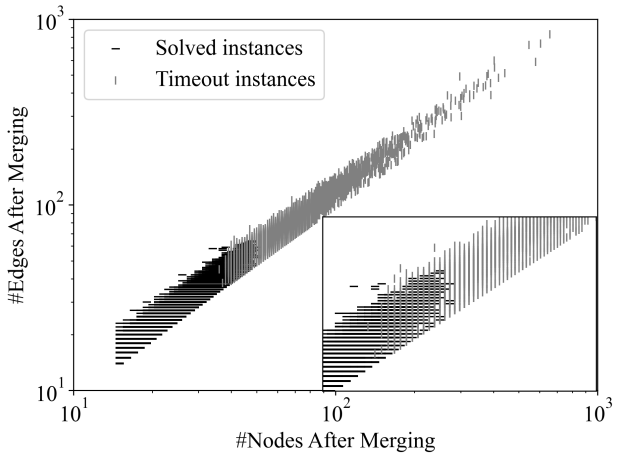

Figure 5: Timeouts as a function of graph size.

node merging resulted in very little reduction for these specific instances: 0% median reduction and only 1.8% in the 90-percentile. When it is 0%, the time required by the base algorithm is very close to the timeout: around 58 seconds on average. Therefore, these timeouts are likely due to noise introduced by worker contention in the parallel experimental environment. When the reduction is very small, this causes the formula's variables and constraints to change, which can trigger unpredictable behavior in the solver. Because the size of the formula is very similar, the slightly smaller formula can be harder to solve.

6.2 Impact of Graph Size

Figure 5 shows the timeouts and solved instances as a function of graph size, considering only the ReGaP with an edge between wildcards. The maximum number of nodes (edges) among solved instances is 50 (64), while the minimum for the instances that timeout is 34 (38). This hints at a strong potential for further performance improvements by investing

	Base				With Node Merging			
	min	max	median	average	min	max	median	average
#Nodes	15	1 070	21	26	1	1 070	18	22
#Edges	14	1 335	24	30.5	0	1 335	21	26
#Var	285	31 601	1 745	2 661	5	25 259	1 423	2 097
#Const	9675	21 079 098	228 175	659 547	15	17 343 652	164 140	490 259

Table 1: Comparison of the graph and encoding size with and without node merging.

				Exec Time (s)		
ReGaP	**W**	**Timeouts**		**avg**	**med**	**std**
call loops	1	337	0.5%	1.3	0.2	4.0
call	2	930	1.3%	2.7	0.6	5.8
ma loops	2	1071	1.5%	3.1	0.8	6.2
afa loops s	3	2003	2.7%	4.8	2.1	7.9
bterm loose	3	8026	11.0%	9.6	5.0	11.5
bterm	3	2010	2.8%	4.5	1.8	7.8
fcall loops	3	1733	2.4%	4.5	1.8	7.7
ma	3	1785	2.4%	4.5	1.8	7.8
afa loops	4	2915	4.0%	6.4	3.2	9.1
fcall	4	2737	3.8%	6.2	3.1	9.0
mfy loops	4	2757	3.8%	6.4	3.2	9.1
afa	5	3983	5.5%	8.0	4.2	10.1
mfy	5	3823	5.2%	7.9	4.2	10.1

Table 2: The number of wildcards (W column) and timeouts, and the baseline execution times for each ReGaP.

in further simplification of the graph. We observed the same behavior for the remaining ReGaPs, with some variation regarding the maximum graph size among solved instances and the minimum for the ones that resulted in timeouts.

6.3 Impact of the Number of Wildcards

Table 2 compares the number of timeouts and execution times obtained without node merging for each ReGaP. The rows are sorted by the respective number of wildcards. Note that the execution time statistics do not consider instances that resulted in a timeout. In most cases, the number of timeouts and the mean/median execution time seem to grow with the number of wildcards. The algorithm also seems to become more unstable as the number of wildcards grows, as evidenced by the increase in the standard deviation.

The obvious exception is the *bterm loose* ReGaP, the one with edges between wildcard nodes, which was expected for the following reason. Lets consider an edge (w, w') between, for example, two any-1+-subgraph wildcards w and w'. The expansion of w replaces the edge (w, w') by $|V|$ new edges from each of the non-wildcard nodes inserted in place of w to w'. In turn, each of those edges will be replaced by another $|V|$ edges when w' is expanded as well.

Note that this is a preliminary evaluation and further experiments should be performed with a more diverse set of ReGaPs. Also, this pattern of performance degradation as the number of wildcards increases is not so clear with node merging. Depending on the narrowness of the constraints, node merging can have a significant impact on performance.

7 Conclusions & Future Work

Solving graph matching problems has many applications. In particular, it is an essential tool for code analysis in visual programming languages. However, the state of the art focuses either on solving graph isomorphism, approximated graph matching or regular-path queries. We propose ReGaP matching, an extension of graph isomorphism that allows one to check complex structural properties through declarative specifications. We propose a SAT encoding for solving ReGaP matching and a simplification technique for reducing encoding size, thus improving the performance of the SAT solver. An extensive experimental evaluation carried on benchmarks from the CodeSearchNet dataset (Husain et al. 2020) shows the effectiveness of the proposed approach.

In the future, we plan to extend ReGaPs with new types of wildcards (e.g. optional nodes/edges and sequence/subgraph wildcards with size limitations). We also wish to explore more compact encodings for ReGaP matching that do not rely on wildcard expansion, as well as further evaluate the impact of the number of wildcards and overall structure of the ReGaPs on the performance of the algorithm. Other SAT solvers and alternative automated reasoning frameworks, such as constraint programming (Rossi, van Beek, and Walsh 2006) and satisfiability modulo theories (de Moura and Bjørner 2011), should also be evaluated. We can also explore tighter bounds for the value of k used for expansion, and an algorithm that starts with a small value for k and iteratively increments k until the formula becomes satisfiable or an upper bound is reached. Lastly, we also plan to develop tools for synthesizing ReGaPs from positive and negative examples. Note that ReGaP matching is a general problem that we believe has the potential to be useful in other applications that deal with graph data, such as computational biology, chemistry and network analysis. We hope to see future work explore such applications.

References

Angles, R.; Arenas, M.; Barceló, P.; Boncz, P. A.; Fletcher, G. H. L.; Gutiérrez, C.; Lindaaker, T.; Paradies, M.; Plantikow, S.; Sequeda, J. F.; van Rest, O.; and Voigt, H. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1421–1432. ACM.

Ansótegui, C.; and Manyà, F. 2004. Mapping Problems with Finite-Domain Variables into Problems with Boolean Variables. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 1–15. Springer.

Audemard, G.; Lagniez, J.; and Simon, L. 2013. Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 309–317. Springer.

Audemard, G.; and Simon, L. 2009. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, 399–404.

Auwatanamongkol, S. 2007. Inexact graph matching using a genetic algorithm for image recognition. *Pattern Recognition Letters*, 28(12): 1428–1437.

Biere, A.; Fleury, M.; and Heisinger, M. 2021. CaDiCaL, Kissat, Paracooba Entering the SAT Competition 2021. In *Proceedings of the SAT Competition 2021 – Solver and Benchmark Descriptions*, Department of Computer Science Report Series B, 10–13. University of Helsinki.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bunke, H. 1997. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8): 689–694.

Carletti, V.; Foggia, P.; and Vento, M. 2013. Performance Comparison of Five Exact Graph Matching Algorithms on Biological Databases. In *Proceedings of the ICIAP International Workshop on New Trends in Image Analysis and Processing*, 409–417. Springer.

Chen, J. 2010. A New SAT Encoding of the At-Most-One Constraint. *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation*.

Conte, D.; Foggia, P.; Sansone, C.; and Vento, M. 2004. Thirty Years Of Graph Matching In Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3): 265–298.

Cordella, L. P.; Foggia, P.; Sansone, C.; and Vento, M. 1999. Performance Evaluation of the VF Graph Matching Algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing (ICIAP)*, 1172–1177. IEEE Computer Society.

Cruz, I. F.; Mendelzon, A. O.; and Wood, P. T. 1987. A Graphical Query Language Supporting Recursion. In Dayal, U.; and Traiger, I. L., eds., *Proceedings of the International Conference on Management of Data (SIGMOD)*, 323–330. ACM.

Dahm, N.; Bunke, H.; Caelli, T.; and Gao, Y. 2012. Topological features and iterative node elimination for speeding up subgraph isomorphism detection. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR)*, 1164–1167. IEEE Computer Society.

de Moura, L. M.; and Bjørner, N. S. 2011. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9): 69–77.

Fan, W.; Li, J.; Ma, S.; Tang, N.; and Wu, Y. 2012. Adding regular expressions to graph reachability and pattern queries. *Frontiers of Computer Science*, 6(3): 313–338.

Feng, Y.; Bastani, O.; Martins, R.; Dillig, I.; and Anand, S. 2017. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*. The Internet Society.

Foggia, P.; Percannella, G.; and Vento, M. 2014. Graph Matching and Learning in Pattern Recognition in the Last 10 Years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(1).

Frisch, A. M.; Peugniez, T. J.; Doggett, A. J.; and Nightingale, P. 2005. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings. *Journal of Automated Reasoning*, 35(1-3): 143–179.

Husain, H.; Wu, H.-H.; Gazit, T.; Allamanis, M.; and Brockschmidt, M. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 428–437. Springer.

Klieber, W.; and Kwon, G. 2007. Efficient CNF encoding for selecting 1 from n objects. In *Proceedings of the International Workshop on Constraints in Formal Verification*, 39.

Larrosa, J.; and Valiente, G. 2002. Constraint Satisfaction Algorithms for Graph Pattern Matching. *Mathematical Structures in Computer Science*, 12(4): 403–422.

Liang, J. H.; Oh, C.; Mathew, M.; Thomas, C.; Li, C.; and Ganesh, V. 2018. Machine Learning-Based Restart Policy for CDCL SAT Solvers. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 94–110. Springer.

Libkin, L.; Martens, W.; and Vrgoc, D. 2013. Querying graph databases with XPath. In *Proceedings of the 16th Joint International Conference on Extending Database Technology and International Conference on Database Theory (EDBT/ICDT)*, 129–140. ACM.

Livi, L.; and Rizzi, A. 2013. The graph matching problem. *Pattern Analysis and Applications*, 16(3): 253–283.

Marques-Silva, J. P.; and Sakallah, K. A. 1996. GRASP - a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 220–227. IEEE Computer Society / ACM.

OutSystems. 2023. Manage technical debt. https://success.outsystems.com/documentation/11/managing_the_applications_lifecycle/manage_technical_debt/. "Accessed: 2023-12-11".

Park, Y. H.; Reeves, D. S.; Mulukutla, V.; and Sundaravel, B. 2010. Fast malware classification by automated behavioral graph matching. In *Proceedings of the 6th Cyber Security and Information Intelligence Research Workshop (CSIIRW)*, 45. ACM.

Piotrowski, P.; and Madeyski, L. 2020. Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review. *Data-Centric Business and Applications*, 77–99.

Prestwich, S. D. 2007. Variable Dependency in Local Search: Prevention Is Better Than Cure. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 107–120. Springer.

Raymond, J. W.; and Willett, P. 2002. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7): 521–533.

Reutter, J. L.; Romero, M.; and Vardi, M. Y. 2017. Regular Queries on Graph Databases. *Theory of Computing Systems*, 61(1): 31–83.

Riveros, O. 2021. SLIME SAT Solver. In *Proceedings of the SAT Competition 2021 – Solver and Benchmark Descriptions*, Department of Computer Science Report Series B, 37–37. University of Helsinki.

Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier.

Sanfeliu, A.; and Fu, K. 1983. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3): 353–362.

Singh, J.; Chowdhuri, S. R.; Gosala, B.; and Gupta, M. 2021. Detecting design patterns: a hybrid approach based on graph matching and static analysis. *Information Technology and Management*, 23(3): 139–150.

Terra-Neves, M.; Amaral, J.; Lemos, A.; Quintino, R.; Resende, P.; and Alegria, A. 2023. SAT-Based Algorithms for Regular Graph Pattern Matching. arXiv:2312.09995.

Terra-Neves, M.; Nadkarni, J.; Ventura, M.; Resende, P.; Veiga, H.; and Alegria, A. 2021. Duplicated code pattern mining in visual programming languages. In *Proceedings of the 29th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1348–1359. ACM.

Ullmann, J. R. 1976. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1): 31–42.

Ullmann, J. R. 2010. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *ACM Journal of Experimental Algorithmics*, 15: 1.1–1.64.

Wang, X.; Wang, Y.; Xu, Y.; Zhang, J.; and Zhong, X. 2020. Extending Graph Pattern Matching with Regular Expressions. In *Proceedings of the 31st International Conference on Database and Expert Systems Applications (DEXA)*, 111–129. Springer.

Zampelli, S.; Deville, Y.; and Solnon, C. 2010. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3): 327–353.

Zaslavskiy, M.; Bach, F. R.; and Vert, J. 2009. Global alignment of protein-protein interaction networks by graph matching methods. *Bioinformatics*, 25(12): 1259–1267.

Zhang, X.; Feng, Z.; Wang, X.; Rao, G.; and Wu, W. 2016. Context-Free Path Queries on RDF Graphs. In *Proceedings of the 15th International Semantic Web Conference (ISWC)*, 632–648. Springer.

Zou, Y.; Ban, B.; Xue, Y.; and Xu, Y. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE)*, 931–942. IEEE/ACM.