Information Hiding and the Complexity of Constraint Satisfaction

Remco C. Veltkamp

Utrecht University, Department of Computing Science
Padualaan 14, 3584 CH Utrecht, The Netherlands
e-mail: Remco.Veltkamp@cs.ruu.nl

Richard H. M. C. Kelleners

CWI, Department of Interactive Systems
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
e-mail: richard@cwi.nl
and
Technical University of Eindhoven, Department of Computing Science
Den Dolech 2, 5612 AZ, Eindhoven, The Netherlands

Abstract This paper discusses the complexity of constraint satisfaction, and the effect of information hiding. On the one hand, powerful constraint satisfaction is necessarily global, and tends to break information hiding. On the other hand, preserving strict information hiding increases the complexity of constraint satisfaction, or severely limits the power of the constraint solver. Ultimately, under strict information hiding, constraint satisfaction on complex objects cannot be guaranteed.

1 Introduction

Graphics systems are typically very large integrated programs that use a wide range of techniques. They may deal with various types of data and allow concurrent interaction with a user and between a large number of agents/actors/active objects. A well founded and appropriate underlying abstraction is needed to deal with the complexity of computer graphics. The aim of any abstraction is to provide a context within which problems can easily be solved, and to provide a formal framework for thinking about the solutions and implementing them. *Object-orientedness* provides such an underlying abstraction to deal with complexity.

The concept of *constraints* is another such abstraction. Constraints specify relations between objects which must be maintained. When a change occurs to one of the constrained objects, all affected objects must be adjusted such that all the constraints remain satisfied. There are two aspects to constraints:

- Description: constraints specify the relation between objects; this is the declarative aspect.

- Satisfaction: in order to solve a set of constraints, methods of constraint satisfaction have to be given; this is the procedural aspect.

Programming with constraints is a form of declarative programming. A declarative program specifies the relations which have to hold in the results of the computation. It specifies what the result of the computation should be, not how to go about the computation. The underlying solving engine takes care of the procedural part, and has to resolve the constraints in order to produce the solution. The principal benefit of declarative approaches is that they shift the burden of deciding how something has to be done from the application programmer to the system environment he is working in.

The use of constraints in managing the complexity of designing interactive graphics systems dates back to the earliest days of interactive graphics (consider Sutherland's Sketchpad from the early sixties [20]), and still receives much attention. It has an intuitive appeal to regard the requirements of a model, interaction, or animation as constraints that have to be maintained. The strength of constraints in computer graphics is the rich set of associations with the modeled reality: reality often behaves like it is driven by constraints.

A concept complementary to abstraction is *encapsulation*. Whereas abstraction focuses on the observable behavior of an object, encapsulation focuses on the implementation behind that behavior. Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior. It serves to separate the contractual interface of an abstraction and its implementation [2].

Encapsulation could be achieved by physical separation, e.g. residing the data structures and methods of each object on a private processor. Encapsulation is most often achieved through *information hiding*, i.e. the process of hiding all the secrets of an object that do not contribute to its essential characteristics. Typically, the structure of an object is hidden, as well as the implementation of its methods [2].

The justification for combining objects and constraints derives from the fact that it addresses the problems of complexity in large interactive graphical systems which arises on two fronts. The first is the complexity inherent in specifying the behavior of animations and interactions with many components or objects. Constraints allow the declarative modeling of the behavior of such systems. The second front is the complexity due to the fact that we are dealing with large software systems. Sound software engineering principles, such as data encapsulation, are needed to cope with large complex software systems.

Abstraction and encapsulation combine well in the object-oriented paradigm. However, the combination of objects with constraints fits less well: a constraint solver looks at, and sets, the constrained objects' internal data, which conflicts with the data encapsulation concept in the object-oriented paradigm. This paper discusses one aspect of this incompatibility. It appears that most constraint systems in an object-oriented environment infringe the information hiding principle to some extent. It is an interesting question if strict information hiding limits the power of constraint satisfaction and may increase the time complexity

to exponential orders. In this paper we consider the effect of information hiding on the time complexity of constraint satisfaction.

2 Constraint satisfaction

Formally, a constraint satisfaction problem (CSP) can be specified by a finite set of variables $v_1, \ldots, v_n$, a domain D_i of possible values for each v_i, and a set of constraints $C_1, \ldots, C_m$. Solving a CSP is finding a valuation $(s_1, \ldots, s_n) \in D_1 \times \ldots \times D_n$ of the variables for which all constraints are satisfied. A domain can be explicit, e.g. a finite enumeration of values, or implicit, e.g. $\mathbb{N}$ for an integer variable. In the latter case the constraint solver must have knowledge about the domain of variables in order to generate correct values. For instance, it may not generate real values for an integer variable.

Possible variants of the CSP are: find a single solution (valuation), find all solutions, find a best solution, find the number of solutions, determine satisfiability (whether or not a solution exists). The satisfiability variant is the simplest one (indeed, it follows from all the others), and even this one is in NP: it is in the class of Nondeterministic Polynomial-bounded problems, i.e. it can be solved by a nondeterministic algorithm in polynomial time, but there is no deterministic algorithm known to solve the problem in polynomial time. It is even NP-complete, i.e. if there were a polynomial-bounded algorithm to solve the problem, then there would be a polynomial-bounded algorithm for each problem in NP. This can be deduced from the fact that the satisfiability variant can be converted in polynomial time into the CNF-satisfiability problem, which is NP-complete [1]. The CNF-satisfiability problem is to determine if there is a truth assignment (a way to assign the values true and false) for the variables in a logical expression in conjunctive normal form (CNF) such that the value of the expression is true. A logical expression in conjunctive normal form is a sequence of clauses separated by conjunctions ($\wedge$), where a clause is a sequence of variables and negations of variables separated by disjunctions ($\vee$).

Because the general CSP is a hard problem, satisfaction algorithms are often slow. One way to speed up satisfaction is to perform computations in parallel. For example, [14] performs massively parallel computations by means of chemical reactions on DNA strands. One can also confine the domain of the constraints and the variables and exploit some of the specific knowledge of the domain. The time complexity of such specialized constraint satisfaction depends on both the domain and the kind of constraints. Symbolic solution of the algebraic expressions describing geometric constraints is known to be NP-hard; tree structured constraint networks can be solved in linear time; linear constraints over real numbers are solvable in polynomial time; a single polynomial constraint of degree higher than four does not even have an analytical solution; and the complexity of solving polynomials on integers of degree higher than two is still unknown.

There are basically two different models that constraint solvers can use for determining solutions (values for variables) to the constraints: the alternation

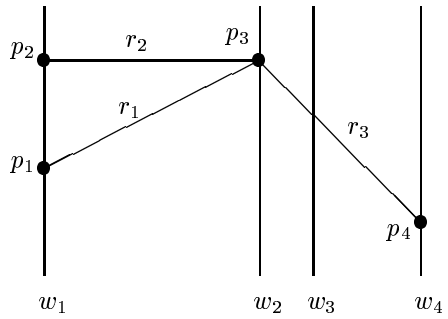

Fig. 1: Example of the Linkage Positioning problem.

and the refinement model. In the alternation (also called perturbation) model, each variable v_i gets assigned a single value s_i. If a variable's value does not satisfy the constraints on it, the constraint solver tries alternative values from its domain (possibly eliminating the incorrect value from the domain). Satisfaction is completed if each variable has a solution such that all constraints hold. After satisfaction, the current domain of each variable can still contain values that do not satisfy constraints. In the refinement model, each variable v_i gets assigned a subset S_i of its domain D_i. If a value s_i in the proposed solution S_i does not satisfy the constraints on v_i, then the constraint solver eliminates that value. Satisfaction is completed when the solution set of each variable satisfies the constraints on that variable. After satisfaction, not every valuation $(s_1, \ldots, s_n) \in (S_1, \ldots, S_n)$ needs to satisfy all constraints, but for every $s_i \in S_i$, values $s_j \in S_j$, $j \neq i$ can be found that satisfy all constraints.

Many constraint solvers in computer graphics use the alternation model with continuous domains of the variables. Many algorithms in artificial intelligence use the alternation model with discrete domains. In constraint logic programming the refinement model with continuous domains is mostly used. The two approaches can also be combined.

3 The Linkage Positioning Problem

In this paper we use the following running example. Let $R = \{r_1, \ldots, r_m\}$ be a set of m rods. The set of endpoints of these rods is $P = \{p_1, \ldots, p_n\}$, a set of n points $p_i = (x_i, y_i)$ in $\mathbb{R}^2$. Each rod r_i has a fixed length d_i. The rods r_i are alternatively denoted as its pair of endpoints: $r_i = (p_{i_1}, p_{i_2})$, $i_1, i_2 \in \{1, \ldots, n\}$. The length of rod (p_{i_1}, p_{i_2}) is alternatively denoted as $d_{i_1 i_2}$. The rods form a single linkage structure, ℓ; if two rods are joined, then they share an endpoint. Let $W = \{w_1, \ldots, w_k\}$ be a set of k vertical walls (in $\mathbb{R}^2$); each w_i is represented by its x-coordinate. The rods must be positioned on the walls in such a way that the endpoints of each rod lie on different walls. See Figure 1 for an example.

This Linkage Positioning problem can be modeled as a constraint problem in various ways.

Constraint Problem 1 (CP-1)

The set of constraint variables is $V = \{p_1, \ldots, p_n\}$. The domain of each variable (x_i, y_i) is $W \times \mathbb{R}$. The constraints are a set of m binary constraints that fix the length between two points that belong to the same rod and prohibit these points to lie on one wall.

Constraint Problem 2 (CP-2)

The set of constraint variables is $V = \{r_1, \ldots, r_m\}$. Each variable is responsible for maintaining its internal integrity, i.e. it retains a fixed length. The domain of each variable $r_i = (p_{i_1}, p_{i_2})$ is $(W \times \mathbb{R})^2$. The constraints are a set of m unary constraints that prevent each rod from being vertical, and q binary constraints to specify incident rods. The q binary constraints construct the linkage, and the value of q depends on the structure of the linkage, but is at least $m - 1$. If the maximal number of rods incident to a point is constant (independent of n or m) then $q = \mathcal{O}(n)$. If a constant number of points are incident to $\mathcal{O}(n)$ points then $q = \mathcal{O}(n^2)$. If $\mathcal{O}(n)$ vertices are incident to $\mathcal{O}(n)$ points then $q = \mathcal{O}(n^3)$.

Constraint Problem 3 (CP-3)

The constraint variable is the whole linkage ℓ. The domain of this variable is $(W \times \mathbb{R})^n$. The linkage should maintain the fixed lengths of the rods, and the incidency relations between the rods. As in CP-2, there are m unary constraints to prevent each rod from being vertical. These are unary constraints on the single linkage variable.

Typically, the internal variables of the objects are the points, and they attain a value from $W \times \mathbb{R}$, or they are unspecified, i.e. have no value yet. Instead of restricting the domain of the x_i's to W in each of the three constraint problems above, we could let the domain be $\mathbb{R}$, and add extra unary constraints to place the points on the wall.

The Linkage Positioning Problem illustrates different types of constraints (unary, binary) and different types of domains (discrete, continuous). It is suitable to demonstrate both the alternation and the refinement models of constraint satisfaction.

The Linkage Positioning Problem is equivalent to a famous problem in computability and complexity theory: (k-)Graph Coloring. The problem there is to assign a color from a given set of k colors to each vertex of the graph, such that vertices connected by an edge have different colors. The set of vertices corresponds to P from the Linkage Problem, the set of edges to R, and the set of colors to W. The condition of 'different colors' corresponds to 'different walls, plus fixed distance'; both conditions take constant time to check. Just like general constraint satisfaction, k-Graph Coloring is known to be NP-complete [1].

Determining if a graph is 2-colorable is easy (polynomial). Determining if it is 3-colorable is NP-complete. It is still NP-complete if the graphs are planar and the maximal degree is four. If the maximal degree is at most two, the k-coloring problem is easy. Coloring bipartite graphs (e.g. trees) is also easy.

4 Constraints vs information hiding

Consider the following constraints on rods $\mathtt{r}_1$, $\mathtt{r}_2$, $\mathtt{r}_3$:

$$\mathtt{r}_1.\mathtt{p}_1 = \mathtt{r}_2.\mathtt{p}_1,\ \mathtt{r}_2.\mathtt{p}_2 = \mathtt{r}_3.\mathtt{p}_1$$
$$\mathtt{r}_1.\mathtt{p}_1.\mathtt{x} \neq \mathtt{r}_1.\mathtt{p}_2.\mathtt{x},\ \mathtt{r}_2.\mathtt{p}_1.\mathtt{x} \neq \mathtt{r}_2.\mathtt{p}_2.\mathtt{x},\ \mathtt{r}_3.\mathtt{p}_1.\mathtt{x} \neq \mathtt{r}_3.\mathtt{p}_2.\mathtt{x}$$

A possible solution is the following (for some y):

$$\mathtt{r}_1.\mathtt{p}_1 = (\mathtt{w}_1,\mathtt{y}),\ \mathtt{r}_1.\mathtt{p}_2 = (\mathtt{w}_2,\ \mathtt{y}+\sqrt{(\mathtt{r}_1.\mathtt{length})^2-(\mathtt{w}_2-\mathtt{w}_1)^2})$$
$$\mathtt{r}_2.\mathtt{p}_1 = (\mathtt{w}_1,\mathtt{y}),\ \mathtt{r}_2.\mathtt{p}_2 = (\mathtt{w}_2,\ \mathtt{y}-\sqrt{(\mathtt{r}_2.\mathtt{length})^2-(\mathtt{w}_2-\mathtt{w}_1)^2})$$
$$\mathtt{r}_3.\mathtt{p}_1 = (\mathtt{w}_2,\ \mathtt{y}-\sqrt{(\mathtt{r}_2.\mathtt{length})^2-(\mathtt{w}_2-\mathtt{w}_1)^2})$$
$$\mathtt{r}_3.\mathtt{p}_2 = (\mathtt{w}_1,\ \mathtt{r}_3.\mathtt{p}_1.\mathtt{y}+\sqrt{(\mathtt{r}_3.\mathtt{length})^2-(\mathtt{w}_1-\mathtt{w}_2)^2})$$

Information hiding is first violated by the constraint expressions, and then by expressing the solution. To avoid this problem, approaches based on message passing have been proposed. In [12], the methods of an object that may violate constraints are guarded by so-called propagators. The propagators send messages to other objects to maintain the constraints. This technique is similar to the pre- and postcondition facilities in Go [5]. It is limited to constraint maintenance (i.e. truth maintenance, as opposed to starting with an inconsistent situation that is then resolved), and not further considered in this paper.

A more powerful technique is presented in [23]. There, the constraint solver produces a set of programs that solve constraints which are stated in the form of equations. It translates a declarative constraint equation into procedural solutions in terms of messages back to objects.

The problem is the local character of the solution. More powerful solutions are necessarily global in nature. The danger is that all objects need methods to get and set their internal data. This however, allows every other object to get and set these values, which is clearly against the object-oriented philosophy.

One way to restrict this, is to have an object allow value setting only when its internal constraints remain satisfied (see [18]). A constraint could be made internal by constructing a 'container object', which contains the constraint and the operand objects, but this does not solve the basic problem. In particular, the state of active objects cannot be changed without their explicit cooperation. (Active objects, or actors, conceptually have their own processor and behave autonomously, which is typical in animation and simulation.) Another approach is to limit access to private data to constraint-objects or the constraint solver-objects only. For example C++ provides the 'friend' declaration to grant functions access to the private part of objects. This is also comparable to the approach taken by [4], where special variables (slots) are accessible by constraints

only. One can argue that encapsulation is still violated (and specifically that the C++ friend construct could be easily misused).

This raises the question how strict information hiding affects (the complexity of) constraint satisfaction. In the following sections we will review a number of constraint satisfaction methods, determine their time complexities, and illustrate that with the particular time complexities for the instances of the Linkage Positioning Problem.

5 Backtracking

The first constraint satisfaction algorithm we consider is backtracking. The basic operation is to pick one variable at a time and assign a value from the domain to it. If the assignment violates some constraints, another value, when available, is chosen. If all variables are assigned a value, the problem is solved. If at any stage no value can be found for a particular variable without violating any constraints, the variable which was last picked is revised and an alternative value, when available, is assigned to that variable. This carries on until either a solution is found or all combinations of values have been tried and have failed.

Below, the backtrack algorithm is given in pseudocode (see [21]).

```
BT (V, C)
{ if (V=={}) return TRUE;
  v = V.select();
  do
  { s = v.domain().select();
    v.assign(s);
    v.domain().remove(s);
    if (v violates no constraints in C)
    { success = BT(V − {v}, C);
      if (success) return TRUE;
    }
  } while (v.domain() ≠ {})
  return FALSE;          // no solution found, backtracking is done
}
```

This recursive algorithm is called with a set of unassigned variables `V` and the set of constraints `C`. Each time BT is called, it picks a value from the domain of the current variable, assigns this to the variable, and removes the value from its domain. Then is tested if the current assignment together with all previous assignments violate any constraints in `C`. If this is the case, another value from the variable's domain is chosen. If the domain runs empty, no solution can be found for this particular variable and backtracking is performed.

Backtracking is applicable in the alternation model of satisfaction with discrete domains. If there are b variables and the maximum initial domain size for a variable is a, there are at most a^b candidate solutions. Every candidate solution can be checked against all constraints in `C`. If the number of constraints is e, then the complexity of the algorithm is $\mathcal{O}(ea^b)$.

5.1 Linkage Positioning Problem

The backtracking algorithm assumes the domains of a variable to be discrete and finite. In Constraint Problem 1 (CP-1) of the Linkage Positioning Problem, the domain of each variable is p_i $(= (x_i, y_i))$ is $W \times \mathbb{R}$. In order to be able to apply the backtracking algorithm, we have to restrict the domain to W only. The y-coordinate of a point is continuous, and its value is not known at the time a value is picked for the x-coordinate. Therefore, we change the binary constraint $d(p_{i_1}, p_{i_2}) = d_{i_1 i_2}$ into $|x_{i_1} - x_{i_2}| < d_{i_1 i_2}$, with $i_1, i_2 \in \{1, \ldots, n\}$. This forces connected points to lie on walls whose distance is smaller than the fixed distance between p_{i_1} and p_{i_2}. If there are no cycles in the linkage structure, the y-coordinates are of no significance: if the x-coordinates have values which do not violate the constraints, proper values can always be found for the y-coordinates in time linear in the number of variables. The complexity is in that case $\mathcal{O}(mk^n)$.

In CP-2, the set of constraint variables is $V = \{r_1, \ldots, r_m\}$. The domain of each variable $r_i = (p_{i_1}, p_{i_2})$ is $(W \times \mathbb{R})^2$. The unary constraints must be extended with $\|x_{i_1}, x_{i_2}\| < d_{i_1 i_2}$, to assure that the rod objects can always retain their fixed length. In order to be able to apply the backtracking algorithm, we restrict the domain to W^2 only, which has size k^2. The time complexity changes accordingly. In the case the linkage structure contains no cycles, there are k^{2m} candidate solutions. For every candidate solution, all constraints will be checked. The number of constraints is $m+q$, so the complexity of the algorithm is $\mathcal{O}((m+q)k^{2m})$.

In the case of CP-3, the single variable is a complete linkage. An assignment to this variable is a tuple of n walls. The size of its domain equals k^n, which is also the number of candidate solutions to the problem. For every candidate solution, all constraints have to be checked. The number of constraints is m, so the complexity of the algorithm is $\mathcal{O}(mk^n)$, which is the same as for CP-1.

6 Node and arc consistency

Node and arc consistency applies to the refinement model of satisfaction. We call a constraint satisfaction problem *node-consistent* if and only if for all variables, all values in its domain satisfy the unary constraints on that variable (cf. [15]).

The following algorithm achieves node-consistency for discrete domains:

```
NC (V, C)
{ for (each v in V)
  { for (each value s in v.domain() )
    { if (unary constraints from C on v are violated)
      remove s from v.domain();
} } }
```

If the size of each domain is at most a and the total number of unary constraints is e, then the time complexity of this algorithm is $\mathcal{O}(ae)$. The algorithm is easily

adapted for continuous domains. The time complexity then is $\mathcal{O}(Re)$, with R the time needed to check a single constraint.

We call a constraint satisfaction problem *arc-consistent* if and only if for all values in the domain of each variable v, and for all constraints c on v, we can find a value in the domain of the other variables of c that satisfy the constraint.

The following algorithm for achieving arc-consistency assumes that the constraints are unary or binary, and have discrete domains. For each binary constraint `c` on variables $\mathtt{v}_1$ and $\mathtt{v}_2$, the domains of the variables are examined separately by the method `revise`. When the domain is examined, all values are deleted which do not satisfy the constraint. If any value is removed, the other constraints on the variable must be reconsidered.

First the method `revise` of constraint `c`:

```
c::revise(v1,v2)
{ changed = FALSE;
  for (each s1 in v1.domain())
  { delete = TRUE;
    for (each s2 in v2.domain())
    { if (check(s1,s2))
      { delete = FALSE;
        break;
    } }
    if (delete)
    { v1.domain().remove(s1);
      changed = TRUE;
  } }
  return changed;
}
```

The following simple algorithm for arc-consistency is called AC-3, after [15]:

```
AC-3 (V, C)
{ NC (V, C);
  while ( NOT C.empty())
  do
  { C.select(c1, v1, v2);
    C.remove(c1, v1, v2);
    if (c1.revise(v1,v2))
      for (each (c2,v2,v3) ≠ (c1, v1, v2))
        C.add(c2,v2,v3);
} }
```

Its time complexity is $\mathcal{O}(ea^3)$, with a the maximum domain size and e the number of binary constraints, which is linear in the number of constraints. If the constraint graph is planar, then the time complexity is also linear in the number of variables [16]. A less simple but optimal algorithm, AC-4, is presented in [17]. It has time complexity $\mathcal{O}(ea^2)$.

The algorithm is easily adapted for continuous domains, but termination of the algorithm then is not assured. The time complexity becomes $\mathcal{O}(\lambda Re)$, with λ the number of loops taken in the algorithm, and R the time necessary to revise a constraint.

Achieving node and arc-consistency does not solve the constraint satisfaction problem. These algorithms are used to reduce the search space of the problem. E.g., when there are no cycles in the constraint graph, achieving node and arc-consistency implies that the graph is backtrack-free, i.e. a solution can be found without backtracking [8]. In a backtrack-free graph, a solution is found in time linear in the number of nodes [6]. When there are cycles in the graph, additional algorithms have to be used to reduce the problem, such as the recognition of certain patterns (e.g. k-trees).

6.1 Linkage Positioning Problem

In CP-1, there are no unary constraints on the variables, thus, by definition it is node-consistent. Node-consistency for CP-2 is achieved by removing all wall-tuples from the domain of a rod r_i on which the segment cannot lie. These are the wall-tuples whose distance is larger than the rod length and the tuples which contain two the same walls. In CP-2 the domain size is k^2 and the number of unary constraints is m. This results is a time complexity of $\mathcal{O}(mk^2)$ The domain size in CP-3 equals k^n, and the number of unary constraints is m. This results is a time complexity of $\mathcal{O}(mk^n)$ for NC.

In CP-1, arc-consistency means that if we assign to a point p_i an arbitrary wall from its domain, then all other points to which it is adjacent, can be placed on walls so that the constraints still hold. The domain size in CP-1 is k, the number of constraints m, thus the complexity for AC-3 is $\mathcal{O}(mk^3)$. If we assume that there are no cycles in the linkage, finding an actual solution takes an additional $\mathcal{O}(n) = \mathcal{O}(m)$ amount of time.

In the case of CP-2, arc-consistency means that once a rod is placed on two walls, all adjacent rods can also be placed. The difference with CP-1 is that the domain size has increased to k^2 and the number of constraints has become $(m + q)$. The complexity of algorithm AC-3 is now $\mathcal{O}((m + q)(k^2)^3)$. Again, if there are no cycles in the linkage, an actual solution can be found in $\mathcal{O}(n) = \mathcal{O}(m)$ additional time.

Since CP-3 only contains one variable which is the complete linkage, there are no binary constraints. So, if it is node-consistent, it is also arc-consistent.

7 Local propagation

Propagation of known states, or just local propagation, can be performed when there are parts in the network whose states are completely known (have no degrees of freedom). The satisfaction system looks for one-step deductions that will allow the states of other parts to be known. This is repeated until all constraints are satisfied or no more known states can be propagated. If not all constraints can be satisfied, the remaining constraints must be resolved by, for

example, numerical relaxation (see next section). Most constraint satisfaction systems use some form of local propagation. For an arbitrary constraint graph, we can use the following local propagation algorithm:

```
LP (v1)
{ for (each constraint c1 on v1)
    fifo.push(c1,v1);
  while(NOT fifo.empty())
  { fifo.pop(c1,v1);
    changed = c1.revise(v1);
    for (each v2 in changed)
      for (each c2 ≠c1 on v2)          // effectively
        fifo.push(c2,v2);              // LP (v2)
} }
```

Algorithm `LP` is invoked with the variable that triggers the propagation. The constraints on the variables are pushed onto a first-in-first-out queue. In the while-loop every constraint in this queue is revised by possibly changing the variables subject to the constraint other than the variable that triggered the constraint. All other variables that have been modified are put in the set `changed`. These variables and the constraints on them are put into the queue. Because constraints on changed variables are placed in a first-in-first-out queue (as opposed to an unordered set), they are guaranteed to be revised later on, whether the propagation is finite or infinite. This is called fair propagation [9].

If there are no cycles in the constraint graph, LP solves each constraint exactly once, and so the time complexity is linear in the number of constraints. Let R be the time needed to revise a constraint. The time complexity for LP is then $\mathcal{O}(Re)$. The time R depends on the problem. For discrete domains of size a, R is typically $\mathcal{O}(a)$ in the alternation model, and $\mathcal{O}(a^2)$ in the refinement model. If there are cycles in the constraint graph, there is in general no way to determine the complexity of the algorithm except for the refinement model on discrete domains. E.g. for binary constraints, it becomes $\mathcal{O}(ea^3)$, equal to the time complexity of AC-3. For the other cases, let λ be the number of loops done by the algorithm. The time complexity is then $\mathcal{O}(\lambda Re)$.

In the case that the constraint graph contains no cycles, the variables of a solved constraint are not changed by the same constraint again during further propagation. However, if that value prevents other variables to satisfy constraints, no solution is found, since backtracking is not possible. This problem could be circumvented by allowing a constraint to change the value of the variable it was triggered from.

Prior to performing local propagation one can perform an analysis to plan the best order to propagate constraints [19]. It is sometimes efficient to plan how to solve constraints before actually doing it. For example if one drags an graphics object which is constrained and must remain constrained, the planning can be done at the beginning of the drag operation, and the execution can be done in real time.

7.1 Linkage Positioning Problem

In CP-1, the constraint variables are the n points. These are connected to each other by m binary constraints. The number of walls, which is also the size of the domain for a point, is k. If we apply LP to CP-1, the time complexity is $\mathcal{O}(km)$, assuming that there are no cycles. However, as is pointed out above, it cannot always find a solution. In constraint problem CP-2, the variables are the m rods. The domain size has increased to k^2, because every rod contains two points. The total number of constraints are the m unary constraints and the q binary constraints. The complexity for LP is then $\mathcal{O}(k^2(q+m))$. In CP-3, the constraint variable is the complete linkage, consisting of n points and m rods. Consequently, the domain size has increased to k^n, the number of unary constraints is m. The resulting complexity is $\mathcal{O}(k^n m)$.

8 Relaxation

Relaxation is an iterative numerical approximation technique that is often used in cases where local propagation fails, such as cycles in the constraint graph. Relaxation can only be used on variables with continuous numeric values. It makes an initial guess at the values of the unknown variables, and then estimates the error that would be caused by assigning these values to the variables. New guesses are then made, and new error estimates calculated. This process repeats until the error is minimized. The number of iterations can also be limited by a constant in case the error doesn't converge fast enough.

An algorithm for relaxation is given below:

```
RX (V, C)
{ times = 0;
  do
  { error = 0;
    times = times + 1;
    for (each v in V)
    { v.guess(C);
      error = max{error,v.error(C)};
    }
  } while (error > ε AND times < λ)
}
```

The algorithm RX applies to all the variables in the set `V`, subject to the constraints in the set `C`. The local variables of the algorithm, `times` and `error`, denote the number of iterations that have elapsed and the maximum error estimate, respectively. The variable `error` could also be a vector containing the error terms for all variables. In the while-loop, first initial values are guessed for the variables. Then the error is determined by testing the values of the variables against the constraints. The while-loop is repeated until the error term is smaller than a certain value ϵ or the maximum number of iterations λ, is reached.

Let again b be the number of variables and e the number of constraints. If we assume that a variable value can be guessed, and its error estimated, in time linear in the number of constraints on the variable, the complexity for guessing and error estimation of all variables is $\mathcal{O}(be)$. Since the number of iterations is at most λ, the time complexity of RX is $\mathcal{O}(\lambda be)$.

Although the relaxation algorithm is linear in the number of variables and constraints, in practice it can be slow because expensive floating point calculations have to be performed to guess values or estimate errors. In combination with local propagation however, relaxation can often be speeded up.

8.1 Linkage Positioning Problem

In order to be able to use the relaxation algorithm on the Linkage Positioning Problem, we allow the x-coordinates of the variables to be continuous and infinite. This means that the domain for points is $\mathbb{R}^2$.

For CP-1, the set of m binary constraints is now extended with n unary constraints, which state that the points must reside on walls. Error estimation is done by calculating distances between points and distances between points and walls. There are n variables and $m + n$ constraints, so the complexity for RX is $\mathcal{O}(n(m + n)\lambda)$. For CP-2, the number of variables is m and the number of constraints is increased by n unary constraints to constrain the endpoints of a rod to lie on walls. The total number of constraints is thus $n + q + m$. The complexity for RX is then $\mathcal{O}(m(q + m + n)\lambda)$. For CP-3, the number of constraints increases to $n + m$, resulting in a complexity of $\mathcal{O}((m + n)\lambda)$.

9 Equate and OOCS

Equate [23] is a constraint satisfaction system that uses *term rewriting* as a guide to find solutions. Constraints are specified as equations. Rewrite rules convert equations into equivalent sets of equations that can more easily be solved. This repeats recursively (zero or more times) until equations are simple enough to be rewritten to a set of instructions (messages to an object). The *rewrite rules* which rewrite the equations are provided by the classes and are similar to the program clauses of logic programs. Several rules may apply to rewrite an equation.

The partial solutions for every constraint have to be combined in order to provide a solution program for the whole constraint problem. Equate uses so-called *read-sets* and *write-sets* to determine which instructions interfere with the accomplishments of others. These sets contain the (internal) variables of objects that will be read or written to during the execution of the instruction. For example, when an instruction A writes to a variable which is read by another instruction B, instruction A can undo some of the achievements of B and is therefore executed first. If no proper order can be found, Equate finds no solution. In this way, a set of solution programs is generated, which is offered to the application. The application must choose a solution program from that set to execute in order to satisfy the constraints.

Another system, which is quite similar to Equate, is the Object-Oriented Constraint System, OOCS [10]. The main difference between these systems is that OOCS does not use term rewriting. Instead, an object supplies a set of *solution program segments* for each constraint that has been imposed upon it. The object guarantees that execution of any of these segments will leave the object in a state which satisfies the constraint. OOCS then solves a set of constraints by determining which program segment steps interfere with each other. This is achieved in the same way as in Equate by using read-sets and write-sets. By arranging the solution steps using these sets, the OOCS solver is able to decide which are feasible solutions.

Constraint solving with Equate takes the following four steps: (i) rewrite all the constraints, (ii) order the partial solutions into solution programs, (iii) make a selection out of the set of solution programs, (iv) execute a program. does not perform the first step. The last two have to be executed by the application program.

Let e be the number of constraints (initial equations), and s the maximum number of rules to rewrite an equation. Let t be the maximum number of *terms* (instructions) in the body of a rewrite rule, and r the maximum number of rewrite iterations. The number of rewrite steps is then $\mathcal{O}(e\sum_{i=1}^{r} s^i t^{i-1}) = \mathcal{O}(es^r t^{r-1})$. The total number of different solution programs (sets of instructions) is $\mathcal{O}(e\sum_{i=1}^{r}(st)^i) = \mathcal{O}(e(st)^r)$, each solution consisting of $\mathcal{O}(et^r)$ terms.

Very little has been written about the complexity of term rewriting. An approach to determine the complexity of term rewriting systems is given in [3], where a notion of the complexity is given by means of the cost of terms. In [13] is mentioned that if the set of terms is strictly left-sequential, there is a fast algorithm which can find a rule in the rule-base in time linear in the length of the expression to be rewritten (the head of the rule). Assuming that this length is bounded by a low constant number, rewriting a single equation takes constant time. The time for rewriting all constraints is then $\mathcal{O}(es^r t^{r-1})$.

In each solution program, the read and write sets of each of the $\mathcal{O}(et^r)$ instructions have to be checked against the read and write sets of all the other instructions in the solution program, requiring $\mathcal{O}(e^2 t^{2r})$ checks. This must be done for all the $\mathcal{O}(e(st)^r)$ solution programs, giving a total of $\mathcal{O}(e^3 s^r t^{3r})$ checks.

The actual satisfaction of the constraints is done by selecting one of the solution programs and executing the $\mathcal{O}(et^r)$ instructions (method calls).

The total complexity of satisfaction is the sum of complexities of the separate steps. It is dominated by the checking of the read and write sets, i.e. $\mathcal{O}(e^3 s^r t^{3r})$. Assuming that r, s, and t are small, relative to the number of constraints e, and bounded above by a constant, the time complexity is cubic in the number of constraints: $\mathcal{O}(e^3)$.

9.1 Linkage Positioning Problem

We shall use Equate here to exemplify how the Linkage Positioning Problem can be solved using this strategy. The line of reasoning for OOCS remains the same. In order to use Equate, classes, constraint equations, and rewrite rules

must be defined. The purpose of Equate is to generate a solution in the form of a sequence of method calls that satisfy constraints, rather than to generate and test individual values from the variables' domains.

In CP-1, the objects involved are points, their domain is the set of walls. It is the responsibility of the points to attain values in the domain. A typical rewrite rule could be:

```
d(point1,point2)=exp ⟵ fail_unless point1.wall(exp) ≠ NIL;
                       point2.move_to(point1.wall(exp))
```

where $point_1$`.wall(`exp`)` returns a position on a wall other than its current wall, at distance exp. In CP-1, the number of constraints is m, so the time complexity becomes $\mathcal{O}(e^3) = \mathcal{O}(m^3)$.

In CP-2 the objects are the rods. Considering the working of Equate, we choose to combine the unary and binary constraints. For example, a typical rewrite rule could be:

```
incident(rod1, rod2)=TRUE ⟵ fail_unless unary_constraint(rod1)=TRUE;
                             rod1.rotate(rod2)
```

where `unary_constraint` needs further rewriting and rod_1`.rotate(`rod_2`)` rotates rod_1 to be incident with rod_2. The number of constraints is now q. The time complexity $\mathcal{O}(e^3)$ thus becomes $\mathcal{O}(q^3)$.

In CP-3 there is only a single variable, the linkage structure. A typical method of this object is to rotate one part of the linkage around a hinge vertex, leaving the other part fixed. The object is responsible for maintaining internal constraints on those points that have a value. As in CP-1, the number of constraints is m, resulting in $\mathcal{O}(m^3)$ time complexity.

Equate need not always find a solution. Checking the read and write sets can be overly restrictive and may abort valid solutions. Another reason is that run time checks may fail. In CP-1 for example, in order to satisfy the distance constraint, Equate moves a point to only one of the possible solutions. This assignment satisfies the current constraint, but it is possible that it obstructs other variables to satisfy their constraints. Equate then succeeds in producing a solution program, but if the application executes the program, it fails, i.e. one of the `fail_unless` statements evaluates to false and the program aborts.

10 Discussion

Below, an overview is given of the complexities we have discussed for the different constraint solving techniques. We discriminate between the alternation and the refinement model of solving, and between discrete and continuous domains. As usual, a is the size of a discrete domain, b is the number of variables, e the number of constraints. R is the time needed to solve a single constraint, and λ the number of loops or iterations made by the algorithms:

	discrete		continuous	
	alt	ref	alt	ref
BT	ea^b			
NC		ae		Re
AC-3		a^3e		λRe
LP	λae	a^3e	λRe	λRe
RX			λbe	
Equate/OOCS	e^3		e^3	

Naturally, all complexities are at least linear in the number of constraints e. It is immediately clear why local propagation LP is such a widely used constraint solving method. It is applicable in a wide variety of applications and linear in the number of constraints, if R is independent of e. Note however that LP need not always find a solution, which is why it is often preceded by a planning stage.

We have seen in the examples CP-1, CP-2, and CP-3 that the way an application is modeled can largely influence the time complexity of constraint satisfaction. Particularly the number of constraints and the size of domains can vary. By encapsulating more variables into objects (like the vertices in the rods), constraints may move into the objects (like the fixed distances of the rods). On the other hand, new constraints may be needed to describe the relations among the objects (like the incidency constraints between the rods). Furthermore, if the domains of the object variables are discrete, the complexity of the solving algorithm is increased when the domains are enlarged.

11 Conclusions

In the alternation model of satisfaction, many algorithms do hard solution assignments to the objects (possibly in combination with domain refinement). E.g., the backtracking algorithm BT by means of `v.assign(s)`, and the arc consistency algorithm AC-3 through `c.revise()`. The complexities above are valid on the assumption that the objects accept the assignment. However, to maintain any form of information hiding, the assignment should be done through message passing, encapsulating the internal implementation of the objects (as Equate and OOCS do).

Equate and OOCS obey information hiding to some extend: neither the constraints nor the procedural solutions refer directly to an object's implementation. Actually, neither the solver nor the object determines a solution alone. The objects offer possible local solutions, and the solver tries to combine them into a global solution. However, a global solution need not be found, and the complexity is cubic in the number of constraints, which limits the use to small scale applications. Note that in the end the use of read-sets and write-sets in Equate and OOCS, which contain implementation specific knowledge of an object, infringes the concept of information hiding after all. In [10] is put forward that encapsulation is maintained from the application programmer's perspective. The read-sets and write-sets are only available to the constraint solver,

thus keeping the benefits of object-oriented programming for the programmer.

If an object has to maintain internal relationships and refuses an assignment, then either the constraints are not satisfied, or the solver has to negotiate with the objects to accept values. In the latter case the time complexity typically becomes exponential in a, or worse for continuous domains. Obviously, to avoid a situation that an object can refuse a solution via a message, the methods should be designed so as to obey the internal relationships, or the solver must take into account these relationships. In the first case the satisfaction power of the solver may be limited, in the second case the information hiding principle is broken. Encapsulating the objects and completely hiding them from the constraint solver prevents the solver from doing any global solution. In the case of multiple constraints, this prevents the solver from doing any work at all.

This leads to the following theorem:

> **Principal Theorem** Under strict information hiding, constraint satisfaction on objects cannot be guaranteed.

The relaxation algorithms were formulated such that the objects themselves determine a value. However, this also gives them the freedom not to satisfy constraints, which destroys the compelling and declarative nature of constraints. So indeed, under strict information hiding, constraint satisfaction is not guaranteed.

For the refinement model of satisfaction similar problems hold. One may model the domain as a separate object, but conceptually it is the exclusive property of the variable, and in fact a part of it. In that sense, changing the domain of a variable is equivalent to assigning a value.

If the constraint system is part of a programming language, the infringement of information hiding is under control of the constraint solver. From the application programmer's point of view the data encapsulation is still preserved. Indeed, as pointed out by [7], requiring the language's internal constraint system to respect information hiding is similar to requiring an optimizing compiler to respect information hiding, which would make part of its task impossible.

If one is to sacrifice strict information hiding in order to facilitate constraint satisfaction, care should be taken not to allow abuse. In [22] we propose a radical separation of the constraint system and the normal object-oriented framework by means of two orthogonal communication strategies for objects: messages on the one hand, and events and data-flow on the other hand. In this way, the process of constraint management via data-flow networks does not interfere with the communication of the object-oriented world via messages. Because of the global and compelling nature of constraints, this strict separation facilitates the design and debugging of constraints and the constraint system.

Acknowledgement

The authors are supported by NWO (Dutch Organization for Scientific Research) under Grants SION-612-31-001 and SION-612-322-212, respectively.

References

[1] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 1978.

[2] Grady Booch. *Object-Oriented Analysis and Design – with applications.* The Benjamin/Cummings Publishing, 1994.

[3] C. Choppy, S. Kaplan, and M. Soria. Complexity analysis of term-rewriting systems. *Theoretical Computer Science*, 67(2/3):261 – 282, 1989.

[4] Eric Cournarie and Michel Beaudouin-Lafon. Alien: a prototype-based constraint system. In Laffra et al. [11], pages 92–110.

[5] Jacques Davy. Go, a graphical and interactive C++ toolkit for application data presentation and editing. In *Proceedings 5th Annual Technical Conference on the X Window System*, 1991.

[6] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *AI*, 34:1–38, 1988.

[7] Bjorn N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92*, LNCS 615, pages 268–286. Springer-Verlag, 1992.

[8] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, January 1982.

[9] Hans-Werner Güsgen and Joachim Hertzberg. Some fundamental properties of local constraint propagation. *AI*, 36:237–247, 1988.

[10] Quinton Hoole and Edwin Blake. OOCS - constraints in an object oriented environment. In [24], pages 215–230, 1994.

[11] C. Laffra, E. H. Blake, V. de Mey, and X. Pintado, editors. *Object Oriented Programming for Graphics*, Focus on Computer Graphics. Springer, 1995.

[12] Chris Laffra and Jan van den Bos. Propagators and concurrent constraints. *OOPS Messenger*, 2(2):68–72, April 1991.

[13] Wm. Leler. *Constraint Programming Languages.* Addison-Wesley, 1988.

[14] Richard J. Lipton. Dna solution of hard computational problems. *Science*, 268:542–545, April 1995.

[15] A. K. Mackworth. Consistency in networks of relations. *AI*, 8:99–118, 1977.

[16] A. K. Mackworth and E. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *AI*, 25:65–74, 1985.

[17] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *AI*, 28:225–233, 1986.

[18] John R. Rankin. A graphics object oriented constraint solver. In Laffra et al. [11], pages 71–91.

[19] Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces.* PhD thesis, University of Washington, Seattle, Washington, 1994.

[20] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference, Detroit, Michigan, May 21-23 1963*, pages 329–345. AFIPS Press, 1963.

[21] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[22] Remco C. Veltkamp and Edwin H. Blake. Event-based.constraints: coordinate.satisfaction -> object.solution. In [24], pages 251–262, 1994.

[23] Michael Wilk. Equate: an object-oriented constraint solver. In *Proceedings OOPSLA'91*, pages 286–298, 1991.

[24] P. Wisskirchen, editor. *Proceedings 4th Eurographics Workshop on Object-Oriented Graphics, Sintra, Portugal, May 1994.*