Math. Struct. in Comp. Science (2002), *vol.* 12, *pp.* 403–422. © 2002 Cambridge University Press
DOI: 10.1017/S0960129501003577 Printed in the United Kingdom

Constraint satisfaction algorithms for graph pattern matching[†]

JAVIER LARROSA[‡] and GABRIEL VALIENTE[§]

Department of Software, Technical University of Catalonia, Barcelona, Spain

Received 18 July 2000; revised 15 December 2000

Graph pattern matching is a central problem in many application fields. Since it is NP-*complete*, we cannot expect to find algorithms with a good worst-case performance. However, there is still room for general procedures with a good average performance. In this paper we explore four different solving approaches within the constraint satisfaction framework, and introduce a new algorithm, which we call nRF+. The algorithm is a refinement of *really full look ahead* that takes advantage of the problem structure in order to enhance the look ahead procedure. We give a formal proof that nRF+ is superior to the other approaches in terms of number of visited nodes. An additional contribution of this paper is the introduction of a new benchmark for testing algorithms in this domain. It is formed by a large set of well-defined graphs of very diverse nature. In this benchmark, we show that nRF+ can efficiently solve a broad range of problems, while still leaving many problem instances unsolved. The use of this challenging benchmark is encouraged for future algorithms evaluation.

1. Introduction

Graph pattern matching is the problem of finding an isomorphic image of a given graph in another graph, and is also known as the *subgraph isomorphism problem*. It is a central problem in several fields, including graph transformation, since the application of a graph transformation rule to a graph succeeds by first finding a subgraph homomorphism or a subgraph isomorphism of the left-hand side of the rule into the graph to be transformed. Indeed, there is an increasing interest in the field of graph transformation in finding better subgraph isomorphism algorithms (Bunke *et al.* 1991; Rudolf 2000; Zündorf 1996).

The practical relevance of the graph pattern matching problem is reflected in a variety of applications, ranging from chemical documentation (Barnard 1993) and pattern recognition and computer vision (Wong 1992; Bunke 1993; Bunke and Messmer 1997) to computer-aided design (Ohlrich *et al.* 1993) and visual languages (Schlieder 1998), as well

[†] This paper is a complete and revised version of a lecture given at the workshop on Theory and Applications of Graph Transformations (GRATRA 2000), which was a satellite event of ETAPS 2000 in Berlin, March 2000.

[‡] Partially supported by the Spanish CICYT project TAP1999-1086-C03-03.

[§] Partially supported by the EC TMR Network GETGRATS and by the ESPRIT Basic Research Working Group APPLIGRAPH through the University of Bremen, and by the Spanish DGES project PB96-0191-C02-02 and CICYT project TIC98-0949-C02-01 HEMOSS.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

as graph transformation. It is also reflected in the diversity of approaches to the problem. Specific methods include backtracking (McGregor 1982; Cheng and Huang 1981), the clique approach (Levi 1972; Barrow and Burstall 1976; Ghahraman *et al.* 1980; Akinniyi *et al.* 1986), which consists of finding the maximum clique in a certain product of the two graphs, the relational approach (Haralick and Kartus 1978; Cortadella and Valiente 2000), which consists of building up a large relation containing all subgraph isomorphisms of one graph into the other, and constraint satisfaction (Ullmann 1976; McGregor 1979; Rudolf 2000). Optimization methods include continuous optimization (Kasif *et al.* 1983; Kuner and Ueberreiter 1988), discrete relaxation (Kitchen and Rosenfeld 1979; Kitchen 1980; Wilson *et al.* 1995), probabilistic relaxation (Christmas *et al.* 1995), and genetic search (Cross *et al.* 1997).

Since the subgraph isomorphism problem is NP-*complete* (Garey and Johnson 1979), all known algorithms have an exponential worst-case behaviour. Thus, the development of a general algorithm able to solve medium and large problem instances within a practical amount of time is beyond the reach of current technology. There is, however, still room for algorithms that exhibit good average performance and with which particular problem instances can be solved in a reasonable amount of time. Given the practical importance of the problem, the development of such algorithms is clearly of interest.

One way to assess algorithms is by means of a benchmark. If the benchmark is formed by a large heterogeneous set of problem instances and there is an algorithm that typically shows a good performance, one can expect that algorithm to be effective in a broader spectrum of problem instances.

Constraint reasoning is a flourishing field in Computer Science (Dechter 1992; Mackworth 1992; Wallace 1996) that provides a common framework for representing many combinatorial problems and a collection of techniques for solving them. In particular, graph pattern matching can be easily expressed as a constraint satisfaction problem.

In this paper, we present two contributions toward the development of practical and general algorithms for the subgraph isomorphism problem. On the one hand, we propose a new benchmark for testing subgraph isomorphism algorithms. This benchmark consists of a library of well-defined subgraph isomorphism problem instances. It is based on a library of standard graphs that are widely used for evaluating combinatorial computing methods and algorithms. On the other hand, we explore the applicability of constraint satisfaction techniques for subgraph isomorphism. We evaluate four previous approaches and present a new algorithm nRF+. Our algorithm is a refinement of *really full look ahead* that includes an enhanced look ahead procedure. nRF+ takes advantage of the problem structure, so it cannot be applied to general constraint satisfaction problems. We show that the new algorithm outperforms previous approaches in terms of number of visited nodes. Our experiments indicate that nRF+ can be effective in a broad range of problem instances.

The paper is structured as follows. In the next section we give some preliminaries on constraint satisfaction. In Section 3, we explore the applicability of constraint satisfaction algorithms to subgraph isomorphism and introduce the new algorithm. Section 4 describes a new benchmark for testing subgraph isomorphism algorithms and presents the results obtained with the algorithms discussed in Section 3. Finally, we give some conclusions in Section 5.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

2. Constraint satisfaction preliminaries

2.1. *Constraint satisfaction problems*

A *constraint satisfaction problem* (CSP) is defined as a tuple $(\mathscr{X}, \mathscr{D}^0, \mathscr{C})$ where:

— $\mathscr{X}$ is an ordered set of variables $(x_1, \ldots, x_n)$.
— $\mathscr{D}^0$ is a set of domains, $\{D_1^0, \ldots, D_n^0\}$, where D_i^0 is a finite set of possible values for variable x_i.
— $\mathscr{C}$ is a set of constraints $\{c_1, \ldots, c_m\}$. A constraint c_i with *scope* $var(c_i) = (x_{i_1}, \ldots, x_{i_{r(i)}})$ specifies the *allowed combinations* of values for the variables in $var(c_i)$.

A constraint c_i is a subset of $D_{i_1}^0 \times \cdots \times D_{i_{r(i)}}^0$ that can be stored explicitly as a *set of tuples*, or implicitly by means of a Boolean function. Without loss of generality, we assume that each constraint has a different scope.

CSPs deal with assignments of domain values to the problem variables. Assignments can also be represented as tuples. Let t be an assignment; its *scope*, denoted $var(t)$, is the ordered set of variables to which t assigns values. An assignment t is *total* if it associates a value to every variable in the problem (that is, $var(t) = \mathscr{X}$), otherwise it is *partial*. An assignment is *consistent* if it satisfies all constraints whose scope is included in the scope of the assignment. A *solution* to a CSP is a total consistent assignment (it assigns a value to every variable and satisfies every constraint).

Theorem 1. (Tsang 1993) Finding a solution in a CSP or proving that there is none is an NP-*complete* problem.

The *arity of a constraint* is the size of its scope. The *arity of a problem* is the arity of its largest constraint. A *binary* CSP has arity two. A *non-binary* CSP has arity greater than two.

2.2. *Local consistency*

Local consistency properties express necessary conditions for consistent partial assignments to be extensible to total ones. If a consistent partial assignment does not satisfy some local consistency property, we can add a new constraint forbidding it without changing the set of problem solutions (Freuder 1982; Mackworth 1977).

Especially useful local consistency properties are those that detect values that can never participate in a problem solution, because these unfeasible values can be safely removed from the domains without changing the set of solutions to the problem. If during this process some variable's domain becomes empty, it proves that the problem does not have any solution. Thus, algorithms that prune locally inconsistent values are incomplete methods for proving unsolvability.

When dealing with algorithms that prune locally inconsistent values it is convenient to distinguish between the initial and the current domains. D_i^0 denotes the initial domain of variable x_i (that is, the domain as defined in the problem formulation), while D_i denotes its current domain after, possibly, some value pruning. We say that a tuple t is *valid* if it assigns values from the current domain.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

A very useful local consistency property is *arc consistency*[†]. A value a of a variable x_i is arc-consistent with a constraint c if and only if $x_i \notin var(c)$ or there exists a valid tuple t in c with value a for x_i. We say that t *supports* the arc consistency of a in x_i. A constraint c is arc-consistent if and only if every value of each variable in its scope is arc-consistent with c. A set of constraints is arc-consistent if and only if each constraint in the set is arc-consistent. A problem is arc-consistent if and only if all its constraints are arc-consistent (Mackworth 1977; Mohr and Masini 1988). Any arc-inconsistent value can be pruned from its domain because there is at least one constraint that it cannot possibly satisfy.

Arc consistency algorithms transform an arc-inconsistent set of constraints into an arc-consistent one by removing arc-inconsistent values from domains until arc consistency is obtained. There are a number of algorithms for achieving arc consistency, ranging from AC1 to AC7. They were initially proposed for binary problems (Mackworth 1977; Mohr and Henderson 1986; Bessière 1994; Bessière *et al.* 1999) and later generalized to the non-binary case (Mohr and Masini 1988; Bessière and Régin 1997).

Theorem 2. (Mohr and Masini 1988; Bessière and Régin 1997) A constraint c_i with scope $var(c_i) = (x_{i_1}, \ldots, x_{i_{r(i)}})$ can be made arc-consistent in time $O(|D_{i_1}| \times \cdots \times |D_{i_{r(i)}}|)$. A set of m constraints can be made arc-consistent in time $O(md^r)$, where d is the size of the largest domain and r is the largest arity.

For particular types of constraints it is possible to design specialized arc consistency algorithms with better complexities. For instance, the n-ary *alldiff* constraint to which we will refer in the next section, can be made arc-consistent in time $O(n^2d^2)$ (n is the number of variables, d is the largest domain size) (Régin 1994). We use AC($\mathscr{C}, \mathscr{D}$) to denote a procedure that enforces arc consistency in the set $\mathscr{C}$ with the current domains $\mathscr{D}$. Procedure AC($\mathscr{C}, \mathscr{D}$) returns the updated domains after value pruning.

2.3. *Constraint satisfaction algorithms*

Most algorithms for solving constraint satisfaction problems start with an empty (trivially consistent) assignment and attempt to extend it by assigning one more variable at a time, while keeping the assignment consistent. When no extension is possible, we say that the algorithm has detected a *dead end*. Then, the algorithm *backtracks* and changes a previous assignment. The algorithm stops when a total consistent assignment has been computed, or when all possibilities have been attempted. Thus, the algorithm *searches* on a tree where intermediate nodes stand for partial assignments and leaves stand for total ones. During search, assigned variables are called *past* variables, unassigned variables are called *future* variables. The variable under consideration is called the *current* variable. At a given search state, we can partition the set of constraints into three subsets. We use $\mathscr{C}_p$ to denote the set of constraints whose scope is included in the set of past variables, $\mathscr{C}_f$

[†] The name comes from the graphical representation of a CSP instance as a graph, in which variables are depicted as nodes and constraints are depicted as arcs. Arc consistency explores local consistency up to the *arc* level.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

Algorithm 1: Generic look-ahead algorithm. P is the set of past variables, F is the set of future variables, $\mathscr{D}$ is the set of current domains and $\mathscr{C}$ is the set of constraints. For clarity, domain restoration after backtracking is omitted.

```
    Solver(P, F, 𝒟, 𝒞);
    if F = ∅ then
1.1 |  print 𝒟 and stop;
    else
1.2 |  𝒟 ← LookAhead(𝒞, 𝒟);
    |  if NoEmptyDomain(𝒟) then
1.3 |  |  i ← PopAVariable(F);
    |  |  Values ← D_i;
    |  |  while (Values ≠ ∅) do
1.4 |  |  |  a ← PopAValue(Values);
    |  |  |  D_i ← {a};
    |  |  |  Solver(P ∪ {i}, F − {i}, 𝒟, 𝒞);
```

to denote the set of constraints whose scope is included in the set of future variables and $\mathscr{C}_{pf}$ to denote the set of constraints whose scope has both past and future variables.

A *look-ahead* algorithm is usually the best choice for solving non-trivial problems (Kondrak and Beek 1997). These algorithms perform a look-ahead procedure at each search state that removes values from future variable domains that cannot participate in a solution extending the current assignment. Surviving values are called *feasible*. Pruned values are called *unfeasible*. When the look-ahead procedure causes an empty domain in a future variable, it is not possible to extend the current assignment to this variable. Therefore, a dead end has been detected and the algorithm has to backtrack.

Some look-ahead algorithms can be described in terms of search plus arc consistency enforcement. The basic idea is that when instantiating a variable x_i with a value a, the domain of the current variable is set momentarily to a single value (that is, $D_i \leftarrow \{a\}$). Then, it is possible to enforce arc consistency in some or all problem constraints. In this paper we will restrict ourselves to two algorithms that are widely used because of their good average-case performance: *forward checking* (FC) (Haralick and Elliot 1980; Bessière *et al.* 1999) and *really full look-ahead* (RF) (Nudel 1988). At each search state, FC enforces arc consistency in $\mathscr{C}_{pf}$, while RF enforces arc consistency in $\mathscr{C}_{pf} \cup \mathscr{C}_f$.

Algorithm 1 is a recursive description of a generic look-ahead algorithm. Each recursive call corresponds to a search node defined by past and future variables, current domains and the problem constraints. The initial call is $\texttt{Solver}(\varnothing, \mathscr{X}, \mathscr{D}^0, \mathscr{C})$ (all variables are future variables and current domains are the initial domains). In an arbitrary call, current domains of past variables are singletons defining the current (consistent) assignment. If there are no future variables, the current assignment is a solution, so the algorithm reports it and stops (line 1.1). Otherwise, the look-ahead procedure is performed (line 1.2). It prunes future values that are detected unfeasible with the current assignment. If the look-ahead causes an empty domain, the algorithm backtracks. Otherwise, a future variable is selected (line 1.3) and its feasible values are sequentially attempted (line 1.4).

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

If the generic look ahead of line 1.2 is replaced by $\mathrm{AC}(\mathscr{C}_{pf}, \mathscr{D})$, the resulting algorithm is *forward checking*. If it is replaced by $\mathrm{AC}(\mathscr{C}_{pf} \cup C_f, \mathscr{D})$, the resulting algorithm is *really full look-ahead*.

The theoretical comparison of different CSP algorithms is a difficult task. One approach is to consider the number of nodes that algorithms visit when solving a problem, because sometimes it is possible to show that one algorithm never visits more nodes than another (Kondrak and Beek 1997). Although useful, the number of nodes does not give the complete picture of the algorithm's efficiency, because different algorithms perform different amounts of work at each node. It may happen that one algorithm visits fewer nodes than another, but performs more computation per node and the extra work does not pay off in terms of CPU time. Normally, theoretical comparison in terms of number of nodes visited is complemented by an empirical comparison in some benchmark.

Since the set of constraints where FC enforces arc consistency is a subset of the set of constraints where RF enforces arc consistency, the look-ahead of RF is stronger than the look ahead of FC (that is, RF detects and prunes more unfeasible values than RF). Therefore, if both algorithms solve the same problem and use the same criterion for variable and value selection (lines 1.3 and 1.4 in Algorithm 1), RF never visits more nodes than FC.

Looking at the overhead per node, we have the following complexities (m denotes the total number of constraints in the problem, d denotes the largest domain size and r denotes the problem arity):

— The look-ahead of FC (that is, $\mathrm{AC}(\mathscr{C}_{pf}, \mathscr{D})$) has complexity $O(md^{r-1})$, because it enforces arc-consistency in at most m constraints each one having at most $r-1$ uninstantiated variables (by definition of $\mathscr{C}_{pf}$, constraints in the set have at least one of the variables in their scope assigned). Instantiated variables do not *add* any cost in the arc-consistency process, since their domain remains *fixed*.
— The look-ahead of RF (that is, $\mathrm{AC}(\mathscr{C}_{pf} \cup C_f, \mathscr{D})$) has complexity $O(md^r)$, because it enforces arc-consistency in at most m constraints each one having at most r uninstantiated variables.

Therefore, RF visits fewer nodes at the cost of a more expensive look ahead. Empirical results show that the extra work of RF might be cost effective when solving sufficiently hard problems (Bessière and Régin 1996).

3. Solving graph pattern matching as a constraint satisfaction problem

3.1. *Problem formulation*

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two undirected[†] graphs with $|V_1| = n$, $|V_2| = m$ and $n \leqslant m$. We say that G_1 is *isomorphic to a subgraph* of G_2 if and only if there is an injection $I : V_1 \to V_2$ such that $(i, j) \in E_1 \Rightarrow (I(i), I(j)) \in E_2$. The *subgraph isomorphism*

[†] For the sake of notational simplicity, we assume undirected graphs. However, all definitions, properties and algorithms discussed in this paper can be easily extended to directed graphs.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

problem is the problem of proving whether G_1 is isomorphic to a subgraph of G_2 or not. It is an NP-*complete* problem (Garey and Johnson 1979).

In the following we use $i, j, \ldots$ to denote nodes of G_1, and $a, b, \ldots$ (and sometimes v) to denote nodes of G_2. We use the boolean predicate $edge(G, i, j)$ that is true if and only if there is an edge from i to j in graph G.

There are several ways to formulate the subgraph isomorphism problem as a constraint satisfaction problem. The most natural one is to associate a problem variable with each vertex of G_1 and let variables take values on the vertices of G_2. Constraints impose the requirement that edges in G_1 must be preseved in the isomorphic subgraph of G_2 (Ullmann 1976; McGregor 1979). Formally, the subgraph isomorphism problem is equivalent to a *binary* constraint satisfaction problem $(\mathscr{X}, \mathscr{D}^0, \mathscr{C})$ defined as:

— *Variables*: $\mathscr{X} = V_1$.
— *Domains*: Let x_i be the variable associated to a vertex $i \in V_1$. If $edge(G_1, i, i)$, then $D_i^0 = \{a \in V_2 \mid edge(G_2, a, a)\}$, else $D_i^0 = V_2$.
— *Constraints*: For each $i, j \in G_1$ $(i < j)$ there is a constraint $c_{(i,j)}$ with scope (x_i, x_j) defined as

$$c_{(i,j)} = \{(a, b) \in (D_i^0 \times D_j^0) \mid (a \neq b) \wedge (edge(G_1, i, j) \Rightarrow edge(G_2, a, b))\}.$$

Observe that domains are defined in such a way that vertices of G_1 that have a loop-edge can only be associated to vertices of G_2 that also have a loop-edge, which is a constraint imposed by the problem.

An alternative to this formulation was proposed in Régin (1994). The condition of different variables taking different values is removed from the binary constraints and replaced by a single n-ary constraint, denoted *alldiff*, with scope $(x_1, \ldots, x_n)$. The new constraint allows only those n-ary tuples that have all their values different to each other. That is

$$alldiff = \{(v_1, v_2 \ldots, v_n) \in (D_1^0 \times D_2^0 \times \cdots \times D_n^0) \mid v_j \neq v_k,\ 1 \leqslant j, k \leqslant n,\ j \neq k\}.$$

Clearly, both formulations of this condition are semantically equivalent. However, arc consistency in the non-binary formulation is a stronger property than arc consistency in the binary formulation.

Theorem 3. (Régin 1994) Let $\mathscr{V}$ be a set of variables of a CSP instance with domains $\mathscr{D}$. Let $c_{(i,j)}$ be a constraint between x_i and x_j that requires that the values assigned to these two variables must be different. Let $\mathscr{C} = \{c_{(i,j)} \mid x_i, x_j \in \mathscr{V},\ i \neq j\}$ be a set of binary constraints requiring that all variables in $\mathscr{V}$ must take pairwise different values. Let $alldiff_{\mathscr{C}}$ be the (semantically equivalent) *alldiff* constraint among variables in $\mathscr{V}$. If $alldiff_{\mathscr{C}}$ is arc-consistent, $\mathscr{C}$ is arc-consistent.

Therefore, algorithms that enforce arc consistency during search (such as FC and RF) prune more values when they use the non-binary formulation. Consequently, if the same algorithm is executed with both formulations, the execution with the non-binary formulation will never visit more nodes than the execution with the binary formulation. However, using a standard arc-consistency enforcement algorithm for the *alldiff* constraint has complexity $O(d^n)$ (since the cost of enforcing arc-consistency in one constraint of arity n

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

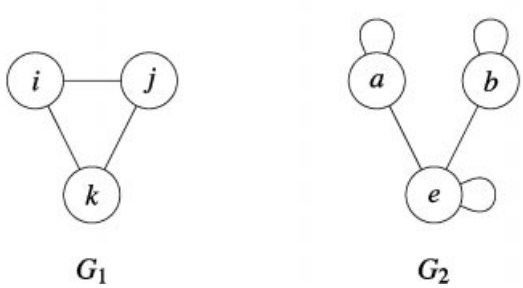


Fig. 1. Two graphs G_1 and G_2 defining a subgraph isomorphism problem.

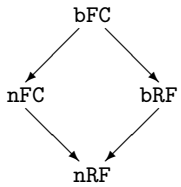


Fig. 2. Hierarchy of four constraint satisfaction algorithms for subgraph isomorphism with respect to the number of visited nodes. $A \rightarrow B$ means that algorithms B never visits more nodes than algorithm A.

is exponential in n, the number of variables in the problem). Clearly, this complexity makes this approach unfeasible. However, Régin (1994) presents a specialized efficient arc-consistency algorithm for the *alldiff* constraint.

Theorem 4. (Régin 1994) AC(*alldiff*) can be achieved in time $O(n^2d^2)$.

This result clearly makes the non-binary formulation of the subgraph isomorphism problem appealing.

Example 1. Consider the subgraph isomorphism problem associated with the graphs of Figure 1. Its *binary* formulation as a CSP renders the following problem:

— $\mathcal{X} = (x_1, x_2, x_3)$.
— $D_1^0 = D_2^0 = D_3^0 = \{a, b, e\}$.
— $c_{(1,2)} = c_{(1,3)} = c_{(2,3)} = \{(a, e), (b, e), (e, a), (e, b)\}$.

The *non-binary* formulation has the same variables and domains. Its binary constraints are

$$c_{(1,2)} = c_{(1,3)} = c_{(2,3)} = \{(a, a), (a, e), (b, b), (b, e), (e, e), (e, a), (e, b)\}$$

and the n-ary constraint allows all ternary tuples having all their values pairwise different:

$$alldiff = \{(a, b, e), (a, e, b), \ldots, (e, b, a)\}.$$

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

3.2. *Constraint satisfaction algorithms for graph pattern matching*

In this paper we consider the binary and non-binary formulations of subgraph isomorphism presented in the previous subsection, and the two algorithms, FC and RF, described in Subsection 2.3. Combining them, there are four different solving strategies. We will denote each formulation-algorithm combination by a letter *b* or *n* indicating binary or non-binary formulation, plus the name of the algorithm. Therefore, the alternatives are: bFC, bRF, nFC and nRF. Except for unimportant details, bFC is the algorithm that was proposed in McGregor (1979) and bRF is the algorithm proposed in Ullmann (1976).

We mentioned in Subsection 2.3 that RF performs more pruning than FC, and in Subsection 3.1 that the non-binary formulation allows for more pruning than the binary formulation. Since value pruning means search space reduction, we can establish a partial order among the four solving approaches in terms of number of visited nodes. nRF is the algorithm that visits the fewest nodes, because it combines the problem formulation with the algorithm that allows for more pruning. Algorithm bFC is the worst in terms of visited nodes. Algorithms bRF and nFC lie between bFC and nRF and are not comparable. Figure 2 illustrates this partial order.

As for the overhead per node, we have the following complexities, where n is the number of variables (vertices of G_1) and d is the maximum number of values in a domain (vertices of G_2):

— The look ahead of bFC has time complexity $O(nd)$, because arc-consistency has to be enforced in a set of at most n binary constraints (the number of constraints the current variable is constrained with), and these constraints have only one of the variables of their scope uninstantiated.
— The look ahead of bRF has time complexity $O(n^2d^2)$ because arc-consistency has to be enforced in constraints between the current variable and future variables, as well as in constraints between future variables (there is a quadratic number of these constraints).
— The look ahead of nFC has time complexity $O(n^2d^2)$ provided that the specialized algorithm of Régin (1994) is used for the *alldiff* constraint.
— The look ahead of nRF has time complexity $O(n^2d^2)$ provided that an optimum arc consistency mechanism is used for the binary constraints and the specialized algorithm of Régin (1994) is used for the *alldiff* constraint

3.3. *A specialized algorithm*

As we will show in Section 4.2, nRF provides a good trade-off between pruning capability and complexity per node, and typically performs better than the other algorithms in subgraph isomorphism. However, we can still increase its look ahead pruning capability. The refinement that we propose renders a new algorithm, nRF+. It takes advantage of the problem structure, so the new algorithm is not a general CSP solving procedure. It can only be applied to the non-binary formulation of the subgraph isomorphism problem.

Consider the non-binary CSP formulation of a subgraph isomorphism problem being solved by nRF and an arbitrary search state. The look ahead prunes every future value that is arc-inconsistent in the current search state. Therefore, surviving values are

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

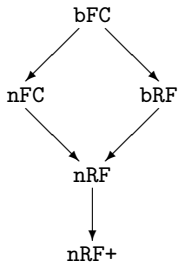


Fig. 3. Hierarchy of five algorithms for subgraph isomorphism with respect to the number of visited nodes. $A \rightarrow B$ means that algorithms B never visits more nodes than algorithm A.

those that have a supporting tuple in each constraint involving their variable. We use $feas_{\mathrm{nRF}}(x_i, a)$ to denote the condition for value a of variable x_i being feasible (that is, $feas_{\mathrm{nRF}}(x_i, a) \iff a \in D_i$). The set of past variables is P, the set of future variables is F and current domains are $\mathscr{D}$. It can be seen that $feas_{\mathrm{nRF}}(x_i, a)$ is

$$
\begin{aligned}
&[\exists (v_1, v_2, \ldots, v_n) \in (D_1 \times D_2 \times \cdots \times D_n) \\
&\quad \text{such that } v_i = a, v_j \neq v_k (1 \leqslant j, k \leqslant n, j \neq k)] \\
&\wedge [\forall x_j \in (P \cup F - \{x_i\}), \exists v_j \in D_j \text{ such that } edge(G_1, i, j) \Rightarrow edge(G_2, a, v_j)]
\end{aligned}
$$

The first condition expresses the arc consistency of (i, a) with respect to the *alldiff* constraint and the second condition expresses the arc consistency with respect to each binary constraint that affects variable x_i. In words, value a of future variable x_i is feasible with nRF if and only if:

(*i*) there is a valid *alldiff* tuple with value a, and

(*ii*) for each node of graph G_1 connected to i there is some node of graph G_2 connected to a in its current domain.

The new algorithm that we propose, nRF+, adds an additional condition to the previous expression. Let x_j be a variable such that i and j are connected vertices in G_1. The set $D_j \cap neigh(G_2, a)$ is the set of values in D_j that are consistent with value a in D_i, which must be non-empty if the current subproblem is arc-consistent. The set,

$$\cup_{j \in neigh(G_1, i)} (D_j \cap neigh(G_2, a))$$

indicates the set of candidate values that we have for the assignment of variables in $neigh(G_1, i)$ if we assign value a to x_i. Clearly, there must be more candidate values than variables, because each variable must get a different value. If this is not the case, value a is unfeasible and can be pruned. Formally, we define $\mathscr{T}(i, a)$ as

$$\mathscr{T}(i, a) = \begin{cases} \varnothing, & \text{if } \exists j \in neigh(G_1, i) \text{ such that } D_j \cap neigh(G_2, a) = \varnothing \\ \cup_{j \in neigh(G_1, i)} (D_j \cap neigh(G_2, a)), & \text{otherwise.} \end{cases}$$

If $|\mathscr{T}(i, a)| < |neigh(G_1, i)|$, value a cannot be in a solution and, therefore, it can be pruned.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

The new algorithm, nRF+, prunes values that do not satisfy $feas_{\mathrm{nRF+}}(x_i, a)$ defined as

$$[\exists(v_1, v_2, \ldots, v_n) \in (D_1 \times D_2 \times \cdots \times D_n)$$
$$\text{such that } v_i = a, v_j \neq v_k (1 \leqslant j, k \leqslant n, j \neq k)]$$
$$\wedge |\mathscr{T}(i, a)| \geqslant |neigh(G_1, i)|.$$

Example. Consider the non-binary CSP formulation of the subgraph isomorphism of Figure 1 and the initial search state (all variables are future variables, $F = (x_1, x_2, x_3)$, and current domains are the initial domains, $\mathscr{D} = \mathscr{D}^0$).

The look ahead of nRF enforces arc consistency in the problem. It is easy to check that this problem is arc-consistent, so the look ahead of nRF does not prune any value. The look ahead of nRF+ prunes value a of D_1 because $\mathscr{T}(i, a) = \{e\}$ has only one element, while $neigh(G_1, a) = \{x_2, x_3\}$ has two elements (the assignment of a to x_i would only leave one value to consistently assign two variables, which is impossible). For the same reason, value a can be pruned from D_2 and D_3. After that, values b and e of D_1 do not have any support with respect to the *alldiff* constraint (there is no tuple of three different values with only two values to choose from) and nRF+ prunes them, producing an empty domain. Thus, the initial look ahead of nRF+ proves that the problem has no solutions.

Property. nRF+ never visits more nodes than nRF provided that both algorithms select variables and values in the same order.

Proof. In the following, we prove that condition $feas_{\mathrm{nRF+}}(x_i, a)$ is stronger than condition $feas_{\mathrm{nRF}}(x_i, a)$. Each of the conditions is a conjunction of two predicates, the first one being equal in both cases. The second predicate in $feas_{\mathrm{nRF}}(x_i, a)$ is

$$\forall x_j \in (P \cup F - \{x_i\}), \exists v_j \in D_j \text{ such that } edge(G_1, i, j) \Rightarrow edge(G_2, a, v_j).$$

If this condition is not satisfied, $|neigh(G_1, i)| \geqslant 1$ and, by definition, $|\mathscr{T}(i, a)| = 0$. Therefore, the second predicate in $feas_{\mathrm{nRF+}}(x_i, a)$,

$$|\mathscr{T}(i, a)| \geqslant |neigh(G_1, i)|,$$

cannot be satisfied.

Since condition $feas_{\mathrm{nRF+}}(x_i, a)$ is stronger than condition $feas_{\mathrm{nRF}}(x_i, a)$, it is clear that nRF+ prunes all the values that nRF prunes, and possibly more. Consequently, nRF+ makes a larger reduction in the search space than nRF. Therefore, nRF+ never visits more nodes than nRF. □

Figure 3 extends Figure 2 by adding nRF+ to the partial order of subgraph isomorphism algorithms with respect to the number of visited nodes according to the previous property. Algorithm 2 implements `nRF+LookAhead`, the look ahead of nRF+. Thus, the algorithmic description of nRF+ is obtained by replacing the generic look ahead of line 1.2 in Algorithm 1 by `nRF+LookAhead`$(\mathscr{X}, \mathscr{D}, \mathscr{C})$. The procedure follows a typical AC schema. Each time a value is found unfeasible it is pruned (see Algorithm 3) and added to a *Stream* of pruned values. Values are sequentially fetched from the stream and the fact that they have been pruned is propagated in the remaining problem (the algorithm checks if there are previously feasible values that have become unfeasible due to this removal).

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

The algorithm stops when *Stream* becomes empty or when value removal causes an empty domain.

Procedure `nRF+LookAhead` uses the following data structures:

— $S(j,a)$ is an integer counting how many feasible values of x_j are neighbours of a ($S(j,a) = |D_j \cap neigh(a, G_2)|$).
If $S(j,a) = 0$, we have a is an arc inconsistent in every x_i such that $edge(G_1, i, j)$ and $a \in D_i$. Therefore, $S(j,a) = 0$ allows for pruning value a of every x_i connected to x_j.

— $R(i,b)$ is an integer counting how many neighbours of x_i have b in their current domain ($R(i,b) = |\{j \in neigh(G_1, i)$ such that $b \in D_j\}|$).

— $CT(i,a)$ is an integer counting how many neighbours of a are in the domain of some neighbour of i ($CT(i,a) = |\{b \in neigh(G_2, a)$ such that $R(i,b) > 0\}|$).
Note that $CT(i,a)$ is equivalent to $|\mathscr{T}(i,a)|$ when value a is arc consistent. Therefore, if $CT(i,a) < |neigh(G_1, i)|$, we can prune a from D_i.

Procedure `nRF+LookAhead` deals with the arc consistency of the *alldiff* constraint using the algorithm of Régin (1994). It uses the following functionalities:

— `ACalldiff` enforces arc consistency in the *alldiff* constraint. It updates domains and adds pruned values to the *Stream*.

— `Propalldiff(`x_i, a`)` propagates the deletion of value a from D_i in the *alldiff* constraint. It updates domains and adds pruned values to the *Stream*.

The procedure starts enforcing arc consistency in the *alldiff* constraint (line 2.1). Then the data structures $S(j,a)$, $R(i,b)$ and $CT(i,a)$ are initialized (line 2.2). Next, unfeasible values are detected, pruned and added to *Stream* (line 2.3). After that, the procedure iterates, each time obtaining one element from the *Stream* and propagating its deletion. The deletion of value b from variable x_j is propagated in two different ways:

1 The first loop (line 2.4) prunes arc inconsistent values. $S(j,a)$ is updated for all a connected to b. If the counter becomes zero, variable x_j does not support a anymore in those variables connected to x_j. Thus, in these variables, value a is arc inconsistent and must be pruned.
2 The second loop (line 2.5) prunes values such that $|\mathscr{T}(i,a)| \geqslant |neigh(G_1, i)|$. Data structures $R(i,b)$ and $CT(i,a)$ are appropriately updated and unfeasible values are pruned.

The following property shows that the look ahead of nRF+ is $O(n^2d^2)$, the same complexity as the look ahead of nRF.

Property. Procedure `nRF+LookAhead` has complexity $O(n^2d^2)$, where n is the number of variables and d is the size of the largest domain.

Proof. Line 2.1 has complexity $O(n^2d^2)$ (Régin 1994). Data structure $S(j,a)$ is initialized in time $O(nd^2)$ (there are nd entries each one requiring d constant time queries). Data structure $R(i,b)$ is initialized in time $O(n^2d)$ (there are nd entries each one requiring n constant time queries). Data structure $CT(i,a)$ is initialized in time $O(nd^2)$ (there are nd entries each one requiring d queries to $R(i,b)$). Therefore, line 2.2 has complexity $O(nd^2 + n^2d)$. The loop of line 2.3 iterates $n \times d$ times, the condition can be evaluated in

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

Algorithm 2: Look ahead of nRF+. $\mathscr{X}$ is the set of variables, $\mathscr{D}$ are the current domains and $\mathscr{C}$ are the constraints. If during its execution the domain of a variable becomes empty, the algorithm stops (this fact is omitted in the algorithm for clarity).

```
   nRF+LookAhead(𝒳, 𝒟, 𝒞)
2.1 ACalldiff(𝒳, 𝒟, 𝒞) ;
2.2 IniStructures ;
2.3 foreach x_i ∈ 𝒳 do  foreach a ∈ D_i do
      if CT(i,a) < |neigh(G_1,i)| ∨ (∃j ∈ neigh(G_1,i) such that S(j,a) = 0) then Prune(x_i, a)
    while Stream ≠ ∅ do
      (x_j, b) ← PopStream(Stream) ;
2.4   foreach a ∈ neigh(G_2, b) do
        Decrement(S(j,a)) ;
        if S(j,a) = 0 then
          foreach i ∈ neigh(G_1, j) with a ∈ D_i do Prune(x_i, a)
2.5   foreach i ∈ neigh(G_1, j) do
        Decrement(R(i,b)) ;
        if R(i,b) = 0 then
          foreach a ∈ neigh(G_2, b) do
            Decrement(CT(i,a)) ;
            if CT(i,a) < |neigh(i, G_1)| then Prune(x_i, a)
```

Algorithm 3: Prunes a value from the current domains, propagates its deletion in the *alldiff* constraint and puts it in the *Stream* for future propagation.

```
Prune(x_i, a)
D_i ← D_i − {a} ;
Propalldiff(x_i, a) ;
PushStream(Stream, (x_i, a)) ;
```

constant time if null $S(j,a)$ are recorded during the initialization of line 2.2, and the cost of pruning a value is $O(nd)$ (due to the call to `Propalldiff` (Régin 1994)). Therefore, the total cost of the loop is $O(n^2d^2)$. The *while* loop iterates at most $n \times d$ times. Each of the inner *foreach* loops iterates $n \times d$ times. Therefore, if we disregard the cost of the pruning, the cost of the *while* loop is $O(n^2d^2)$. During the execution of the loop, there are at most $n \times d$ calls to `Prune`, because each value can only be pruned once. Pruning a value has complexity $O(nd)$, so the overall time spent pruning values is bounded by $O(n^2d^2)$. Therefore, the *while* loop requires time $O(n^2d^2)$. Summing up the complexities of each part, the final complexity is $O(n^2d^2)$. □

4. Empirical results

4.1. *A benchmark for graph pattern matching*

The experimental comparison of algorithms for subgraph isomorphism, as it is found for instance in Akinniyi *et al.* (1986), Cheng and Huang (1981), and McGregor (1979), is

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

often based on the results reported in Ullmann (1976). The experiments described therein consisted of finding one and finding all subgraph isomorphisms of a random undirected graph of order n into a random undirected graph of order d, for selected values n and d such that $n < d$ and for a fixed edge probability value. For each generated pair (G_1, G_2), the adjacency matrix of graph G_1 are added (*or*'ed) to the adjacency matrix of graph G_2 in order to guarantee the existence of at least one solution.

Some highly regular graphs have been also considered in the experiments described in Ullmann (1976), and another series of subgraph isomorphism experiments on random directed graphs has been described in Berztiss (1973) in which the in-degree and out-degree of every vertex is exactly half the number of vertices of the graph, but that choice of graphs seems to be tailored to the particular algorithm described therein. More recent experiments, reported in Cortadella and Valiente (2000) and Valiente and Martínez (1997), have shown that random graphs generated using the multiplicative linear congruential random number generator algorithm (Park and Miller 1988), do not show the expected number of edges reported in Ullmann (1976), meaning that the experiments described therein cannot be repeated unless further details of that implementation become available.

As a matter of fact, no benchmark has been developed yet for the subgraph isomorphism problem. In order to facilitate experimental comparison of different algorithms for subgraph isomorphism, we propose a benchmark that is based on the Stanford GraphBase[†].

The Stanford GraphBase (Knuth 1993) provides a large library of well-defined graphs. It presents the advantages of being stable, machine-independent and freely available. A machine-independent random number generator is included, thus providing a way to construct random graphs that can be reconstructed elsewhere.

The library of graphs comprises both irregular graphs derived from cultural artifacts and highly regular graphs. Among the irregular graphs are graphs derived from encounters between the characters that appear in famous novels, graphs derived from a thesaurus, from a digitized image, from the economy of the United States, from college football scores, from highway distances between North American cities, directed acyclic graphs that define logical circuits based on fundamental Boolean operations, and example graphs that appear in standard textbooks, such as graphs based on generalized chess pieces on d-dimensional chess boards, on generalized triangular configurations, on the subsets of a multi set, on the partitions of an integer, and on binary trees. Among the regular graphs are random graphs and expander graphs, the latter having relatively large girth and small diameter. Furthermore, it also provides standard operations on graphs, such as copying, complementing, and forming unions, intersections, line graphs, and products, as well as inducing subgraphs.

The benchmark proposed in this paper consists of all pairs of graphs (G_1, G_2) generated using the example graph parameters from Knuth (1993, Appendix C), with G_1 being not larger (in number of vertices) than G_2. The undirected graphs range from 10 to

[†] The Stanford GraphBase is available from `ftp://labrea.stanford.edu/pub/sgb/`. It can be used on any computer that supports CWEB (Levy and Knuth 1994), which can be obtained from `ftp://labrea.stanford.edu/pub/cweb/`.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

	undirected		directed	
	yes	no	yes	no
simple	102	11	54	5
acyclic	1	112	34	25
connected	71	42	47	12
biconnected	53	60	39	20
triconnected	43	70	26	33
bipartite	32	81	14	45
planar	19	94	7	52

Fig. 4. Number of graphs in the proposed benchmark which are simple, acyclic, connected, biconnected, triconnected, bipartite, and planar.

16796 vertices and from 0 to 328960 edges, while the directed graphs range from 10 to 16796 vertices and from 0 to 657920 arcs (that is, directed edges). Since there are 113 undirected graphs, and 59 directed graphs, there are, respectively, $(113 \cdot 114)/2 = 6441$ and $(59 \cdot 60)/2 = 1770$ such problem instances available[†]. Figure 4 illustrates the different nature of this set of graphs by indicating the total number that satisfy one of the following properties: simple, acyclic, connected, biconnected, triconnected, bipartite and planar.

Due to the high variability of structure in the considered graphs, many problem instances are likely not to have a solution (that is, graph G_1 is *not* isomorphic to a subgraph of G_2). It is our opinion that this is a nice feature of the proposed benchmark, because proving unsolvability is a fundamental task of the subgraph isomorphism problem. For instance, in the field of graph transformation one has to find a transformation rule such that the left-hand side of the rule is an isomorphism of the graph to be transformed. Presumably, different rules will unsuccessfully be attempted before finding a matching rule. In addition, when solving a subgraph isomorphism problem with a backtracking-based algorithm, each time that a wrong assignment is made (no solution exists extending the current assignment) the algorithm is condemned to unsuccessfully traverse the corresponding subtree until backtracking to the point where the mistake was made and then attempt a new value. This is precisely proving unsolvability in the subproblem associated with the point where the wrong assignment is made. As a matter of fact, when solving non-trivial problems with backtracking-based algorithms, *most of the computational effort is spent proving unsolvability after making wrong assignments*, in order to recover from the error.

The experiments reported in this paper are restricted to a subset of 1631 instances, namely all pairs (G_i, G_j) with $1 \leqslant i < j \leqslant 50$ for undirected graphs (1225 problem instances) and with $1 \leqslant i < j \leqslant 29$ for directed graphs (406 problem instances). The undirected graphs range from 10 to 128 vertices and from 0 to 9900 edges, while the directed graphs range from 10 to 151 vertices and from 10 to 4902 arcs. Figure 5 gives some additional information about the graphs used in our experiments. Plots on the left refer to undirected graphs. Plots on the right refer to directed graphs. The row on

[†] The benchmark is available from `http://www.lsi.upc.es/~valiente/` for use with the Stanford GraphBase or with LEDA (Mehlhorn and Näher 1999).

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

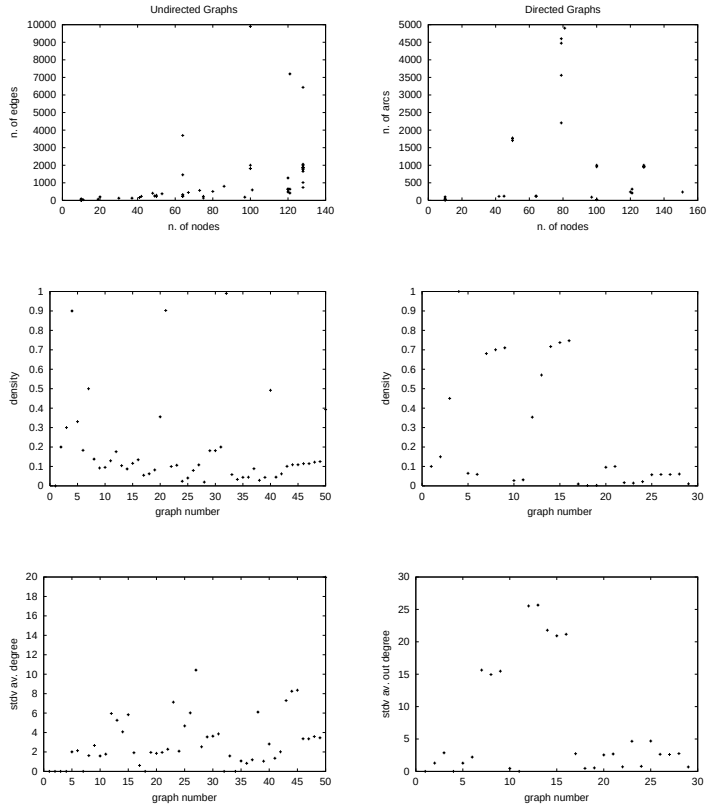


Fig. 5. Graphical description of the benchmark. Left column refers to the undirected graphs. Right column refers to the directed graphs. Top row depicts number of vertices *vs* number of edges at the left, and number of vertices *vs* number of arcs at the right. The row in the middle depicts graph number *vs* density. Bottom row depicts graph number *vs* standard deviation of the vertices degree.

the top depicts number of vertices *vs* number of edges (arcs in the directed case) and shows the large variability with respect to these two parameters. The row in the middle depicts for each graph its density defined as the ratio between the number of edges (arcs) and the maximum number of edges (arcs). It shows that our benchmark includes all sorts of graphs ranging from very sparse to very dense. Finally, the row on the bottom considers for each graph the degree of each vertex and depicts the standard deviation. A small standard deviation indicates a highly regular graph such that all its vertices have a similar number of neighbours. A large standard deviation indicates that some vertices are highly connected while others are sparsely connected. The plot shows that our benchmark includes graphs of all kinds.

4.2. *Experimental results*

The set of 1631 proposed problem instances has been attacked using the five algorithms: bFC, bRF, nFC, nRF and nRF+. All algorithms have been implemented in C, sharing

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

Table 1. *Number of problem instances in which unsolvability has been proved, in which a solution has been found, and that could not be solved within a fixed time limit.*

Undirected Graphs

	bFC	bRF	nFC	nRF	nRF+
Proved unsolvable	524	447	675	701	731
Solution found	306	305	317	328	319
Not solved	395	473	233	196	175

Directed Graphs

	bFC	bRF	nFC	nRF	nRF+
Proved unsolvable	260	256	267	281	271
Solution found	86	86	93	81	93
Not solved	60	64	46	44	42

code and data structures. The experiments have been performed on a Sun SparcStation 4. In preliminary executions, we observed that most problems were solved either in a few minutes, or they could not be solved in hours or days of search. For this reason, in our experiments we have set a limit of 5 minutes to the time invested to solve each problem instance (it bounds the total time required to run the benchmark with any algorithm to $1631 \times 5 = 8155$ minutes, that is, 5.6 days).

Table 1 reports the results of our experiments in terms of solved problems for both undirected and directed graphs (we consider that a problem is solved if the algorithm either finds a solution or proves that there is none). For each type of graph and each of the five algorithms, the table reports how many problem instances were proved unsolvable, how many were proved solvable (by finding a solution) and how many could not be solved because the time limit was reached. The main conclusion that can be extracted is that the non-binary representation seems to be more appropriate for this problem. Comparing algorithms under the binary formulation, the two algorithms seem very close with respect to solvable instances. However, nFC seems to outperform nRF in proving unsolvability. Comparing algorithms under the non-binary formulation, it seems that enforcing arc consistency in every constraint (nRF and nRF+) is cost effective. In every case nRF+ was the algorithm that solved most instances, closely followed by nRF.

Clearly, these results are context dependent because they depend on machine and implementation details. Further work needs to be done in order to establish an environment under which experiments can be replicated and algorithms compared in a context independent manner.

5. Conclusions

Graph pattern matching is a central problem in many application fields. Since it is an NP-*complete* problem, we cannot expect algorithms with good worst-case performance. However, there is still room for algorithms with good average-case performance. Con-

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

straint reasoning offers a general framework for representing combinatorial problems and developing solving techniques. Graph pattern matching can easily be formulated as a constraint satisfaction problem.

In this paper, we have explored the applicability of constraint satisfaction techniques in graph pattern matching. We have introduced a new algorithm that improves general constraint satisfaction techniques in terms of visited nodes without increasing the asymptotic complexity per node. The algorithm can be applied to undirected or directed graphs, and also to labelled and attributed graphs (Löwe *et al.* 1993) by just taking attributes into account in the problem formulation.

We have also proposed a benchmark for graph pattern matching that is based on the Stanford GraphBase. In this benchmark we have shown that the new algorithm can solve more problem instances than general constraint satisfaction algorithms (within a fixed limit of time). In addition, the benchmark has many problem instances that none of the considered algorithms can solve (and do not seem to be easily solved with current technology). For this reason, we encourage the use of this benchmark for evaluation purposes in further algorithmic research.

References

Akinniyi, F. A., Wong, A. K. C. and Stacey, D. A. (1986) A new algorithm for graph monomorphism based on the projections of the product graph. *IEEE Trans. on Systems, Man, and Cybernetics* **16** (5) 740–751.

Barnard, J. M. (1993) Substructure searching methods: Old and new. *J. Chem. Inf. Comp. Sci.* 33 (4) 532–538.

Barrow, H. G. and Burstall, R. M. (1976) Subgraph isomorphism, matching relational structures and maximal cliques. *Inform. Process. Lett.* **4** (4) 83–84.

Berztiss, A. T. (1973) A backtrack procedure for isomorphism of directed graphs. *J. ACM* **20** (3) 365–377.

Bessière, C. (1994) Arc consistency and arc consistency again. *Artif. Intell.* **65** (1) 179–190.

Bessière, C., Freuder, E. C. and Régin, J.-C. (1999) Using constraint metaknowledge to reduce arc consistency computation. *Artif. Intell.* **107** (1) 125–148.

Bessière, C., Meseguer, P., Freuder, E. C. and Larrosa, J. (1999) On forward checking for non-binary constraint satisfaction. In: Principles and Practice of Constraint Programming. *Springer-Verlag Lecture Notes in Computer Science* **1713** 88–102.

Bessière, C. and Régin, J.-C. (1996) MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proc. 2nd Int. Conf. Constraint Programming*, Cambridge, MA, USA 61–75.

Bessière, C. and Régin, J.-C. (1997) Arc consistency for general constraint networks: Preliminary results. *Proc. 15th Int. Joint Conf. Artificial Intelligence* 398–404.

Bunke, H. (1993) Structural and syntactic pattern recognition. In: Chen, C. H., Pau, L. F. and Wang, P. S. P. (eds.) *Handbook of Pattern Recognition and Computer Vision*, Chapter 1.5, World Scientific 163–209.

Bunke, H., Glauser, T. and Tran, T.-H. (1991) An efficient implementation of graph grammars based on the RETE matching algorithm. Graph-Grammars and their Application to Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **532** 174–189.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

Bunke, H. and Messmer, B. T. (1997) Recent advances in graph matching. *Int. J. Pattern Recogn.* **11** (1) 169–203.

Cheng, J. K. and Huang, T. S. (1981) A subgraph isomorphism algorithm using resolution. *Pattern Recognition* **13** (5) 371–379.

Christmas, W. J., Kittler, J. and Petrou, M. (1995) Structural matching in computer vision using probabilistic relaxation. *IEEE T. Pattern Anal.* **17** (8) 749–764.

Cortadella, J. and Valiente, G. (2000) A relational view of subgraph isomorphism. *Proc. 5th Int. Seminar on Relational Methods in Computer Science*, Québec, Canada 45–54.

Cross, A. D. J., Wilson, R. C. and Hancock, E. R. (1997) Inexact graph matching using genetic search. *Pattern Recogn.* **30** (6) 953–970.

Dechter, R. (1992) Constraint networks. In: Shapiro, S. C. (ed.) *Encyclopedia of Artificial Intelligence*, Volume 1, second edition, Wiley 276–285.

Freuder, E. C. (1982) A sufficient condition for backtrack-free search. *J. ACM* **29** (1) 24–32.

Garey, M. R. and Johnson, D. S. (1979) *Computers and intractability: A guide to NP-completeness*, Freeman.

Ghahraman, D. E., Wong, A. K. C. and Au, T. (1980) Graph monomorphism algorithms. *IEEE Trans. on Systems, Man, and Cybernetics* **10** (4) 189–197.

Haralick, R. M. and Elliot, G. (1980) Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.* **14** (3) 263–313.

Haralick, R. M. and Kartus, J. S. (1978) Arrangements, homomorphisms, and discrete relaxation. *IEEE T. Syst. Man Cyb.* **8** (8) 600–612.

Kasif, S., Kitchen, L. and Rosenfeld, A. (1983) A Hough transform technique for subgraph isomorphism. *Pattern Recogn. Lett.* **2** (2) 83–88.

Kitchen, L. (1980) Relaxation applied to matching quantitative relational structures. *IEEE T. Syst. Man Cyb.* **10** (2) 96–101.

Kitchen, L. and Rosenfeld, A. (1979) Discrete relaxation for matching relational structures. *IEEE T. Syst. Man Cyb.* **9** (12) 869–874.

Knuth, D. E. (1993) *The Stanford GraphBase: A Platform for Combinatorial Computing*, ACM Press.

Kondrak, G. and Beek, P. V. (1997) A theoretical evaluation of selected backtracking algorithms. *Artif. Intell.* **89** (1–2) 365–387.

Kuner, P. and Ueberreiter, B. (1988) Pattern recognition by graph matching—combinatorial versus continuous optimization. *Int. J. Pattern Recogn.* **2** 527–542.

Levi, G. (1972) A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo* **9** (4) 1–12.

Levy, S. and Knuth, D. E. (1994) *The CWEB System of Structured Documentation*, Addison-Wesley.

Löwe, M., Korff, M. and Wagner, A. (1993) An algebraic framework for the transformation of attributed graphs. In: Sleep, R., Plasmeijer, R. and van Eekelen, M. (eds.) *Term Graph Rewriting: Theory and Practice*, John Wiley & Sons 185–199.

Mackworth, A. K. (1977) Consistency in networks of relations. *Artif. Intell.* **8** (1) 99–118.

Mackworth, A. K. (1992) Constraint satisfaction. In: Shapiro, S. C. (ed.) *Encyclopedia of Artificial Intelligence*, Volume 1, second edition, Wiley 285–293.

McGregor, J. J. (1979) Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences* **19** 229–250.

McGregor, J. J. (1982) Backtrack search algorithms and the maximal common subgraph problem. *Software—Practice and Experience* **12** (1) 23–34.

Mehlhorn, K. and Näher, S. (1999) *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press

Mohr, R. and Henderson, T. C. (1986) Arc and path consistency revisited. *Artif. Intell.* **28** (1) 225–233.

Mohr, R. and Masini, G. (1988) Good old discrete relaxation. *Proc. 8th European Conf. Artificial Intelligence* 651–656.

Nudel, B. (1988) Tree search and arc consistency in constraint satisfaction algorithms. In: Kanal, L. N. and Kumar, V. (eds.) *Search in Artificial Intelligence*, Springer-Verlag 287–342.

Ohlrich, M., Ebeling, C., Ginting, E. and Sather, L. (1993) SubGemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. *Proc. 30th Annual ACM/IEEE Design Automation Conf.* 31–37.

Park, S. K. and Miller, K. W. (1988) Random number generators, good ones are hard to find. *Commun. ACM* **31** (10) 1192–1201.

Régin, J.-C. (1994) A filtering algorithm for constraints of difference in CSPs. *Proc. 12th Conf. American Assoc. Artificial Intelligence* **1** 362–367.

Rudolf, M. (2000) Utilizing constraint satisfaction techniques for efficient graph pattern matching. In: Theory and Application of Graph Transformations. *Springer-Verlag Lecture Notes in Computer Science* **1764** 215–227.

Schlieder, C. (1998) Diagrammatic transformation processes on two-dimensional relational maps. *Journal of Visual Languages and Computing* **9** (1) 45–59.

Tsang, E. (1993) *Foundations of Constraint Satisfaction*, Academic Press.

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

Valiente, G. and Martínez, C. (1997) An algorithm for graph pattern-matching. Proc. 4th South American Workshop on String Processing. *Int. Informatics Series* **8**, Carleton University Press 180–197.

Wallace, M. (1996) Practical applications of constraint programming. *Constraints: An International Journal* **1** (1–2) 139–168.

Wilson, R. C., Evans, A. N. and Hancock, E. R. (1995) Relational matching by discrete relaxation. *Image and Vision Computing* **13** (5) 411–421.

Wong, E. K. (1992) Model matching in robot vision by subgraph isomorphism. *Pattern Recogn.* **25** (3) 287–303.

Zündorf, A. (1996) Graph pattern matching in PROGRES. Graph Grammars and their Application to Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **1073** 454–468.

https://doi.org/10.1017/S0960129501003577 Published online by Cambridge University Press