Datamining to Capture Geometric Design Intent *

Meera Sitharam †

CISE department, University of Florida

Abstract

We indicate how a careful mining of past data plays a natural in efficiently dealing with several recurring issues in the imple tation of mechanical computer aided design and assembly (MC systems. All of these issues concern the general question "hc facilitate intuitive expression of design intent, and how to auto its interpretation? ?" and more specifically "how to exploit th potential of declarative geometry representations during the design stage?" We provide the required background, formalize key underlying mathematical problems, and present initial approaches to solving them.

1 Introduction

We consider the use past data in mechanical computer aided design, assembly and manufacturing applications (MCAx) in the context of *variational geometric constraint representations*. For recent reviews of the extensive literature on geometric constraint solving in the MCAx context see, e.g, [31], [14], [18, 20]. [23].

More formally, a *geometric constraint problem* consists of a finite set of geometric objects and a finite set of constraints between them. These are typically input as designer sketches using a graphical user interface. The geometric objects are drawn from a fixed set of types such as points, lines, circles, conics, and sometimes also spline curves in the plane; or points, lines, planes, cylinders, spheres, and sometimes also spline surfaces in 3 dimensions. The constraints are spatial and include logical constraints such as incidence, tangency, perpendicularity; and metric constraints which include distance, angle, radius, enclosed area/volume in the case of spline curves/surfaces, involving numerical parameters.

Sometimes idealized global constraints such as symmetry constraints are added to the repertoire. All constraints can usually be written as algebraic equations whose variables are the coordinates of the participating geometric objects. For example, the distance constraint of d between two points on the plane results in the quadratic equation $(x_1 - x_2)^2 + (y_1 - y_2)^2 = d^2$, which constrains the positions (x_1, y_1) and (x_2, y_2) of the two points.

The solution or realization of a geometric constraint problem is a real zero of the corresponding algebraic system. In other words solution is a class of valid instantiations of the geometric elements such that all constraints are satisfied. Here, it is understood that such a solution is in a particular geometry, for example the Euclidean plane, the sphere, or Euclidean 3 dimensional space.

Most of today's MCAx systems restrict variational constraint representations mainly to 2D cross sections. This persists despite the general consensus that advocates a judicious use of variational 3 dimensional or spatial constraints for the intuitive expression and maintanence of certain complex and cyclic relationships that often occur between features, parts or subassemblies [6] [13] [9] [24] [27] [26] [44] [45] [49].

It is generally understood that purely procedural, history based representations risk tediousness and inefficiency in the generation and maintanence of examples such as the toy variational constraint system shown in Figure 1 and the assembly example shown in Figure 2. Such representations often require the designer to perform hand calculations, or explicit trial and error constructions on screen. Furthermore, intuitively local changes and updat es could require long "rollbacks," at worst repeating the design process from scratch. These difficulties persist, even if the procedural representation incorporates standard solid modeling languages such as B-rep or CSG, parametric constraints or even variational constraints on 2D cross sections combined with sweeps, extrusions, etc.

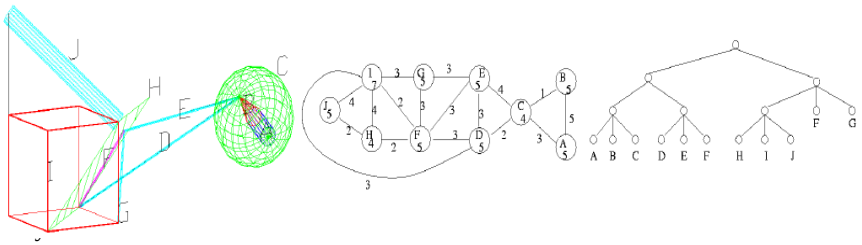

Figure 1: Toy spatial wellconstrained example, constraint graph and decomposition plan tree; blue bars and cylinder have fixed length, pencil length is sphere diameter, two bars are tangent to sphere and incident on pencil tip which is incident on extended top cube face, diagonal plane of cube contains one of the bars, cylinder ends at cube's extended edge and is perpendicular to diagonal plane; numbers in graph represent dofs

We identify three factors which dictate why declarative geometry representations, and in particular variational spatial constraints are not used to their full potential in today's MCAx systems. Each of these factors indicates the importance of past design data, and gives rise to datamining problems.

1.1 Use of past Data

Within the overall datamining context, our emphasis is on *extracting conceptual patterns* and *utilizing relevant* data in an efficient manner, rather than *isolating* the relevant raw data from large amounts of other data (which is an important issue, but one that we do not deal with). Hence, whenever a certain type of past raw data are useful for our purposes, we assume that such data are clearly isolated and easily available, and we provide background justification for this assumption. We do, however, consider the question of how best to *organize, index and store* the relevant raw data. We generally distinguish between: the time needed to *preprocess* the past raw data, and organize into a convenient datastructure; and the time needed to *use* the processed past data on the current input at run time.

We formalize and treat the underlying technical problems problems at an abstract mathematical level. We concentrate on careful problem statements or reductions to already formalized problems, and suggest initial approaches to solving them. Our succesful experiences [21, 22, 40, 39] at building the FRONTIER geometric constraint solver have consistently shown that this process of formalization and reduction constitutes a significant portion of the solu-

* Supported in part by NSF Grants EIA0096104, CCR9902025.

† Corresponding author: sitharam@cise.ufl.edu

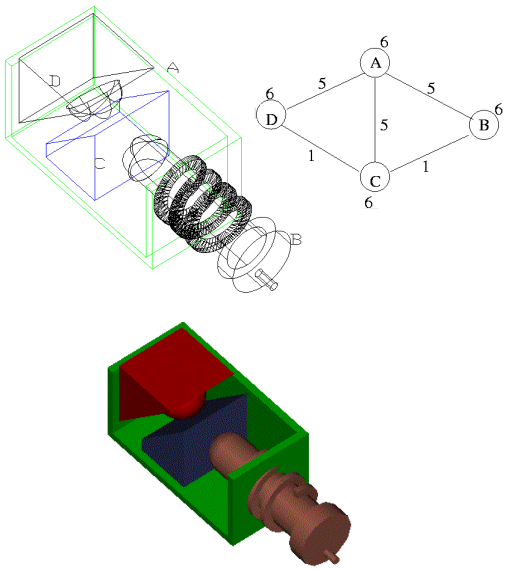

Figure 2: Underconstrained assembly example with 1 extra degree of freedom, and constraint graph

tion effort, and forms a crucial foundation for efficient algorithm implementation. More significantly, since it is tempting to suggest the use of general methods such as statistical clustering techniques on past data as a cure for all ills, we take particular care to critically consider the usefulness of past data. In particular, by careful formalization we avoid the trap of converting one combinatorially difficult problem into another equally difficult one which additionally has the drawback of requiring past data. In particular, we begin with a minmax formulation to measure the worst case complexity of an algorithm as a function of both the quantity of past data available and the size of the input *assuming* that the past data is of the best possible quality, for each input size. In particular, this formalization and its straightforward variants permit comparison with algorithms that do not use past data at all. Moreover, their generality permits their use outside our current application, in the general datamining context.

As a related note: we completely exclude certain related issues that could conceivably fall under the general category of datamining issues, but which are best dealt with as direct algebraic-numeric problems. One example concerns solution extrapolation for a geometric constraint system whose algebraic structure is identical to an already solved constraint system, differing only in actual numerical parameter values. Other examples include dealing with nonlinear algebraic dependencies and nongeneric solutions, as well as topological solution changes caused by parameter perturbations.

Finally, our overall emphasis is on using past data for combinatorially difficult computational problems that have an exact output specification. For such problems, we consider the possibility of using past solutions to speed up inefficient current algorithms, with exponential or higher time complexity. However, we also consider computational problems for which the "correct" output is unknown. The "correctness or quality" of an output or its "closeness to the actual" is only measured by comparison with an expert's idea of the ground truth, or using a variety of quality measures that still need to be formalized.

1.2 Organization

Generally, the overall background and and definitions necessary to motivate each problem immediately precede the problem formulation. However, simply by way of being the first technical section, Section 2 contains the bulk of the preliminary background and definitions.

The three sections correspond to a rough classification of problems into three categories.

The first category involves mainly the combinatorial information in geometric constraint problems, and the object/constraint type information, but not the actual numerical parameter information attached to objects/constraints. These problems are formalized in Section 2 and relate to: the decomposition of the constraint problem; generation of manufacturing and other constraint views from the design view; interactive completion of of partial constraint specifications and underconstrained problems; determining dependent or redundant constraints in overconstrained or constraint reconciliation problems.

The second problem category discussed in Section 3 additionally involves numerical parameter information and deals with solution realization issues such as: taming combinatorial explosion by choice of solution possiblities; automating the modification of special features that are generated during the realization process; and dealing with soft constraints such as approximate and fuzzy constraints. As noted previously, we do not consider many other important issues that involve numerical parameter information such as classsifying nongeneric solutions; nonlinear algebraic dependencies between constraints; and changes in solution topologies caused by numerical perturbations. For these problems, the task of extrapolation from past data instances would involve algebraic-numeric techniques, which are outside the scope of our current discussion.

In the problem categories listed above, we assume that the geometric constraint system is input, and the issue is to effectively use past data to process this input system, in particular, to obtain solution or realization instances of the input system.

2 Decomposition and Completion

The following rule of thumb has emerged from years of experimentation with geometric, spatial constraint solvers in engineering design and assembly: the use of direct algebraic/numeric solvers for solving large subsystems renders a geometric constraint solver practically useless (see [12] for a natural example of a geometric constraint system with 6 primitive geometric objects and 15 constraints, which has repeatedly defied attempts at tractable solution). The overwhelming cost in a geometric constraint solving is directly proportional to the size of the largest subsystem that is solved using a direct algebraic/numeric solver. This size dictates the practical utility of the overall constraint solver, since the time complexity of the constraint solver is at least exponential in the size of the largest such subsystem.

Therefore, an effective constraint solver would first decompose the constraint system into small subsystems, whose solutions can be recombined by solving other small subsystems. The primary aim of this decomposition plan is to restrict the use of direct algebraic/numeric solvers to subsystems that are as small as possible. Hence the *optimal* or most efficient decomposition plan would minimize the size of the largest such subsystem. Any geometric constraint solver should first solve the problem of efficiently finding a close-to-optimal *decomposition-recombination (DR) plan*, because that dictates the usability of the solver. See DR-plans in Figures 1, 2, 3, and 4. Finding a DR-plan can be done as a pre-processing step by the constraint solver: a robust DR-plan would *generically*[1] remain unchanged even as minor changes to numerical parameters or

[1] we omit the formal definition of generic, it intuitively means "for all but finitely many choices of parameter values"

other such on-line perturbations to the constraint system are made during the design process.

The intermediate subsystems in the DR-plan should be generically *rigid*. A *rigid* subsystem of the constraint system is one for which the set of real-zeroes of the corresponding algebraic equations is generically discrete (i.e. the corresponding real-algebraic variety is zero dimensional), *after* the local coordinate system is arbitrarily, i.e after an appropriate number of *degrees of freedom* D, are fixed. In other words, the constraints force a finite number of isolated solutions, so that one realization cannot be obtained by an infinitesimal flexing perturbation of another. The constant D is usually the number of (translational and rotational) degrees of freedom available to any rigid object in the given geometry (3 in 2 dimensions, 6 in 3 dimensions, typically $\binom{d+1}{2}$ for d dimensions) and in some cases, D depends on other symmetries of the subsystem. For example, both a point and a fixed radius circle in 2 dimensions have two translational degrees of freedom and no rotational degree of freedom. A fixed-length line segment in 2 dimensions has 3 degrees of freedom, but in 3 dimensions only 5 degrees of freedom. An *underconstrained* system is not rigid, i.e, its set of real zeroes is not discrete (non-zero-dimensional). A *wellconstrained* system is a rigid system where removal of any constraint results in an underconstrained system. An *overconstrained* system is a rigid system in which there is a constraint whose removal still leaves the system rigid. Directly and independently solvable systems of equations are therefore rigid, i.e, wellconstrained or overconstrained. A *consistently* overconstrained system is one which has atleast one real zero.

Most DR-planners ignore the algebraic information and use only the combinatorial information in the constraint system to generate the DR-plan. A *geometric constraint graph* $G = (V, E, w)$ corresponding to geometric constraint problem is a weighted graph with n vertices (representing geometric objects) V and m edges (representing constraints) E; $w(v)$ is the weight of vertex v and $w(e)$ is the weight of edge e, corresponding to the number of degrees of freedom available to an object represented by v and number of degrees of freedom removed by a constraint represented by e respectively. See the constraint graphs in Figures 1, 2, 3, and 4. Note that the constraint graph could be a *hypergraph*, each hyperedge involving any number of vertices. A subgraph $A \subseteq G$ that satisfies

$$\sum_{e \in A} w(e) + D \geq \sum_{v \in A} w(v) \tag{1}$$

is called *dense*, where D is a dimension-dependent constant, to be described below. The function $d(A) = \sum_{e \in A} w(e) - \sum_{v \in A} w(v)$ is called the *density* of a graph A. The constant D, introduced in the previous paragraph, captures the degrees of freedom associated with the *cluster* of geometric objects corresponding to the dense graph. In most naturally occuring cases, it turns out that a dense graph with density strictly greater than $-D$ generically corresponds to an overconstrained cluster in the corresponding constraint system. A graph that is dense and all of whose subgraphs (including itself) have density at most $-D$ is wellconstrained. A dense graph is *minimal* if it has no dense proper subgraph. It turns out that all minimal dense subgraphs generically correspond to rigid clusters, but the converse is not the case. A graph that is not rigid is *underconstrained*.

Thus a DR-plan can now be viewed as a hierarchical decomposition of a constraint graph into a *DR-forest*, each of whose nodes is a rigid subgraph, and whose roots are a complete set of maximal rigid clusters in the constraint graph. An *optimal* DR-plan minimizes the largest *fan-in* that occurs in the DR-forest. See the DR-trees in Figures 1, 2, 3, and 4: in these cases, the entire graph is rigid, resulting in a tree rather than a forest with many roots. The optimal decomposition problem is NP-hard as shown in [23], with no currently known constant factor approximation algorithms [35].

These observations form the basis for *generalized degree of freedom analysis* used by certain DR-planners such as [1, 43, 34, 31],

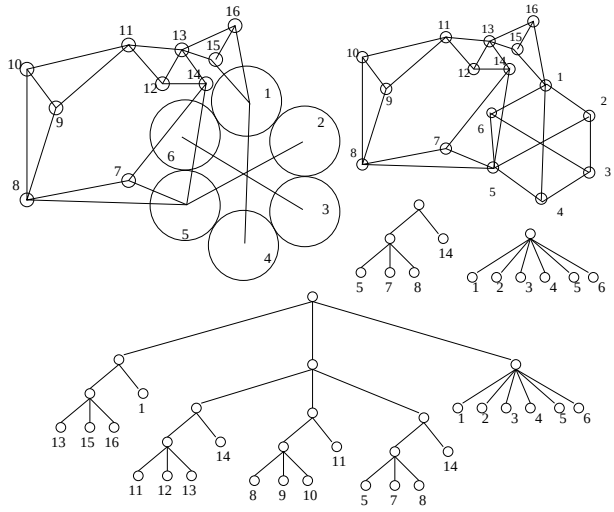

Figure 3: 2D variational example, constraint graph and DR plan incorporating the 2 input feature hierarchies shown; all vertices and edges have dof weights 2 and 1

and [23, 22]. Many other graph-based DR-planners such as e.g.[4, 41, 42, 4], [24, 25, 15, 16], use decomposition based on specific patterns occuring in the constraint graph resulting in a crippling lack of generality for spatial constraints. Other combinatorial matroid based and linear algebraic characterizations of generic rigidity [33, 50, 51] and DR-planners based on them [8, 17, 28] work on a larger class of constraint systems, but result in exponential algorithms even for the subproblem of detecting rigid clusters, which is only part of the problem of optimal decomposition.

Most graph-based DR-planners are classified and their performance is formally analyzed with respect to a number of relevant measures in [21]. Most of these DR-planners can be proven to output far-from-optimal decompositions in natural cases. In particular, all except the DR-planner of [22] would perform poorly on atleast one of the three natural examples shown in Figures 3 and 4. The DR-planner of [22] - based on network flow and so-called frontier vertex algorithm - was specifically designed to defeat the drawbacks of the previous DR-planners and excel in the performance measures of [21], and does well in the above examples. However, this or any other polynomial time algorithm is unlikely to perform optimally in all natural cases since the problem, as mentioned earlier, is NP-hard and its approximability status is unknown.

2.1 Using past data for efficient decomposition

Given the graph-based DR-planning scenario, it is natural to consider whether past data, consisting of geometric constraint graphs and their close-to-optimal DR-plans can be used to speed up the process of obtaining optimal DR-plans for the currently input constraint graph. This gives rise to the following formal computational problem.

Note. We state computational problems by simply specifying the *input*, *desired output*, *past data available*, and complexity measure being used. It is understood that the actual problem is to design an efficient algorithm for those specifications.

Problem 2.1

Input: *a geometric constraint graph G with n vertices;*

Past Data: *a set P_n of at most k tuples (G_i, T_i), where G_i are geometric constraint graphs with at most n vertices, and T_i are the corresponding DR-plans whose maximum fan-in is within a constant factor $c \geq 1$ of the optimal;*

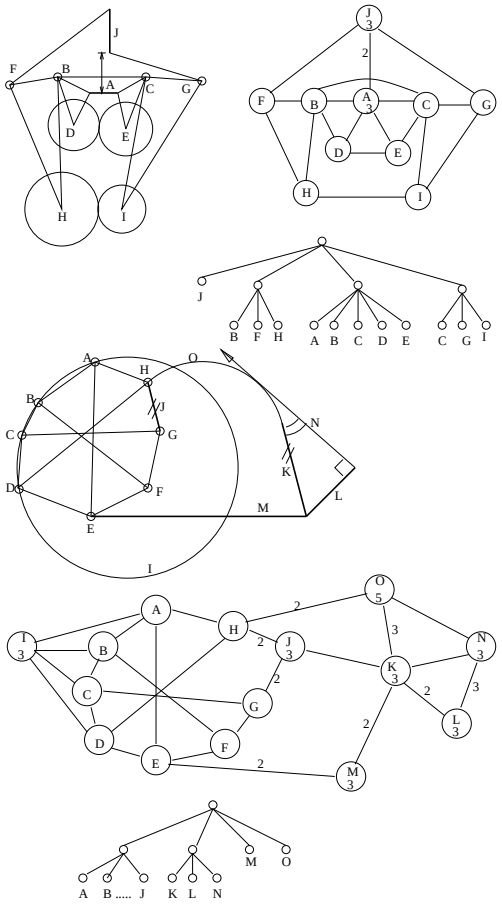

Figure 4: Two more 2D variational examples requiring sophisticated decomposition; unnumbered vertices and edges have dof weights 2 and 1

Output: *an optimal DR-plan for G, or a DR-plan whose maximum fan-in is atmost c-factor of the optimal.*

For a fixed c, the efficiency or complexity of such approximation algorithms is measured by a function $T_c(n,k) = O(\min_{\{P_n:|P_n|\leq k\}} \max_{\{G:|G|\leq n\}} [$ the time taken by the algorithm on input G and past data set $P_n)]$. The function $T_c(n,k)$ does not include the preprocessing time needed to organize P_n into in a convenient data structure, but it does include the time needed to access the required elements of P_n from this datastructure. Hence the algorithm description includes a description of the data structure.

Intuitively, the *min* in the above expression permits the most useful past data to be used, and and the *max* captures the worst case time complexity.

Of particular interest is the efficiency of such algorithms when k is some function of n. I.e, we are interested in the complexity: $T(n, f(n))$, where $f(n)$ could be for example, $O(\log n)$ or $O(poly(n))$ etc. As one variation, if we were interested in average case as opposed to worst case complexity, the max in the above expression could be replaced by the expected value over some probability distribution of the inputs; and similarly the choice of past data could be randomized and min could be replaced by the expected value over some probability distribution over the past data. As another variation, one could consider Las Vegas (probably fast, always correct) or Monte Carlo (always fast, probably correct) randomized algorithms. As a final variation, if one is interested in algorithms that admit closer approximations by running longer, one could introduce c as an argument, i.e, use a function $T(n,k,c)$. In a somewhat less likely scenario, one could conversely be interested in algorithms which crucially *assume* that k is a certain function of n, i.e, their correctness crucially depends on k being a certain function of n. All of these variations are interesting and can be formulated as straightforward modifications of the above problem definition. Some of these variations coincide with models used in computational learning theory, such as the PAC model see for example [30], exact learning model [3], and the AAC model [46, 48, 47].

We can now formally define when past data is *useful* for the (approximate) optimal DR-planning problem. I.e, there should exist an algorithm whose complexity $T_c(n,k)$ decreases nontrivially as k increases, atleast on some ranges of k (keeping n unchanged). To define "nontrivial," note the following. Since the optimal DR-planning problem is NP-hard, unless $P = NP$, there is no polynomial time algorithm for determining the optimal DR-tree with an approximation factor of 1, using an amount of past data that is independent of the size of the input; in other words, for any algorithm, $T_1(n, O(1))$ is superpolynomial. On the other hand, if the set of past data P_n consists of all labeled graphs of size n and their optimal DR-forests, one can use a direct indexing by, say the adjacency matrix, to immediately locate the optimal DR-tree for the input graph. Since the size of such a dataset P_n is $O(2^{n^2})$, there is a simple algorithm for which $T_1(n, 2^{n^2})$ is $O(n^2)$. Hence as the size of the past data set increases from $O(1)$ to $O(2^{n^2})$, the complexity drops from superpolynomial (or even exponential), to $O(n^2)$. This elementary observation indicates that even an inverse linear dependence is not particularly interesting. Hence we require the complexity $T_c(n,k)$ to have an inverse superlinear dependence on k, atleast on certain ranges of k. In other words, for past data to be considered useful, there must be an algorithm that shows a *speedup that is superlinear* in k, atleast on certain nontrivial ranges of k.

Note. the above conceptual definition of usefulness applies not only to this type of problem, but to fairly general situations and types of data.

2.2 Circumventing the isomorphism problem

Straightforward approaches to Problem 2.1 fail to show superlinear speedup by the use of past data due to the following reason. The information in a particular past data pair (G_i, T_i) is the same for all the labelings of the vertices in G_i. Since there are potentially superpolynomially many labeled isomorphic copies of G_i, one should avoid storing all the corresponding pairs, for the efficient use of past data. However, if only one (unlabeled) representative pair (G_i, T_i) is stored for each isomorphism type, in order to use this data, one would have to solve the graph isomorphism or correspondence problem in order to locate the unlabeled past data graph that is closest to the given labeled input graph (or one of its subgraphs). This is an unsatisfactory situation, since it is unknown whether the graph isomorphism problem is tractable, i.e, whether it has polynomial time complexity.

To circumvent this problem, we recapitulate the MCAx origins of the DR-planning problem, and two observations come to our rescue. The **first** observation is the following. The second observation will be discussed in the next section. A competing requirement to optimality in MCAx applications is that the DR-plan should be consistent with a design decomposition: in particular, the designer often has a multi-layered or hierarchical conceptual decomposition in mind, reflecting *features* or conglomerates of features in the case of product design and parts or subassemblies in the case of assembly. We stay consistent with FEMEX and other standard definitions of feature hierarchy. See [7], [2] [5, 10, 11], [36, 37]. The designer would typically wish the DR-plan to further decompose the components of her feature hierarchy, but also to treat these components (recursively) as units that can be independently manipulated. See Figure 3 for an example of a *feature hierarchy tree*, and Figure 5, for an example of a constraint system that relates features at an intermediate level of a feature hierarchy containing features specified in different representation languages.

For example, the geometric objects within a feature are manipulated with respect to a local coordinate system, and a feature as a whole unit is manipulated with respect to a local coordinate system of the next level feature. The DR-plan should therefore be a consistent extension and/or refinement of this conceptual design decomposition. Recent DR-planners such as the Frontier vertex algorithm of [22] have incorporated the above requirement in addition to optimality. The availability of the input feature hierarchy leads to the following modification of Problem 2.1.

Problem 2.2

> **Input:** *a geometric constraint graph G with n vertices; and and a feature hierarchy forest F;*
>
> **Past Data:** *a set P_n of at most k tuples (G_i, F_i, T_i), where G_i are geometric constraint graphs with at most n vertices, and F_i and T_i are the corresponding feature hierarchies and DR-plans respectively. The T_i have maximum fan-in within a constant factor $c \geq 1$ of the optimal;*
>
> **Output:** *a DR-plan for G, whose maximum fan-in is atmost a c-factor of the optimal.*

We trace the crucial steps in solving this problem, The details of these steps yield interesting open problems. The complexity measure of interest here is the average case variant of the function $T_c(n, k)$ in Problem 2.1. Assuming a natural input distribution would ensure that nontrivial feature forests are part of the average input. Thus we can circumvent the graph isomorphism problem by designing an indexing (hash) function - during the past data preprocessing stage - that will map an input feature hierarchy F to a set of the tuples in P_n, whose F_i is *closest* to being isomorphic to F. This process can be done recursively, moving down the hierarchy forest

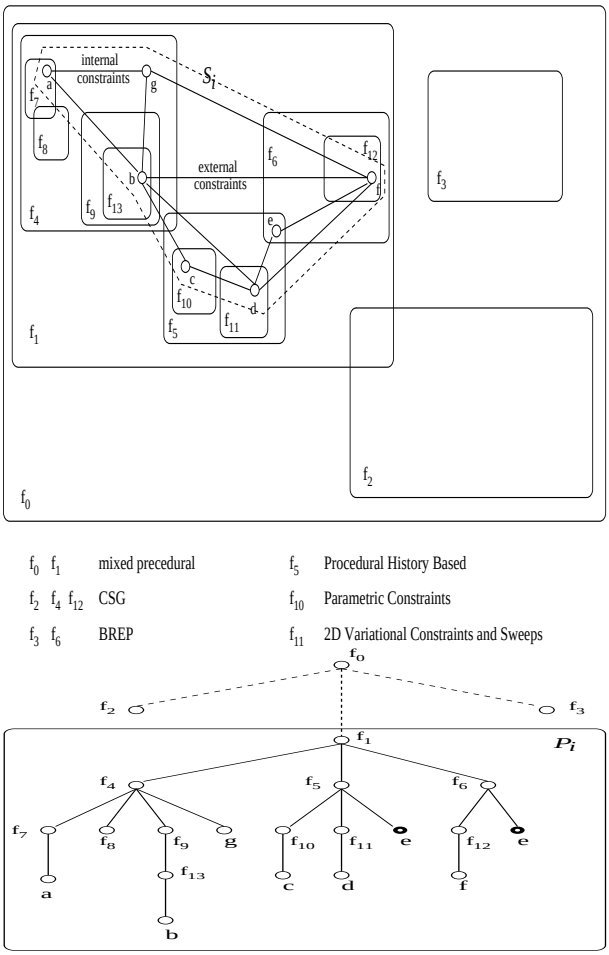

Figure 5: Constraint system S_i and underlying feature hierarchy P_i; features $f_0, \ldots, f_{11}$ are specified in nondeclarative or procedural representation languages; a, b, c, d, e are geometric elements from lower level features participating in S_i; $f_0, \ldots, f_3$ are features at a higher level of hierarchy

F (in a breadth-first manner), and finding the F_i that are closest to being isomorphic to the subhierarchies. I.e, different portions of of F could be mapped to different F_i.

Several straightforward and natural definitions of "closest" are valid. An interesting issue is the design of a hash function whose computation, on input F, involves the solution of much easier *tree* isomorphism problems between (parts of) F and the F_i's; After the hash function is computed, in the average case, one can restrict graph isomorphism occurences to much smaller cluster subgraphs of G and G_i: these clusters correspond to the lowest level vertices in the matched portions of the F and F_i.

A challenging issue is determining the exact tradeoffs between the sizes of the tree and graph isomorphism problems that need to be solved. For a given constraint graph G, the size of the tree isomorphism problem that needs to be solved is directly related to the size and level of detail of the feature hierarchy F; however, this size is inversely related to the size of the the cluster graphs corresponding to the vertices of F; thus solving larger tree isomorphisms results in smaller graph isomorphism problems.

2.3 Feature matching and recognition

A second observation specific to the MCAx application not only helps to circumvent the graph isomorphism barrier to solving Problem 2.1, but also arises in other related problems.

Both Problem 2.1 and Problem 2.2 rely on the constraint graph information alone, which contains degrees of freedom but not the information on the *types* of primitive elements or constraints. This is well-justified for standard DR-planners that do not use past data, as explained in Section 2. However, in order to use past data for DR planning, the complete type information contained in the geometric constraint problem can be crucial. By augmenting the the constraint graphs G and G_i in Problems 2.1 and 2.2 by the type information to get the augmented constraint graphs C and C_i, determining the required isomorphisms becomes significantly easier. Note that we omit the actual numerical parameters associated with the geometric objects and constraints in the complete constraint problem, since these are highly specific to the particular data instance. However, in the presence of only the complete type information, the problem of determining the tree isomorphism between the feature hierarchies F and F_i now embeds the natural problem of *feature matching*, where the features being matched correspond to the vertices in F and F_i.

In the cases where the feature hierarchies F or F_i are not very detailed, Problem 2.2 largely reduces to Problem 2.1; In this case, there are not enough features (vertices in F and F_i) that can be matched to make a significant dent in the sizes of the isomorphism problems to be solved. Therefore, the isolation of features becomes the central issue. This leads to Problem 2.3 which needs feature recognition and isolation in addition to matching.

The past data available for Problem 2.3 is assumed to include a list of *significant feature types* that are used for the feature recognition and isolation at run time. An example of a feature type is "a sharp corner at the meeting of 3 plane faces." These feature types are extracted from the set of past geometric constraint problems and their DR-plans during the preprocessing step. It is a natural assumption that the number of significant feature types is no larger than the number of past data tuples available. Several valid definitions of "significant feature type" are possible and it is an interesting problem to choose one that fits the given scenario. For example, one could define significant purely as "statistically significant," or additionally based on an expert's idea of mechanically significant features: the properties of these feature types could themselves be specified as logical geometric constraints.

Problem 2.3

> **Input:** *C, a geometric constraint graph augmented by object and constraint type information, with n geometric objects; and a feature hierarchy forest F;*
>
> **Past Data:** *a set P_n of at most k tuples (C_i, F_i, T_i), and a list L_n of significant feature types of size at most k; here C_i are type-augmented geometric constraint graphs with at most n geometric objects, and F_i and T_i are the corresponding feature hierarchies and DR-plans respectively. The T_i have maximum fan-in within a constant factor $c \geq 1$ of the optimal. The higher level feature types in L_n can be represented as a feature hierarchy forest, containing lower level constituent features, also from L_n; this forest is isomorphic to a subforest of one of the F_i's. Moreover, the DR-plan of each element of L_n is embedded as a subtree or forest in one of the T_i's. Thus there is a reference pointer from each element of L_n both to its constituent elements in L_n, to a subforest of one of the T_i's, and to a subforest of one of the F_i's.*
>
> **Output:** *A DR-plan for C whose maximum fan-in is atmost a c-factor of the optimal.*

Based on the chosen definition significant feature type, it is a nontrivial problem to determine the significant feature types from the past data tuples (C_i, T_i, F_i) : it calls for a judiciously chosen combination of general and standard techniques such as clustering; learning algorithms; Bayesian networks and neural nets [32], [29]; as well as less standard and more specific geometric correlation methods.

Note again that the extraction of significant feature types from P_n is part of the preprocessing step, and the time spent on it is *not included* in the complexity measure $T_c(n, k)$. However, $T_c(n, k)$ does include the time taken to recognize features in C that conform to types from L_n, and to match these features to corresponding ones in C_i, thereby circumventing the isomorphism or correspondence problem. In other words, an indexing function needs to be designed to incorporate the above recognition and matching processes to locate the past data tuple (C_i, T_i, F_i) closest to the current input (C, F).

Feature recognition and matching are also useful for solving other related problems that involve the use of past data. These are discussed in the next section.

2.4 Partial specifications, redundancies, multiple views

Partly due to the expressiveness of declarative geometry representations. it is important to support the management of a wide variety of ambiguities and inconsistencies of both input and output. Well documented problems, such as those in [18] can benefit from the use of past data. We formalize 3 problems by indicating their exact relationship to Problem 2.3.

Partial constraint specifications

The first question is how to automate of the completion of partial constraint specifications. This falls into the following broad categories.

- To speed up the sketching process: for example, the designer sketches only a few key objects and relationships which define a commonly occuring type of part or feature, and would like an instance of of that feature type to appear on the screen. The designer would then be allowed to reassign the actual numerical parameter values thereby completing the specification of that part or feature, and would then proceed with their sketch. A noninteractive version of this scenario is also possible: the designer inputs only a few key objects and constraints for an entire composite of commonly occuring types of features and parts, along with crucial hierarchy information; and would

like a complete instance of the desired composite to appear on the screen, allowing numerical parameter values to be further edited.

- To find a wellconstrained completion of an underconstrained geometric composite: this occurs for example when the designer wishes to specify a geometric composite as a complex cyclical constraint problem. In such cases, under- or over-specifications are unavoidable since the problem of determining if a system is wellconstrained is, in general, as hard as the entire DR-planning problem! This also occurs as an intermediate problem when one wishes to render valid realizations of a series of partial sketches *online* as the designer adds each constraint and object one by one.

The former application can benefit directly from a list of commonly occuring types of parts and features, i.e, a list L_n as in Problem 2.3, extracted from past data. Ideally, each low level feature type in this list would simply be indexed by a *minimal* set of objects and relationships that define that feature type; and every *higher level type* of feature or part would be indexed by a minimal set of *key lower level features* and their relationships. There could be more than one minimal defining set, in which case the indexing function maps each one of these minimal defining sets to to the corresponding feature type. Note as mentioned earlier that this indexing function is designed during the preprocessing stage, which is not included in the complexity function $T_c(n, k)$. In fact, such an indexing of L_n could contribute significantly to the efficient solution of Problem 2.3 as well; moreover, in the presence of such an indexing, even the non-interactive version of the above application reduces to an abbreviated and easier version of Problem 2.3 (the difference is that in the current case, a DR-plan is not needed).

The latter application of finding a wellconstrained completion of the input underconstrained system is more challenging. In the next section, we will consider the issue of what numerical parameters to assign to the additional constraints, and how to obtain a valid realization. Here we are primarily interested in the combinatorial and type information that would make the system wellconstrained.

One possibility is to embed the input underconstrained system and its DR-forest consistently into previously seen wellconstrained systems or DR-trees. Such an embedding directly indicates what additional constraints and objects need to be added to make the input system wellconstrained. This results in a *direct generalization* of Problem 2.3, where the output sought is an optimal DR-tree (or a constant factor approximation) for a *wellconstrained completion* of the input C, and where the list of past feature types L_n is indexed as described in the previous application.

The second possibility is to save and use past completion data, i.e, past data where an underconstrained system was input and a similar completion occurred. In this case, crucial additional information available, namely what degrees of freedom were fixed or what additional constraints were added in order to obtain the wellconstrained completion. Formally, this results in a *further generalization* of the Problem 2.3: besides the extensions described in the previous paragraph, the past data additionally contains the DR-trees for the wellconstrained completions of the C_i.

Note that the constraints that were added to obtain wellconstrained completions need not only be local constraints. Idealized *global constraints*, in particular *symmetry constraints*, are often used to obtain wellconstrained completions. These are natural in many physical applications as they reflect minimum energy configurations. Symmetries are also natural requirements in artistic design applications. These often force solution uniqueness and moreover permit the reduction of a large global constraint system into a local constraint system. Global constraints are also used to force an otherwise underconstrained system, with infinitely many solutions to become a well-constrained system with a single solution. Such past data involving the effect of imposition of global constraints is extremely valuable since it typically involves the intuition and extensive experimentation by the designer.

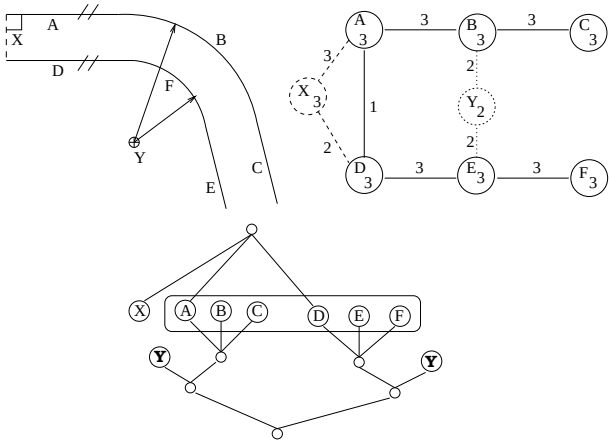

Figure 6: Multiple views (dotted and dashed) with different referencing elements x and y; numbers represent degrees of freedom; two views have intertwined feature hierarchies (above and below); box contains netshape elements common to both views

Generation of multiple product views

The second related question concerns the use of the designer's constraint model of a product to automatically generate different constraint models that facilitate, say, the manufacturing of the product, including machining of the parts, assembly, tolerancing, etc. More specifically, in client server architectures, these constraint models are different views of the same master model [19], [11]. Hence the question deals with the automatic generation of multiple product views. Views may be significantly different, their feature hierarchies may not even be refinements of one another, but intertwined. Furthermore, each view could contain different referencing geometric elements that are not physically part of the final composite (or netshape) and therefore, not part of the other views. See Figure 6.

For commonly occurring features, parts and assemblies, the usage of past data - correspondences between past manufacturing views and their design views - leads to the following generalization of Problem 2.3.

The past data tuples are now of the extended form: $(C_i^1, C_i^2, \ldots, F_i^1, F_i^2, \ldots, T_i)$, where the C_i^j and F_i^j are the various constraint and feature hierarchy views of the product, labeled consistently, so that the correspondence is clear. Note that a wellconstrained cluster remains wellconstrained in all the C_i^j's since they represent the same product. Due to this simple fact, and based on the DR-planning algorithm of the FRONTIER system [40], it is reasonable to assume that the DR-plans T_i in fact commonly incorporate *all* of the feature hierarchies F_i^j. Each such DR-plan is a union of forests, and is therefore best represented as a directed acyclic graph or DAG. Similarly, there are several lists L_n^j of significant feature types, each corresponding to a different view. The elements of each L_n^j contain references to other elements in that list as well as the corresponding C_i^j, and F_i^j as described in Problem 2.3.

Detecting redundant constraints

The third related question concerns the detection of overconstraints and simple dependencies between constraints. We point out that complex algebraic dependencies can exist between (even parameterless) constraints. As mentioned in the Section 1.1, these are best treated using algebraic-numeric techniques outside the scope of this

manuscript. The question of detecting simple constraint dependencies arises in overconstrained situations and in reconciling constraints from multiple product views. However, as observed in [40], this problem turns out to be reducible to the DR-planning problem, and a good DR-planner such as FRONTIER finds overconstraints and performs constraint reconciliation as a direct byproduct of the DR-planning process. Therefore, the issue of past data usage for this problem is in fact reducible to Problem 2.3. Once the redundant constraints have been detected, their consistency, which depends on their numerical parameter values, is directly checked by measurement during the realization phase discussed in the next section.

3 Realization of the geometric composite

In this section, as opposed to the previous one, we deal with a problem that arises during the solving or realization phase, which uses the decomposition or DR-plan found during the planning phase. These phases could also proceed in parallel, with the solver operating on those parts of the DR-forest that the planner has already completed. The solving phase depends on more than just the combinatorial and type information in the constraint model; it depends on the actual numerical parameter values as well.

During the realization phase, the clusters corresponding to the vertices of the DR-forest are resolved, moving bottom-up. The resolution of each cluster involves translating and rotating the realizations of each of its (rigid) child clusters in the DR-forest, in such a way that the constraints relating the children are satisfied. If each child cluster i has $m_i \geq 1$ realizations (m_i is finite since each cluster is rigid), then the number of potential realizations of the parent cluster is $\prod_i^h m_i$, where h here is the number of children (fan-in). This means that even if the there are atmost 2 possible solutions to each leaf cluster, and the number of leaf clusters is n/h, where h is the maximum fan-in of the DR-tree, and n is the total number of primitive geometric objects in the constraint problem, it follows that the number of realizations of the entire wellconstrained composite (root of the DR-tree) is atleast $\Omega(2^{n/h})$. Recall that one main function of the DR-planner is to *minimize* the maximum fan-in h of the DR-tree to speed up the solution of each cluster or subsystem. This implies that there is a *combinatorial explosion* of solution possibilities: there could be exponentially many in the size of the geometric constraint problem.

These solutions or realizations are often called *bifurcations*. For example, a system of 5 distance constraints on 4 points, in the form of a quadrilateral with a diagonal typically has 2 bifurcations depending on whether the quadrilateral is "folded" across the diagonal or not. These bifurcations often have different topological properties. It is therefore crucial to give the designer a systematic guide to search for and categorize solutions, to force solution uniqueness or isolate particular solutions, preferably *without* actually solving any of the subsystems. The reason for this was pointed out earlier: even small subsystems are often challenging to solve, and it is worthwhile to plan the bifurcation choices apriori, as opposed to a trial and error approach for searching through solution possibilities.

Past data can be highly useful in this context. For example, before solving or realizing a commonly occuring type of cluster or feature, it would help to display the the most-often-chosen solution possibility, or list of solution possibilities of this feature type, based on previous realizations of features of the same type. While the displayed realization could be based on somewhat different numerical parameter values than the actual cluster or feature that is currently being realized, such a display generically conveys important, guiding information concerning topological types of the possible realizations. (Here again, as mentioned in Section 1.1, we do not consider algebraic degeneracies that could cause unexpected topological changes in the realizations, even on small numerical perturbations. Those are best treated by algebraic-numeric techniques outside the scope of our current discussion).

Often, idealized global constraints such as symmetry are additionally thrown in at this realization stage, since they force a unique solution possibility, while ensuring design intent, since symmetric configurations are often desirable both physically, due to minimum energy requirements, as well as in aesthetic applications. Such past information about successful use of global constraints during realization is also valuable. These considerations lead to the following computational problem formulation.

Problem 3.1

Input: *K, a complete geometric constraint problem, with all numerical parameter values included, containing n geometric objects; and a DR-plan T whose maximum fan-in is within a $c \geq 1$ factor of the optimal;*

Past Data: *a set P_n of at most k tuples (K_i, A_i), and a list L_n of at most k significant feature types; here K_i are geometric constraint problems on at most n geometric objects, and A_i are the corresponding c-factor-optimal DR-plans that have been solved or realized. More precisely, the A_i contain all the information of the DR-plan for K_i, augmented as follows. Each vertex in A_i contains two additional pieces of information.*

The first is a list of solutions of the cluster corresponding to that vertex; each such solution is based on either the most commonly occuring solution possibility of each child; this could be generalized to look ahead l lower levels, i.e, based on the most commonly occuring solution possibilities of the l^{th} generation descendants. In some cases, the most commonly occuring solution possibility is the replaced by the unique solution possibility that resulted from the application of a global constraint such as symmetry; in such cases, the parameters of the global constraint are specified.

The second piece of information at each cluster u of A_i is the most commonly occuring realization of u. Again, this solution could be the result of the application of a specific global constraint, whose parameters are given.

As in Problem 2.3, higher level feature types in L_n can be represented as a feature hierarchy forest, containing lower level constituent features, also from L_n. And as in Problem 2.3, there is a reference pointer from each element e of L_n both to its constituent elements in L_n, and to a subforest of one of the A_i, which corresponds to the realized or solved DR-plan for e.

Output: *A solution or realization of T that has the desired form intended by the designer: this could be either the most commonly occuring realization, or the unique realization forced by specific global constraints such as symmetry.*

The time complexity measure $T_c(n, k)$ still applies; however, Problem 3.1 extends only a subproblem of Problem 2.3, since the DR plan T is already furnished as part of the input. During the bottom-up solving phase, the solution of one of T's clusters u is interspersed with a computation that locates the closest feature type from L_n that corresponds to u (i.e, that vertex and subtree) of T. These are used to link and display a list of solution types through the A_i's. This allows the correct choice of solution possibility of each of the child clusters of u, allowing u to be realized in a tractable manner, without a combinatorial explosion. This could still result in more than one realization of u. However, at the next stage, when u's parent cluster is being resolved, a single candidate is chosen in

the same manner from list of u's realizations, as well as those of its siblings.

Now we formalize four other problems by specifying their exact relationship to Problem 3.1.

Valid realizations of underconstrained systems

In Section 2, we discussed the related problem of how to use past data to augment an underconstrained system with additional constraints and objects in order to make it wellconstrained and at the same time obtain an optimal DR plan for it. However, we have so far only dealt with combinatorial and type information for these augmenting constraints. It is the actual numerical parameter values of these constraints that determine the realizability of the resulting wellconstrained augmentation.

Without the availability of past data, determination of these numerical values is typically achieved as follows. See [38]. Since the optimal DR-plan for the wellconstrained augmentation is known, we proceed with the bottom-up realization phase discussed earlier in this section. However, now the clusters have constraints whose numerical parameters are unknown. We proceed by fixing them arbitrarily, solving the cluster if possible, backtracking when an inconsistency is located, reassigning some of these values judiciously, and proceeding iteratively.

The simple but classic example of this type is the resolution of a n distance constraints connecting n points $v_1, \ldots v_n$, forming a large cycle as the constraint graph. The wellconstrained augmentation adds $n-3$ additional distance constraints relating v_1 to $v_3, \ldots, v_{n-1}$ to triangulate this cycle. The DR-plan is a single path consisting of $n-2$ cluster-vertices, with the leaf cluster consisting of the points $\{v_1, v_2, v_3\}$ and i^{th} cluster on the path consisting of the $(i-1)^{st}$ cluster along with the vertex v_{i+2}. By arbitrarily fixing the added distances to conform to the triangle inequality, valid realizations can be found for all the clusters except possibly the root cluster. However, while a brute force backtracking search can combinatorially explode, in fact there is a single backtracking sequence, with judicious (re)settings of the additional distance values, which allows the root cluster to be solved efficiently (or allows us to determine that the original cycle of n distance constraints is unrealizable.) Useful past data that have been developed for the resolution of commonly occuring types of underconstrained systems - for example, an effective backtracking sequence, and good settings of unknown parameter values as a function of the other parameters etc - can be succinctly stored and used.

More formally, the problem is a minor generalization of Problem 3.1, and can be tackled in a similar manner. The generalization is the following: where the input constraint problem K may have unknown numerical parameter values, as do the the clusters in T. The output sought is the realization of the clusters in T based on *some* setting of these parameter values. Similarly, the past data now consists of K_i with unknown parameter values, whereas the A_i are based on some setting of these values. In addition, the past data includes the following information for each cluster u in A_i: the exact iterative sequence in which the parameter values were set, including resettings that occur when backtracking down the subtree under u; and an easily computable function which (based on heuristic) gives each (re)setting of a parameter value in this iterative sequence, in terms of parameter values that are already set. This additional past data information can be viewed as the description of a procedure to be followed in setting the unknown numerical parameter values for commonly occuring types of underconstrained features, in order to ensure valid realizability. Such a procedure is used to resolve each cluster u in the current input DR-plan T. The procedure is found by locating a commonly occuring feature closest to u, using the list L_n, and then linking to the information stored in the A_i's, as described in Problem 3.1.

Automatic modification of generated features

The second highly related problem we consider is the following. It is a common occurence during the design process that the designer is interested in whether certain special features - such as two line-segments meeting at an acute angle - is generated as a result of the resolution or realization of other constraints. If the feature is in fact generated, the designer may wish to further constrain its constituent geometric elements, for example, the acute angle is rounded off by the introduction of an arc object and tangency constraints. See [38] for approaches to these problems without using past data. It is useful to automate this process, based on past information.

This leads to another minor generalization of Problem 3.1, and can be tackled similarly. The past data should now additionally contain a list M_n of those feature types that were usually modified when they occured during the realization of previous constraint problems K_i. The list M_n is organized with reference pointers in a manner similar to the list L_n as described in Problem 2.3. As in the first variant of Problem 3.1 discussed above, A_i's vertices - those that are referred by a feature in M_n - now contain a description of how the feature and its forest of subfeatures was modified in the past. i.e, what new objects and constraints were then introduced to modify those (sub)features, and how numerical parameter values were assigned to them. At run time, when each cluster in the input DR plan T is realized in a bottom-up fashion, these special features are detected and located in M_n as they are generated, and the feature modification procedure is automatically applied to them.

Soft constraints

We briefly consider a third related question on the use of past data for dealing with (a) certain types of approximate constraints given as inequalities; (b) fuzzy, optional constraints that are weighted and correspondingly penalized for not being satisfied; and (c) constraints whose meaning and exact parameter values need to be inferred by their approximate position on the sketch screen. All of these types of constraints allow a user to give *conceptual, inexact specifications*. All of these cases reduce to previously discussed problems. We explain the reductions below. Approaches to these problems without the use of past data can be found in [38].

Case (a) can be split into two further subcases. First, it is determined during the DR-planning phase if the inequalities represent redundant constraints, i.e, whether they are part of a minimal defining constraint set, as discussed in Section 2. If not, then they are treated like constraints with unknown parameter values in a wellconstrained augmentation of an underconstrained system, reducing this to the problem of finding a valid realization of such a system. The only difference is that the (re)settings of the parameter values for these constraints should now satisfy the corresponding inequalities.

If, on the other hand, the inequalities do represent redundant constraints in a cluster u, then they are first ignored during the realization of u. Thereafter it is determined by measurement if the inequality is in fact satisfied for atleast one of the solution possibilities for u, the current cluster. If not, a search through the solution possibilities of the child clusters ensues, recursively travelling down the subtree under u. Again, this is similar to the problem of choosing good numerical parameter values for underconstrained systems, and this search can result in a combinatorial explosion. A good search order can be obtained by locating past data on similar redundant constraints with similar inequalities arising within similar features or clusters.

It is possible that both the above subcases coexist, with some inequality constraints being redundant and others not. In this case, when trying to enforce the inequality on a redundant constraint in a cluster u, the above search through u's subtree in the DR-plan is now amalgamated with the process of resetting the parameter

values for the nonredundant constraints as in problem of finding a valid realization of an underconstrained system.

In Case (b), the objective is to minimize an overall penalty which is usually a simple and direct function of the weights $p(e)$ of the constraints e that are not satisfied; this function usually also depends on the distance of the given parameter value d of a constraint e to the closest parameter value d^* that would ensure satisfiability of e. In other words, the penalty function depends directly on $p(e)|d - d^*|$. The problem is nontrivial only if the complete constraint problem is inconsistent, i.e, if it has no valid realization (this could occur even if the problem is underconstrained). The problem reduces to a combination of two problems discussed earlier: finding wellconstrained augmentations and valid realizations, respectively, of underconstrained systems. By removing all the fuzzy constraints and throwing in the global constraint requiring the minimization of the given penalty function, we first find a wellconstrained augmentation of the problem as described in Section 2. Following this, we treat the augmenting constraints as having unknown parameter values, and obtain a valid realization of the augmented wellconstrained system.

To see that Case (c) is largely a subcase of (a) and (b) consider the following. The measurements of the designer's sketch can be read off the screen within an acceptable margin of error. The interpretation of these measurements into actual constraint values is best done by treating them as constraints that are both fuzzy and approximate. I.e, their parameter value has a range, resulting in inequalities which can be treated as in Case (a), and moreover, there is a penalty weight attached to the constraint, which is inversely proportional to the probability or confidence that the inferred constraint was indeed intended by the designer. These weights and the penalty for not satisfying the corresponding constraints are treated as in Case (b). The penalty weights to be assigned are parameters that can be learned from past data using a standard neural net or Bayesian learning algorithm [29, 32].

4 Conclusions

We have motivated and precisely formalized 4 problems in the MCAx context, which deal with the usefulness of past data for exploiting the full potential of declarative geometry representations and for capturing design intent. We have taken particular care to quantify the usefulness of past data. We have shown how several other problems can be reduced to these 4 primary problems, and furthermore we have indicated the crucial steps and approaches for solving them.

References

[1] S. Ait-Aoudia, R. Jegou, and D. Michelucci. Reduction of constraint systems. In *Compugraphics*, pages 83–92, 1993.

[2] V. Allada and S. Anand. Feature-based modeling approaches for integrated manufacturing: state-of-the-art survey and future resear ch directions. *International Journal for Computer Integrated Manugfacturing*, 8:411–440, 1995.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. A geometric constraint solver. *Computer Aided Design*, 27:487–501, 1995.

[5] W.F. Bronsvoort and F.W. Jansen. Multiview feature modeling for design and assembly. In J.J. Shah, M. Mantyla, and D.S. Nau ed.s, editors, *Advances in Feature Based Modeling*, pages 315–330. Elsevier Science, 1994.

[6] B. Bruderlin. Constructing three-dimensional geometric object defined by constraints. In *ACM SIGGRAPH*. Chapel Hill, 1986.

[7] G. Brunetti and B. Golob. A feature based approach towards an integrated product model including conceptual design information. *Computer Aided Design*, 32:877–887, 2000.

[8] G. Crippen and T. Havel. *Distance Geometry and Molecular Conformation*. John Wiley & Sons, 1988.

[9] R.P. Zuffante D.C. Gossard and H. Sakurai. Representing imensions, tolerances, and features in mcae systems. *IEEE Computer Graphics and Application*, 8:51–59, 1988.

[10] K.J. de Kraker, M. Dohmen, and W.F. Bronsvoort. Multiway feature conversion to support concurrent engineering. In *ACM Symposium on Solid Modeling*, pages 105–114. ACM press, 1995.

[11] K.J. de Kraker, M. Dohmen, and W.F. Bronsvoort. Maintaining multiple views in feature modeling. In *ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 123–130. ACM press, 1997.

[12] C. Durand. *Symbolic and Numerical Techniques for Constraint Solving*. PhD thesis, Purdue University, Computer Science Dept, 1998.

[13] L. Eggli, C. Hsu, G. Elber, and B. Bruderlin. Inferring 3d models from freehand sketchers and constraints. *Computer Aided Design*, 29:101–112, 1997.

[14] I. Fudos. *Geometric Constraint Solving*. PhD thesis, Purdue University, Dept of Computer Science, 1995.

[15] I. Fudos and C. M. Hoffmann. Correctness proof of a geometric constraint solver. *Intl. J. of Computational Geometry and Applications*, 6:405–420, 1996.

[16] I. Fudos and C. M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Trans on Graphics*, pages 179–216, 1997.

[17] T. Havel. Some examples of the use of distances as coordinates for Euclidean geometry. *J. of Symbolic Computation*, 11:579–594, 1991.

[18] C. M. Hoffmann. Solid modeling. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook on Discrete and Computational Geometry*. CRC Press, Boca Raton, FL, 1997.

[19] C. M. Hoffmann and R. Joan-Arinyo. Distributed maintanence of multiple product views. *Manuscript*, 1998.

[20] C. M. Hoffmann and J. Rossignac. A road map to solid modeling. *IEEE Trans. Visualization and Comp. Graphics*, 2:3–10, 1996.

[21] Christoph M. Hoffmann, Andrew Lomonosov, and Meera Sitharam. Decomposition of geometric constraints systems, part i: performance measures. *Journal of Symbolic Computation*, page in press, 1998.

[22] Christoph M. Hoffmann, Andrew Lomonosov, and Meera Sitharam. Decomposition of geometric constraints systems, part ii: new algorithms. *Journal of Symbolic Computation*, page in press, 1998.

[23] Christoph M. Hoffmann, Andrew Lomonosov, and Meera Sitharam. Geometric constraint decomposition. In Bruderlin and Roller Ed.s, editors, *Geometric Constraint Solving*. Springer-Verlag, 1998.

[24] Christoph M. Hoffmann and Pamela J. Vermeer. Geometric constraint solving in R^2 and R^3. In D. Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry*. World Scientific Publishing, 1994. second edition.

[25] Christoph M. Hoffmann and Pamela J. Vermeer. A spatial constraint problem. In *Workshop on Computational Kinematics*, France, 1995. INRIA Sophia-Antipolis.

[26] C. Hsu, G. Alt, Z. Huang, E. Beier, and B. Bruderlin. A constraint-based manipulator toolset for editing 3d objects. In *1997 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, 1997.

[27] C. Hsu and Bruderlin B. A degree of freedom graph approach. In *Geometric Moeling: Theory and Practice*. Springer Verlag, 1997.

[28] Ching-Yao Hsu. *Graph-based approach for solving geometric constraint problems*. PhD thesis, University of Utah, Dept. of Comp. Sci., 1996.

[29] S. Judd. *Neural Network Design and the Complexity of Learning*. The MIT Press, 1990.

[30] M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, 1994.

[31] G. Kramer. *Solving Geometric Constraint Systems*. MIT Press, 1992.

[32] P.D. Laird. *Learning from Good and Bad Data*. Kluwer Academic publishers, 1988.

[33] G. Laman. On graphs and rigidity of plane skeletal structures. *J. Engrg. Math.*, 4:331–340, 1970.

[34] R. Latham and A. Middleditch. Connectivity analysis: a tool for processing geometric constraints. *Computer Aided Design*, 28:917–928, 1996.

[35] Andrew Lomonosov and Meera Sitharam. Approximation algorithms for optimal constraint decomposition. In *in preparation*, 2000.

[36] M. Mantyla, J. Opas, and J. Puhakka. Generative process planning of prismatic parts by feature relaxation. In *Advances in Design Automation, Computer Aided and Computational Design*, pages 49–60. ASME, 1989.

[37] A. Middleditch and C. Reade. A kernel for geometric features. In *ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*. ACM press, 1997.

[38] J. J. Oung and M. Sitharam. A fast, simple solver for some geometric constraint problems. Technical report, University of Florida, CISE department, 2001.

[39] J. J. Oung, M. Sitharam, B. Moro, and A. Arbree. Frontier: fully enabling geometric constraints for feature based design. In *ACM Symp. on Solid Modeling*, Ann Arbor, Michigan, to appear, 2000.

[40] J. J. Oung, M. Sitharam, B. Moro, and A. Arbree. Full technical report on the frontier geometric constraint solver. Technical report, University of Florida, http://www.cise.ufl.edu/ sitharam/full-frontier.ps, 2000.

[41] J. Owen. Algebraic solution for geometry from dimensional constraints. In *ACM Symp. Found. of Solid Modeling*, pages 397–407, Austin, Tex, 1991.

[42] J. Owen. Constraints on simple geometry in two and three dimensions. In *Third SIAM Conference on Geometric Design*. SIAM, November 1993. To appear in Int J of Computational Geometry and Applications.

[43] J.A. Pabon. Modeling method for sorting dependencies among geometric entities. In *US States Patent 5,251,290*, Oct 1993.

[44] A. Rappoport and M.J.G.M. van Emmerik. User interface devices for rapid and exact number specification. *ACM Transactions on Graphics*, 12:348–354, 1993.

[45] Jaroslaw P. Rossignac. Constraints in constructive solid geometriy. In *Workshop on Interactive 3D Graphics*, pages 23–24. Chapel Hill, 1986.

[46] M. Sitharam and T. Straney. Derandomized learning of boolean functions. In *Proc. of* 8^{th} *International Workshop, ALT*. Springer Lecture Notes in Artificial Intelligence, Vol. 1316, 1997.

[47] M. Sitharam and T. Straney. Derandomized learning of boolean functions finite abelian groups. In *International Journal on Foundations of Computer Science, accepted*, 2000.

[48] M. Sitharam and T. Straney. Sampling boolean functions over abelian groups and applications. In *Applicable Algebra in Engineering, Computation and Communication, accepted*, 2000.

[49] W. Sohrt and B. Bruderlin. Interaction with constraints in 3d modeling. *International Journal of Computational Geometry and Application*, pages 405–425, 1991.

[50] W. Whiteley. Matroids and rigid structures. In *Matroid Applications*, pages 1–53. Cambridge University Press, 1992.

[51] W. Whiteley. Rigidity and scene analysis. In *Handbook of Discrete and Computational Geometry*, pages 893 –916. CRC Press, 1997.