# A non-rigid cluster rewriting approach to solve systems of 3D geometric constraints

Hilderick A. van der Meiden and Willem F. Bronsvoort

Faculty of Electrical Eng., Mathematics and Computer Science,
Delft University of Technology,
Mekelweg 4, 2628 CD Delft, The Netherlands
H.A.vanderMeiden@tudelft.nl / W.F.Bronsvoort@tudelft.nl

August 27, 2009

**Abstract**

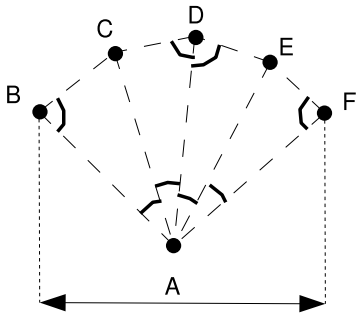
We present a new constructive solving approach for systems of 3D geometric constraints. The solver is based on the cluster rewriting approach, which can efficiently solve large systems of constraints on points, and incrementally handle changes to a system, but can so far solve only a limited class of problems. The new solving approach extends the class of problems that can be solved, while retaining the advantages of the cluster rewriting approach. Whereas previous cluster rewriting solvers only determined rigid clusters, we also determine two types of non-rigid clusters, i.e. clusters with particular degrees of freedom. This allows us to solve many additional problems that cannot be decomposed into rigid clusters, without resorting to expensive algebraic solving methods. In addition to the basic ideas of the approach, an incremental solving algorithm, two methods for solution selection, and a method for mapping constraints on 3D primitives to constraints on points are presented.

keywords: geometric constraint solving, cluster rewriting, solution selection

## 1 Introduction

Geometric constraints are used in current CAD systems to specify dimensions in 2D sketches and to position parts in 3D assemblies. To create and position geometry such that a system of constraints is satisfied, the system is solved by a geometric constraint solver, which is part of the CAD system.

The most successful geometric constraint solvers are so-called constructive solvers, which determine a decomposition of a problem into generically rigid subproblems, known as clusters. These clusters are solved independently, and




(a) graphical representation

$Angle(A, B, C)$ $Angle(A, D, C)$
$Angle(A, D, E)$ $Angle(A, F, E)$
$Angle(B, A, C)$ $Angle(C, A, D)$
$Angle(D, A, E)$ $Angle(E, A, F)$
$Distance(B, F)$

(b) textual representation

Figure 1: System of constraints on points $[A, \ldots, F]$. An angle constraint is represented graphically by an arc between dashed lines. A distance constraint is represented graphically by a two-sided arrow.

the solutions of the clusters are used to construct a solution for the complete problem. The advantages of using constructive solvers in CAD, and requirements for such solvers, are discussed in [1].

The simplest constructive solving approach is the cluster rewriting approach, also referred to as the bottom-up approach. In this approach, patterns of geometric constraints that are known to be generically rigid, are recognised as clusters. Certain patterns of clusters are also recognised, and merged into larger clusters. If the problem is well-constrained, a single cluster will remain at the end of the rewriting process.

To be able to efficiently determine solutions for problems with incremental changes, the cluster rewriting approach is very attractive. It is fast and an incremental algorithm can be easily implemented. However, the cluster rewriting approach can only be used to solve a relatively small class of problems, in particular, problems that can be decomposed into fairly small rigid subproblems.

Consider, for example, the 2D constraint problem in Figure 1. Here, we have a number of points (the variables), constrained by several angle constraints, and one distance constraint. The whole constraint system is rigid, but there is no subset of variables and constraints that forms a rigid system. A solver based on the cluster rewriting approach will not be able to solve the system, because it cannot find any rigid clusters, unless the solver has been explicitly programmed to recognise this particular system as a cluster.

This problem has been recognised in constraint solving literature, and various solutions have been proposed. Along with more general background, these are discussed in Section 2.

Previous solutions are mostly 2D and not compatible with the cluster rewriting approach. Therefore, we present a new 3D solving algorithm that identifies not only *rigid clusters*, but also two types of *non-rigid clusters*, in particular *scalable clusters* and *radial clusters*. The three types of clusters we distinguish

are defined in Section 3.

Our solving approach, based on cluster rewriting, is presented in Section 4. The basic idea is to exhaustively apply a small set of rewrite rules to a system of rigid and non-rigid clusters. The set of clusters remaining when no more rewrite rules can be applied, represents the generic solution of the system. From the generic solution, we can compute particular solutions, and determine whether the system is well-constrained, underconstrained or overconstrained.

It is relatively easy to update the solution(s) of a system when changes are made to it, i.e. when values of constraint parameters are changed, or when constraints are added to or removed from the system. An efficient incremental algorithm is presented in Section 5.

In general, a well-constrained problem can have a large number of solutions. Typically, in CAD applications, only one or a few specific solutions are needed. In Section 6, we present two methods for solution selection that can be used in the presented solving algorithm: declarative solution selection and prototype-based solution selection.

The cluster rewriting algorithm can only solve systems of constraints on points, whereas in many applications constraints are imposed on 3D primitives, e.g. planes, spheres and cylinders. Our approach to solve a system of geometric constraints on 3D primitives is discussed in Section 7. Basically, constraints on such primitives are mapped to a system of distance and angle constraints on point variables. This system is then mapped to a system of rigid and non-rigid clusters, which is solved. The solutions of the system of clusters are used to construct the solutions for the original problem involving 3D primitives.

Some conclusions are given in Section 8.

## 2 Background and related work

In principle, geometric constraint problems can be considered as algebraic problems. However, generic algebraic solving methods are either too expensive for use in interactive systems, or they cannot find all solutions for a given problem, resulting in ambiguity and robustness problems in practice. Symbolic algebraic methods, e.g. methods based on characteristic sets, such as Wu's method [2], have exponential running times in relation to the number of constraints. Numerical methods, such as Newton-Raphson iteration, although fast, cannot find all solutions to a given problem, and may not even find a solution without a suitable starting configuration. Homotopic continuation techniques have been used to find all solutions for small problems, e.g. the octahedral problem [3]. However, for larger problems, this technique is also too expensive, because the number of homotopy paths grows exponentially.

Constructive solvers determine a decomposition of a problem into generically rigid subproblems, known as clusters. For 2D problems, generic rigidity is characterised by Laman's theorem [4]. This theorem formulates generic rigidity for graphs, where edges in the graph correspond to distance constraints and vertices correspond to point variables, as follows:

> Let a graph $G$ have exactly $2n-3$ edges, where $n$ is the number of vertices in $G$. Then $G$ is generically rigid in $\mathbb{R}^2$ if and only if $e' \leq 2n'-3$ for every subgraph of $G$ with $n'$ vertices and $e'$ edges.

In other words, Laman's rule states that a 2D constraint problem with $n$ points and $e$ distance constraints is generically rigid, or well-constrained, if and only if $e = 2n-3$, and for every subproblem with $n'$ points and $e'$ distance constraints, $e' \leq 2n'-3$. If the number of constraints is smaller than specified by the rule, the system is called underconstrained. If, for any subproblem, the number of constraints is larger, the system is overconstrained. Note that a well-constrained system is not necessarily consistent, i.e. depending on the actual parameter values of the constraints there may or may not be any solutions. Also, overconstrained systems are not necessarily inconsistent; for some values of the constraint parameters, the system may have solutions, and is called consistently overconstrained.

Most 2D solvers use a bottom-up cluster rewriting approach with optimised data structures for representing systems of clusters, e.g. the graph-constructive approach used in [5, 6, 7]. For some of these algorithms a proof of correctness has been given, e.g. [8, 9], showing that if a system is reduced to a single cluster, then this cluster represents a correct solution for the original system.

The cluster rewriting approach is not complete, because no set of rewrite rules is known that will reduce all well-constrained systems, even if they consist only of distance constraints, to a single rigid cluster. In practice this means that some well-constrained systems may not be solved, and may be classified incorrectly as underconstrained.

As an alternative, using Laman's theorem, a complete top-down decomposition algorithm can be devised for 2D problems, e.g. [1, 10]. However, such a complete algorithm is expensive, and cannot deal with consistently overconstrained situations.

Solving geometric constraints in 3D is significantly more difficult than in 2D. Some 3D solvers based on the cluster rewriting approach have been presented [7, 3]. However, in 3D, the incompleteness issues of this approach are even more severe than in 2D. Until recently, there was no 3D equivalent to Laman's theorem, but a characterisation of generic well-constrainedness for 3D systems of distance constraints has now been found [11]. This result may lead to the development of a complete top-down algorithm for solving systems of 3D geometric constraints. However, at the time of writing, no solving algorithm based on this result is known.

Most 3D solvers, e.g. [12, 13, 14], create top-down decompositions using a technique called degrees-of-freedom analysis. In general, the term 'degrees of freedom' (DOF) refers to the number of parameters of a system that can be varied independently, e.g. a rigid body in 3D space has 6 DOF, corresponding to 3 translational and 3 rotational parameters. In the DOF-based approach, each constraint corresponds to a DOF reduction, and heuristic rules determine the DOF reduction of combinations of constraints. In this way, the generic rigidity of a problem can be determined, as well as a minimal set of rigid subprob-

lems. In practice, DOF-based rules correctly determine well-constrainedness for many systems of constraints. However, the DOF-based approach is also not complete, since current DOF-analysis techniques are based on heuristics for well-constrainedness in 3D. In particular, DOF-based algorithms sometimes incorrectly classify overconstrained systems as well-constrained.

The DOF-based approach can solve a larger class of problems than the cluster rewriting approach, but is generally more expensive, and no incremental algorithm is known.

When a problem cannot be decomposed into rigid clusters, it cannot be solved efficiently with any of the previous approaches. Using the cluster rewriting approach, the system will be considered underconstrained, and no solution will be found. Using a DOF-based approach, the system may be found to be well-constrained, but the system will be considered as a single cluster, and must be solved using expensive symbolic algebraic methods.

Problems that cannot be decomposed into rigid clusters can sometimes be solved efficiently by propagating angle constraints. For example, in the problem of Figure 1, one can derive $\measuredangle CDE = \measuredangle CDA + \measuredangle ADE$. In this way, new constraints can be derived, and the system can be solved by local propagation. A matrix-based propagation algorithm for distance and angle constraints on points in 2D is presented in [15]. A drawback of this algorithm is that a large number of redundant distances and angles are derived, resulting in high space and time complexity.

In [16], a constraint solving approach is presented that considers clusters that are invariant under the direct similarity group, i.e. invariant under any transformation that is angle preserving. This allows to decompose systems that cannot be decomposed into the usual rigid clusters, which are invariant under the Euclidean group, i.e. invariant under rotations and translations. A solving algorithm is presented that can solve systems of angle constraints, distance constraints and distance-ratio constraints on points in 2D. It also supports propagation of angle constraints and distance-ratio constraints.

Our solving approach, unlike the previous, works in 3D, and we identify not one but two types of non-rigid clusters, in particular *scalable clusters* and *radial clusters*. Scalable clusters are clusters that are invariant under the direct similarity group, as discussed above. Radial clusters are new, and provide a more general mechanism for propagating angles than other approaches. Thus, it is now possible to do angle propagation in 3D, and to incorporate this in the efficient cluster rewriting approach. The different types of clusters are described in detail in the next section.

# 3 Clusters

A cluster basically represents a collection of distance and angle constraints on a set of points. We define three types of clusters: *rigid clusters*, *scalable clusters* and *radial clusters*. The type of a cluster determines which distances and angles are constrained by it. Also, a set of *configurations* is associated with a cluster,

each of which determines an alternative set of values for the distances and angles constrained by the cluster.
The distances $\delta(p, q)$ constrained by a cluster are defined as:

$$\delta(p, q) = \sqrt{(q - p) \cdot (q - p)}$$

and the angles $\measuredangle(p, q, r)$ as:

$$\measuredangle(p, q, r) = cos^{-1}(\frac{p - q}{\delta(p, q)} \cdot \frac{r - q}{\delta(r, q)})$$

where $p, q, r \in \mathbb{R}^2$ or $\mathbb{R}^3$ are points in the cluster.

By this definition, distances and angles are unsigned, i.e. $\delta(p, q) \geq 0$ and $0 \leq \measuredangle(p, q, r) \leq \pi$. Constraints with signed angle parameters can be mapped to clusters with particular configurations, as will be discussed later in this section.

A *configuration* is a set of assignments of coordinates to point variables. For a set of point variables $A = [p_1, \ldots, p_n]$, each point $p_i$ is assigned a vector $v_i$, and for this configuration we write: $c_A = \{p_1 = v_1, p_2 = v_2, \ldots, p_n = v_n\}$.

The actual values of the distances and angles constrained by a cluster are determined by the configurations associated with the cluster. When there are no configurations associated with a cluster, the cluster is unsatisfiable, i.e. there are no solutions for this cluster. If there are several configurations associated with a cluster, then these determine alternative values for the distances and angles, i.e. the distance and angle values determined by one of the configurations must be satisfied.

For a cluster with a given type and set of configurations, the distance and angle values can be determined as follows. Suppose, the type of the cluster specifies that the distance $\delta(p_1, p_2)$ is constrained, and associated with the cluster are two configurations, $c_1 = \{p_1 = (0, 0, 0), p_2 = (1, 1, 1)\}$ and $c_2 = \{p_1 = (2, 0, 0), p_2 = (1, 0, 0)\}$. Configuration $c_1$ specifies a constraint $\delta(p_1, p_2) = \sqrt{3}$. Alternatively, configuration $c_2$ specifies a constraint $\delta(p_1, p_2) = 1$. When solving a system containing this cluster, one of these constraints must be satisfied.

The system of distance and angle constraints represented by a cluster, when considered as independent constraints, can be overconstrained. However, the values of these distance and angle constraints are determined by a configuration, and therefore these constraints are in fact not independent. Because a configuration assigns a point in $\mathbb{R}^3$ to each variable in the cluster, the system of constraints is always consistent (see Section 4). Consequently, a cluster with one or more configurations can be used to represent a constraint, and, at the same time, the solutions of a system of constraints.

A *rigid cluster* is a constraint on a set of points $[p_1, \ldots, p_n]$ such that the relative position of all points is constrained, i.e. all distances and angles in the set of points are constrained (see Figure 2(a)). This type of cluster has no internal DOF. Note that a number of solutions may exist for the system of distance constraints (each represented by a configuration associated with the cluster), but these are not considered DOF. The notation for a rigid cluster on a set of points $[p_1, \ldots, p_n]$ is: $Rigid([p_1, \ldots, p_n])$.

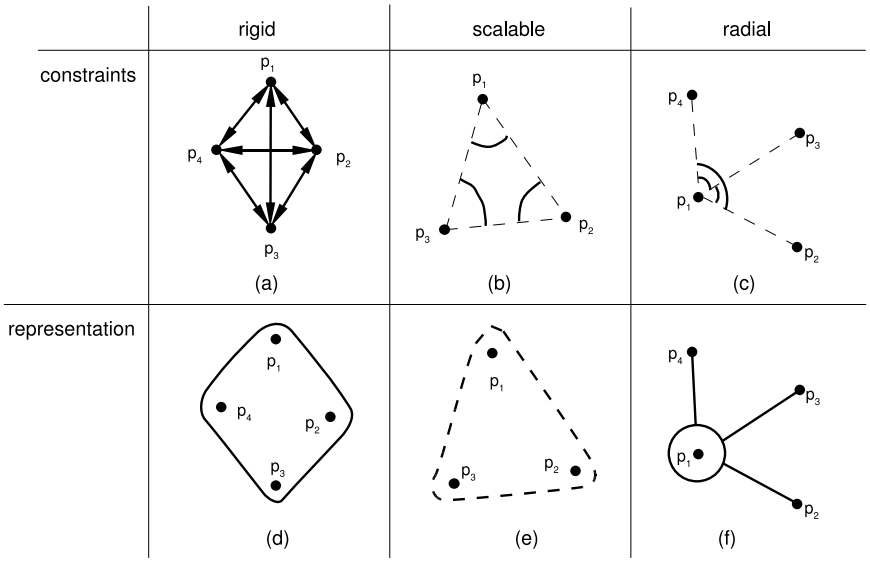


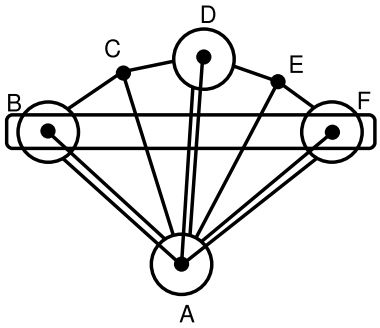

Figure 2: Constraints imposed by different cluster types (a,b,c) and their graphical representation (d,e,f).

A *scalable cluster* is a constraint on a set of points $[p_1, \ldots, p_n]$ such that for all $i, j, k \in [1, n]$ the angles $\measuredangle(p_i, p_j, p_k)$ are constrained (see Figure 2(b)). The constraint has one internal DOF, namely it may be scaled uniformly. The notation for a scalable cluster on this set of points is: $Scalable([p_1, \ldots, p_n])$.

A *radial cluster* is a constraint on a set of points $[p_c, p_1, \ldots, p_n]$ such that for all $i, j \in [1, n]$ the angles $\measuredangle(p_i, p_c, p_j)$ are constrained (see Figure 2(c)). Point $p_c$ is called the centre point and points $p_1, \ldots, p_n$ are called the radial points. This constraint has $n$ internal DOF (each point $p_1, \ldots, p_n$ can move along a line from the centre point). The notation for a radial cluster on these points is: $Radial(p_c, [p_1, ..., p_n])$.

We use a graphical notation for clusters, as shown in Figures 2(d)-(f). A point variable is represented by a dot with the name of the corresponding variable next to it. A rigid cluster is represented by a solid curve enclosing the set of points constrained by the cluster. A scalable cluster is represented by a dashed curve enclosing the set of points constrained by the cluster. Finally, a radial cluster is represented by a circle around the centre point and lines connecting the circle to the radial points.

Scalable clusters allow to solver to propagate angles that can be derived in triangles and tetrahedra, i.e. $\measuredangle(ABC) = \pi - \measuredangle(BCA) - \measuredangle(CAB)$. Radial clusters allow to propagate angles that can be derived by adding up concentric angles, i.e. $\measuredangle(AMC) = \measuredangle(AMB) + \measuredangle(BMC)$. Furthermore, angle constraints can now be represented by radial clusters, just like distance constraints are often represented by rigid clusters in other constraint solving approaches. This results




(a) graphical representation

$Radial(A, [B, C, D, E, F])$
$Radial(B, [A, C])$
$Radial(D, [C, A, E])$
$Radial(F, [A, E])$
$Rigid([B, F])$

(b) textual representation

Figure 3: The system of clusters corresponding to the problem in Figure 1.

in a clean representation of constraint data, and a simple solving algorithm.

Distance and angle constraints on points are easily mapped to clusters and configurations, as follows.

A distance constraint between two points is equivalent to a rigid cluster of two points with one associated configuration. For example, a distance constraint $\delta(p_1, p_2) = 1$ can be represented by a cluster $Rigid(p_1, p_2)$ and a configuration $\{p_1 = (0, 0), p_2 = (1, 0)\}$. Obviously, the choice of this particular configuration is somewhat arbitrary: infinitely many different configurations can be used to set the distance value.

An unsigned angle constraint on three points is equivalent to a radial cluster with one centre point and two radial points, and one associated configuration. For example, the angle constraint $\measuredangle(p_1, p_2, p_3) = \frac{1}{2}\pi$ can be represented by a cluster $Radial(p_2, [p_1, p_3])$ and a configuration $\{p_1 = (1, 0), p_2 = (0, 0), p_3 = (0, 1)\}$.

In 2D, a signed angle $\measuredangle(p_1, p_2, p_3)$ is uniquely defined as the angle of rotation to transform the normalised vector $p_1 - p_2$ to the normalised vector $p_3 - p_2$. A signed angle can be used in the construction of the configuration, resulting in a signed angle constraint.

In 3D, an angle $\measuredangle(p_1, p_2, p_3)$ is always unsigned. To define a signed angle constraint in 3D, an additional reference point $p_4$ is needed. Thus, a signed angle constraint in 3D is mapped to a 4 point radial cluster, $Radial(p_2, [p_1, p_3, p_4])$, and a configuration of those points. The vector $p_4 - p_2$ in the configuration is either the left-handed or right-handed cross product $(p_1 - p_2) \times (p_3 - p_2)$. The handedness of the cross product is not changed when the constraint parameter is changed and the configuration is re-determined. Therefore, the orientation of the points in the configuration can be used to select the appropriate solution, if needed (see Section 6).

A system of distance and angle constraints on points can thus be mapped to a system of clusters. Figure 3 shows the system of clusters corresponding to the

problem in Figure 1. Note that some angle constraints (e.g. $\measuredangle ADC$ and $\measuredangle ADE$) that are initially mapped to overlapping radial clusters (i.e. $Radial(D, [A, C])$ and $Radial(D, [A, E])$, respectively) have been merged (into $Radial(D, [C, A, E])$). Merging these clusters is actually the first step of the solving algorithm, which is explained in the next section using this example problem. However, in the visual representation such overlapping radial clusters cannot be easily distinguished. To make the example easier to follow, this merging step has already been performed here, and the result is taken as a starting point in the next section.

# 4 Solving approach

To solve a system of clusters, we basically try to rewrite the system to a single rigid cluster, by exhaustively trying to apply a set of rewrite rules. A *rewrite rule* specifies a *pattern*, describing its *input clusters*, and its *output cluster* in a generic way, and it specifies a *procedure* to determine the configurations of the output cluster from the configurations of the input clusters. A rewrite rule can be applied if a set of clusters is found in the system that matches the input clusters in the pattern. The corresponding output cluster is added to the system, and the configurations of the output cluster are determined by the procedure.

A pattern specifies a number of input clusters of a given type and a number of pattern variables. These pattern variables are matched by the solving algorithm to the point variables of the clusters in the system, such that the number of variables and the type of the cluster match. A pattern may also specify that an input cluster can match any cluster with a superset of the given variables. If a variable name occurs several times in the pattern, it must be matched with a single point variable that is constrained by several clusters in the constraint system.

To determine the configurations of the output cluster of a rewrite rule, the procedural part of the rule is applied for every combination of input cluster configurations. Suppose, for example, that two clusters are used as the input of a rewrite rule, and that each cluster has two configurations associated with it, then four different configurations for the output cluster are computed.

A set of rewrite rules for 2D and 3D problems is given in Appendix A. Of the 14 rules in total, 3 rules are specific for 2D problems, 6 rules are specific for 3D problems, and 5 more can be used in both 2D and 3D. In general, each rule determines an output cluster with new constraints that are not in the input clusters, either because the new cluster involves more points, or because the new cluster is of a type with fewer DOF. We have built this rule set starting with rules for every combination of two clusters that results in new constraints. Then, rules were added for combinations of three clusters that cannot be solved by using the simpler rules. In this way, rewrite rules for even more complex subproblems might also be found.

In the remainder of this section, we show how these rewrite rules are used to solve the problem illustrated in Figure 3. Consider Rule 7 from Appendix A.

**Rule 7** Derive a scalable cluster from two radial clusters

Pattern: $Radial(p_1, [p_3, p_2, \ldots]) \cup Radial(p_2, [p_1, p_3, \ldots])$
$\rightarrow Scalable([p_1, p_2, p_3])$
Procedure: $c_1 \times c_2 \rightarrow c_R$
$c_R(p_1) = (0, 0, 0)$
$c_R(p_2) = (1, 0, 0)$
$c_R(p_3) =$ intersection
ray from $c_R(p_1)$ direction $\measuredangle(c_1(p_3), c_1(p_1), c_1(p_2))$
ray from $c_R(p_2)$ direction $\measuredangle(c_2(p_1), c_2(p_2), c_2(p_3))$

This rule can be applied (in 2D or 3D) when two radial clusters share three points, including the centre point of each cluster. When a match is found, a new scalable cluster ($R$) is added to the system, and a configuration ($c_R$) is computed by intersecting two rays (directed half-lines).

This rule can be applied to the problem in Figure 3, as follows. We find the following matches:

$Radial(A, [B, C, D, E, F]) \cup Radial(B, [A, C]) \rightarrow Scalable([A, B, C])$
$Radial(A, [B, C, D, E, F]) \cup Radial(D, [C, A, E]) \rightarrow Scalable([A, C, D])$
$Radial(A, [B, C, D, E, F]) \cup Radial(D, [C, A, E]) \rightarrow Scalable([A, D, E])$
$Radial(A, [B, C, D, E, F]) \cup Radial(F, [A, E]) \rightarrow Scalable([A, E, F])$

By applying the rewrite rule to the first match, the system shown in Figure 4(a) is obtained. Repeated application of the rule for all the matches listed above, results in the system shown in Figure 4(b).

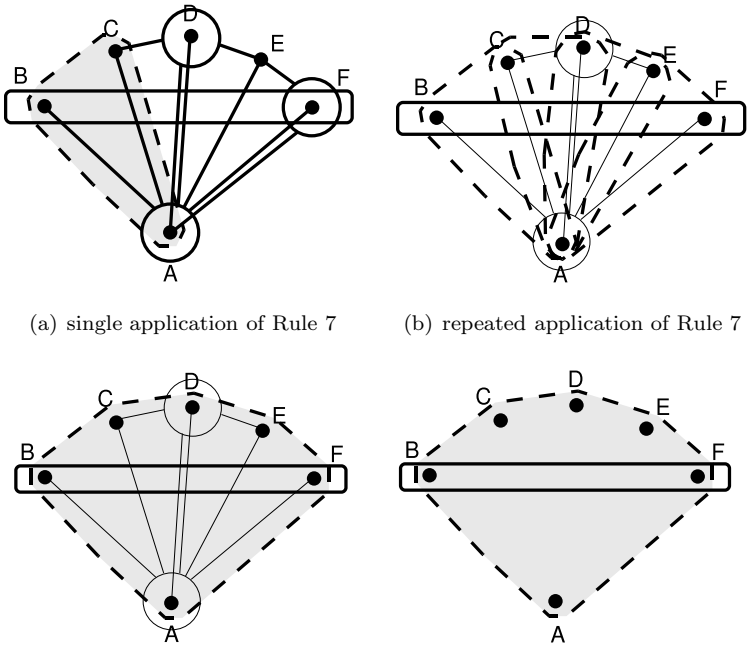
When a rewrite rule is applied, the input clusters may become *redundant* and should be removed from the system. A cluster is redundant if all distances and angles constrained by the cluster are also constrained by newer clusters. Thus, an input cluster is removed from the system if all the distances and angles in the cluster are also in the output cluster.

In the system in Figure 4(a), the cluster $Radial(B, [A, C])$ is redundant and removed, because the angle $\measuredangle ABC$ constrained by this cluster, is also constrained by the newer cluster $Scalable([A, B, C])$. The cluster $Radial(A, [B, C, D, E, F])$, however, is not removed, even after repeated application of the rewrite rule (result shown in Figure 4(b)), because it constrains angles that are not in any of the scalable clusters (e.g. $\measuredangle BAF$).

If a rewrite rule is defined such that its output cluster contains all distances and angles that are in its input clusters, then all input clusters are removed after the rule has been applied, and we say that the rewrite rule *merges* the input clusters. The scalable clusters in Figure 4(b) can be merged using Rule 3 from Appendix A.

**Rule 3** Merge two scalable clusters with two shared points

Pattern: $Scalable(A = [p_1, p_2, \ldots]) \cup Scalable(B = [p_1, p_2, \ldots])$
$\rightarrow Scalable(A \cup B)$
Procedure: $c_1 \times c_2 \rightarrow c_R$




(a) single application of Rule 7 (b) repeated application of Rule 7

(c) after applying Rule 3 (d) after removing redundant clusters

Figure 4: Intermediate results for solving the system in Figure 3.

T = rotation, translation and scaling such that $p_1$ and $p_2$ in $c_2$ are mapped onto $p_1$ and $p_2$ in $c_1$
$c_R = c_1 \cup T(c_2)$

This rule basically takes two scalable clusters, and combines their configurations by rigidly transforming one of them such that the shared points between the configurations coincide. New configurations obtained in this way are associated with a new scalable cluster. The rule can be applied repeatedly, in the example problem, as follows:

$Scalable([A, B, C]) \cup Scalable([A, C, D]) \rightarrow Scalable([A, B, C, D])$
$Scalable([A, D, E]) \cup Scalable([A, E, F]) \rightarrow Scalable([A, D, E, F])$
$Scalable([A, B, C, D]) \cup Scalable([A, D, E, F]) \rightarrow Scalable([A, B, C, D, E, F])$

Applying these rewrites results in the system shown in Figure 4(c). Now we can remove the clusters $Radial(A, [B, C, D, E, F])$ and $Radial(D, [C, A, E])$, because all angles in those clusters are also constrained by the newer cluster $Scalable([A, B, C, D, E, F])$, resulting in Figure 4(d). Finally the cluster $Scalable([A, B, C, D, E, F])$ can be merged with $Rigid([B, F])$, using the following rule from Appendix A.

**Rule 8** Derive a rigid cluster from a scalable and a rigid cluster with two shared points

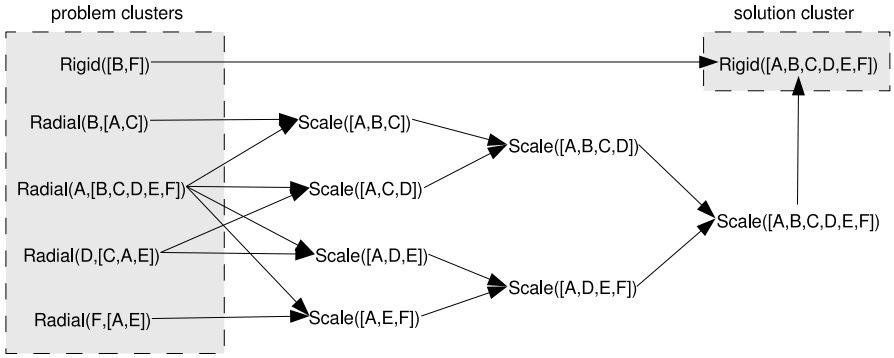



Figure 5: Generic solution for the problem in Figure 3. Note that $Scale(\ldots)$ represents a scalable cluster.

Pattern: $Scalable(A = [p_1, p_2, \ldots]) \cup Rigid([p_1, p_2, \ldots]) \rightarrow Rigid(A)$
Procedure: $c_1 \times c_2 \rightarrow c_R$
T = scale configuration by $\frac{\delta(c_2(p_2),c_2(p_1))}{\delta(c_1(p_2),c_1(p_1))}$
$c_R = T(c_1)$

This rule basically scales the configuration of the input cluster, such that the distance between the shared points becomes equal to the distance specified by the configuration of the rigid input cluster. New configurations obtained in this way are associated with a new rigid cluster.

The rule can be applied to the example system, resulting in a cluster $Rigid([A, B, C, D, E, F])$, which constrains all the variables of the problem. No other rewrite rules can be (nor need to be) applied.

The *generic solution* of a problem is a recipe for computing all configurations that satisfy the constraints, and consists of a set of clusters related by rewrite rules. The generic solution of the problem of Figure 3 is shown in Figure 5. In this figure, arrows indicate dependencies between clusters created by rewrite rules (the rules are not explicitly represented). The clusters in the generic solution can be classified as *problem clusters*, i.e. clusters specified in the original problem, intermediate clusters, and *solution clusters*, i.e. clusters that are not used as input for any rewrite rule. In Figure 5, the clusters with no incoming arrows are problem clusters, and the cluster with no outgoing arrows is the solution cluster.

The configurations associated with the solution cluster are the *particular solutions* of the problem. If there are no configurations associated with the solution cluster, then the problem is *inconsistent*. If there are one or more configurations, then the problem is *consistent*.

From the generic solution we can also determine whether a problem is structurally underconstrained, overconstrained or well-constrained. These classifications are usually defined in terms of the number of solutions, which may be infinite, zero, or finite, respectively. However, such definitions are problematic

here, because, for certain valuations of the parameters of the problem, the problem may degenerate and should be classified differently. We therefore use the following operational definitions:

- A problem is *underconstrained* if its generic solution has more than one solution cluster or a single non-rigid solution cluster.
- A problem is *overconstrained* if any distance or angle constraint has more than one *source cluster*. This is elaborated below.
- A problem is *well-constrained* if it is not underconstrained and not overconstrained. Note that these conditions are not mutually exclusive.

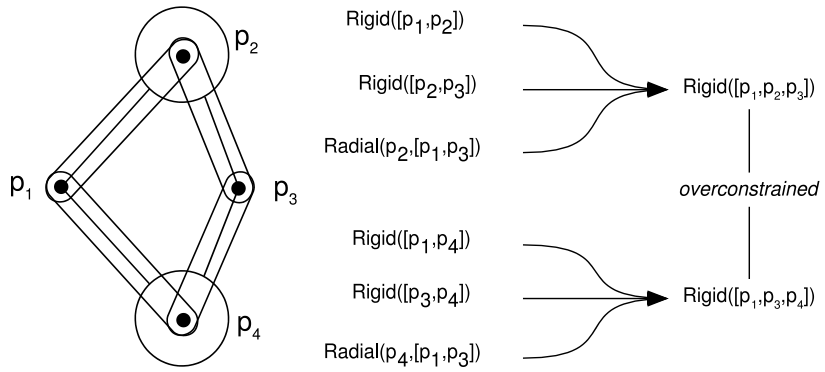
These definitions depend on our particular solving algorithm, which is not complete, and therefore they are not equivalent to other, more formal definitions found in literature. However, our version of well-constrainedness is easily decidable and of practical value when using our solver.
If any distance or angle is constrained in two or more clusters in the generic solution, then the system may be overconstrained, depending on which clusters these distances and angles occur in.

In particular, if two or more problem clusters, i.e. clusters determined from constraints specified by the user, constrain the same distance or angle, then those distances and angles are generally not consistent, and thus the problem is overconstrained. However, if some distance or angle is constrained in several clusters that are not all problem clusters, then the problem is not necessarily overconstrained. The values of this distance or angle in different clusters, are derived by rewrite rules from the original problem clusters, and these values may in fact be the same in all clusters.

To determine whether a problem is overconstrained, we use the following procedure. When adding new clusters to the generic solution, we determine, for each distance or angle in that cluster, its *source clusters*, i.e. the first clusters in the generic solution that constrain that distance or angle. The source cluster of a distance or angle in some cluster can be found by following dependencies in the generic solution in the reverse direction, checking for each cluster encountered whether it constrains the distance or angle. A cluster that constrains a given distance or angle is a source cluster, if no other cluster on which the cluster depends already constrains that distance or angle.

If there is exactly one source for each distance or angle, then the system is not overconstrained, because each rewrite rule ensures that all distance/angle constraints in its input clusters are also satisfied in its output clusters. Otherwise, if there is more than one source for a distance or angle, then there is no guarantee that it will have the same value in the different source clusters, and therefore the system is overconstrained.

During the cluster rewriting process, sets of clusters may be created by the solver that constrain the same distances and angles, but do not result in an overconstrained system. Consider, for example, the system in Figure 4(c). Here, all the angles in the clusters $Radial(A, [B, C, D, E, F])$ and $Radial(D, [C, A, E])$




(a) problem (b) generic solution

Figure 6: An overconstrained 2D problem and its generic solution.

are also constrained by the cluster $Scalable([A, B, C, D, E, F])$. However, from the generic solution in Figure 5, we can infer that for each angle, there is only one source, in these cases a single problem cluster. Thus, the system is not overconstrained.

In Figure 6(a), an overconstrained problem is shown. The generic solution for this problem is shown in Figure 6(b). Here, the clusters $Rigid([p_1, p_2, p_3])$ and $Rigid([p_1, p_3, p_4])$ both constrain the distance $\delta(p_1, p_3)$. So, this distance is determined twice, by different rewrite rule applications, using different input clusters that do not constrain this distance. Thus, there are two source clusters for the distance: $Rigid([p_1, p_2, p_3])$ and $Rigid([p_1, p_3, p_4])$, and therefore the problem is overconstrained.

# 5 Incremental algorithm

The solving algorithm incrementally updates the generic solution of a constraint problem whenever changes are made to the problem, i.e. clusters are added or removed. The generic solution (symbol `G` in Algorithms 1, 2 and 3) is represented by a bi-partite graph, in which nodes are clusters or rewrite rules. Directed edges connect clusters and rewrite rules. If a cluster is an input cluster of a rewrite rule, then there is an edge from the cluster to the rewrite rule. If a cluster is the output cluster of a rewrite rule, then there is an edge from the rewrite rule to the cluster.

The solving algorithm also keeps track of the set of *active clusters*, i.e. the clusters that represent the problem after all rewriting steps so far (symbol `A` in Algorithms 1, 2 and 3).

The generic solution and the active set are initially empty. When the user adds a cluster to the problem (method `AddCluster`, see Algorithm 1), the cluster is added to the generic solution and to the active set. Because it is not the output of a rewrite rule, the cluster can be identified in the generic solution as a

Algorithm 1 Adding a cluster

```
method AddCluster (G,A,c)
  G: generic solution
  A: active set
  c: cluster
begin
  G.add(c)
  A.add(c)
  SearchRewrites(G,A,c)
end
```

Algorithm 2 Removing a cluster

```
method RemoveCluster (G,A,c)
  G: generic solution
  A: active set
  c: cluster
begin
  G.remove(c)     (* also removes rewrite rules on c *)
  A.remove(c)
  for each x in DependentClusters(G,c)
    RemoveCluster(x)
  for each y in DeactivatedClusters(A,c)
    A.add(y)
    SearchRewrites(G,A,y)
end
```

problem cluster. The algorithm then searches for possible rewrite applications on that new cluster, i.e. rewrite rule applications where the cluster is used as input (method `SearchRewrites`, see Algorithm 3).

When a cluster is removed (method `RemoveCluster`, see Algorithm 2), it is removed from the generic solution and the active set. All dependent clusters are also removed from the generic solution. The dependent clusters (function `DependentClusters`) are all clusters in the generic solution that are directly or indirectly determined by rewrite rules that use the given cluster as input. The algorithm must then determine a new set of active clusters. For this purpose, it determines which clusters were removed from the active set when this particular cluster was added (function `DeactivatedClusters`). The active set is restored by re-adding those clusters to the active set. It is possible that after restoring the active set, combinations of clusters can be rewritten (method `SearchRewrites`).

Searching for possible rewrite rule applications (method `SearchRewrites`, see Algorithm 3) can be done efficiently because we search only for rewrites on newly added clusters. Since each rewrite rule involves a small number of overlapping clusters (i.e. clusters sharing one or more point variables), we construct a subset of the set of active clusters consisting only of the newly added

Algorithm 3 Searching for rewrite rule applications

```
method SearchRewrites(G,A,c)
  G: generic solution
  A: active set
  c: cluster
begin
  subset := c + OverlappingClusters(A,c)
  reference := ReferenceGraph(subset)
  for each rule in AllRewriteRules
    pattern := PatternGraph[rule]
    matches = SubgraphIsomorphisms(pattern,reference)
    for each match in matches
      rewrite := instantiate rule from match
      if IsProgressive(rewrite) then
        G.add(rewrite)     (* also adds output cluster *)
        A.add(rewrite.output)
        for each i in rewrite.inputs
          if IsRedundant(i) then
            A.remove(i)
        SearchRewrites(rewrite.output)
end
```

cluster and the clusters that overlap with it (function `OverlappingClusters`), and search in that subset for possible rewrite rule applications. The pattern matching algorithm thus searches only through a small number of clusters and variables.

The pattern matching algorithm used in our implementation is basically a subgraph matching algorithm that finds all subgraph isomorphisms. The subset of the active set in which we look for rewrite rule applications is converted to a graph (function `ReferenceGraph`). The input pattern specified by a rewrite rule is also represented by a graph (`PatternGraph[rule]`). For each subgraph isomorphism returned by the graph matching algorithm (function `SubgraphIsomorphisms`), we determine which point variable is assigned to which pattern variable, and from that the actual rewrite rule can be instantiated.

For each possible rewrite rule application found, the algorithm first checks whether it is *progressive* (function `IsProgressive`), and only if it is, the algorithm adds it to the generic solution. A rewrite rule application is progressive if it either increases the number of distances and angles constrained by the active set, or reduces the number of active clusters. This ensures that the algorithm does not add redundant clusters to the system, except to remove overconstrained clusters.

Generally, when a rewrite rule is added to the generic solution, its output cluster becomes part of the set of active clusters, and one or more input clusters may be removed from the active set. A cluster is removed from the active set

| intersection | condition |
|---|---|
| $Rigid(A) \cap Rigid(B) = Rigid(A \cap B)$ | $\lvert A \cap B \rvert > 1$ |
| $Rigid(A) \cap Scalable(B) = Scalable(A \cap B)$ | $\lvert A \cap B \rvert > 2$ |
| $Rigid(A) \cap Radial(p_c, B) = Radial(p_c, A \cap B)$ | $p_c \in A, \lvert A \cap B \rvert > 2$ |
| $Scalable(A) \cap Scalable(B) = Scalable(A \cap B)$ | $\lvert A \cap B \rvert > 2$ |
| $Scalable(A) \cap Radial(p_c, B) = Radial(p_c, A \cap B)$ | $p_c \in A, \lvert A \cap B \rvert > 2$ |
| $Radial(p_c, A) \cap Radial(p_c, B) = Radial(p_c, A \cap B)$ | $\lvert A \cap B \rvert > 2$ |

Table 1: Pairwise cluster intersections. If none of the cases listed here matches, then the intersection is empty, i.e. the intersection contains no distances or angles.

| cluster | distances | angles |
|---|---|---|
| $Rigid([p_1, \ldots, p_n])$ | $\binom{n}{2}$ | $3\binom{n}{3}$ |
| $Scalable([p_1, \ldots, p_n])$ | 0 | $3\binom{n}{3}$ |
| $Radial(p_c, [p_1, \ldots, p_n])$ | 0 | $\binom{n}{2}$ |

Table 2: Number of distance and angles constrained by clusters.

if it is redundant, i.e. if all distances and angles constrained by it are already constrained by the other clusters in the active set.

To determine whether a cluster is redundant (function `IsRedundant`), the algorithm needs to determine whether the set of distances and angles constrained by the cluster is a subset of the set of distances and angles constrained by the other clusters in the active set. Determining these sets explicitly is too expensive. Instead, we determine the number of distances and angles constrained by the cluster, and the number of distances and angles constrained by each intersection of the cluster with any other overlapping cluster in the active set.

We define the intersection of two clusters as a cluster that constrains only those distances and angles that are constrained by both clusters. The intersection can be determined efficiently using the rules listed in Table 1. For example, given two clusters, $Rigid([p_1, p_2, p_3])$ and $Rigid([p_2, p_3, p_4])$, the intersection is determined by the first rule in the table as $Rigid([p_2, p_3])$. The table shows how the type of the intersection cluster is determined by the types of the original clusters. The set of point variables constrained by the intersection cluster is the set of point variables shared by the original clusters. The set of shared points must satisfy additional conditions to ensure that the intersection cluster is a valid cluster, e.g. a minimum number of shared points is needed and the centre point of two radial clusters must be the same.

The number of distances and angles constrained by a cluster can be determined from Table 2. Note that the number of distances and angles constrained by a cluster is larger than the number of constraints typically needed for a

well-constrained system. However, because the allowable combinations of values of the distances and angles constrained by a cluster, are determined by a configuration, these values are always consistent.

If the number of distances and angles constrained by a cluster is larger than the total number of distances and angles constrained by the intersections of the cluster with each other overlapping cluster in the active set, then the cluster is not redundant. Otherwise, the number of distances and angles in the cluster is equal to the total number of distances and angles in the intersections (it cannot be smaller), and the cluster is redundant.

After a new rewrite rule has been added to the generic solution, the algorithm is called recursively to find possible rewrites on the output cluster of the added rewrite rule. Note that, for compactness, a recursive algorithm is given here, but our implementation is iterative, using a stack of clusters to be processed.

The generic solution of a problem can be used to determine its particular solutions, by evaluating the procedural part of each rewrite rule in the generic solution, for each combination of its input clusters' configurations. This may also be done in an incremental way. When the set of configurations associated with a problem cluster is changed, the dependent rewrite rules can be determined from the generic solution, i.e. the rules which use this cluster as input cluster. Only these rewrite rules need to be re-evaluated.

The determining factor for the complexity of the algorithm is the subgraph matching algorithm (`SubgraphIsomorphisms`). In general, subgraph isomorphism is an NP-complete problem, but for many problem instances, heuristic algorithms can find matches more quickly, see e.g. [17]. Also, the matching is simplified a great deal by taking into account the types of clusters. Further, because no rewrite rule currently requires more than three input clusters, and because of the incremental nature of the algorithm, we only match against pairs and triples of clusters that contain the cluster last added to the active set. Finally, we are not interested in all possible permutations of pattern variables in rewrite rules, because variables in clusters are not ordered (with the exception of the first variable of radial clusters), thus we do not have to generate all possible matches. The actual cost of the algorithm is therefore much lower than the theoretical bound for general subgraph isomorphism.

The performance of the solver was tested on a set of randomly generated well-constrained 3D problems. The problems were created by first constructing simple well-constrained systems using distance constraints only. Then, angles were randomly introduced by replacing some distances in triangular configurations. Finally, we replaced combinations of angle constraints that formed scalable and radial clusters with randomly selected angle constraints from those clusters, resulting in problems with non-triangular subproblems. The generated set of problems seems to represent the class of problems that can be solved quite well, since all rewrite rules have been used in solving them.

Figure 7 shows the average time for completely solving a problem with a given number of variables, and several incremental solving times for randomly removing one constraint from, and adding another to, these problems. As can be seen, complete solving time increases approximately quadratically with the size

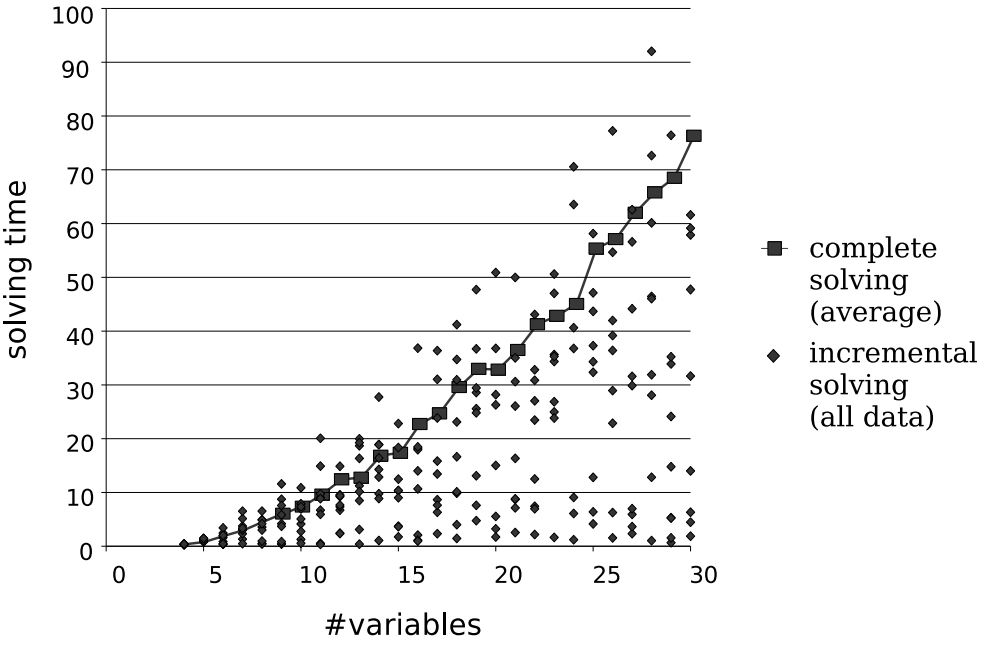



Figure 7: Plot of average complete solving time and all incremental solving times, against the number of problem variables.

of the problem. Performance for incrementally solving random changes varies wildly, but is typically lower than the average time for solving the complete system.

The solver was implemented in Python, using only standard libraries. The experiments were performed on a a 2.6 Ghz Pentium IV system. Significant speedups should be possible by using special libraries for graph matching and vector and matrix algebra.

In these experiments, the measured times are for determining the generic solution. The number of particular solutions of a problem may be exponential to the number of variables, and computing all solutions may be very expensive. For most applications, however, not all solutions are required or desirable. Therefore, a solution selection mechanism is needed to reduce the number of solutions, and thus reduce the computation time.

# 6 Solution selection

Solution selection is an important problem in geometric constraint solving; it is also known as the multiple-solution problem or the root identification problem. The number of solutions for geometric constraint problems is generally exponential to the number of geometric elements. In fact, the problem of finding all real solutions of an arbitrary system of geometric constraints has been shown to be NP-complete [6]. Therefore, various solution selection schemes have been proposed, e.g., interactive approaches [5, 18], genetic algorithms [19] and sketch-based methods [20]. Our solver supports two solution selection schemes, namely declarative solution selection [21] and prototype-based solution selection [22].

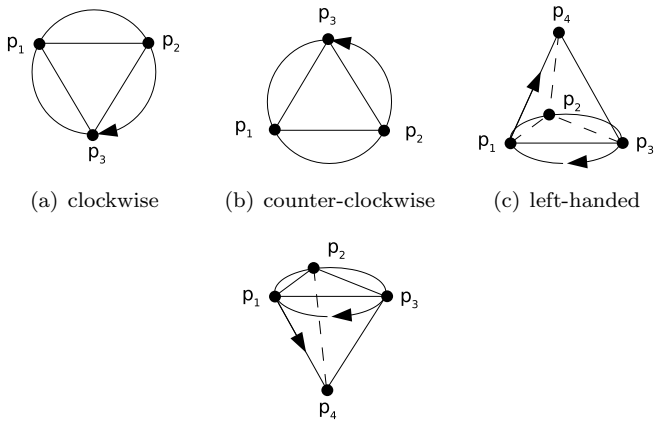
Declarative solution selection is supported in our solver by selection constraints. These specify the orientation of a set of points. In 2D, solutions can be selected based on whether a set of three points $\{p_1, p_2, p_3\}$ is oriented clockwise or counter-clockwise, as shown in Figure 8(a) and Figure 8(b), respectively. In 3D, the handedness of a set of points is a useful selection criterion. A set of points $\{p_1, p_2, p_3, p_4\}$ can be classified as left-handed or right-handed, as shown in Figure 8(c) and Figure 8(d), respectively.

In general, the orientation of a set of $n+1$ points in $\mathbb{R}^n$ can be determined by taking one point as a reference, and computing the determinant of the ordered set of offset vectors for the other points. We define the orientation of an ordered set of points $\{p_1, ..., p_{n+1}\}$ as:

$$Orientation(p_1, ..., p_{n+1}) = sign(Det[p_2 - p_1, ..., p_{n+1} - p_1])$$

The set of points is *positively oriented* if the determinant is positive, *negatively oriented* if the determinant is negative, and indeterminate if the determinant is zero. Note that the points may be represented as either the rows or the columns of the matrix of which the determinant is computed.

Obviously, solution selection should take place as soon as possible, i.e. as soon as enough information is available to evaluate the selection constraint. In




(a) clockwise (b) counter-clockwise (c) left-handed

(d) right-handed

Figure 8: Solution selection constraints.

our solver, selection constraints are evaluated as soon as a cluster has been determined that contains a superset of the point variables in the constraint.

There are two problems with the declarative approach to solution selection. The first problem is that, in the worst case, all solutions of a geometric constraint problem have to be generated, before selection constraints can be evaluated. The second problem is that a large number of selection constraints generally needs to be specified, in order to determine a single solution. In addition, this solution, as a function of the parameters of the problem, is often discontinuous. Discontinuous behaviour is undesirable for most parametrisations. For example, in [23], requirements are identified for freeform feature classes and their instances. For users of the feature class, who generally have no knowledge of the constraints used for the parametrisation, discontinuities in the behaviour of the feature are unexpected and undesirable.

An alternative method to declarative solution selection, which does not suffer from these problems, is prototype-based solution selection. A prototype is simply a configuration of all the geometric variables in the problem that 'looks like' the desired solutions. Typically, the prototype is obtained from a sketch created by the user. In [20] a formal framework is presented for sketch-based solution selection.

We use a prototype-based solution selection mechanism that always determines at most one solution, the so-called *intended solution*, described in [22]. The intended solution satisfies the following properties:

- the intended solution is a continuous function of the parameters of the problem
- the intended solution uniquely resembles a given prototype.

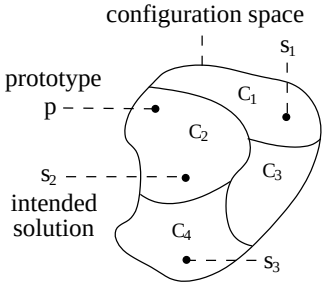



Figure 9: The resemblance relation partitions the configuration space into a number of equivalence classes $C_i$. The intended solution ($s_2$) is the solution that is in the same equivalence class ($C_2$) as the prototype ($p$).

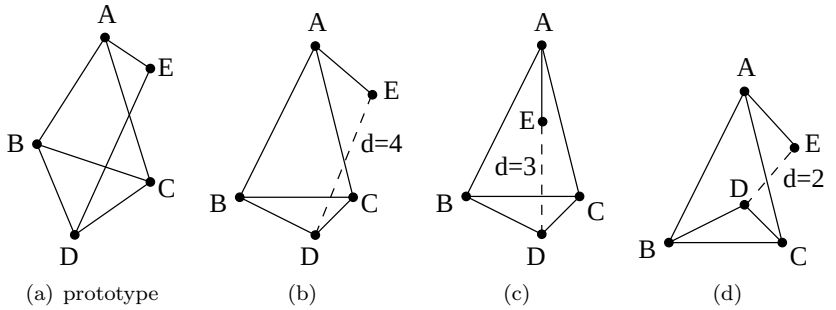
The first property ensures that there is a predictable and intuitive relation between the intended solution and the parameters of the problem. The second property ensures that there is an intuitive relation between the intended solution and the prototype. A unique resemblance between the intended solution and the prototype means that other solutions, found for the same parameter values, must not resemble the prototype, by some definition of resemblance. Thus, the intended solution is uniquely determined by the prototype.

Resemblance is defined by a resemblance relation. This relation is an equivalence relation, which partitions the configuration space into a number of equivalence classes. The intended solution is the solution in the same equivalence class as the prototype. This is illustrated in Figure 9.

For some combinations of the parameter and prototype, there may not be an intended solution, even though a real solution exists. Consider, for example, the 2D system in Figure 10. For any parameter value $d \geq 3$, the solution is a continuous function of $d$. If, however, we instantiate the problem with parameter $d = 2$, there is no solution in the same equivalence class, i.e. there is no solution for $d = 2$ that can be reached continuously from the previous solutions. Such behaviour is intuitive for most applications, but if it is not desirable, declarative solution selection can be used.

The intended solution can be found by using the cluster rewriting algorithm presented in the previous sections. Basically, for each subproblem that is solved, selection constraints are generated, such that a single solution is selected for each subproblem. The selection constraints that are generated for a specific subproblem depend on the type of the subproblem, i.e. the specific rewrite rule, and the prototype. Since at most one solution is found for each subproblem, and no back-tracking search is needed, computing the intended solution in this way is inexpensive.

For example, consider the 2D system in Figure 10 again. This problem can be decomposed into three simple triangular problems: $ABC$, $BCD$ and $ADE$. To solve each of these subproblems, the intersection of two circles is determined, as shown in Figure 11. The two solutions can be distinguished



(a) prototype (b) (c) (d)

Figure 10: A constraint problem with a prototype (a) and a distance parameter $d$. The intended solution exists only for $d \geq 3$ (b, c). For $d = 2$ there is a solution, but there is no intended solution (d).

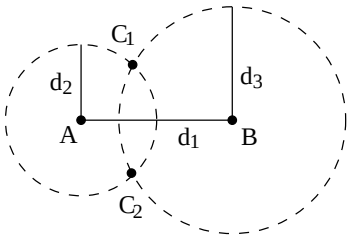



Figure 11: A simple triangular subproblem, where three distance constraints are given. The solution can be found by intersecting two circles.

by the orientation of the points in the solution; either the points are counter-clockwise oriented, i.e. $ABC_1$ or the points are clockwise oriented, i.e. $ABC_2$. If in the prototype the points $ABC$ are oriented clockwise, then a selection constraint $Clockwise(A, B, C)$ is added to the system. If, on the other hand, the points in the prototype are oriented counter-clockwise, then a selection constraint $CounterClockwise(A, B, C)$ is added. If the orientation of the prototype points is clockwise nor counter-clockwise, i.e. the prototype points are on a line, then we must make an arbitrary choice, or warn the user. For the other subproblems in the example, the same procedure is followed.

In general, for each type of subproblem, selection constraints can be generated that can distinguish between the possible solutions of the subproblem. The particular selection constraint that is satisfied by the prototype is added to the problem. For most 2D subproblems, the *Clockwise* and *Counterclockwise* selection constraints can be used, and for most 3D subproblems the $Lefthanded$ and *Righthanded* selection constraints. For some subproblems, inequality constraints on angles are used.

In [22], it is shown, for a somewhat simpler constraint solver, which only recognises triangular and tetrahedral rigid subproblems, that the intended solution can be found using selection constraints. Basically, we formally define

the properties that the resemblance relation must satisfy, in order to find the intended solution. For each type of subproblem, the selection constraints that are used define a resemblance relation. First we show, for several types of subproblems, that these resemblance relations satisfy the given properties. Then we show that by combining subproblems, a resemblance relation results for the whole problem, which also satisfies those properties. Thus, the solution found in this way is the intended solution. For the solver discussed in this paper, which can also find non-rigid clusters, we believe that a similar proof can be given.

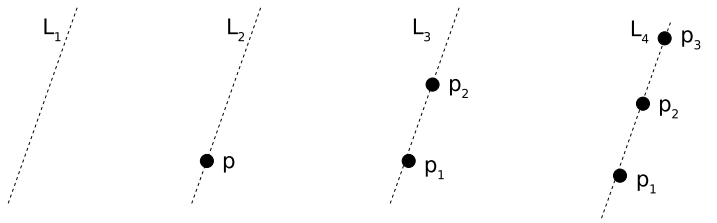
# 7 Constraints on 3D primitives

So far, we have only considered systems of distance and angle constraints on points, in particular, systems of clusters. In typical CAD models, however, constraints are imposed on other types of geometric primitives, e.g. lines, planes, spheres and cylinders.

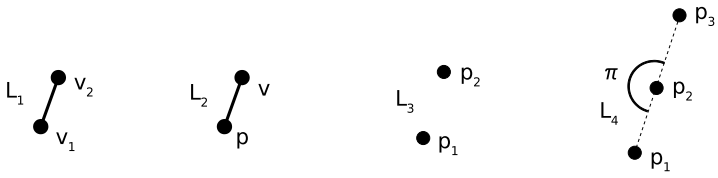
In this section, we present a mapping from constraint systems on primitives to a system of distance and angle constraints on points. Basically, a primitive is represented by a small number of points, e.g. a line can be represented by two points. Constraints on primitives can then be expressed by distance and angle constraints on points. A system of distance and angle constraints on points, in turn, can be represented by a system of clusters, which can be solved with the solving algorithm presented in the previous sections.

The DOF of a primitive cannot always be represented by a set of point variables. For example, a line in 3D has 4 DOF. It cannot be represented by a single point variable, which has 3 DOF, or by two point variables, which together have 6 DOF. However, we can represent the DOF of a *system* of primitives and constraints, at least in many cases where the system is well-constrained. In such cases, the solutions of a particular system of constraints on points can be used to construct the solutions of a particular system of constraints on primitives. The key idea is that the representation of a primitive depends on the number of points that are constrained to be coincident with it.

Figure 12 illustrates the mapping of lines with different numbers of coincident points. A line $L_1$, with no constraints imposed on it, is represented by two points $v_1$ and $v_2$, with an arbitrary distance constraint such that $\delta(v_1, v_2) \neq 0$. The represented line is the line through the points $v_1$ and $v_2$. By itself, this system is well-constrained. If a single point $p$ is constrained coincident with a line $L_2$, by a constraint $Coincident(p, L_2)$, then line $L_2$ is represented by that point $p$ and another point $v$. Again, this system is well-constrained. If two points are constrained coincident with a line $L_3$, i.e. $Coincident(p_1, L_3)$ and $Coincident(p_2, L_3)$, then both points are used in the representation, i.e. line $L_3$ is represented by $p_1$ and $p_2$. If these points are part of a well-constrained system, then the line is also well-constrained. Note that the distance between the points is not constrained by the mapping, because if the system is well-constrained, then the distance between the points is already determined. If more than two points are constrained with a line $L_4$, i.e. $Coincident(p_i, L_4)$ for




(a) lines with coincident points




(b) mapping

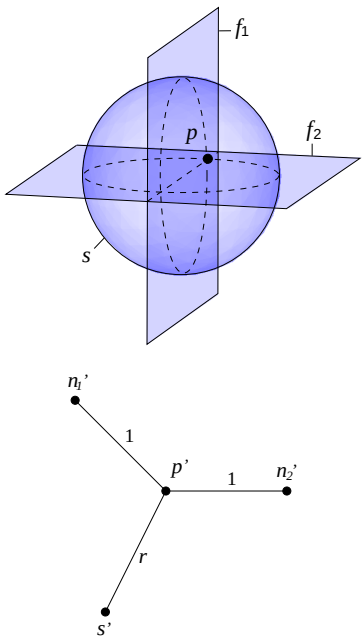
Figure 12: Mapping of lines ($L_1 - L_4$), with different numbers of coincident points

$1 <= i <= n$ with $n > 2$, then only the first two points, $p_1$ and $p_2$ are used in the representation. The other points are constrained to be coincident with the line, using an angle constraint that specifies that the angle between points is either 0 or $\pi$: $\measuredangle(p_1, p_2, p_i) = 0|\pi$ for $3 \leq i \leq n$. Such constraints with alternative values can be represented by clusters with several configurations.

The mapping of planes is similar to the mapping of lines. A plane $f$, with no constraints imposed on it, is represented by two point variables, $v_1$ and $v_2$, and a distance constraint $\delta(v_1, v_2) \neq 0$. The represented plane is the plane through $v_1$ with normal $v_2 - v_1$. If there is a constraint $Coincident(p, f)$, then the plane is represented by $p$ and a point $v$, with a non-zero distance constraint between them. Any other points $p_*$ constrained coincident with the plane are constrained in plane by a constraint $\measuredangle(v, p, p_*) = \frac{1}{2}\pi$.

For a sphere, we have to consider its centre point and its radius, since these are often constrained in applications. Fixed radius spheres are easily mapped. The centre of the sphere is represented by a point variable, and any point constrained coincident with the sphere is constrained with a distance equal to the given radius.

Variable radius spheres cannot be easily represented, because all points coincident with the sphere must have an equal, unknown distance from its centre. Such equalities are not supported by our solver, because distance constraints must have a fixed parameter value. In some cases, however, variable radius circles, spheres or cylinders can be supported, by using propagation to solve the equality constraint. If the radius can be determined by first solving other constraints in the system, then this value can simply be propagated to those distance constraints that should be equal to the radius. However, these cases




point $p$
plane $f_1$
plane $f_2$
sphere $s$, radius $r$
$p$ on $s$
$p$ on $f_1$
$p$ on $f_2$

$\mathcal{R}(p) = p'$
$\mathcal{R}(f_1) = \{p', n_1'\}$
$\delta(p', n_1') = 1$
$\mathcal{R}(f_2) = \{p', n_2'\}$
$\delta(p', n_2') = 1$
$\mathcal{R}(s) = s'$
$\delta(s', p') = r$

Figure 13: Mapping plane and sphere primitives with coincidences to constraints on points.

will not be further considered here.

Table 3 shows the different representations for primitives depending on the number of points constrained coincident with it. For a sphere, we also consider the case where a point is concident with its centre, and for a cylinder, the case where a line is coincident with its axis. The function $\mathcal{R}$ maps a primitive to a set of point variables and constraints on those variables, depending on the constraints imposed on the primitive.

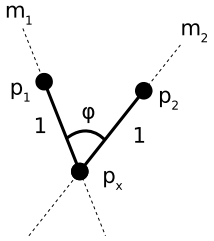
Figure 13 shows an example of how a set of primitives with coincidence constraints can be mapped to a system of constraints on points.

More general constraints on primitives are mapped to a set of points, a set of coincidence constraints between the primitives and those points, and a set of distance and angle constraints between those points. The representation of the primitive is determined by the points that are constrained coincident with it.

For example, an angle between two lines $\angle(m_1, m_2) = \phi$, where $0 < \phi < \pi$, is mapped to three points, $p_1$, $p_2$ and $p_x$, and a number of constraints, as shown in Figure 14. The constraint may or may not affect the representations of the lines, depending on the constraints already present in the system. If no other constraints are imposed on the lines, $m_1$ is represented by $p_1$ and $p_x$, and $m_2$ is represented by $p_2$ and $p_x$. The angle between two planes can be constrained by constraining the angle between the normals of those planes, using the same construction.

| primitive | constraints | mapping ($\mathcal{R}$) |
|---|---|---|
| point $p_1$ | - | $\mathcal{R}(p_1) = v_1$ |
| point $p_2$ | $Coincident(p_1, p_2)$ | $\mathcal{R}(p_2) = \mathcal{R}(p_1)$ |
| line $l_1$ | - | $\mathcal{R}(l_1) = \{v_1, v_2\}$<br>$\delta(v_1, v_2) = 1$ |
| line $l_2$ | $Coincident(l_2, p)$ | $\mathcal{R}(l_2) = \{\mathcal{R}(p), v\}$<br>$\delta(\mathcal{R}(p), v) = 1$ |
| line $l_3$ | $Coincident(l_3, p_1)$<br>$Coincident(l_3, p_2)$ | $\mathcal{R}(l_3) = \{\mathcal{R}(p_1), \mathcal{R}(p_2)\}$ |
| line $l_4$ | $Coincident(l_4, p_1)$<br>$Coincident(l_4, p_2)$<br>$Coincident(l_4, p_*)$ | $\mathcal{R}(l_4) = \{\mathcal{R}(p_1), \mathcal{R}(p_2)\}$<br>$\measuredangle(\mathcal{R}(p_1), \mathcal{R}(p_2), \mathcal{R}(p_*)) = 0 \vert \pi$ |
| plane $f_1$ | - | $\mathcal{R}(f_1) = \{v_1, v_2\}$<br>$\delta(v_1, v_2) = 1$ |
| plane $f_2$ | $Coincident(f_2, p)$ | $\mathcal{R}(f_2) = \{\mathcal{R}(p), v\}$<br>$\delta(\mathcal{R}(p), v) = 1$ |
| plane $f_3$ | $Coincident(f_3, p_1)$<br>$Coincident(f_3, p_*)$ | $\mathcal{R}(f_3) = \{\mathcal{R}(p_1), v\}$<br>$\delta(\mathcal{R}(p_1), v) = 1$<br>$\measuredangle(v, \mathcal{R}(p_1), \mathcal{R}(p_*)) = \frac{1}{2}\pi$ |
| sphere $s_1$ | $Radius(s_1) = r$<br>$Coincident(s_1, p_*)$ | $\mathcal{R}(s_1) = v$<br>$\delta(v, \mathcal{R}(p_*)) = r$ |
| sphere $s_2$ | $Radius(s_2) = r$<br>$Center(s_2, p_1)$<br>$Coincident(s_2, p_*)$ | $\mathcal{R}(s_2) = \mathcal{R}(p_1)$<br>$\delta(\mathcal{R}(p_1), \mathcal{R}(p_*)) = r$ |
| cylinder $c_1$ | $Radius(c_1) = r$<br>$Coincident(c_1, p_*)$ | $\mathcal{R}(c_1) = \{v_1, v_2\}$<br>$\delta(v_1, v_2) = 1.0$<br>$\delta(v_*, \mathcal{R}(p_*)) = r$<br>$\measuredangle(v_1, v_*, \mathcal{R}(p_*)) = \frac{1}{2}\pi$ |
| cylinder $c_2$ | $Radius(c_2) = r$<br>$Axis(c_2, l)$<br>$Coincident(c_2, p_*)$ | $\mathcal{R}(c_2) = \mathcal{R}(l)$<br>$Coincident(v*, l)$<br>$\delta(v_*, \mathcal{R}(p_*)) = r$ |

Table 3: Mapping of primitives with different incidence constraints. Points are represented by $p_*$, lines by $l_*$, planes by $f_*$, spheres by $s_*$ and cylinders by $c_*$, where $*$ can be any integer.




$Coincident(p_1, m_1)$
$Coincident(p_2, m_2)$
$Coincident(p_x, m_1)$
$Coincident(p_x, m_2)$
$\delta(p_x, p_1) = 1.0$
$\delta(p_x, p_2) = 1.0$
$\measuredangle(p_1, p_x, p_2) = \phi$

Figure 14: Mapping of an angle between lines, $\measuredangle(m_1, m_2) = \phi$.

In general, many useful constraints on primitives can be mapped in this way, and the resulting systems of constraints on points can be solved using the cluster-based approach presented in this paper. From the solutions of the latter systems, the geometry of the primitives can be completely determined.

# 8 Conclusions

In this paper we have introduced two non-rigid types of clusters, scalable and radial clusters, which can be used, together with traditional rigid clusters, to solve geometric constraint problems. Using a simple cluster rewriting approach, we can solve a larger class of problems than is possible with only rigid clusters. We have presented an incremental algorithm for this.

We have also presented methods for solution selection in the new solving approach. These can reduce the number of solutions found by the solver in an efficient way.

Finally, we have presented a way to solve systems of constraints on 3D primitives. The basic cluster rewriting algorithm can only solve systems of constraints on points. By mapping constraints on 3D primitves, such as planes, spheres and cylinders, to a system of distance and angle constraints on point variables, we can also solve such constraints.

The advantages of this approach over more general solving approaches, such as DOF-based decomposition, are that large problems can be solved efficiently, and that an incremental solving algorithm is easy to implement.

We have not yet done any extensive comparison of our solver to other solving algorithms. It would be interesting to see how it compares in terms of the class of problems that can be solved and in terms of algorithmic complexity.

The set of rewrite rules for 3D problems that we have developed is not complete; there are known well-constrained geometric constructions that cannot be solved with the current rule set. It is not even known whether there exists a complete set of rewrite rules that can be used to solve every 3D system of rigid, scalable and radial clusters. This is related to the more fundamental problem whether a generic, combinatorial description can be given of rigidity in 3D [11].

Altogether, we believe that the use of non-rigid clusters in geometric constraint solving is a promising approach to solve more complex systems more efficiently.

# Acknowledgements

We thank Chris Hoffmann and two anonymous reviewers for their helpful comments.

H.A. van der Meiden's work has been supported by the Netherlands Organisation for Scientific Research (NWO).

# References


[1] C. M. Hoffmann, A. Lomonosov, M. Sitharam, Decomposition plans for geometric constraint systems, Part I: performance measures for CAD, Journal of Symbolic Computation 31 (4) (2001) 376–408.

[2] W.-T. Wu, Basic principles of mechanical theorem proving in elementary geometries, Journal of Automated Reasoning 2 (3) (1986) 221–252.

[3] C. Durand, C. M. Hoffmann, A systematic framework for solving geometric constraints analytically, Journal of Symbolic Computation 30 (5) (2000) 493–519.

[4] G. Laman, On graphs and rigidity of plane skeletal structures, Journal of Engineering Mathematics 4 (4) (1970) 331–340.

[5] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, R. Paige, A geometric constraint solver, Computer-Aided Design 27 (6) (1995) 487–501.

[6] I. Fudos, C. M. Hoffmann, A graph-constructive approach to solving systems of geometric constraints, ACM Transactions on Graphics 16 (2) (1997) 179–216.

[7] C. M. Hoffmann, P. J. Vermeer, Geometric constraint solving in $\Re^2$ and $\Re^3$, in: D. Du, F. Hwang (Eds.), Computing in Euclidean Geometry, Second Edition, World Scientific Publishing, Singapore, 1995, pp. 266–298.

[8] I. Fudos, C. M. Hoffmann, Correctness proof of a geometric constraint solver, International Journal of Computational Geometry and Applications 6 (4) (1996) 405–420.

[9] R. Joan-Arinyo, A. Soto, A correct rule-based geometric constraint solver, Computers & Graphics 21 (5) (1997) 599–609.

[10] R. Joan-Arinyo, A. Soto, S. Vila-Marta, J. Vilaplana-Pasto, Revisiting decomposition analysis of geometric constraint graphs, Computer-Aided Design 36 (2) (2004) 123–140.

[11] M. Sitharam, Wellformed systems of point incidences for resolving collections of rigid bodies, International Journal of Computational Geometry and Applications 16 (5) (2006) 591–615.

[12] G. A. Kramer, Solving Geometric Constraint Systems: a Case Study in Kinematics, The MIT Press, Cambridge, MA, USA, 1992.

[13] C. M. Hoffmann, A. Lomonosov, M. Sitharam, Decomposition plans for geometric constraint systems, Part II: new algorithms, Journal of Symbolic Computation 31 (4) (2001) 409–427.

[14] X. Gao, Q. Lin, G. Zhang, A C-tree decomposition algorithm for 2D and 3D geometric constraint solving, Computer-Aided Design 38 (1) (2006) 1–13.

[15] D. Podgorelec, A new constructive approach to constraint-based geometric design, Computer-Aided Design 34 (11) (2002) 769–785.

[16] P. Schreck, E. Schramm, Using invariance under the similarity group to solve geometric constraint systems, Computer-Aided Design 38 (5) (2006) 475–484.

[17] J. Ullmann, An algorithm for subgraph isomorphism, Journal of the ACM 23 (1) (1976) 31–42.

[18] M. Sitharam, J.-J. Oung, Y. Zhou, A. Abree, Geometric constraints within feature hierarchies, Computer-Aided Design 38 (1) (2006) 22–38.

[19] R. Joan-Arinyo, M. V. Luzón, A. Soto, Genetic algorithms for root multiselection in constructive geometric constraint solving, Computers & Graphics 27 (1) (2003) 51–60.

[20] C. Essert-Villard, P. Schreck, J.-F. Dufourd, Sketch-based pruning of a solution space within a formal geometric constraint solver, Artificial Intelligence 124 (1) (2000) 139–159.

[21] B. Bettig, J. Shah, Solution selectors: a user-oriented answer to the multiple solution problem in constraint solving, Journal of Mechanical Design 125 (3) (2003) 443–451.

[22] H. A. van der Meiden, W. F. Bronsvoort, An efficient method to determine the intended solution for a system of geometric constraints, International Journal of Computational Geometry and Applications 15 (3) (2005) 279–298.

[23] E. van den Berg, H. A. van der Meiden, W. F. Bronsvoort, Specification of freeform features, in: G. Elber, V. Shapiro (Eds.), Proceedings of Solid Modelling '03, Eighth ACM Symposium on Solid Modeling and Applications, June 16–20, Seattle, Washington, USA, ACM Press, New York, NY, USA, 2003, pp. 56–64.

# A Appendix

A collection of rewrite rules for clusters is presented here. Each rewrite rule consists of a pattern and a procedure. The pattern is of the form: $C_1 \cup C_2 \cup \ldots \cup C_{n-1} \rightarrow C_n$. Here $C_1$ to $C_{n-1}$ represent input clusters and $C_n$ represents the output cluster. Each cluster in the pattern specifies a type, i.e. *Rigid*, *Scalable* or *Radial*, and a set of variables. A cluster with a fixed number of variables may be specified, e.g. $Rigid([p_1, p_2])$ or a cluster with an unknown number of variables, using an ellipsis, e.g. $Rigid([p_1, p_2, \ldots])$. A radial cluster specifies one center variable and a set of radial variables, e.g. $Radial(p_1, [p_2, p_3])$. A set of variables of input clusters may also be given a name, e.g. $Rigid(A = [p_1, p_2, \ldots])$.

Such named sets of variables may be combined in the specification of the output cluster, e.g. $Rigid(A) \cup Rigid(B) \rightarrow Rigid(A \cup B)$.

The procedure specified by a rule is a function $c_1 \times c_2 \times \ldots \times c_{n-1} \rightarrow c_n$. Here $c_1$ to $c_{n-1}$ represent input configurations and $c_n$ represents the output configuration. The number of configurations in the procedure is always equal to the number of clusters specified in the pattern. In a procedure, the union of two configurations, represented as $c_1 \cup c_2$, is a configuration containing all the point variables in $c_1$ and $c_2$. Point variables shared by $c_1$ and $c_2$ take the value specified by $c_1$.

We present a set of rules applicable only in 2D (Subsection A.1), a set of rules applicable in 2D and 3D (Subsection A.2), and a set of rules applicable only in 3D (Subsection A.3).

## A.1 2D rewrite rules

**Rule 1** *Merge two rigid clusters with two shared points*

*Pattern:* $Rigid(A = [p_1, p_2, \ldots]) \cup Rigid(B = [p_1, p_2, \ldots])$
$\rightarrow Rigid(A \cup B)$
*Procedure:* $c_1 \times c_2 \rightarrow c_R$
$T$ = *rotation and translation such that* $p_1$ *and* $p_2$ *in* $c_2$ *are mapped onto* $p_1$ *and* $p_2$ *in* $c_1$
$c_R = c_1 \cup T(c_2)$

**Rule 2** *Merge two radial clusters with two shared points*

*Pattern:* $Radial(p_x, A = [p_1, \ldots]) \cup Radial(p_x, B = [p_1, \ldots])]$
$\rightarrow Radial(p_x, A \cup B)$
*Procedure:* $c_1 \times c_2 \rightarrow c_R$
$T$ = *rotation, translation and scaling, such that* $p_x$ *and* $p_1$ *in* $c_2$ *are mapped onto* $p_x$ *and* $p_1$ *in* $c_1$
$c_R = c_1 \cup T(c_2)$

**Rule 3** *Merge two scalable clusters with two shared points*

*Pattern:* $Scalable(A = [p_1, p_2, \ldots]) \cup Scalable(B = [p_1, p_2, \ldots])$
$\rightarrow Scalable(A \cup B)$
*Procedure:* $c_1 \times c_2 \rightarrow c_R$
$T$ = *rotation, translation and scaling such that* $p_1$ *and* $p_2$ *in* $c_2$ *are mapped onto* $p_1$ *and* $p_2$ *in* $c_1$
$c_R = c_1 \cup T(c_2)$

## A.2 2D/3D rewrite rules

**Rule 4** *Derive a triangle from three distances*

*Pattern:* $Rigid([p_1, p_2, \ldots]) \cup Rigid([p_1, p_3, \ldots]) \cup Rigid([p_2, p_3, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3])$
*Procedure:* $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = intersection$
*circle centre* $c_R(p_1)$ *radius* $\delta(c_2(p_3), c_2(p_1))$
*circle centre* $c_R(p_2)$ *radius* $\delta(c_3(p_3), c_3(p_2))$

**Rule 5** *Derive a triangle from two distances and an angle (by rotation)*

*Pattern:* $Rigid([p_1, p_2, \ldots]) \cup Radial(p_2, [p_1, p_3, \ldots]) \cup Rigid([p_2, p_3, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3])$
*Procedure:* $c_1 \times c_2 \times c_3 \rightarrow c_R$
$\phi = \measuredangle(c_2(p_1), c_2(p_2), c_2(p_3))$
$c_R(p_1) = (\delta(c_1[p_1], c_1[p_2]), 0)$
$c_R(p_2) = (0, 0)$
$c_R(p_3) = \delta(c_3[p_3], c_3[p_2])(cos(\phi), sin(\phi))$

**Rule 6** *Derive a triangle from two distances and an angle (by intersection)*

*Pattern:* $Rigid([p_1, p_2, \ldots]) \cup Radial(p_2, [p_1, p_3, \ldots]) \cup Rigid([p_1, p_3, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3])$
*Procedure:* $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = intersection$
*circle centre* $c_1(p_1)$ *radius* $\delta(c_3(p_3), c_3(p_1))$
*ray from* $c_1(p_2)$ *direction* $\measuredangle(c_2(p_1), c_2(p_2), c_2(p_3))$

**Rule 7** *Derive a scalable cluster from two radial clusters*

*Pattern:* $Radial(p_1, [p_3, p_2, \ldots]) \cup Radial(p_2, [p_1, p_3, \ldots])$
$\rightarrow Scalable([p_1, p_2, p_3])$
*Procedure:* $c_1 \times c_2 \rightarrow c_R$
$c_R(p_1) = (0, 0, 0)$
$c_R(p_2) = (1, 0, 0)$
$c_R(p_3) = intersection$
*ray from* $c_R(p_1)$ *direction* $\measuredangle(c_1(p_3), c_1(p_1), c_1(p_2))$
*ray from* $c_R(p_2)$ *direction* $\pi - \measuredangle(c_2(p_1), c_2(p_2), c_2(p_3))$

**Rule 8** *Derive a rigid cluster from a scalable and a rigid cluster with two shared points*

*Pattern:* $Scalable(A = [p_1, p_2, \ldots]) \cup Rigid([p_1, p_2, \ldots]) \rightarrow Rigid(A)$
*Procedure:* $c_1 \times c_2 \rightarrow c_R$
$T$ = *scale configuration by* $\frac{\delta(c_2(p_2), c_2(p_1))}{\delta(c_1(p_2), c_1(p_1))}$
$c_R = T(c_1)$

## A.3 3D rewrite rules

**Rule 9** *Merge two rigid clusters with three shared points*

*Pattern:* $Rigid(A = [p_1, p_2, p_3, \ldots]) \cup Rigid(B = [p_1, p_2, p_3, \ldots])$
$\rightarrow Rigid(A \cup B)$
*Procedure:* $c_1 \times c_2 \rightarrow c_R$
$T$ = *rotation and translation such that* $p_1$, $p_2$ *and* $p_3$ *in* $c_2$ *are mapped onto* $p_1$, $p_2$ *and* $p_3$ *in* $c_1$
$c_R = c_1 \cup T(c_2)$

**Rule 10** *Merge two scalable clusters with three shared points*

*Pattern:* $Scalable(A = [p_1, p_2, p_3, \ldots]) \cup Scalable(B = [p_1, p_2, p_3, \ldots])$
$\rightarrow Scalable(A \cup B)$
*Procedure:* $c_1 \times c_2 \rightarrow c_R$
$T$ = *rotation, translation and scaling such that* $p_1$, $p_2$ *and* $p_3$ *in* $c_2$ *are mapped onto* $p_1$, $p_2$ *and* $p_3$ *in* $c_1$
$c_R = c_1 \cup T(c_2)$

**Rule 11** *Merge two radial clusters with three shared points*

*Pattern:* $Radial(p_x, A = [p_1, p_2, \ldots]) \cup Radial(p_x, B = [p_1, p_2, \ldots])$
$\rightarrow Radial(p_x, A \cup B)$
*Procedure:* $c_1 \times c_2 \rightarrow c_R$
$T$ = *rotation, translation and scaling such that* $p_x$, $p_1$ *and* $p_2$ *in* $c_2$ *are mapped onto* $p_x$, $p_1$ *and* $p_2$ *in* $c_1$
$c_R = c_1 \cup T(c_2)$

**Rule 12** *Derive a tetrahedron from three rigid clusters*

*Pattern:* $Rigid([p_1, p_2, p_3, \ldots]) \cup Rigid([p_1, p_2, p_4, \ldots]) \cup Rigid([p_3, p_4, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3, p_4])$
*Procedure:* $c_1 \times c_2 \times c_3 \rightarrow c_R$

$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = c_1(p_3)$
$c_R(p_4) = intersection$
*sphere centre* $c_R(p_1)$ *radius* $\delta(c_2(p_4), c_2(p_1))$
*sphere centre* $c_R(p_2)$ *radius* $\delta(c_2(p_4), c_2(p_2))$
*sphere centre* $c_R(p_3)$ *radius* $\delta(c_3(p_4), c_3(p_3))$

**Rule 13** *Derive a radial cluster from three angles*

*Pattern:* $Radial(p_x, [p_1, p_2, \ldots]) \cup Radial(p_x, [p_1, p_3, \ldots]) \cup Radial(p_x, [p_2, p_3, \ldots])$
$\rightarrow Radial(p_x, [p_1, p_2, p_3])$
*Procedure:* $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_x) = c_1(p_x)$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = intersection$
*cone apex* $c_R(p_x)$ *axis* $c_R(p_1) - c_R(p_x)$ *angle* $\measuredangle(c_2(p_1), c_2(p_x), c_2(p_3))$
*cone apex* $c_R(p_x)$ *axis* $c_R(p_2) - c_R(p_x)$ *angle* $\measuredangle(c_3(p_2), c_3(p_x), c_3(p_3))$
*sphere centre* $c_R(p_x)$ *radius 1*

**Rule 14** *Derive a rigid cluster from two rigid clusters with two shared points and an angle*

*Pattern:* $Rigid([p_1, p_2, p_3, p_4, \ldots]) \cup Rigid([p_3, p_4, p_5, \ldots]) \cup Radial(p_1, [p_2, p_5, \ldots])$
$\rightarrow Rigid([p_1, p_2, p_3, p_4, p_5])$
*Procedure:* $c_1 \times c_2 \times c_3 \rightarrow c_R$
$c_R(p_1) = c_1(p_1)$
$c_R(p_2) = c_1(p_2)$
$c_R(p_3) = c_1(p_3)$
$c_R(p_4) = c_1(p_4)$
$c_R(p_5) = intersection$
*cone apex* $c_R(p_1)$ *axis* $c_R(p_2) - c_R(p_1)$ *angle* $\measuredangle(c_3(p_5), c_3(p_1), c_3(p_2))$
*cylinder axis* $c_R(p_4) - c_R(p_3)$ *radius* $distance(c_2(p_5), line(c_2(p_4), c_2(p_3)))$
*plane normal* $c_R(p_4) - c_R(p_3)$ *through* $c_R(p_3) + (c_2(p_5) - c_2(p_3)) \cdot (c_2(p_4) - c_2(p_3))$