A Practical Algorithm for Structure Embedding

Charlie Murphy and Zachary Kincaid

Princeton University

Abstract. This paper presents an algorithm for the *structure embedding problem*: given two finite first-order structures over a common relational vocabulary, does there exist an injective homomorphism from one to the other? The structure embedding problem is NP-complete in the general case, but for *monadic* structures (each predicate has arity ≤ 1) we observe that it can be solved in polytime by reduction to bipartite graph matching. Our algorithm, MatchEmbeds, extends the bipartite matching approach to the general case by using it as the foundation of a backtracking search procedure. We show that MatchEmbeds outperforms state-of-the-art SAT, CSP, and subgraph isomorphism solvers on difficult random instances and significantly improves the performance of a client model checker for multi-threaded programs.

1 Introduction

This paper introduces and addresses the *structure embedding problem*, an algorithmic problem in finite model theory. The task is to determine whether a given first-order structure contains an isomorphic copy of another (e.g., if both structures in question are graphs, this is exactly the subgraph isomorphism problem). The structure embedding problem is NP-complete in general, but applications in software model checking demand algorithms that work well on instances that arise in practice.

A *finite relational structure* (simply *structure* in the following) consists of a finite set (the structure's *universe*) and a collection of relations over that set. For example, a graph is a structure where the universe is the set of vertices and which has a single binary relation, incidence. Structures are objects of interest in the fields of finite model theory and the theory of databases. A *structure embedding* is an injective homomorphism from one structure to another, and the *structure embedding problem* is to determine whether such an embedding exists between two given structures.

In the context of model checking, the structure embedding problem arises in abstract state space exploration of parameterized concurrent programs — multi-threaded programs with arbitrarily many threads each running the same code. Analogously to the way that the state of a (non-parameterized) program can be modeled by a valuation of a finite set of predicates, the state of a parameterized program can be modeled by a structure: the universe of the structure is a set of threads, and each relation represents a collection of program properties that hold. For example, a structure with universe $\{1, 2, 3\}$ and two monadic relations

$X = \{1\}$ and $Y = \{2, 3\}$ might represent a configuration with three threads $\{1, 2, 3\}$ where thread 1 is at location X and threads 2 and 3 are at location Y. Inter-thread relationships are represented with higher-arity predicates — e.g., a linear order on process identifiers might be represented by a binary relation $PidLt = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$. The structure embedding problem is exactly the problem of determining whether one such abstract state subsumes another (and so can be pruned from the state exporation).

Predicate automata are an automaton model that has been proposed for use in verification of multi-threaded programs [8,9], that utilizes structures to model program states. The state space of a predicate automaton is infinite, and the emptiness problem — the fundamental problem of interest for these automata — is not decidable in general. However, Farzan et al [8] give a semi-algorithm that can determine emptiness of a predicate automaton without enumerating all reachable states, employing ideas from well-structured transition systems [1,10]. The idea is to exploit structure embeddings to prune the state space: if there is an embedding from a structure $\mathfrak{A}$ to another $\mathfrak{B}$ and $\mathfrak{A}$ cannot reach an accepting state, then neither can $\mathfrak{B}$. By retaining only those states in the search space that are minimal w.r.t. embedding, it is often possible to make the search space finite. In particular, for *monadic* predicate automata (in which each relation has arity ≤ 1, corresponding to a program property that refers to the local variables of only one thread), this is always the case.

A single predicate automaton emptiness problem can involve thousands of structure embedding queries, and each structure embedding query can potentially take exponential time. Fortunately, there are two properties that make the situation less dire: first, we expect each embedding query to be small (e.g., we would not expect to observe a configuration involving hundreds of threads); second, we can expect structures to be dominated by monadic predicates (i.e., the correctness argument of a multi-threaded program somewhat rarely requires inter-thread properties).

This paper presents MatchEmbeds, an algorithm for the structure embedding problem that is based on the observation that the embedding problem for *monadic* structures can be solved in polynomial time by reduction to bipartite graph matching. We develop a practical algorithm for general structure embedding that uses bipartite graph matching as the backbone of a backtracking search procedure. Graph matching is used to inform the backtracking search procedure both on which decision points to branch on and which decisions to make. We show that this algorithm is practical for both structure embedding problems that result from predicate automaton emptiness checking and difficult randomly generated instances.

Paper organization The remainder of the paper is structured as follows. Section 2 formalizes the structure embedding problem and presents the main contribution of this paper: an efficient algorithm for the structure embedding problem. Section 3 discusses various heuristics and implementation issues that are important for practical performance. Section 4 presents experimental results, which compares our embedding algorithm against state-of-the-art SAT, Constraint Satisfaction Problem (CSP), and subgraph isomorphism solvers. Section 5 discusses related work, and Section 6 concludes.

2 Structure Embedding

This section describes our algorithm for the structure embedding problem. We begin by formalizing finite relational structures and embeddings. We then describe a reduction of the special case of monadic structure embedding to bipartite graph matching. Finally, we show how to use bipartite graph matching as the core of a backtracking search algorithm for the general structure embedding problem.

2.1 Finite Relational Structures and the Embedding Problem

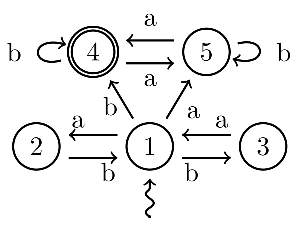
First we recall the definition of finite relational vocabularies and structures:

Definition 1 (Vocabulary, structure). *A* ***(finite relational) vocabulary*** *$\sigma = \langle Q, ar\rangle$ is a pair consisting of a finite set of predicate symbols $Q = \{q_1, ..., q_n\}$ and a function $ar : Q \to \mathbb{N}$ associating an* ***arity*** *to each predicate symbol. We say that σ is* ***monadic*** *if for each predicate symbol $q \in Q$, the arity of q is at most 1.*

A ***(finite) σ-structure*** *$\mathfrak{A} = \langle A, \{q^{\mathfrak{A}}\}_{q\in Q}\rangle$ consists of a finite universe A together with an interpretation $q^{\mathfrak{A}} \subseteq A^{ar(q)}$ of each predicate symbol $q \in Q$ as a relation over A of arity $ar(q)$.*

Example 1. Consider the class of non-deterministic finite automata (NFA) over the alphabet $\Sigma = \{a, b\}$. This class of NFAs can be represented as $\sigma_{\mathrm{NFA}(\Sigma)}$-structures, where $\sigma_{\mathrm{NFA}(\Sigma)}$ is the vocabulary consisting of two monadic predicates *Start* and *Final* (representing the start and final states of an automaton, respectively), and two binary relations Δ_a and Δ_b (representing the transition relation on the letters a and b, respectively).

For example, the automaton pictured below to the left (which recognizes sequences consisting of pairs of a and b followed by an even number of as) can be represented by the $\sigma_{\mathrm{NFA}(\Sigma)}$-structure $\mathfrak{F}$ pictured below to the right.



$\mathfrak{F} \triangleq \langle\{1, 2, 3, 4, 5\}, Start^{\mathfrak{F}}, Final^{\mathfrak{F}}, \Delta_a^{\mathfrak{F}}, \Delta_b^{\mathfrak{F}}\rangle$, where:

$Start^{\mathfrak{F}} \triangleq \{1\}$

$Final^{\mathfrak{F}} \triangleq \{4\}$

$\Delta_a^{\mathfrak{F}} \triangleq \{\langle 1, 2\rangle, \langle 1, 5\rangle, \langle 3, 1\rangle, \langle 4, 5\rangle, \langle 5, 4\rangle\}$

$\Delta_b^{\mathfrak{F}} \triangleq \{\langle 1, 3\rangle, \langle 1, 4\rangle, \langle 2, 1\rangle, \langle 4, 4\rangle, \langle 5, 5\rangle\}$

Next, we define structure homomorphisms and embeddings. Intuitively, one structure embeds into another if a "copy" of it appears in the second structure (modulo renaming of universe elements). Formally,

Definition 2 (Homomorphism, embedding). *Let $\sigma = \langle Q, ar \rangle$ be a vocabulary, and let $\mathfrak{A}$ and $\mathfrak{B}$ be σ-structures. A **homomorphism** is a function $h : A \to B$ such that for all $q \in Q$ and all $\langle a_1, ..., a_{ar(q)} \rangle \in q^{\mathfrak{A}}$, we have $\langle h(a_1), ..., h(a_{ar(q)}) \rangle \in q^{\mathfrak{B}}$. We say that a homomorphism is an **embedding** if h is injective.*

Note that the usual notion of embedding from model theory additionally requires that a "reverse homomorphism" condition hold: if $\langle h(a_1), ..., h(a_{ar(q)}) \rangle \in q^{\mathfrak{B}}$ then we must have $\langle a_1, ..., a_{ar(q)} \rangle \in q^{\mathfrak{A}}$. This condition is not required within the scope of this paper, but if it is desired it can be encoded by introducing for each relation q in the vocabulary a second relation q^C that holds the complement of q: a function that is homomorphic w.r.t. q^C is reverse homomorphic w.r.t. q.

The **structure embedding problem** is as follows: *given two finite structures over a common relational vocabulary, determine whether there is an embedding from one to the other.* The structure embedding problem is NP-complete, following immediately from the fact that subgraph isomorphism is a special case.

2.2 Monadic Structure Embedding

Although the structure embedding problem is NP-complete in the general case, it can be solved in polytime for monadic structures. This section describes a polytime reduction from monadic structure embedding to bipartite graph matching, which can be solved in $O(N^{5/2})$ time (where N is the number of vertices) [13].

First, recall the definitions of bipartite graphs and matchings:

Definition 3 (Bipartite graph, matching). *A **bipartite graph** $G = \langle U, V, E \rangle$ consists of two sets of vertices U and V and a set of edges $E \subseteq U \times V$. A **matching** in G is a set $M \subseteq E$ of edges such that no two edges share a common vertex. A matching M is **total** if its cardinality is equal to that of U. A total matching defines an injective function $f_M : U \hookrightarrow V$ where for each $u \in U$, $f_M(u)$ is defined to be the unique $v \in V$ such that $\langle u, v \rangle \in M$.*

The reduction of the monadic structure embedding problem to bipartite graph matching is based on the observation that the homomorphism condition acts on each element of the universe independently. That is, a function $h : \mathfrak{A} \to \mathfrak{B}$ is a homomorphism of monadic structures iff for each $a \in \mathfrak{A}$, $h(a)$ satisfies all the monadic predicates in $\mathfrak{B}$ that a does in $\mathfrak{A}$ (and additionally, the nullary predicates that hold in $\mathfrak{A}$ also hold in $\mathfrak{B}$, which is trivially checked). To capture this idea, we introduce *signatures* and *signatures graphs.*

Definition 4 (Signature, Signature graph). *Let $\sigma = \langle Q, ar \rangle$ be a vocabulary, let $\mathfrak{A}$ be a σ-structure, and let $a \in A$ be a member of its universe. The signature $sig(\mathfrak{A}, a)$ of a in $\mathfrak{A}$ is defined to be*

$$sig(\mathfrak{A}, a) \triangleq \{q \in Q : \exists \langle a_1, ..., a_n \rangle \in q^{\mathfrak{A}}. \exists i. a = a_i\}$$

*Let $\mathfrak{A}$ and $\mathfrak{B}$ be structures over a common vocabulary. The **signature graph** of $\mathfrak{A}$ and $\mathfrak{B}$ is a bipartite graph $Sig(\mathfrak{A}, \mathfrak{B}) = \langle A, B, E \rangle$ where the vertices $E \triangleq \{\langle a, b \rangle \in A \times B : sig(\mathfrak{A}, a) \subseteq sig(\mathfrak{B}, b)\}$.*

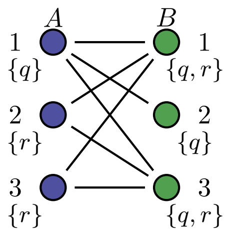
The intuitive idea behind the above definition is that b is a candidate target of a homomorphism for a iff $sig(\mathfrak{A}, a) \subseteq sig(\mathfrak{B}, b)$. The signature graph $Sig(\mathfrak{A}, \mathfrak{B})$ draws an edge from each element of $\mathfrak{A}$'s universe to its candidate targets in $\mathfrak{B}$. There is an embedding from $\mathfrak{A}$ to $\mathfrak{B}$ precisely when it is possible to select a distinct candidate target for each element of the universe — that is, there exists a total matching for $Sig(\mathfrak{A}, \mathfrak{B})$. Summarizing:

Observation 1 *Let $\mathfrak{A}$ and $\mathfrak{B}$ be structures over a common vocabulary σ.*

1. *For any embedding $f : A \hookrightarrow B$, the graph of f (the set $\{\langle a, f(a)\rangle : a \in A\}$) is a total matching in $Sig(\mathfrak{A}, \mathfrak{B})$.*
2. *If σ consists only of monadic predicates, then for every total matching M in $Sig(\mathfrak{A}, \mathfrak{B})$, f_M is an embedding.*

Example 2. Two monadic structures $\mathfrak{A}$ and $\mathfrak{B}$ over the vocabulary consisting of two monadic predicates q and r appears below to the left. In the center is the signature graph $Sig(\mathfrak{A}, \mathfrak{B})$; the signature of each element appears below it. To the right are the two total matchings of the signature graph (equivalently, the two embeddings of $\mathfrak{A}$ into $\mathfrak{B}$).

$\mathfrak{A} \triangleq \langle\{1,2,3\}, q^{\mathfrak{A}}, r^{\mathfrak{A}}\rangle$,
$\mathfrak{B} \triangleq \langle\{1,2,3\}, q^{\mathfrak{B}}, r^{\mathfrak{B}}\rangle$
where:
$q^{\mathfrak{A}} \triangleq \{1\}$
$r^{\mathfrak{A}} \triangleq \{2,3\}$
$q^{\mathfrak{B}} \triangleq \{1,2,3\}$
$r^{\mathfrak{B}} \triangleq \{1,3\}$



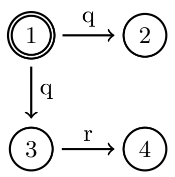
Total Matchings:
$M_1 = \{\langle 1,2\rangle, \langle 2,1\rangle, \langle 3,3\rangle\}$
$M_2 = \{\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,1\rangle\}$

2.3 General Structure Embedding

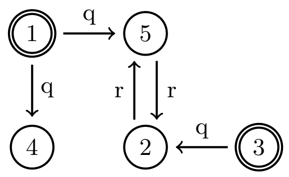
This section presents the MatchEmbeds algorithm, the main contribution of this paper. MatchEmbeds is a backtracking search algorithm that uses bipartite graph matching to guide search. The algorithm is designed to be fast on monadic (or nearly monadic — contains at most a small constant number of non-empty, non-monadic relations) structures, and have good practical performance on general structures. In the case that MatchEmbeds is applied to monadic structures, it operates in polytime, effectively applying the reduction to matching described in the previous section. In general (non-monadic structures) it can (in the worst case) take time proportional to the number of total matchings in the signature graph for the instance.

First, we give an example showing why the reduction to bipartite graph matching does not work for general structures:

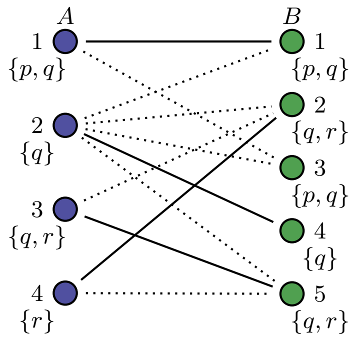
Running Example 1. Consider a vocabulary σ consisting of one monadic relation p and two binary relations q, r, and the two σ-structures $\mathfrak{A}$ and $\mathfrak{B}$ visualized in Figure 1(a) and (b). Members of the monadic relation p are illustrated with double circles; the binary relations are illustrated with labeled edges.



(a) Structure $\mathfrak{A}$



(b) Structure $\mathfrak{B}$



(c) Signature graph $Sig(\mathfrak{A}, \mathfrak{B})$.

$M_1 = \{\langle 1,1\rangle, \langle 2,4\rangle, \langle 3,2\rangle, \langle 4,5\rangle\}$
$\mathbf{M_2} = \{\langle \mathbf{1},\mathbf{1}\rangle, \langle \mathbf{2},\mathbf{4}\rangle, \langle \mathbf{3},\mathbf{5}\rangle, \langle \mathbf{4},\mathbf{2}\rangle\}$
$M_3 = \{\langle 1,3\rangle, \langle 2,4\rangle, \langle 3,2\rangle, \langle 4,5\rangle\}$
$M_4 = \{\langle 1,3\rangle, \langle 2,4\rangle, \langle 3,5\rangle, \langle 4,2\rangle\}$
$M_5 = \{\langle 1,1\rangle, \langle 2,3\rangle, \langle 3,2\rangle, \langle 4,5\rangle\}$
$M_6 = \{\langle 1,1\rangle, \langle 2,3\rangle, \langle 3,5\rangle, \langle 4,2\rangle\}$
$M_7 = \{\langle 1,3\rangle, \langle 2,1\rangle, \langle 3,2\rangle, \langle 4,5\rangle\}$
$M_8 = \{\langle 1,3\rangle, \langle 2,1\rangle, \langle 3,5\rangle, \langle 4,2\rangle\}$

(d) Total matchings (**Embeddings**).

Fig. 1: Running example.

The signature graph $Sig(\mathfrak{A}, \mathfrak{B})$ is depicted in Figure 1(c). All edges (dotted and solid) belong to $Sig(\mathfrak{A}, \mathfrak{B})$; the solid edges belong to an embedding, dotted do not. Observe while $Sig(\mathfrak{A}, \mathfrak{B})$ has eight total matchings, only one of them corresponds to an embedding.

Intuitively, the reason that the reduction to bipartite graph matching does not work for general structures is that the homomorphism condition for relations of arity greater than one is not captured by the signature graph. As a result, for a given candidate matching M there may be tuples belonging to relations of the source structure that have no corresponding tuple in the image of f_M. This idea is encapsulated by the following definition of *conflict*:

Definition 5 (Conflict set). *Let $\mathfrak{A}$ and $\mathfrak{B}$ be structures over a common vocabulary $\sigma = \langle Q, ar\rangle$, and let $f : A \to B$ be a function. The* ***conflict set*** *of f is the set*

$$conflict(f) \triangleq \{q(a_1, ..., a_n) : q \in Q, \langle a_1, ..., a_n\rangle \in q^{\mathfrak{A}}, \langle f(a_1), ..., f(a_n)\rangle \notin q^{\mathfrak{B}}\} \ .$$

Note that f is a homomorphism iff its conflict set is empty.

Running Example 2. Conflict sets for $M_1, \dots, M_8$.

$conflict(f_{M_1}) \triangleq \{q(1,3)\}$	$conflict(f_{M_5}) \triangleq \{q(1,2), q(1,3)\}$
$conflict(f_{M_2}) \triangleq \emptyset$	$conflict(f_{M_6}) \triangleq \{q(1,2)\}$
$conflict(f_{M_3}) \triangleq \{q(1,2)\}$	$conflict(f_{M_7}) \triangleq \{q(1,2)\}$
$conflict(f_{M_4}) \triangleq \{q(1,2), q(1,3)\}$	$conflict(f_{M_8}) \triangleq \{q(1,2), q(1,3)\}$

The MatchEmbeds algorithm searches the space of total matchings of $Sig(\mathfrak{A}, \mathfrak{B})$, trying to find a matching that corresponds to an embedding (following point 1 of Observation 1). If a given candidate matching is *not* an embedding, its conflict set tells us *what went wrong*, which we can use to guide the search away from the current candidate matching and hopefully other candidates that will fail for the same reason. The choices that can be made to guide search away from a failed candidate matching are encapsulated by *decisions*:

Definition 6 (Decision). *Let $\mathfrak{A}$ and $\mathfrak{B}$ be structures over a common vocabulary σ, $G = \langle A, B, E\rangle$ a bipartite graph over A and B, and M a total matching on G. A **decision of** M is an edge $\langle a, b\rangle \in M$ such that (1) the degree of a is greater than one in G (i.e., there is some other choice available for a), and (2) there is some conflict $q(a_1, ..., a_{ar(q)}) \in conflict(f_M)$ that involves a ($a = a_i$ for some i).*

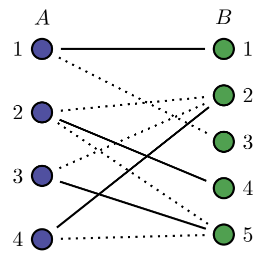
MatchEmbeds represents a search space of candidate matchings as a bipartite graph G. It can split the search space by choosing a decision $\langle a, b\rangle$ and either *committing* to it (by removing every edge incident to a and b in G excluding $\langle a, b\rangle$) or *eliminating* it (by removing $\langle a, b\rangle$ from G). But in either case, we would like to avoid exhaustively searching through all candidate matchings. Furthermore, when we explore the branch of the search space in which we commit to the decision $\langle a, b\rangle$ (which we know was part of a conflict in the "current" candidate matching) we would like to be able to make progress — to remove matchings from the search space that fail for the same essential reason. Both of these goals are accomplished by employing *constraint propagation*, a classic technique in CSP solving (more precisely, generalized arc consistency) [23, Ch. 6]. The idea is to identify edges in G that cannot be part of any embedding and to remove them (and thereby eliminate any matching that uses them). We formalize this idea with the notion of *consistent* graphs (which do not contain such edges):

Definition 7 (Consistency). *Let $\mathfrak{A}$ and $\mathfrak{B}$ be structures over a common vocabulary $\sigma = \langle Q, ar\rangle$. Given a bipartite graph $G = \langle A, B, E\rangle$, we say that an edge $\langle a, b\rangle \in E$ is **consistent with** $\langle a_1, ..., a_{ar(q)}\rangle \in q^{\mathfrak{A}}$ when for all positions $i \in [1, ar(q)]$ such that $a = a_i$, there is some $\langle b_1, ..., b_{ar(q)}\rangle \in q^{\mathfrak{B}}$ such that $b = b_i$ and for all positions $j \in [1, ar(q)]$, $\langle a_j, b_j\rangle \in E$. We say that G is **consistent** when for all $\langle a, b\rangle \in E$, all $q \in Q$, and all $\alpha \in q^{\mathfrak{A}}$, $\langle a, b\rangle$ is consistent with α.*

Definition 8 (Maximum Consistent Sub-Graph). *Let $\mathfrak{A}$ and $\mathfrak{B}$ be structures over a common vocabulary σ. Given a bipartite graph $G = \langle A, B, E\rangle$, the **maximum consistent sub-graph** of G is a graph $G' = \langle A, B, E'\rangle$ such that (1) $E' \subseteq E$ (2) G' is consistent (3) there is no G'' such that 1 and 2 hold and $|G'| < |G''|$ (G'' contains more edges than G'). For any G, we define $filter(G)$ to be the maximum consistent sub-graph of G.*

Efficient implementation of *filter* is discussed in Section 3. The crucial property of filtering is that it preserves all embeddings:

Proposition 1. *Let $\mathfrak{A}$ and $\mathfrak{B}$ be structures and let $G = \langle A, B, E\rangle$ be a bipartite graph. For any embedding $f : A \hookrightarrow B$ such that G contains the graph of f (for all $a \in A$, $\langle a, f(a)\rangle \in E$), $filter(G)$ also contains the graph of f.*



Running Example 3. The picture to the right illustrates the maximum consistent subgraph of the signature graph from the running example (Figure 1(c)). The edge $\langle 2, 1\rangle$ is inconsistent with the constraint $q(1, 2)$ (i.e., 2 cannot map to 1 because 2 has an incoming q-edge in $\mathfrak{A}$ and 1 has no incoming q-edge in $\mathfrak{B}$); similarly, the edge $\langle 2, 3\rangle$ is inconsistent with the constraint $q(1, 2)$. The edges $\langle 2, 1\rangle$ and $\langle 2, 3\rangle$ are removed from the signature graph, which eliminates half of the candidate total matchings (M_5 through M_8). The one total matching corresponding to an embedding (M_2) remains, along with three other candidate total matchings.

Now we have all of the machinery necessary to define our algorithm. We define MatchEmbeds in terms of the recursive sub-procedure **embeds** as shown in Algorithm 1. At a high-level, **embeds** explores the space of total matchings in the given bipartite graph G, searching for an embedding (G is initially the signature graph, $Sig(\mathfrak{A}, \mathfrak{B})$). We first try to compute a total matching on G. If we fail then we backtrack, returning false if no further decision is left to backtrack. Otherwise, we have a candidate total matching M and we check if f_M is an embedding. If so, we return true; otherwise, we select a decision $\langle a, b\rangle$ and branch on it.

Some more care is needed to understand how a decision $\langle a, b\rangle$ is selected from M. How can we be assured that there is some decision to select? When control reaches the decision selection point, we know that (1) G is consistent, (2) it contains a total matching M, and (3) f_M is not an embedding. Since f_M is not an embedding, it must have at least one conflict — say $q(a_1, ..., a_{ar(q)})$. Some $\langle a_i, f_M(a_i)\rangle$ must be a decision, because if none are then G is inconsistent with $\langle a_1, ..., a_{ar(q)}\rangle$. Thus, there is always at least *some* decision to choose. How do we choose which one? While any choice is enough to ensure correctness of MatchEmbeds, in practice we found that choosing a decision $\langle a, b\rangle$ that minimizes the degree of a works well (this is essentially the minimum remaining values heuristic in CSP literature).

Next, we remark on the design choice that MatchEmbeds explores the branch that *commits* to the decision $\langle a, b\rangle$ *first* (after all, we know that $\langle a, b\rangle$ is involved in a conflict). The reason is two-fold. First, observe that for a binary proposition $p(a_1, a_2)$ to be involved in a conflict, both $\langle a_1, f_M(a_1)\rangle$ and $\langle a_2, f_M(a_2)\rangle$ must be decisions (otherwise, G is inconsistent) — we must change one of the decisions, but *which* one is arbitrary. In either case, the same matching M cannot be computed in the next recursive call to the algorithm. (For a conflict involving an n-ary predicate, we must decide on $n - 1$ decisions in the conflict to ensure we discard the candidate matching). Second, observe that we need not recompute a matching from scratch: many edges may be shared between the previous candidate and the next one. Our implementation uses the algorithm of Ford and Fulkerson [11] to compute matchings, which benefits from starting from a *partial matching* consisting of the edges of the previous candidate matching that were not removed by *filter*.

Algorithm 1: MatchEmbeds

```
Data: $\mathfrak{A}$ and $\mathfrak{B}$ finite structures over a relational vocabulary $\sigma = \langle Q, ar \rangle$.
Result: true $\iff$ there exists an embedding from $\mathfrak{A}$ to $\mathfrak{B}$.

Function embeds($G$)
    $G \leftarrow$ filter($G$)
    $M \leftarrow$ maximum_matching($G$)
    if $|M| \neq |G.A|$ then
        return false
    end
    if $f_M$ is an embedding then
        return true
    end
    Select a decision $\langle a, b \rangle$ of $M$
    if embeds($G \setminus \{\langle u, v \rangle \in E : u = a \text{ xor } v = b\}$) then
        return true
    else
        return embeds($G \setminus \{\langle a, b \rangle\}$)
    end

if there is some $q \in Q$ with $ar(q) = 0$, $q^{\mathfrak{A}} \neq \emptyset$, and $q^{\mathfrak{B}} = \emptyset$ then
    return false
else
    return embeds(Sig($\mathfrak{A}, \mathfrak{B}$))
end
```

Running Example 4. Figure 2 illustrates the execution of MatchEmbeds on the embedding instance from the Running Example. We start by computing a total matching M on G. We observe that f_M is not an embedding, and compute its conflicts and decisions. We select the decision $\langle 3, 2 \rangle$ and filter the graph — the result is empty. Unable to compute a total matching, we backtrack and blame the decision $\langle 3, 2 \rangle$; we remove it from the graph and once again filter. We then compute another total matching on the graph. This matching corresponds to an embedding from $\mathfrak{A}$ to $\mathfrak{B}$, so we return **true**.

3 Discussion

This section discusses some additional ideas that are important for the practical performance of MatchEmbeds. We also discuss a data structure for organizing a collection of structures for a multi-source single-target variation of the structure embedding problem, which is useful for our target application of testing emptiness of predicate automata.

Refined signatures Definition 4 shows how to associate a bipartite graph with a pair of structures $\mathfrak{A}$ and $\mathfrak{B}$ over a common vocabulary $\sigma = \langle Q, ar \rangle$ by drawing an edge from $a \in \mathfrak{A}$ to $b \in \mathfrak{B}$ iff the set of predicates that involve a in $\mathfrak{A}$ — the *signature* of a in $\mathfrak{A}$ — is a subset of the predicates that involve b in $\mathfrak{B}$. A simple generalization of this idea is to define a partial order $\langle P, \leq \rangle$ and a

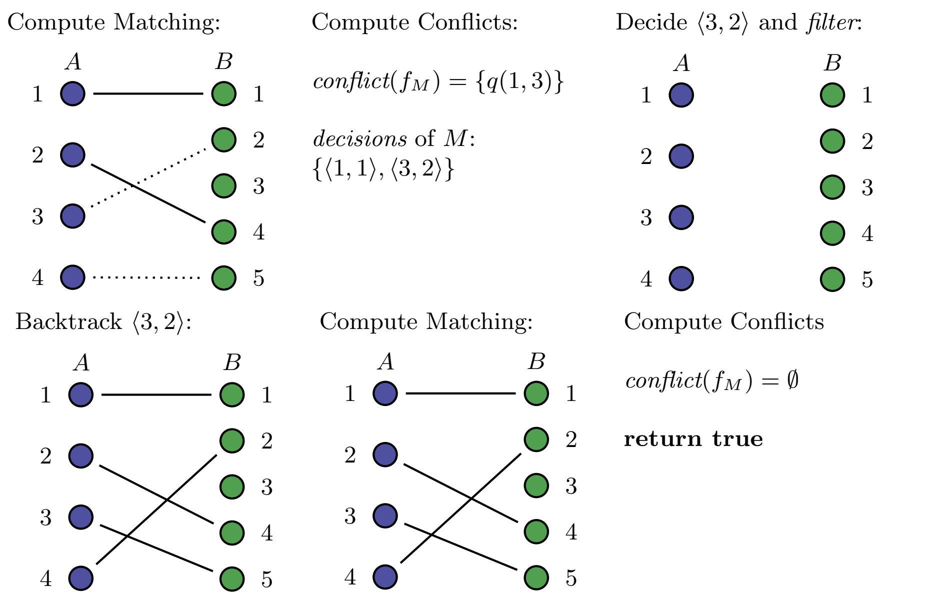


Fig. 2: The operation of MatchEmbeds on the running example.

function $sig(\cdot, \cdot)$ that maps a structure and a member of its universe into P such that

1. If $h : \mathfrak{A} \hookrightarrow \mathfrak{B}$ is an embedding, then $sig(\mathfrak{A}, a) \leq sig(\mathfrak{B}, h(a))$, and
2. If $sig(\mathfrak{A}, a) \leq sig(\mathfrak{B}, b)$ then for all monadic $q \in Q$ such that $a \in q^{\mathfrak{A}}$ we have $b \in q^{\mathfrak{B}}$.

The associated bipartite graph for such a refined notion of signature is formed by drawing an edge from a to b iff $sig(\mathfrak{A}, a) \leq sig(\mathfrak{B}, b)$.

Signatures can be used to encode various properties that are monotone w.r.t homomorphism (e.g., the size of the connected component of a binary relation to which an element belongs). Using a more refined notion of signature can yield smaller signature graphs, which results in graph matching being a more informative heuristic. In our implementation of MatchEmbeds, we use the partial order $(Q \times \mathbb{N}) \to \mathbb{N}$ (i.e., multisets of predicates indexed by the position an element appears), with

$$sig(\mathfrak{A}, a)(q) \triangleq |\{(\langle a_1, ..., a_{ar(q)} \rangle, i) : \langle a_1, ..., a_{ar(q)} \rangle \in q^{\mathfrak{A}} \wedge a_i = a\}|$$

E.g., for the special case of a binary predicate e, $sig(\mathfrak{A}, a)(e)$ is the total degree of a in the graph $\langle A, e^{\mathfrak{A}} \rangle$.

An algorithm for enforcing consistency A crucial factor in the practical performance of MatchEmbeds is the algorithm *filter* that enforces consistency of a graph. To support this operation we make use of an auxiliary bipartite graph $G_q = \langle q^{\mathfrak{A}}, q^{\mathfrak{B}}, E_q \rangle$ for each predicate q of arity ≥ 2. Our implementation of *filter* repeatedly iterates over each G_q as well as the graph $G = \langle A, B, E \rangle$ while performing the following update rules:

1. If $(\langle a_1, ..., a_{ar(q)}\rangle, \langle b_1, ..., b_{ar(q)}\rangle) \in E_q$ and $\langle a_i, b_i\rangle \notin E$ for some i, remove it from E_q.
2. If $\langle a, b\rangle \in E$, $\langle a_1, ..., a_{ar(q)}\rangle \in q^{ar(q)}$ with $a = a_i$, and there is no edge $\langle\langle a_1, ..., a_{ar(q)}\rangle, \langle b_1, ..., b_{ar(q)}\rangle\rangle \in E_q$ with $b = b_i$, remove $\langle a, b\rangle$ from E.

The *filter* algorithm keeps applying these two rules until no more apply (a fixed point is reached).

A structure embedding database In the context of predicate automaton emptiness checking, the problem of interest is to check whether any structure within a given set of structures (i.e., the states of the automaton that have already been explored) embeds into another given structure (i.e., some new candidate state). To solve this problem, we require a data structure that stores a set of structures, and that supports an embedding query operation that can test if any member of this set embeds into a given structure (ideally without simply testing embedding for each structure in the set).

We use k-d trees [2] to organize the database of structures, and use range queries to support multi-source single-target embedding problems. The idea is to associate each structure $\mathfrak{A}$ with a vector $v(\mathfrak{A}) \in \mathbb{N}^d$ (for some fixed dimension d) such that if $\mathfrak{A}$ embeds into $\mathfrak{B}$, then $v(\mathfrak{A}) \leq v(\mathfrak{B})$. By storing structures in a k-d-tree-based map that is keyed by these vectors, we can support multi-source embedding queries by using a range query to search for structures keyed by vectors less than a given target vector, and attempt structure embedding only for the subset of structures returned. In our implementation, we use $|Q|$-dimensional binary vectors where $v(\mathfrak{A})_q = 0$ if $q^{\mathfrak{A}} = \emptyset$ and $v(\mathfrak{A})_q = 1$ otherwise.

4 Experiments

In this section, we evaluate the performance of MatchEmbeds by comparing it against three CSP solvers (Gecode [24], HaifaCSP [27], and Google's or-tools [20]), two SAT solvers (Lingeling [3] and CryptoMiniSat [26]), and two subgraph isomorphism solvers (Boost's implementation of VF2 [19] and Glasgow [16]). Our experiments are designed to answer three questions:

1. Does MatchEmbeds improve the performance of its intended client application of subsumption checking in state-space exploration of parameterized concurrent programs?
2. Does the k-d tree data structure improve the performance of many-to-one structure embedding queries in predicate automata emptiness checking?
3. Is MatchEmbeds capable of solving difficult problem instances?

4.1 Predicate automata emptiness

We integrated MatchEmbeds into a prototype implementation of the software model checking algorithm proposed in [8]. The model checking algorithm operates by iteratively constructing a predicate automaton that recognizes a *safe* set of executions and checking whether all program traces are contained inside the safe one (so a single verification task involves many predicate automaton emptiness tests, each of which involves many structure embedding instances). We experimented with small synthetic benchmark programs that were designed to stress-test the structure embedding procedure. While these programs are small and synthetic it allows us to control both the universe and arity of predicates involved in any structure. We used the API provided with Gecode and Boost to integrate it into the prototype. We used the text interface provided by HaifaCSP, or-tools, Lingeling, and CryptoMiniSat which bears a performance penalty.

Count threads : we consider a family of programs wherein the main thread spawns some number of threads N, each of which atomically increments a global variable `count`, and then finally asserts that `count` is no greater than N. We expect the count threads benchmark to produce monadic, or nearly monadic structures, but to vary in size with many structures having a universe size close to N, as we need to explore the execution of all threads to verify that the assertion holds.

```
main():
  count = 0
  for i = 1 to N:
    fork thread
  assert(count <= N)
```

```
thread():
  count = count + 1
```

Secret sharing : we consider a family of programs wherein all threads execute a protocol that results in having a shared secret. The significance is that the shared secret forces the use of a binary predicate that expresses that two threads have the same secret value. The correct protocol is shown in `main_safe` below: it allocates a secret positive number, spawns an arbitrary number of threads, sends it to each using the `to` variable, then checks if it has received a message in the `from` variable. If so, it asserts that the received message is equal to its secret. The incorrect protocol is shown in `main_bug`. It does the same, except that it computes a new secret for every N threads, where N is a parameter to the system. The assertion may fail if at least N threads are spawned. The correct version can be verified in 0.77 seconds with MatchEmbeds, 0.78 with VF2, 0.80 with Gecode, 5.36 with Lingeling, 11.02 with CryptoMiniSat, 30.99 with HaifaCSP, and 41.25 with or-tools.

```
main_safe():
 local secret = *
 assume(secret > 0)
 from = 0
 while (*):
   to = secret
   fork thread
   while (to > 0): skip
 if (from > 0):
   assert(from == secret)
```

```
main_bug():
 from = 0
 while (*):
   local secret = *
   assume(secret > 0)
   for (i = 1 to N):
     to = secret
     fork thread
     while (to > 0): skip
   if (from > 0):
     assert(from == secret)
```

```
thread():
 local m = to
 to = 0
 from = m
```

We expect the secret sharing benchmark to produce mostly binary structures, that form a hub-and-spoke topology, since the local threads only interact with the main thread.

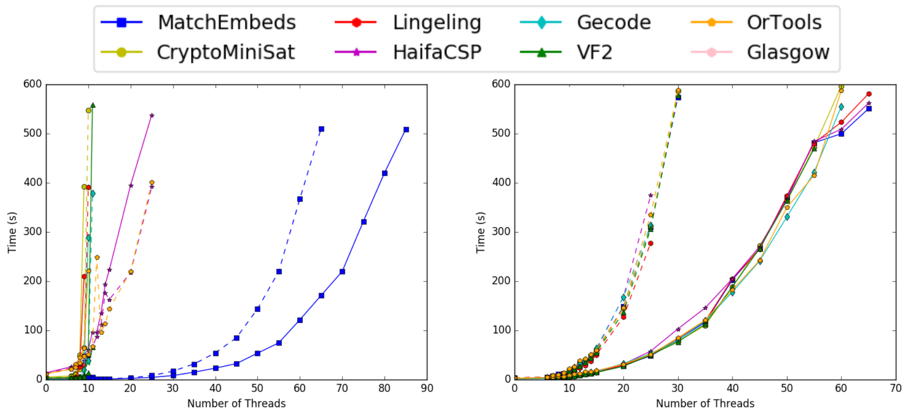
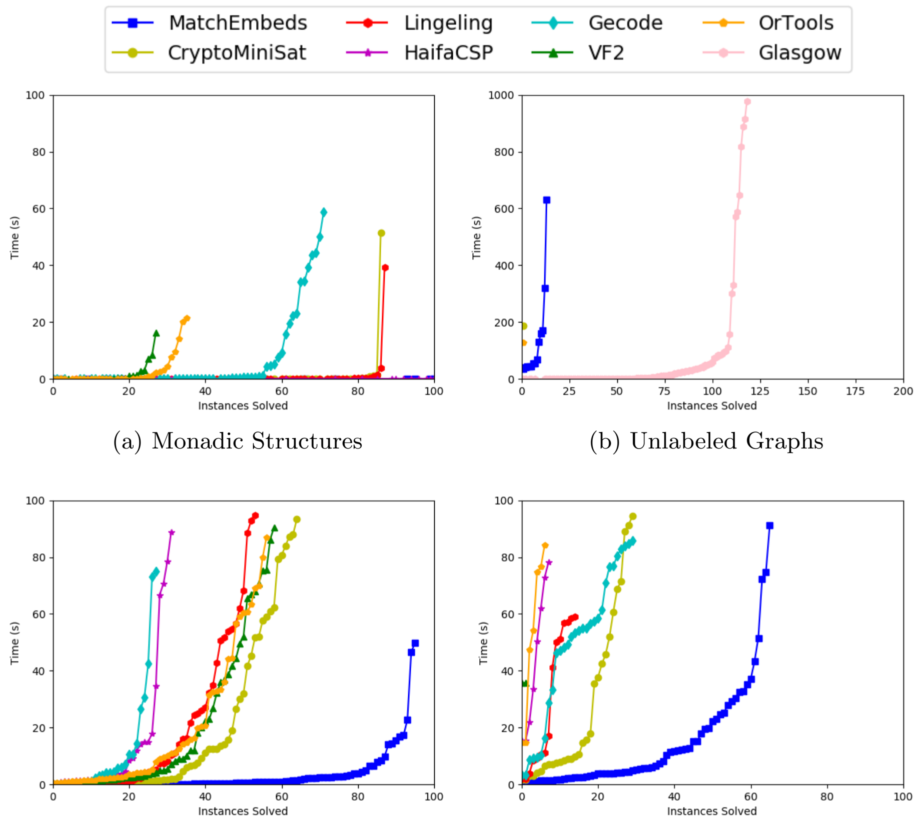


Fig. 3: Proof Space Benchmarks: Count Threads (Left), Secret Sharing (Right). Solid line indicates k-d tree, dashed line indicates list data structure.

In Figure 3, we compare the results of each tool, with and without the use of the k-d tree data structure, on the *Count Threads* and *Secret Sharing* parametric benchmarks. In the *Count Threads* benchmark, MatchEmbeds substantially outperforms all other solvers, verifying up to 85 threads; HaifaCSP and CryptoMiniSat, the two next closest, reached up to 25. In the *Secret Sharing* benchmark we see a different story: almost identical performance from each solver, but a large performance gain from the k-d tree. When using the k-d data structure, only a small fraction — less than 1/50th — of the time was spent on embedding queries. In contrast, without the k-d tree data structure almost the entirety of the verification task was spent solving embedding instances. We see similar improvements in the *Count Threads* benchmark when using MatchEmbeds. For the 30 thread secret sharing benchmark, we see that the k-d tree only performs 220 thousand embedding queries while the naive method explores 318 million embeddings. We note a similar reduction in number of embedding queries for all benchmarks. We suspect the similar performance of all solvers in the *Secret Sharing* benchmark is due to the topology of the structures resulting in easy embedding instances.

4.2 Hard instances

The previous experiment demonstrates that MatchEmbeds is able to very quickly solve the (typically easy) embedding problems that arise in predicate automaton emptiness checking. The second question we would like to answer is whether MatchEmbeds also works well for larger, more difficult instances. To answer this question, we compared the performance of MatchEmbeds against SAT, CSP, and subgraph isomorphism solvers on a suite of *hard* randomly generated benchmarks.



(a) Monadic Structures (b) Unlabeled Graphs

(c) Binary Structures (c) Ternary Structures

Fig. 4: Embedding instances solved within time for each benchmark.

4.3 Random embedding

Results We randomly generated a suite of difficult monadic, binary, and ternary structure embedding problems, pictured in Figure 4. For each monadic instance, the source universe size is 40, target universe size is 50, and the vocabulary consists of 3 monadic predicates; the suite has 53 satisfiable and 47 unsatisfiable instances. For each binary instance, the source universe size is 20, target universe size is 30, and the vocabulary has 3 monadic and 3 binary predicates; the suite has 46 satisfiable, 49 unsatisfiable, and 5 unsolved instances. For each ternary instances, the source universe size is 10, the target universe size is 30, the vocabulary had 3 monadic, 3 binary, and 3 ternary predicates; the suite has 35 satisfiable, 32 unsatisfiable, and 33 unsolved instances.

Cactus plots comparing the performance of the structure embedding solvers on the random embedding instances are pictured in Figure 4. In a cactus plot, the x-axis denotes the total number of instances solved, and the y-axis denotes time. A point (x, y) denotes that within a timeout of y, x of the instances can be solved.

For monadic structures, both HaifaCSP and MatchEmbeds perform well: all instances can be solved in less than a second—their graphs are barely visible above the x-axis; the next best solvers, CryptoMiniSat and lingeling, solved 87 and 86 instances respectively. For binary structures, MatchEmbeds is able to solve 95 instances, solving 60 of these in under one second. CryptoMiniSat,the next best solver, solved 64 instances and required substantially longer to solve 40 of those instances. VF2 solves 58, OrTools 56, Lingeling 53, HaifaCSP 31, and Gecode 27 in the 100s time limit. For the ternary benchmark, MatchEmbeds was able to solve 65 instances where the next best solvers, OrTools and Gecode could solve only 29 instances, taking more time on many of those instances. VF2 does not appear in the ternary figure as it failed to solve any instances.

Method We now describe our methodology for randomly generating hard problem instances.

Generalizing the Erdős-Rényi method for generating random graphs, we generate random structures as follows: given as parameters a universe size n, a finite relational vocabulary $\sigma = \langle Q, ar \rangle$, and a density function $d : Q \to [0, 1]$, we generate a random structure $\mathfrak{A}(n, \sigma, d)$ by iterating over all k-tuples $(a_1, ..., a_k) \in \{1, ..., n\}^k$ and including the proposition $q(a_1, ..., a_n)$ in $\mathfrak{A}(n, \sigma, d)$ with probability $d(q)$. We generate a random embedding problem by generating two such random structures (with the same vocabulary, but possibly different universe sizes and density functions).

We now turn to the problem of how to randomly sample parameters (universe sizes and predicate densities) that result in *hard* problem instances, following the insight of Cheeseman et. al. that hard random instances lie near the phase shift — the parameters used to generate the instances have probability $\sim$0.5 to produce satisfiable instances [5]. Similarly to McCreesh et al.'s method for generating hard subgraph ismorphism instances [17], we fix the source and target universe sizes and the vocabulary, and search only over densities. We aim to sample density functions that achieve near parity between the number of satisfiable and unsatisfiable embedding instances.

Our method is based on the assumption that parameters with similar expected number of solutions also have similar probabilities of there existing at least one solution. The expected number of solutions for a given set of parameters, which we denote $E(n, m, d^{\mathfrak{A}}, d^{\mathfrak{B}})$, has a simple closed form formula (which we derive below). Using this formula, we find a target expected number $T(n, m, d^{\mathfrak{A}}, d^{\mathfrak{B}})$ experimentally, using a binary search for an expected number such that the ratio between satisfiable and unsatisfiable instances, estimated by generating 10 instances at random, is nearly 1/2 (there is a 5-5 or 6-4 split). We then generate a random hard instance as follows. First, we randomly sample parameters that achieve (nearly) the target number of solutions by using stochastic gradient decent to minimize $|T(n, m, d^{\mathfrak{A}}, d^{\mathfrak{B}}) - E(n, m, d^{\mathfrak{A}}, d^{\mathfrak{B}})|$. Specifically, we uniformly at random pick a parameter, and perform 1 iteration of stochastic gradient descent using the partial derivative w.r.t. the chosen parameter, and repeat this process until E converges (within 0.005) to T. We then use these parameters to produce random embedding problems until we find one that is non-trivial (at least one solver takes more than 1 second to solve). To generate the benchmark suites pictured in Figure 4, we repeat this process 100 times.

We conclude with a derivation of a formula to calculate the expected number of embeddings for a random embedding problem. Observe that the number of injective functions from a set of size n to a set of size m is given by $m!/(m-n)!$. For any given injective function h, recall that an $ar(q)$-tuple belongs to $q^{\mathfrak{A}}$ with probability $d^{\mathfrak{A}}(q)$ and its image, $\langle h(a_1), ..., h(a_{ar(q)})\rangle$, belongs to $q^{\mathfrak{B}}$ with probability $d^{\mathfrak{B}}(q)$; thus the probability that h satisfies the homomorphism condition for a given predicate q and $ar(q)$-tuple is $d^{\mathfrak{A}}(q)d^{\mathfrak{B}}(q) + (1 - d^{\mathfrak{A}}(a))$. Since there are $n^{ar(q)}$ such tuples and each event is independent, we raise the probability to the $n^{ar(q)}$ power to get the probability of all $ar(q)$-tuples satisfying the homomorphism conditions for q. Taking the product over all $q \in Q$, then gives the probability that h is a homomorphism. We can multiply this probability by the total number of injective functions to arrive at the function E that computes the expected number of embeddings for structures sampled using the given parameters:

$$E(n, m, d^{\mathfrak{A}}, d^{\mathfrak{B}}) = \frac{m!}{(m-n)!} \prod_{q \in Q} (d^{\mathfrak{A}}(q)d^{\mathfrak{B}}(q) + (1 - d^{\mathfrak{A}}(q)))^{n^{ar(q)}}$$

4.4 Unlabeled subgraph isomorphism

(Unlabeled) subgraph isomorphism is a special case of the structure embedding problem that has received considerable attention (see Section 5). Figure 4(b) compares the performance of CSP, SAT, and subgraph isomorphism solvers on a suite of 200 *hard* random subgraph isomorphism instances from [17]. We included the Glasgow subgraph isomorphism solver in this benchmark, as it was the leading solver in [17]; it is excluded from our other experiments because it does not support labeled subgraph isomorphism. In this benchmark, all source graphs consist of 30 vertices and all target graphs contain 150 vertices. Glasgow outperforms all other solvers on these instances, solving 118; MatchEmbeds performs second best, solving just 13 instances in the 1000s time limit. All other solvers solved at most one instance. The poor performance of MatchEmbeds (relative to the previous experiments) is expected: MatchEmbeds matching-based heuristic search is uninformative in this setting. Since the signature of any vertex in an unlabeled graph is exactly the total degree of that vertex and random graphs are likely to have many vertices with similar degree, the expectation is that signature graphs will be dense and MatchEmbeds will have little information to exploit. We expect more informative signatures to result in MatchEmbeds performing better on unlabeled graphs.

4.5 Encoding structure embedding into CSP

A constraint satisfaction problem consists of a finite set of variables $X = \{x_1, ..., x_n\}$, with each variable $x_i \in X$ associated with a finite domain D_i that determines which values that x_i, and a finite set of constraints among those variables. Given two structures $\mathfrak{A}$ and $\mathfrak{B}$ over a common vocabulary $\langle Q, ar\rangle$, we construct the following CSP. We introduce a variable x_a for each $a \in A$ with domain $D_a = \{b \in B : sig(\mathfrak{A}, a) \subseteq sig(\mathfrak{B}, b)\}$. We add the constraint $\texttt{alldifferent}(X)$, which asserts that each a must map to a unique b (i.e. $\forall x_a, x_{a'} \in X.x_a \neq x_{a'}$). Then for each $\langle a_1, ..., a_n\rangle \in q^{\mathfrak{A}}$ we introduce a constraint $C_\alpha = \bigvee_{\langle b_1,...,b_n\rangle \in q^{\mathfrak{B}}} (\bigwedge_i x_{a_i} = b_i)$ to ensure the homomorphism condition. The CSP is satisfiable iff $\mathfrak{A}$ embeds into $\mathfrak{B}$.

4.6 Encoding structure embedding into SAT

Let $\mathfrak{A}$ and $\mathfrak{B}$ be structures over a common vocabulary $\langle Q, ar\rangle$. For each edge $\langle a, b\rangle$ in the signature graph $\langle A, B, E\rangle = Sig(\mathfrak{A}, \mathfrak{B})$, we introduce one propositional variable $p_{a,b}$, with the interpretation that $p_{a,b}$ is set iff a maps to b in an embedding from $\mathfrak{A}$ to $\mathfrak{B}$. For each $a \in A$ we introduce a constraint $\bigvee_{\langle a,b\rangle \in E} p_{a,b}$ to encode that a must have an image. We encode that a has at most one image ensuring that for each $a \in A$, at most one of $\{p_{a,b} | b \in Adj(a)\}$ holds, using the sequential counter encoding of that at-most-1 constraint [25]. Similarly, we enforce injectivity by ensuring that for each $b \in B$, at most one of $\{p_{a,b} | a \in Adj(b)\}$ holds. Finally, for each $q \in Q$ and $\langle a_1, ..., a_n\rangle \in q^{\mathfrak{A}}$, we introduce a constraint $\bigvee_{\langle b_1,...,b_n\rangle \in q^{\mathfrak{B}}} (p_{a_1,b_1} \wedge \cdots \wedge p_{a_n,b_n})$ to encode the homomorphism condition. The resulting formula is satisfiable iff $\mathfrak{A}$ embeds into $\mathfrak{B}$.

4.7 Encoding structure embedding into labeled subgraph isomorphism

Given a structures $\mathfrak{A}$ with vocabulary $\langle Q, ar\rangle$, we generate a graph $G(\mathfrak{A})$. For each $q \in Q$ we introduce a vertex label, l_q and for each $i \in [1, ar(q)]$ we introduce an edge label l_{qi}. Then for each universe element, $a \in A$, we introduce a vertex a in G. Additionally, for each tuple $\alpha \in q^{\mathfrak{A}}$, we introduce a vertex v_α and for each $i \in [1, ar(q)]$ we introduce an edge $\langle v_\alpha, a_i\rangle$ with edge label l_{qi}. Note, for both the binary and monadic embeddings, the VF2 implementation allows labeling vertices and edges with sets of labels and using set inclusion for matching vertex and edge labels, we are able to directly encode monadic and binary structures without adding any additional vertices and edges (i.e. $l(v) = \{q : v \in q^{\mathfrak{A}}\}$ and $l(\langle u, v\rangle) = \{q : \langle u, v\rangle \in q^{\mathfrak{A}}\}$).

5 Related Work

Constraint satisfaction problems Constraint satisfaction problems (CSPs) are a broad class of combinatorial problems that includes structure embedding. A good introduction appears in [23, Ch. 6]. MatchEmbeds employs several ideas that are commonly used in resolution algorithms for CSPs, including backtracking search, filtering (constraint propagation), and heuristics for decision selection (variable and value selection). The hypothesis of our work, validated in Section 4, was that by exploiting the injectivity feature of structure embedding we could outperform general-purpose CSP solvers.

Of particular relevance to structure embedding is work on the `alldifferent` constraint, which requires a specified subset of variables in the problem to be assigned distinct values (mirroring the injectivity condition of structure embeddings). The work most relevant to ours is Régin's domain consistency algorithm for `alldifferent`. Régin's algorithm uses biparite graph matching to discover all edges in a value graph of a CSP (analogous to the signature graph of a structure embedding problem) that do not belong to *any* total matching and deletes them. Efficient algorithms for weaker notions of consistency (namely bounds consistency) have also been developed [21,15]. A survey on the `alldifferent` constraint can be found in [12]. Considering the CSP solvers included in Section 4: HaifaCSP implements Régin's algorithm [22] and Gecode and or-tools implement the algorithm of Lopez-Ortiz et al [15]. The superior performance of HaifaCSP (particularly for monadic structures) demonstrates the importance of the `alldifferent` constraint.

A commonality of these works is that they are *constraint propagation* techniques: they infer additional constraints on the problem that must be satisfied by any solution. In contrast, the algorithm presented in Section 2.3 uses graph matching as the central search mechanism, guiding both which decisions to make and when to make them (value and variable selection in the terminology of constraint programming). Our algorithm exploits the fact that structure embedding problems involve an `alldifferent` constraint on *all variables*, which makes matching more informative for structure embedding than it is for general CSPs.

Subgraph isomorphism Given two graphs G and H, the *subgraph isomorphism* problem is to determine if there exists a subgraph of H that is isomorphic to G, or equivalently, to determine whether there exists an injective homomorphism from G to H. Subgraph isomorphism has a number of applications, including subcircuit identification [18] and finding motifs in biochemical graph data [4] — see [6] for a broad survey of techniques for and applications of subgraph isomorphism. An accessible account describing the differences between some subgraph isomorphism algorithms and an experimental comparison between them can be found in [14].

The subgraph isomorphism problem is a special case of structure embedding, where the vocabulary of structures is fixed to the vocabulary of graphs consisting of a single binary incidence relation. A reduction from structure embedding to labelled subgraph isomorphism (wherein the signature consists only of monadic and binary predicates) is also possible through constraint binarization [23, Ch.

6]. However, the applications of subgraph isomorphism differ from the setting of this paper: typically, the source graph is small and the target is very large, and the problem of interest is to enumerate *all* injective homomorphisms. In this paper, the problem of interest is the decision problem to determine whether there is at least one injective homomorphism, and the expectation is that the source and target are of similar size.

A common theme in algorithms for subgraph isomorphism is to exploit local edge structure. In contrast, the algorithm we propose exploits the global structure of the problem by using graph matching as the foundation of the backtracking search. We are not aware of an existing algorithm for subgraph isomorphism that operates in polytime for labelled graphs without edges, which is the analogue of monadic structures. We see from Figure 4 that VF2 is not competitive on monadic structures, unable to exploit any local edge structure.

6 Conclusion

In this paper we presented MatchEmbeds, a practical algorithm for the problem of testing whether one finite relational structure embeds into another. The core idea is to use bipartite graph matching to drive a backtracking search procedure. The algorithm operates in polytime for monadic structures, but may take exponential time for general structures. The procedure has been shown to be effective for problems that arise in practice and for difficult random instances.

It would be interesting to apply matching-based search to other problems where injectivity constraints are important. For instance, in entailment checking for separation logic formulas, separately conjoined heap cells in a source formula must map to separately conjoined heap cells in a target formula. Entailment checking for the list fragment can be done in polytime using graph homomorphism [7], but entailment checking for formulas with existential quantifiers may benefit from matching-based search.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS. pp. 313–321 (1996)
2. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (Sep 1975)
3. Biere, A.: Lingeling, plingeling and treengeling entering the sat competition 2013. In: Balint, A., Belov, A., Heule, M.J., Järvisalo, M. (eds.) SAT Competition 2013. pp. 51–52. Department of Computer Science Series of Publications B (2013)
4. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC bioinformatics **14**(7), S13 (2013)
5. Cheeseman, P.C., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: IJCAI. vol. 91, pp. 331–340 (1991)
6. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. International Journal of Pattern Recognition and Artificial Intelligence **18**(03), 265–298 (2004)

7. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: CONCUR. pp. 235–249 (2011)
8. Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. In: POPL. pp. 407–420 (2015)
9. Farzan, A., Kincaid, Z., Podelski, A.: Proving liveness of parameterized programs. In: LICS. pp. 185–196 (2016)
10. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! TCS **256**(1), 63–92 (2001)
11. Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. Canadian Journal of Mathematics **8**(3), 399–404 (1956)
12. van Hoeve, W.J.: The alldifferent constraint: a survey. In: In 6th Annual Workshop of the ERCIM Working Group on Constraints (2001)
13. Hopcroft, J.E., Karp, R.M.: A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. In: SWAT. pp. 122–125 (1971)
14. Lee, J., Han, W.S., Kasperovics, R., Lee, J.H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. In: PVLDB. pp. 133–144 (2013)
15. Lopez-Ortiz, A., Quimper, C.G., Tromp, J., Van Beek, P.: A fast and simple algorithm for bounds consistency of the all different constraint. In: IJCAI. pp. 245–250 (2003)
16. McCreesh, C., Prosser, P.: A parallel, backjumpig subgraph isomorphism algorithm using supplemetal graphs. In: International conference on principles and practice of constraint programming. pp. 295–312. Springer (2015)
17. McCreesh, C., Prosser, P., Trimble, J.: Heuristics and really hard instances for subgraph isomorphism problems. In: IJCAI. pp. 631–638 (2016)
18. Ohlrich, M., Ebeling, C., Ginting, E., Sather, L.: SubGemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. In: DAC. pp. 31–37 (1993)
19. P. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (Oct 2004)
20. Perron, L.: Operations research and constraint programming at google. In: CP. pp. 2–2 (2011), `https://developers.google.com/optimization/`
21. Puget, J.F.: A fast algorithm for the bound consistency of alldiff constraints. In: AAAI. pp. 359–366 (1998)
22. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: AAAI. pp. 362–367 (1994)
23. Russell, S.J., Norvig, P.: Artificial intelligence - a modern approach, 3rd Edition. Prentice Hall series in artificial intelligence, Prentice Hall (2009)
24. Schulte, C., Lagerkvist, M., Tack, G.: GECODE - An open, free, efficient constraint solving toolkit. `www.gecode.org`
25. Sinz, C.: Towards an optimal cnf encoding of boolean cardinality constraints. In: van Beek, P. (ed.) Principles and Practice of Constraint Programming - CP 2005. pp. 827–831 (2005)
26. Soos, M., Nohl, K., Castelluccia, C.: Cryptominisat. SAT Race solver descriptions (2010)
27. Veksler, M., Strichman, O.: Learning general constraints in csp. In: CPAIOR. pp. 410–426 (2015), `http://strichman.net.technion.ac.il/haifacsp/`