Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching

Michael Rudolf, TU Berlin
mich@cs.tu-berlin.de

September 15, 1998

Abstract

This paper presents a way to represent and solve the problem of graph matching – also known as the *subgraph homomorphism problem* – as a *constraint satisfaction problem (CSP)*, opening up direct access to the large variety of research findings on optimized solution algorithms for CSPs. By decoupling the solution algorithm from the concrete graph model, the approach allows for variations of the model without affecting the algorithm. Furthermore, complementing the standard CSP definition, a query concept is introduced to allow abstract representation of concrete implementation properties for optimization purposes.

1 Introduction

In the common rule-based approaches to graph transformation, we use a graph to describe the state of a modelled system at a given time, and we have graph rules to describe operations on the system's state in a "before/after" manner. To apply a graph rule to a state graph, first we have to find an embedding, also called a *match*, of the rule's left side into the state graph. In the field of graph theory, this problem is known as the *subgraph homomorphism problem*[1], and it is known to be NP complete [Meh84]. As a consequence in practice, finding a match for a given rule has a worst-case complexity which is exponential to the size of the rule's left side[2]. Since the actual rewriting operation, where the embedding of the left side of a rule is replaced by its right side, can typically be performed in about linear time, it is obvious that matching performance will be decisive to the overall performance of any implemented graph transformation system.

Of course, an algorithm with exponential time complexity is inacceptable for serious system implementation. While we won't be able to decrease the worst-case complexity of the matching problem unless we prove $P = NP$, it is fortunately possible to considerably reduce its average-case complexity at least. Implemented systems like PROGRES [Zün95, Sch97] have shown that it is possible to cut down the cost of graph matching to a level well suited for application development. In general, to achieve acceptable performance it is inevitable to consequently exploit the special properties of both the concrete graph and graph transformation model, and even of their implementation. Since it is almost impossible to reuse an algorithm developed like that, it is common practice today to re-develop a new algorithm from scratch whenever the need for matching functionality arises in the context of a new software project. Still there is a striking commonality in all those efforts: they are all based on the backtracking paradigm. Knowing that the overwhelming majority of theoretical as well as empirical studies on the optimization of backtracking algorithms is done in the context of constraint satisfaction problems (CSPs), it is near at hand to open this knowledge base for graph matching algorithms by reformulating the subgraph homomorphism problem as a CSP.

Apart from gaining direct access to the rich findings of the research work done in the field of CSPs, solving the graph homomorphism problem as a CSP has another significant advantage: by introducing a new level of abstraction, it allows for variation of the graph model with-

[1] or *subgraph isomorphism problem*, if we want to restrict ourselves to injective embeddings

[2] The size of a graph is given by the sum of its nodes and arcs.

out affecting the actual matching algorithm. To avoid performance drawbacks which often come with abstraction, in section 5 we propose a *query concept* to make crucial properties of the graph model and its implementation available in the abstract CSP representation.

Note that all the ideas and concepts presented in the following sections have been successfully applied in the implementation of the matching subsystem of AGG, a graph transformation system developed at TU Berlin [AGG98, Rud97] which is implemented in Java.

2 Constraint Satisfaction Problems

A lot of problems in computer science, most notably in artifical intelligence, can be interpreted as special cases of the *constraint satisfaction problem (CSP)*. Research work in the CSP area is focussed on the theoretical as well as the empirical investigation of methods to solve such problems. While research activity is lively in the area, results are manifold already. Hence, representing and solving a given problem as a CSP pays off for complex problems where no efficient analytical solutions are known, as we will then be able to directly apply the latest findings in CSP technology.

Our definition of a CSP is based on [DvB97]; it is called *constraint network* there, however.

Definition 2.1 (*CSP*) A *constraint satisfaction problem (CSP)* consists of

- a finite set of *variables* $X = \{x_1, \ldots, x_n\}$,
- a finite and discrete *domain* D_i of possible values for every variable $x_i \in X$, and
- a finite set R of *constraints* on the variables of X.

$\triangle$

We still have to define the central notion of a constraint:

Definition 2.2 (*Constraint*) A *constraint* C_S on a tuple of variables $S = (x_1, \ldots, x_r)$ is a relation on the product of these variables' domains: $C_S \subseteq D_1 \times \cdots \times D_r$.

The number r of variables a constraint is defined upon is called *arity* of the constraint. $\triangle$

A constraint comprises the values a variable is allowed to take with respect to other variables. Of special interest are the constraints with arity two, the so-called *binary constraints*, as any arbitrary n-ary CSP can be transformed into an equivalent CSP with only unary[3] and binary constraints [RDP90]. Hence, most literature is only concerned with such *binary CSPs*, and we will consider binary CSPs exclusively, too. A nice property of binary CSPs is that they can easily be depicted as simple graphs, where the variables are denoted as nodes and the constraints as directed arcs between them; the direction of an arc gives the order of the variables in the corresponding constraint's tuple. The graph representation of a CSP is called *constraint graph*.

Now what is a solution of a CSP? To explain, we need some more definitions:

Definition 2.3 (*Instantiation of Variables*) Let $X = \{x_1, \ldots, x_n\}$ be a set of variables with their respective domains $D_i, i \in \{1, \ldots, n\}$. Then any n-tuple $\Gamma = (a_1, \ldots, a_n)$ where $a_i \in D_i$ denotes an *instantiation* of each variable x_i with the corresponding value a_i. We also write $\Gamma(x_i) = a_i$ for the value of x_i under an instantiation Γ. $\triangle$

Definition 2.4 (*Satisfied Constraint*) A constraint C_S on a tuple of variables $S = (x_1, \ldots, x_r)$ is *satisfied* by an instantiation Γ if and only if $(\Gamma(x_1), \ldots, \Gamma(x_r)) \in C_S$. $\triangle$

Definition 2.5 (*Solution of a CSP*) An instantiation Γ is a *solution* of a CSP if it satisfies all the constraints of the problem. $\triangle$

The following example illustrates the definitions.

Example 2.6 (*Map Coloring*) In the map coloring problem, we want to color the regions of a map in a way that no two adjacent regions have the same color. The actual problem is that only a certain limited number of colors is available. To cast a CSP out of this problem, we take the regions to be colored as the variables, and the set of colors as the variables' domains. Let's say we have four regions as shown in figure 1, and only three colors available. We get a set of variables $X = \{x_1, x_2, x_3, x_4\}$ with domains, e.g., $D_1 = D_2 = D_3 = D_4 =$

[3] Note that a unary constraint is easily represented by a binary one containing the same variable twice.

$\{\text{red}, \text{green}, \text{blue}\}$. Now we just have to find a set of constraints to express our demand for adjacent regions having different colors. To ensure that x_1 gets another color than x_2, we may define $C_{(x_1,x_2)} = \{(u,v) \in D_1 \times D_2 \mid u \neq v\}$, and just the like for $C_{(x_1,x_3)}, C_{(x_1,x_4)}, C_{(x_3,x_4)}$ and $C_{(x_2,x_4)}$. A solution for the CSP would then be $\Gamma = (\text{red}, \text{green}, \text{green}, \text{blue})$, for instance.

Note that since our constraints are symmetric, the direction of the constraint arcs (i.e., the order in a constraint's variable tuples) is insignificant in this example. △

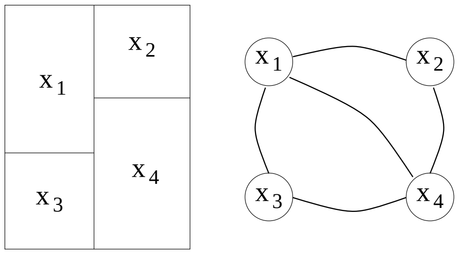

Figure 1: A sample map coloring problem and its representation as a constraint graph.

3 Graph Matching as a CSP

In the graph matching problem, we're trying to find a mapping between the object sets of two graphs. Several restrictions apply to make this mapping a valid graph morphism. The exact definition of these restrictions depends on the concrete graph model, but usually at least a morphism has to preserve the graph structure in some way, and also the typing information of the graph objects if available. The analogy to the constraint satisfaction problem is quite obvious: Here we are also looking for a mapping between two sets, namely between a set of variables and a set of values, where some restrictions apply which are called constraints. So basically, what we have to do to obtain an equivalent CSP for a given graph matching problem is:

- take the objects of the graph to be matched as the CSP's set of variables,
- take the objects of the graph to be matched into as the variables' domain, and
- find a proper translation of the restrictions that apply to a graph morphism into a set of constraints.

This way, we'll get a CSP whose solutions are actually graph morphisms. To go into some more detail on how to construct an equivalent CSP, especially on how to find the proper constraints, first we have to fix the concrete graph model to consider. For the purpose of demonstration, we choose a simple model with typed graph objects:

Definition 3.1 (*Simple Graph*) A *simple graph* G is a tuple $G = (G_V, G_E, L, s, t, \ell)$ consisting of

- a finite set of nodes (or "vertices") G_V and a finite set of arcs (or "edges") G_E where $G_V \cap G_E = \emptyset$,
- two total mappings $s, t : G_E \to G_V$ ("source" and "target"),
- a set of types (or "labels") L and
- a total mapping $\ell : G_V \cup G_E \to L$.

The nodes and arcs of a graph are also collectively called the "objects" of the graph (or "graph objects"). △

A match of one graph into another is given by a graph morphism, which is a mapping of one graph's object sets into the other's, with some restrictions to preserve the graph's structure and its typing information:

Definition 3.2 (*Graph Morphism*) A *graph morphism* $m : L \to G$ between two simple graphs $L = (L_V, L_E, L_L, s_L, t_L, \ell_L)$ and $G = (G_V, G_E, L_G, s_G, t_G, \ell_G)$ is a pair of total mappings $m = (m_V : L_V \to G_V, m_E : L_E \to G_E)$, where the following restrictions apply:

1. $\forall v \in L_V : \; \ell_L(v) = \ell_G(m_V(v))$
2. $\forall e \in L_E : \; \ell_L(e) = \ell_G(m_E(e))$
3. $\forall e \in L_E :$
   - $m_V(s_L(e)) = s_G(m_E(e))$
   - $m_V(t_L(e)) = t_G(m_E(e))$

The image of a graph object o with respect to a morphism m we simply denote by $m(o)$, since the index can always be derived from the context. △

Now that we have defined a concrete graph model and its morphisms, let us consider an example on how to construct a CSP out of a given graph matching problem:

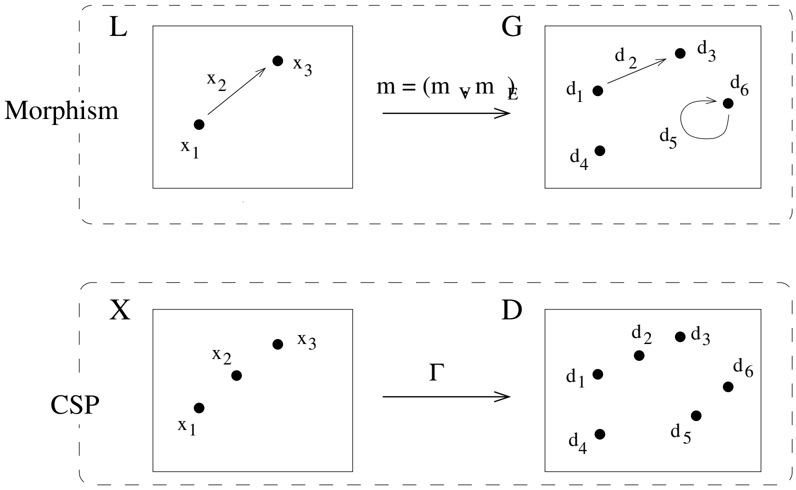

Figure 2: Translation of a graph matching problem into CSP terminology.

Example 3.3 (*CSP for graph matching*) Figure 2 shows a very simple graph matching problem concerning the two graphs L and G; objects' type information is omitted for simplicity. Now we are seeking for all the possible morphisms $m : L \to G$. To construct a CSP out of this problem, we advance as outlined before:

- The set of variables X is given by $X = L_V \cup L_E = \{x_1, x_2, x_3\}$.
- As the variables' domain, we could simply use $D = G_V \cup G_E$. But since we don't want a node to be accidentially mapped to an arc or vice versa, we rather use different domains for the nodes and arcs of L: $D_1 = D_3 = G_V = \{d_1, d_3, d_4, d_6\}$ and $D_2 = G_E = \{d_2, d_5\}$.

Still missing in our CSP is the proper set of constraints to enforce morphism properties on the solutions. For example, it must not be allowed to assign x_2 to d_2 when x_1 is already mapped to d_3. Hence, we need a constraint to ensure source compatibility:

$$\begin{aligned} C^{\text{src}}_{(x_2,x_1)} &= \{(d_E, d_V) \in D_2 \times D_1 \mid s(d_E) = d_V\} \\ &= \{(d_2, d_1), (d_5, d_6)\} \end{aligned}$$

Analogously, we need a constraint to ensure target compatibility for the mapping of x_2 w.r.t. x_3:

$$\begin{aligned} C^{\text{tar}}_{(x_2,x_3)} &= \{(d_E, d_V) \in D_2 \times D_3 \mid t(d_E) = d_V\} \\ &= \{(d_2, d_3), (d_5, d_6)\} \end{aligned}$$

And as a matter of fact, for our tiny example these two constraints already do the trick; now our CSP yields only the solutions $\Gamma_1 = (d_1, d_2, d_3)$ and $\Gamma_2 = (d_6, d_5, d_6)$ which are valid morphisms. Furthermore, it is easy to see that these are actually the only two morphisms that can exist between the graphs L and G.

So far we deliberately ignored the typing of graph objects which is provided by our graph model. If we had differently typed objects in our graphs and therefore would like the solutions of our CSP to be type compatible too, we'd simply have to add another constraint $C^{\text{type}}_{(x_i)} = \{d \in D_i \mid \ell(x_i) = \ell(d)\}$ for every object $x_i \in X$. These are unary constraints which can of course be notated as binary constraints $C^{\text{type}}_{(x_i,x_i)}$ as well. Figure 3 shows the complete constraint graph of the resulting CSP. △

Instead of adding unary type constraints to our CSP, we could also simply remove those values from any domain D_i that will not satisfy the constraint $C^{\text{type}}_{(x_i)}$. This is the preferred method to deal with unary constraints, as it avoids dynamically checking the constraints over and over again in the backtracking solution process. It is known as establishing *node consistency*[4] [Mac77, Kum92] on a CSP. However in section 5 we will introduce a concept using *queries* as a much more flexible approach to domain reduction.

We want to generalize the construction to be able to automatically generate the proper CSP for any two given graphs:

[4]this refers to the nodes of the constraint graph, i.e. the variables of a CSP

Condition $\longleftrightarrow$	Constraint $\in R$
$x_i = x_k$	$C^{\text{type}}_{(x_i)} = \{d \in D_i \mid \ell_L(x_i) = \ell_G(d)\}$
$x_i \in L_E,\ x_k \in L_V,\ s_L(x_i) = x_k$	$C^{\text{src}}_{(x_i,x_k)} = \{(d_i, d_k) \in D_i \times D_k \mid s_G(d_i) = d_k\}$
$x_i \in L_E,\ x_k \in L_V,\ t_L(x_i) = x_k$	$C^{\text{tar}}_{(x_i,x_k)} = \{(d_i, d_k) \in D_i \times D_k \mid t_G(d_i) = d_k\}$

Table 1: Conditions for when to establish a constraint between two variables x_i, x_k.

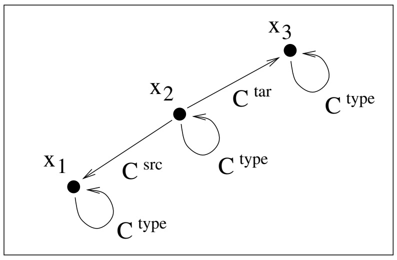

Figure 3: The constraint graph of the CSP in example 3.3.

Construction 3.4 (*CSP for graph matching*) Given two graphs $L = (L_V, L_E, L_L, s_L, t_L, \ell_L)$ and $G = (G_V, G_E, L_G, s_G, t_G, \ell_G)$ according to def. 3.1. Then we construct a CSP as follows:

- $X = L_V \cup L_E = \{x_1, \ldots, x_n\},\ n = |X|$.
- $D_i = \begin{cases} G_V, & \text{when } x_i \in L_V \\ G_E, & \text{else} \end{cases}$, $i \in \{1, \ldots, n\}$.
- The constraint set R is built according to table 1: whenever a condition listed in the left column of the table holds for a given pair of variables (x_i, x_k), the corresponding constraint is to be included in R.

$\triangle$

It can be easily shown that the resulting CSP is equivalent to the original graph matching problem. For a straightforward proof of this proposition, cf. [Rud97].

4 Variable Ordering & Intelligent Backtracking

In this section we give a brief sketch of some basic issues arising when it comes to solve a CSP algorithmically. These are issues well-known to the CSP community, which are discussed in more detail in the literature on the subject. We recommend [Kum92] as a good starting point for further reading.

The standard approach to solve a CSP is using backtracking. Given a CSP like the one in figure 4 with certain variable and value orders, backtracking essentially performs a depth-first search on the depicted search space of potential solutions.

The good thing about backtracking is that whenever a constraint check fails for a variable x_i, it eliminates an exponential sub-problem. Let's elaborate on that. In a CSP with a number of $|X|$ variables and an average domain size of $|D_\emptyset|$, the total number of variable instantiations to be checked amounts to about $|D_\emptyset|^{|X|}$. Now if instantiation fails for a variable x_i with some value, we have $|D_\emptyset|^{|X|-i}$ instantiation variants less left to consider. Interesting enough, the optimization effect grows exponentially with decreasing i. This leads us directly to the most prominent issue in variable ordering: the *first fail principle* suggests that variables with the highest probability of instantiation failure should always come first. A good starting point for how to determine a variable's "failure probability" is the number of constraints it is involved in, but in the end a good variable order is a matter of heuristics highly dependent on the particular application environment. Even the first fail principle, as evident as it may be, is not appropriate in any case; if you are not interested in all the solutions, but want to find only one solution as fast as possible, you are likely to be better off using a *succeed first* strategy instead.

Since the consequences of the applied variable ordering heuristic are so drastic, it is always worth to investigate and experiment. Of course there's a great variety of studies already available on the topic, e.g. [GMP+96, Smi97].

The bad thing about backtracking is, however, that it is dumb in that it often keeps looking for consistent instantiations in some subspace of the search tree where for some reason it

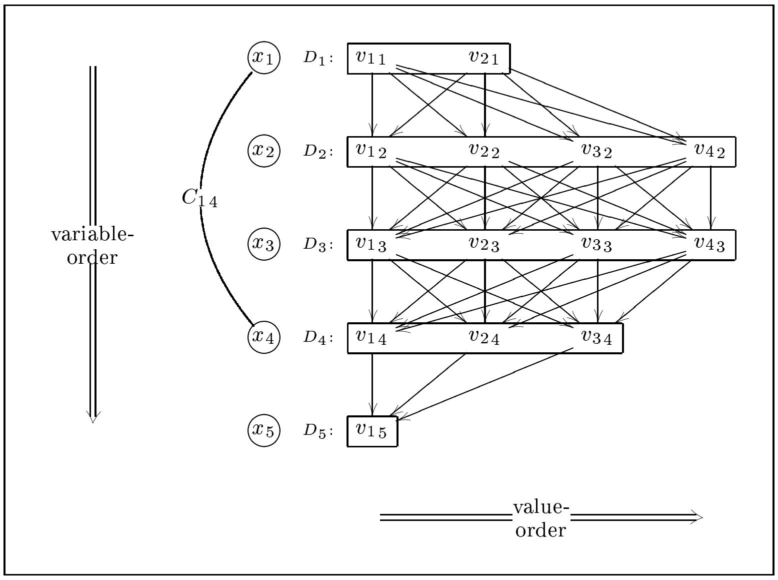

Figure 4: The search space of a sample CSP.

is already evident that the search must fail. This kind of systematic misbehaviour is commonly known as *thrashing*, and it is mostly responsible for the poor performance of simple backtracking in practice. Diverse efforts have been made to overcome these deficiencies; they are subsumed under the collective term *intelligent backtracking*. As an example, we'll now have a look at a typical thrashing situation and at the idea how to resolve it.

Example 4.1 (*Thrashing*) Given the CSP constellation in figure 4, let's say we have successfully instantiated the variables x_1 through x_3 with values v_{11}, v_{12}, and v_{13}, respectively, using the standard backtracking procedure. Now suppose there's no consistent instantiation of x_4 w.r.t. C_{14} and the current value of x_1: $\forall v \in D_4 : (v_{11}, v) \notin C_{14}$. As a consequence, the subsequent attempt to instantiate x_4 will fail. A back step is due, and the next possible value v_{23} will be assigned to x_3. However, since the constraint C_{14} having caused the failure in the first place is independent of x_3, reattempting to instantiate x_4 will inescapably fail again. Obviously, the only chance to break out of the "dead end" at x_4 is to change the value of x_1. But the simple backtracking scheme will not revisit x_1 unless it has finished "thrashing" at all the possible value combinations of the variables x_2 and x_3. Evidently, this kind of thrashing situation imposes an overhead growing exponentially with the number of variables enclosed by the constraint which caused the failure. △

This is where *backjumping* comes in. Backjumping is a simple intelligent backtracking algorithm that tries to be smart about just the kind of situation discussed above. Its basic idea is just as simple as it is appealing: "Whenever a dead-end occurs during variable instantiation, determine the culprit of failure and jump directly back to revise the variable concerned, without revisiting all the variables in between." But the backjumping algorithm has some drawbacks, too, and it only addresses one of the many causes for thrashing behaviour. Therefore, numerous approaches are out there trying to be much more intelligent about eliminating virtually any kind of thrashing. For further reading on intelligent backtracking, you may refer e.g. to [Dec97, Bru81, Kon94, Pro93].

But be aware that, as [Kum92] puts it, "a simple intelligent backtracking scheme may turn out to have less overall complexity than a more complicated intelligent backtracking"; this is due to a tendency towards an increasing expense on the maintenance of auxiliary data structures in the more sophisticated approaches.

5 Domain Reduction by Queries

By now, we have focussed our attention on the optimization of only one dimension of the search space: both intelligent backtracking and variable ordering address optimization *in depth*. These approaches are most promising as they directly affect the exponential factor of the worst-case complexity term $|D_\emptyset|^{|X|}$, so they are likely to have substantial impact on the actual run-time performance.

Although it may seem less profitable to consider optimization *in breadth* since it is only concerned with the base factor $|D_\emptyset|$, it is still worthwhile to have a closer look. What optimization in breadth actually means is trying to cut down on the overall number of constraint checks by reducing the domains in some way. In particular, the query-approach to dynamic domain reduction presented below will often be able to cut down a variable's domain by orders of magnitude, having considerable implications on run-time performance.

Of course we may only remove a value from a domain if it is certainly not part of any of the CSP's solutions. Obviously, for a value $v \in D_{x_i}$ to be part of a solution, it is a necessary condition that v satisfies every constraint $C_{(x_i,x_k)}$, $k \leq i$.[5] We call these conditions *dynamic* as they depend on the state of other variables which have been instantiated previously in the solution process. The observation is that sometimes, the value domain is structured in a way providing us with operations to obtain exactly the set of all values satisfying such a condition; this set can then be used as a properly reduced domain. Let us consider an example:

Example 5.1 (*Dynamic domain reduction*) Suppose that in the situation depicted in fig. 2, x_2 is instantiated by d_2 while no other variables have any values assigned. As a necessary condition for a consistent value $\Gamma(x_3)$ for x_3, we have $(d_2, \Gamma(x_3)) \in C^{\text{tar}}_{(x_2,x_3)}$ (cf. fig. 3). With the definition of the target constraint, we get $\Gamma(x_3) = t(d_2)$. The term $t(d_2)$, however, can be evaluated in constant time yielding the only consistent value d_3, whereas the standard procedure would have taken us linear time to find the same result in the domain D_3 by trial and error.

Things look different though when we start the other way around, looking for a consistent instantiation of x_2 while d_3 is already assigned to x_3. The necessary condition imposed by $C^{\text{tar}}_{(x_2,x_3)}$ leads to $d_3 = t(\Gamma(x_2))$. By "looking close" we can easily see that only d_2 matches this condition; but since there is no inverse counterpart to the target operation available in our graph model, we cannot resolve the equation to $\Gamma(x_2)$. At this time, we have to descend from the abstract model down to the implementation level. Considering e.g. AGG, its graph model implementation does in fact provide direct access to the outgoing and incoming arcs of a node, representing exactly the concept of inverse relations to the source and target operations we were missing in the abstract model. Thus, we can still get constant time access to the reduced domain we were looking for. △

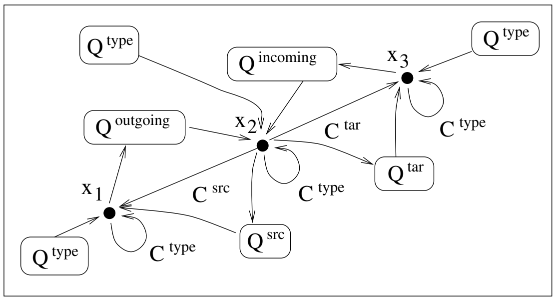

Figure 5: The constraint graph of fig. 3 with some typical queries.

The example teaches us that in the end, to do this kind of dynamic domain reduction efficiently we need to know about the properties of the domain's implementation. This gives rise to a problem seeming contradictory: On one hand, we want to exploit concrete knowledge about the properties of domain implementation for the sake of optimization, on the other hand we want to settle the solution algorithms which are to utilize that knowledge on the abstract level of a general CSP, just to decouple them from the issues of the concrete graph model. Fortunately, this kind of problem has been solved before: hiding their concrete definition behind the abstract notion of satisfaction, the constraints themselves are the perfect example for the concept of abstract representation of concrete model properties. Following this example, we introduce the notion of a *query*[6] to the abstract framework of the CSP:

Definition 5.2 (*Query*) Given a CSP Z with variable set $X = \{x_1, \ldots, x_n\}$ and corresponding domains D_i, $i \in \{1, \ldots, n\}$. Also given a

[5] We assume that a variable's index reflects the variable order here.

[6] Name and idea have been inspired by the use of queries in [Zün96].

subset $Y \subset X$, a consistent instantiation Γ of Y and a variable $x_t \in X \setminus Y$. A *Query* $Q_{Y,x_t}(\Gamma)$ is then a subset of D_t so that the following condition holds:

$$\begin{array}{lll} \forall d \in D_t \setminus Q_{Y,x_t}(\Gamma) : & & \\ \quad \nexists \Gamma_X & \text{where} & \Gamma_X(x_t) = d \\ & \text{and} & \Gamma_X \text{ is a solution of } Z. \end{array}$$

The elements of Y are called the *sources* of $Q_{Y,x_t}(\Gamma)$, while x_t is known as the *target* of the query.

A query $Q_{Y,x_t}(\Gamma)$ is called *constant* if and only if $Y = \emptyset$. We may also write Q_{x_t} in that case. $\triangle$

Queries may be used for the purpose of dynamic as well as static domain reduction, where the latter is the job of constant queries. Note that static domain reduction corresponds to the notion of *node consistency* mentioned in section 3.

In a constraint graph, a query Q_{Y,x_t} is represented by a hyperedge originating in the variables of Y and terminating in x_t. Figure 5 shows the constraint graph from figure 3, complemented by some simple queries. Since none of these queries has more than one source variable, there are no real hyperedges displayed. As an example on how the definition of a query may look like, we consider $Q^{\text{tar}}_{\{x_2\},x_3}$: $Q^{\text{tar}}_{\{x_2\},x_3}(\Gamma) = \{t(\Gamma(x_2))\}$. Note that the queries for incoming and outgoing arcs cannot be defined in terms of the definition of our graph model.

6 Conclusion

In this paper, we proposed a way to represent and solve the graph matching problem as a CSP. The main benefit of this approach is that we gain direct access to the rich research findings in the CSP area; instead of inventing new algorithms for graph matching from scratch, we can now apply well-elaborated CSP solution algorithms right "out of the box".

Another important advantage is that the actual solution algorithm becomes independent of the concrete graph model, allowing us to change the model without affecting the algorithm; we only have to adapt the translation step from the graph model into the CSP representation. In spite of this additional level of abstraction, solution algorithms can still be highly optimized by exploiting crucial properties of the graph model implementation. To accomplish this, we introduced the query concept to represent properties of the concrete model on the more abstract CSP level. As an additional means to provide model knowledge for optimized heuristics, we may assign weighting information to constraints, queries, and variables [BMR97], reflecting e.g. the "tightness" of a constraint or the estimated size of a query result.

As an effect of the decoupling of the graph model from the solution process, it becomes feasible now to develop and reuse software libraries of algorithms for CSP solution and variable ordering. To some extent, the CSP framework implemented for the AGG system [Rud97] can be seen as a prototype of such a library.

As for the drawbacks of the CSP aproach to graph matching, we have to accept a linear computation overhead for the translation steps between the graph model and the CSP representation. In general, this overhead is neglectable, since the problem itself is NP hard. But even then, since the constraint graph of the CSP can be generated from the graph rule's left-hand side alone, most of the overhead can be moved away from matching time to editing time.

We could only give a very limited insight into the vast field of optimization issues. For example we did not consider *relational consistency algorithms* at all [Mac77] (also known as *path consistency* or *constraint propagation*), which are frequently used to reduce the amount of backtracking needed in the solution process. Based on a minimal graph model, McGregor [McG79] did some experimental comparison of several relational consistency algorithms when applied to the subgraph isomorphism problem.[7] He found that only the simplest consistency algorithms were cost-effective. However, recent studies [FD96] suggest that the larger and harder the problem instances are, the more constraint propagation may pay off. Note that the query concept introduced here may be seen as a very limited, yet extremely cost-effective, form of constraint propagation as well.

A lot of future work remains to be done, most notably systematical empirical testing is due in order to compare the performance of various solution algorithms and ordering heuristics on typical instances of the graph matching problem.

[7] Actually, he used a simple CSP representation of the problem, without starting any systematical investigation on that matter.

References

[AGG98] AGG Homepage, 1998. http://tfs.cs.tu-berlin.de/agg.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[Bru81] M. Bruynooghe. Solving Combinatorial Search Problems by Intelligent Backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[Dec97] R. Dechter. *Backtracking algorithms for constraint satisfaction problems - a survey.* Technical Report, University of California, Irvine, January 1997. ftp://ftp.ics.uci.edu/pub/CSP-repository/papers/backtracking.ps.

[DvB97] R. Dechter and P. van Beek. Local and Global Relational Consistency. *Theoretical Computer Science*, 173:283–308, 1997. ftp://ftp.cs.ualberta.ca/pub/vanbeek/papers/tcs97.ps.

[FD96] D. Frost and R. Dechter. Looking at full look-ahead. In *Proc. 2nd Intl. Conf. on Constraint Programming (CP-96)*, 1996.

[GMP+96] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. In E. C. Freuder (ed.), *Proc. 2nd Intl. Conf. on Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*, pages 179–193. Springer Verlag, 1996. http://www.cs.strath.ac.uk/~apes/papers/CRCcp96gmpsw.ps.gz.

[Kon94] G. Kondrak. *A Theoretical Evaluation of Selected Backtracking Algorithms.* Technical Report TR94-10, University of Alberta, 1994. ftp://ftp.cs.ualberta.ca/pub/TechReports/TR94-10.ps.Z.

[Kum92] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992. http://www.cirl.uoregon.edu/constraints/archive/kumar.ps.

[Mac77] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[McG79] J. J. McGregor. Relational Consistency Algorithms and Their Application in Finding Subgraph and Graph Isomorphisms. *Information Sciences*, 19:229–250, 1979.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness.* Springer Verlag, 1984.

[Pro93] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[RDP90] F. Rossi, V. Dahr, and C. Petrie. On the equivalence of constraint satisfaction problems. In *Proc. Europ. Conf. on Artificial Intelligence (ECAI90)*, August 1990. Also MCC Technical Report ACT-AI-222-89.

[Rud97] M. Rudolf. *Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation.* Master's thesis, Technical University of Berlin, Dep. of Comp. Sci., 1997. http://tfs.cs.tu-berlin.de/publikationen/public1997.html.

[Sch97] A. Schürr. Programmed Graph Replacement Systems. In G. Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997.

[Smi97] B. M. Smith. Succeed-first or Fail-first: A Case Study in Variable and Value Ordering Heuristics. In G. Smolka (ed.), *Proc. 3rd Intl. Conf. on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 321–330. Springer Verlag, 1997. file://www.scs.leeds.ac.uk/scs/doc/reports/1996/96_26.ps.Z.

[Zün95] A. Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme.* PhD thesis, RWTH Aachen, 1995.

[Zün96] A. Zündorf. Graph Pattern Matching in PROGRES. In *Proc. 5th Intl. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 454–468. Springer Verlag, 1996.