Automating the Construction of Patchers That Satisfy Global Constraints

Kerstin Voigt*
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
ARPAnet:voigt@aramis.rutgers.edu

Chris Tong*
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
ARPAnet:ctong@aramis.rutgers.edu

Abstract

Generate-and-test algorithms to solve constraint satisfaction problems are often inefficient, but can be constructed fairly easily by knowledge compilation techniques that convert declarative problem knowledge and domain knowledge into a procedural format [Liew and Tong, 1987]. Current research is focusing on methods to improve the efficiency of generate-and-test algorithms by completely incorporating *local* constraints into generators of *parts* of composite solutions [Braudaway, 1988]. More *global* constraints on multiple parts cannot necessarily be incorporated into the part generators. Their satisfaction must be ensured in a different way.

We describe an (unimplemented) method for transforming a generate-and-test algorithm into a generate-test-and-patch algorithm that efficiently hillclimbs toward a solution satisfying a particular global constraint. Our method is based on constructing an evaluation function from the global constraint, that reflects the "degree" to which the constraint has been satisfied. Some of the steps in this method rely on *categorizing* the global constraint into a generic class. In this paper, the constraint classes on which we focus are quota-meeting and covering constraints.

We illustrate the general approach by applying it to a simple generate-and-test algorithm for house floorplanning. We provide empirical results that corroborate our claim that the efficiency of the algorithm has been significantly improved.

*The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number N00014-85-K-0116 in part by the National Science Foundation (NSF) under Grant Number DMC-8610507, and in part by the Center for Computer Aids to Industrial Productivity (CAIP), Rutgers University, with funds provided The opinions expressed in this paper are those of the authors and do not reflect any policies, either expressed or implied, of any granting agency.

Currently on leave at IBM Watson Research Center.

Room lengths must be at least minValue.	(C1)
Room widths must be at least minValue.	(C2)
Rooms have to be inside the house.	(C3)
Rooms must be adjacent to the house boundary.	(C4)
Rooms must not overlap.	(C5)
The rooms must completely fill the house space.	(C6)

Figure 1: Constraints on house floorplans

1 Introduction

In this paper, we address part of the problem of *compiling* a declarative representation of a class of problems and knowledge relevant to solving it into an efficient problem-solving system.

Example domain. We will use the domain of *house floorplanning* to illustrate several domain-independent ideas for addressing this research problem. Floorplans are arrangements of rectangular rooms in a rectangular house space. Rooms and houses are aligned with an integer-valued grid of points in a plane. The constraints in Figure 1 define a particular *class* of floorplanning problems. The number of rooms, the dimensions of the house space, and minimum values for room lengths and widths are *problem-specific* parameters. [Liew and Tong, 1987] h as shown how to automatically construct a simple but inefficient generate-and-test algorithm that solves particular problems for a problem class such as this one. Given the length and width of the house, and n, the number of rooms, the algorithm generates n room rectangles, and then tests the candidate solution against each of the above constraints. Failure results in chronological backtracking.

Constraint incorporation. Various methods have been developed for automatically improving the efficiency of a generate-and-test algorithm [Mostow, 1983, Tappel, 1980, Braudaway, 1988]. One of the best improvements is to *completely incorporate* the constraints in the generator, that is, to modify the generator so that it only produces objects satisfying the constraints. When a constraint has been completely incorporated, the test corresponding to the constraint can be removed. The method described in [Braudaway, 1988] completely incorporates those constraints which are *local* to individual room generation (C1 - C4), or that can be localized to the set of rooms generated thus far (C5).

Unfortunately, the more global constraints (e.g., C6) are not readily incorporated into generators by such methods. Modifying a generator that only generates a solution *pari* (e.g., an individual room) generally is insufficient to guarantee satisfaction of a constraint on the entire solution.

The research problem. We define *local* constraints to be those which can be incorporated into a solution generator in such a way that the resulting generator runs in polynomial time. *Global* constraints are constraints that are not local. Our research is aimed at constructing <u>an efficient procedural embedding for global constraints:</u>

GIVEN
: a set CSET of constraints on the solution; declaratively represented domain knowledge elucidating the constraints CSET; a constrained generator in which all constraints in CSET' CCSET have been completely incorporated;

FIND
: a procedural embedding of the unincorporated constraints CSET - CSET' such that the resulting algorithm is significantly more efficient than embedding CSET - CSET' solely as tests.

The domain knowledge is provided by a human knowledge engineer, whose task we aim to simplify by only requiring that the knowledge be in a declarative form. Our method requires knowledge about the *solution part hierarchy* (e.g., solutions are floorplans; floorplans are houses having rooms as parts; rooms have four parameters as parts: length, width, xCoord, and yCoord), *solution part typing* (e.g., rooms and houses are rectangles; the xCoord of a room is a coordinate; rooms and gridpoints inside rooms are space-filling units; the house and gridpoints inside the house are space units,[1] and *generic knowledge* (e.g., definitions of predicates such as inside; items of type "coordinate" can take on any integer value between [-maxlnt,maxlnt]).

The constrained generator is produced by the RICK knowledge compiler described in [Braudaway, 1989]; CSET - CSET' are constraints which that compiler could not completely incorporate.

In this paper, we illustrate a method that optimizes a generate-and-test floorplanner into a generate-test-and-*patch* floorplanner. When a candidate solution fails to satisfy a global constraint, it is passed to a patcher, which hillclimbs its way to solutions that are increasingly better with respect to that constraint. For instance, failure to satisfy the constraint C6 means the house space contains "holes" unassigned to any room. The patcher incrementally reduces the amount of unassigned space by extending rooms.

2 Categorization of global constraints

The English statement of C6, "The rooms must completely fill the house space", seems "global" in that it appears we must simultaneously consider all the rooms to determine whether it is satisfied. However, when we express the constraint more formally.

VF,P {floorplan(F) A gridpoint(P,F) A inside(P,house,F)(C6)
  => 3R [room(R,F) A inside(P,R,F)]}

we notice that the constraint is only existentially (and not universally) quantified over rooms R. *Ignoring* CI through C5 would allow C6 to be satisfied by a single room that covered the entire house. What is more accurately called a "global" constraint is the *conjunction:* C7<=>C3 A C5 A C6. Together, these three constraints define a *covering* relationship that must be satisfied. C6, by itself, could also be called "global" with respect to a constrained generator in which C3 and C5 have already been incorporated.

One of our major long-term goals is to construct *global constraint schemas* or *normal forms* for a diverse range of global constraints. Such constraint schemas may contain multiple components (e.g., Figure 2 contains three components). A successful match of a constraint schema S to a set of constraints occurs when each schema component matches a constraint in the set. The conjunction of the matching constraints (e.g., C7) is "a global constraint of type S". Constraint schemas S can have associated knowledge for helping to construct a patcher that eventually produces a solution that satisfies a constraint of type S.

In this paper, we focus on global constraints that can be viewed as constraints on *global resources.* In particular, certain parts of our approach are currently restricted to fit two generic global constraint schemas: *quota-meeting constraints* and *covering constraints.*

Quota-meeting constraints. Quota-meeting constraints have the following schema:

(QNF)
VS [solution(S) => cardinality(unitsOfType(t,S)) = quota]

where quota is an integer, quota>0, t is a given unit type, and the number of units of type t is always less than or equal to quota. Quota-meeting constraints require that the solution contain exactly quota units of type t. C7 is an example of a constraint from which we can construct another constraint C8 such that C8 and C7 are logically equivalent and C8 matches QNF (Quota Normal Form); for instance, C8 could be "the number of gridpoints in the filled part of the house must equal the total number of gridpoints in the house."

Covering constraints. Satisfying a covering constraint involves constructing an assignment:

> Assign units of type T_{SF} to units of type T_5
> so that the units of type T5 are covered.

We will call units of type T_{SF} *space-filling units*[2] and units of type T5 *space units.* Covering constraints have the normal form given in Figure 2, consisting of the conjunction of three constraints. The number of space units and space-filling units is finite, all candidate solutions share the same set of space units, and the space-filling units (and how they are assigned) varies, depending on the solution. The predicate corresponds defines the assignment. Examples of constraints that could be

[1] The utility of this kind of domain knowledge will become clear in section 3.

[2] We will use the notational convention of capitalizing variable names.

Injection **(SNF-I)**

$\forall X1,X2,Y,S$ [**solution**(S) $\wedge$ **spaceUnit**(Y,S)
$\wedge$ **corresponds**(X1,Y,S) $\wedge$ **spaceFillingUnit**(X1,S)
$\wedge$ **spaceFillingUnit**(X2,S) $\wedge$ **corresponds**(X2,Y,S)
$\Rightarrow$ **equal**(X1,X2)]

Complete assignment **(SNF-II)**

$\forall S,X$ {**solution**(S) $\wedge$ **SpaceFillingUnit**(X,S)
$\Rightarrow \exists Y$ [**spaceUnit**(Y,S) $\wedge$ **corresponds**(X,Y,S)]}

Complete covering **(SNF-III)**

$\forall S,Y$ {**solution**(S) $\wedge$ **spaceUnit**(Y,S)
$\Rightarrow \exists X$ [**spaceFillingUnit**(X,S) $\wedge$ **corresponds**(X,Y,S)]}

Procedures

Re-express constraint to match QNF form.

Figure 2: The SNF constraint schema

matched to SNF (Space-filling Normal Form) include: "All available jobs must be assigned a person to carry them out"; "All available time slots must be assigned a job"; etc.

C7 can be written in SNF form because C3, C5, and C6 can be shown to match II, I, and III, respectively. After being re-written in SNF, C7 is a covering constraint where the space units are the gridpoints in the house rectangle, and the space-filling units are the gridpoints in all the rooms. The assignment is the obvious one: if a room gridpoint is "on top of" a house gridpoint, the two gridpoints correspond.

Constraint schemas, logically equivalent constraints, and reformulation procedures. Given any constraint C_a of type SNF, we can construct a logically equivalent constraint C_b of type QNF. This is because SNF-II and SNF-III together imply that the total number of space units and the total number of space-filling units must be the same (call it n); if C_a is satisfied, then *all* the space units have been covered by space-filling units and therefore the number of *covered space units* is n. Thus by defining C_b to be "The number of covered space units equals the total number of space units", C_b is implied by C_a and C_b matches the QNF schema. Conversely, if C_b is true, then all the space units have been covered, and covering constraint C_a is satisfied.

Thus we can associate with schema SNF a *procedure* that takes a constraint matching SNF and constructs a logically equivalent constraint matching QNF. The use of such procedures will be demonstrated in the next section.

3 Incremental Construction of a Patcher

Our approach to the research problem is to compile the unincorporated constraints into a *knowledge-based solution patcher.* The key steps in our approach are now outlined (see also [Voigt, 1988]). We have begun implementing this method in a program called MENDER. We illustrate the steps of our approach on the constraint C6, and presume the existence of a constrained generator in which constraints Cl through C5 have already been completely incorporated.

STEP 1. Extract an evaluation function from the global constraint. Constraints per se are predicates; when they are evaluatable, they are either "satisfied" or "not satisfied". A key to making effective use of a global constraint is to be able to recognize the *degree* to which it has been satisfied (e.g., to what degree has the house space been filled by rooms?). Thus we want to convert a predicate into an evaluation function, which can be used to guide a hillclimbing patcher.

Some constraint schemas (e.g., QNF) contain procedures for creating an evaluation function corresponding to a constraint (or set of constraints) that fit the schema. However, we may discover that a constraint C_a is expressed in a form that does match a schema S (e.g., SNF), but S does not contain a procedure for constructing an evaluation function.

However, S may have an associated procedure for constructing a constraint C_b that is logically equivalent to C_a but which matches S', a constraint schema that *does* contain a procedure for constructing an evaluation function f measuring improvement with respect to satisfaction of C_b. The important observation is that, if f takes on its maximum value only in states when C_b is true, then f will also aid in hillclimbing to a solution that completely satisfies C_a, because C_b and C_a are logically equivalent. The purpose of the patcher, is to eventually produce a state in which C_a is *completely* true. Thus we don't insist that C_a necessarily become "more and more true" as f increases its value, so long as C_a is completely true when f takes on its maximum value.

We will now illustrate these points by matching constraints to the SNF schema, re-expressing them in QNF, and then constructing an evaluation function.

Re-expressing constraints to match SNF. Because of the large number of possible matches, *syntactic* matching is used to suggest plausible constraint/constraint schema component matches in a cost-effective manner. The matching leaves in the parse trees of these plausible matches correspond to theorems which are then proved to verify that the constraint fits the constraint schema (see, e.g., Figure 3).

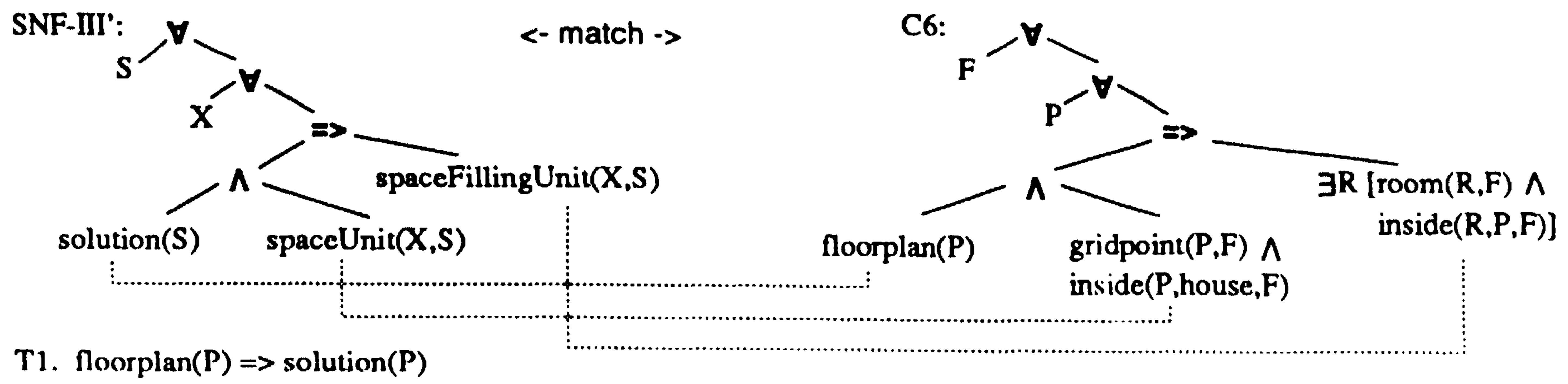
We illustrate this process by matching constraint C6 against the components of the SNF schema. In general, schema components can contain special cases. If the prototypical case fails to match, the special cases are tried. C6 does not match any of the SNF prototypical components. However, it does match a special case of SNF-III (see Figure 3):

$\forall S,X$ [**solution**(S) $\wedge$ **spaceUnit**(X,S) (SNF-III')
$\Rightarrow$ **spaceFillingUnit**(X,S)] .

where the space units and space filling units arc drawn from the same set of objects, and corresponds(X,Y,S) is equals(X,Y).

The fit of C6 to SNF-III' is then confirmed by proving the three theorems suggested by the match, using the domain knowledge indicating that house gridpoints are space units and room gridpoints are space-filling units.

Since one component of SNF has been successfully matched, the search for matches next tries to find domain constraints to match the remaining components. It finds that C3 matches SNF-II and C5 matches SNF-I.



T1. floorplan(P) => solution(P)

T2. gridpoint(P,F) ∧ inside(P,house,F) => spaceUnit(P,F)

T.3 ∃R [room(R,F) ∧ inside(R,P,F)] => spaceFillingUnit(P,F)

Figure 3: Matching constraint C6 and schema SNF-III′

setOf(SpaceFillingUnit(X,s))

match: room(R,f) ∧ inside(P,R,f)

type definition:

A1. P is of type gridpoint: gridpoint(P,f) ↔ P=<X,Y> where X and Y are of type coordinate
t1: <X,Y> | X∈[-maxInt,maxInt] ∧ Y∈[-maxInt,maxInt]

A2. room(R,f):
t2: R ∈ rooms in floorplan f

A3. inside(<X,Y>,r,f):
t3(r,f): <X,Y> | X∈[xCoord(r,f),xCoord(r,f) + length(r,f)] ∧ Y∈[yCoord(r,f),yCoord(r,f) + width(r,f)]

A1∧A2∧A3:
t4: $\bigcup_{R \in t2}$ <X,Y>∈t3(R,f)

cardinality	of types:
t1	4 × maxInt × maxInt
t2	cardinality(input set of rooms)
t3(r,f)	length(r,f) × width(r,f)
t4	$\sum_{R \in t2}$ length(R,f) × width(R,f)

Figure 4: Converting predicates into types with cardinality

Matching has thus constructed constraint C7 (the conjunction of C3, C5, C6) and shown that it fits the SNF schema.

Re-expressing SNF constraints to match QNF. Our method next re-expresses C7 to match QNF using the following pre-compiled correspondence between SNF and QNF:

setOf(spaceFillingUnit(X,S))	unitsOfType(T,S)
cardinality(setOf(spaceUnit(Y,S)))	quota

The re-expression is done by converting predicate definitions into type definitions and then finding an expression for the cardinality of these types, as illustrated in Figure 4. The first half of re-expressing constraint C7 in QNF is thus given by:

cardinality(unitsOfType(t4,s))
  = cardinality(setOf(spaceFillingUnit(X,s)))
  = $\sum_{R \in roomsinfloorplanf}$ length(R,f) × width(R,f)

In a similar, tightly focused manner, we derive the other half (not shown here):

quota = cardinality(setOf(spaceUnit(Y,s)))
  = length(house,f) × width(house,f)

Finally, the QNF schema contains the information that the desired evaluation function f(S) is given by f(S) = cardinality(unitsOfType(t4,S)). The derived evaluation function for measuring progress in satisfying C7 is "the number of gridpoints in the filled part of the house":

f(F) = $\sum_{R \in roomsinfloorplanF}$ length(R,F) × width(R,F)

STEP 2. Eliminate <parameter,direction> pairs that do not immediately improve a solution with respect to the evaluation function. We assume the patcher eventually constructed by this method will make *patching moves* that vary only one parameter value at a time, either increasing or decreasing its value by some quantity. This patcher will hillclimb, thus requiring repeated improvement of the solution with respect to the evaluation function. We could easily construct an "unintelligent" patcher which simply allows any parameter to be varied, in any direction, by any amount. In the floorplanning example, these parameters and directions are:

±ΔxCoord(R), ±ΔyCoord(R), ±Δlength(R), ±Δwidth(R)

Such a patcher wastes time repeatedly determining that most patching moves do not improve the solution.

Fortunately, in many cases, we can prove (at compile time) that changing certain parameters in certain directions (e.g. increasing) *never* (immediately) improves the solution. The "theorems" we would like to prove take a very restricted form:

∀S [state(S) ⇒ f(S) ≤ f(patch(S,Par))]

where patch(S,Par) is the result of applying the patcher to parameter Par in state S. Where we can prove such theorems, we achieve a compile time speed-up by simply not including these patching moves in the patcher. The number of such prunable "silly parameter changes" tends to increase if we assume that certain local problem constraints (i.e. the ones that can be fully incorporated by other methods, as, for example, in [Braudaway, 1988]) are true before the patching move and must remain true after the patching move.

For example, suppose we assume that constraints C1 through C5 must hold before and after the patching move. We can then prove that the following patching moves will *not* (immediately) improve the solution:

±ΔxCoord(R), ±ΔyCoord(R), -Δlength(R), -Δwidth(R)

Moving a room I(by changing either its x or y coordinates) will keep fixed the amount of the house that is filled, since the room is not allowed to overlap any other rooms, or be placed outside the house. *Shrinking* a room (by decreasing either its length or its width) will decrease the amount ofthe house that is filled. After constructing proofs that reflect these observations (based on the definitions of the predicates), the incorporation method constrains the patcher to the only moves left, +Δlength(R) and +Δwidth(R).

STEP 3. Re-express the evaluation function in relative and local terms. We can re-express the evaluation function f for a patched solution, patch(S,Par), as:

$$f(patch(S,Par)) = f(S) + \Delta f(S,patch(S,Par))$$

where Af is the improvement due to the patch. Rewriting this, we have:

$$\Delta f(S,patch(S,Par)) = f(patch(S,Par)) - f(S)$$

The value of Af for a length-increasing patch of room R is derived to be:

Af(F,patch(F,length(R,F))) = width(R,F) x Alengthl(R,F)

We compute Af for each possible patching move. In this manner, we can evaluate potential patches without having to execute them. This will be needed for doing greedy patching, which is described in STEP 5.

STEP 4. Construct a patcher in which the local constraints have been incorporated. Because we assumed in STEP 2 that the patching moves do not violate any completely incorporated constraints, we must restrict the parameter value modifications ofthe patcher so as to guarantee this. The patcher has several components, one for each modifiable parameter. The "increase room length" component ofthe patcher must ensure that the longer room is still inside the house (C3), still does not overlap another room (C5), still is adjacent to the house boundary (C4), and still has dimensions at least minValuo long (C1,C2). Cl and C2 are obviously always true; C4 is guaranteed to be true because the <X,Y> corner has not been moved off the house boundary.

The RICK program [Braudaway, 1989] automatically constructs a constrained generator in which several local constraints have been incorporated. A function g(Par,F) constructed by RICK at compile time is attached to each floor plan parameter's range; this function dynamically recomputes the legal values for that parameter. The patcher is constructed by modifying the constrained generator produced by RICK. The patcher differs from the constrained generator only in that the range of legal patching moves in a particular direction (+/-) is the subset of the range of legal moves (computed by g(Par,F)) bounded by (but not including) the current value and varying in the desired direction as far as possible, while still satisfying the incorporated constraints (C3 and C5).

STEP 5. Convert the patcher into a greedy patcher. STEP 2 of this method only determines desirable directions for changing parameter values, but does not constrain the *amount* by which the parameter value should be varied. One easily implement able idea is to take a *greedy* approach; for a particular parameter, a greedy patcher simply tries values from that parameter's

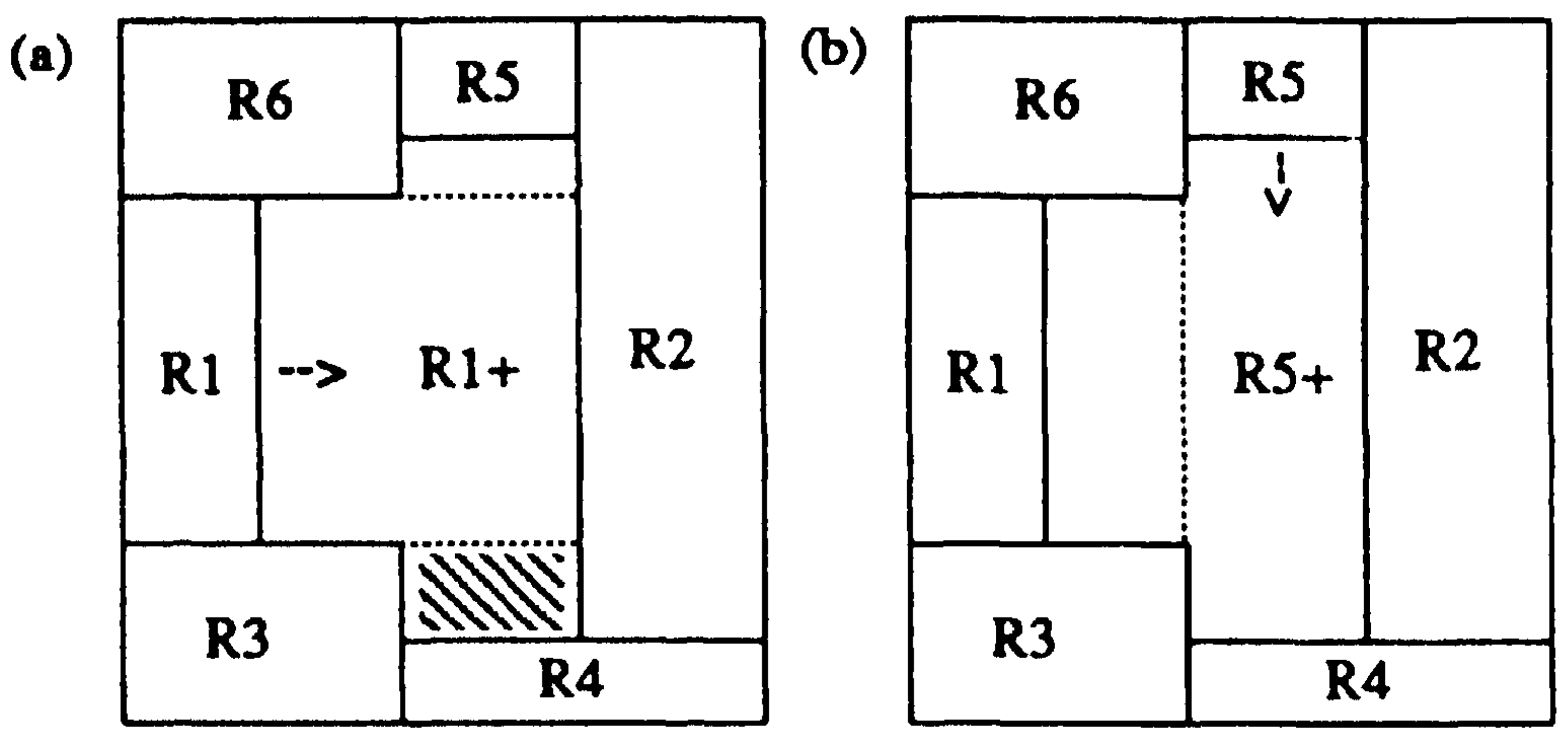


Figure 5: (a) Greedy patch creating an unsolvable subproblem (shaded area); (b) the corresponding block-preventing patch.

legal range, in *decreasing* order of improvement with respect to the evaluation function. From the re-expressed evaluation function created in STEP 3, we see that, for a proposed increase in the length of room r in state s, <r,Δlength>, the change in evaluation function value is:

$$\Delta f = width(r,s) \times \Delta length(r,s)$$

Since Af is proportional to Alength(r,s) (width(r) is not changed by the patch), the greatest evaluation function change occurs when the length change is greatest. Thus, the greedy patcher is made to select values in decreasing order of length. If Af had been *inversely* proportional to width(r,s) then the greedy patcher would be designed to select values in increasing order. In either of these special cases, no extra work is required to sort the parameter range values using the evaluation function. More generally, we can compile an expression for computing Af and sorting the range elements at run time.

STEP 6. Offset the negative aspects of the greedy patcher by trying to prevent blocking moves. The major problem with a greedy strategy is that it neglects to check for long-term negative consequences. Thus, as is illustrated in Figure 5, extending room Rl as far as possible to the right inadvertently *blocks* room R5 from being extended maximally in the downward direction. Unfortunately, the shaded area in Figure 5 is a hole that is *only* tillable by extending R5 into it. The problem with a greedy strategy is that it may solve one subproblem (e.g. filling in part of a hole) at the expense of rendering some later subproblem unsolvable.

Fortunately, we can offset some ofthe negative aspects of a greedy patching strategy by including a *look-ahead component* in the strategy. The purpose of this component is to detect patching subproblems that can *only* be solved by a *single* move in one way (e.g. a maximal increase in the width of room R5). *As soon as* we detect that some patching move has this property, we make that move, to prevent other patching moves from inadvertently *blocking* such a necessary move. We will call such moves *block-preventing moves.* For example, the unsolvable patching subproblem in Figure 5 (a) can be avoided only by extending R5 down to R4 (see Figure

5(b)).

Description of the look-ahead component. Given a covering constraint, the function of the patcher derived in STEP 1 through STEP 5 is to construct a complete assignment: "Assign space-filling units of type T_{SF} to space units of type T_s so that the units of type T5 are covered." The space units are invariant (e.g., the gridpoints in the house), while the space-filling units are created (and implicitly assigned) by the generator and patcher. We use a data structure D (e.g., an array) that contains a single element for each space unit. D will represent the current assignment. Before trying to detect possible block-preventing moves, we initialize D so that the space units currently filled by rooms are recorded. Making a patch move involves marking those space units in D that are filled in by the patch move; in particular, we place pointers to the patching move itself in the appropriate elements of D. We detect space units that can only be filled in by a single patching move in two steps. First, we update D to reflect the result of applying *all* patches applicable in the current state to the current state; then we look for elements of D that only have one associated patching move.

Construction of the look-ahead component. The most difficult part of automatically compiling the look-ahead component is to construct procedures that define and maintain the data structure D. Compiling a procedure for defining D is relatively straightforward. The cardinality C of the space units is presumed to be fixed and was computed in STEP 1. For our example, C is length(house,s)xwidth(house,s). D is an array with C records (one for each space unit), each having a "space unit" field (containing a pointer to the space unit that that record represents) and a "space fillers" field (a list of all space-filling units covering that space unit). Compiling a procedure to initialize D is based on the following information determined in STEP 1:

$$\text{setOf[spaceUnit(P,s)]} = \langle X,Y\rangle \mid X\in[0,\text{length(house,s)}] \wedge Y\in[0,\text{width(house,s)}]$$

The values for the "space unit" fields of the elements in D are initialized from this set.

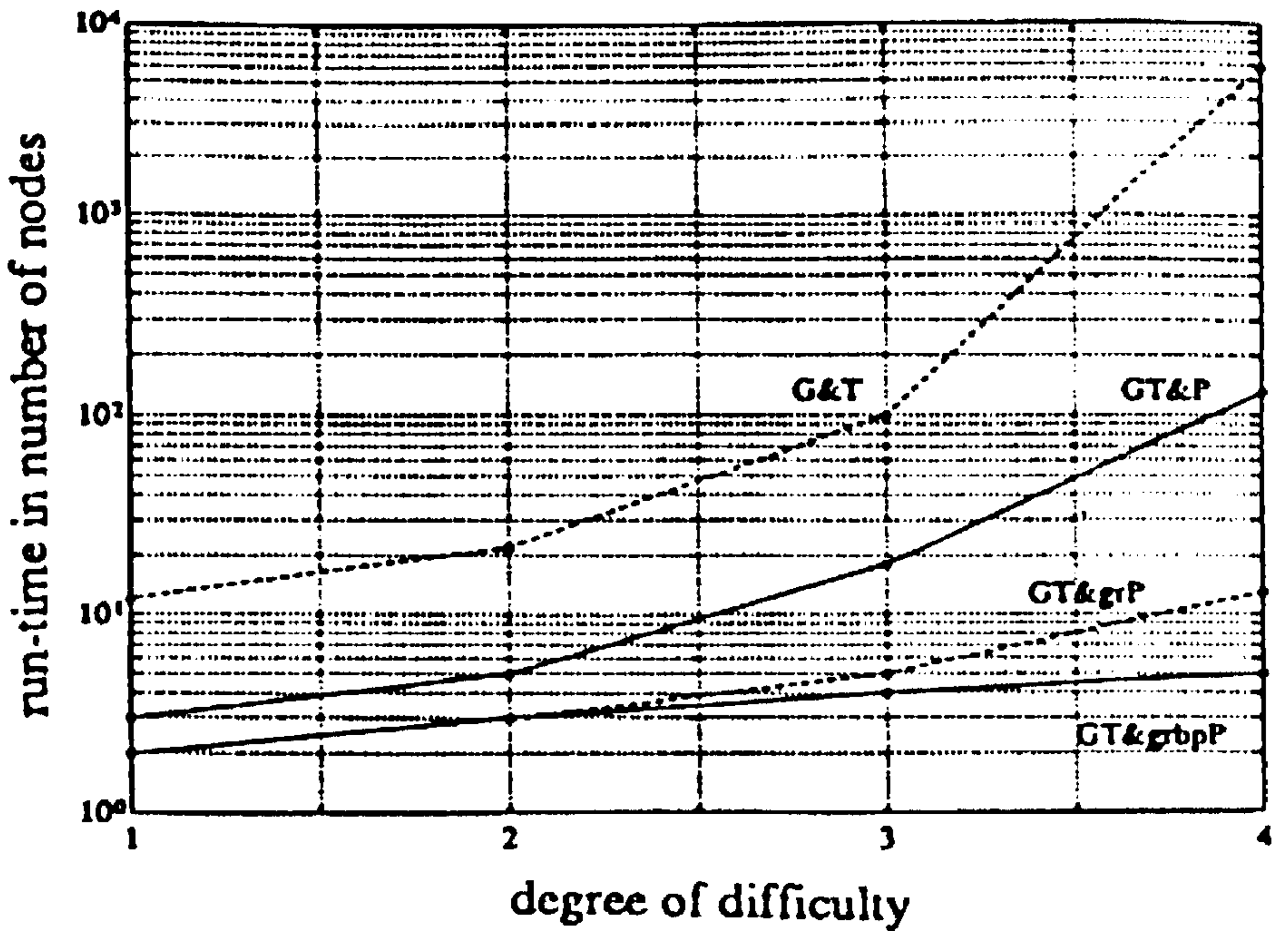
Compiling a procedure for maintaining D is based on the following information determined in STEP 1:

$$\text{setOf[spaceFillingUnit(P,s)]} = \bigcup_{R\in rooms in floorplan_f} P\in t3(R,f$$

From this we determine the change in the covering resulting from applying a patch:

$$\begin{aligned}&\text{setOf[spaceFillingUnit(P,patch(}\langle r,\text{length}\rangle\text{,s))]}\\&-\text{setOf[spaceFillingUnit(P,s)]} =\\&\langle X,Y\rangle \mid X\in[\text{xCoord(r)} + \text{length(r,s)},\\&\qquad \text{xCoord(r)}\\&\qquad + \text{length(r,patch(}\langle r,\text{length}\rangle\text{,s))]}\\&\quad \wedge\ Y\in[\text{yCoord(r,s)},\text{yCoord(r,s)} + \text{width(r,s)}]\end{aligned}$$

After each patch, the maintenance procedure computes the above set difference. For each space-filling unit in the set difference, it looks for the element D[i] containing the corresponding space unit. (Automatic programming techniques such as in [Kant and Barstow, 1978] can be used to optimize this operation.) It then adds the current patch to the value of D[i]'s "space fillers"



G&T: generate-and-test, **GT&P**: generate-test-and-patch **GT&grP**: generate-test-and-greedy-patch, **GT&grbpP**: generate-test-and-greedy-patch with block-preventing patches

Figure 6: Relative performance of algorithms

4 Empirical Results

We empirically evaluated and compared the performances of generate-and-test, generate-test-and-patch, generate-test-and-greedy-patch, and generate-test-and-greedy-patch with block-preventing patches on a test set of floorplans in which the rooms did not entirely fill the house area. We manually constructed the test set of floorplans to have varying degrees of *difficulty* (which we defined as the *length of the shortest path* of patching operations that would convert the floorplan into one that filled the house). In our study, the degree of difficulty of the example floorplans ranged between 1 and 4. The relative performances of the algorithms were measured by the *number of nodes* (floorplans) generated before finding a solution. Note that the cost of generating a node can be shown to take time polynomial in the area of the house.

The graph in Figure 6 clearly depicts that the three generate-test-and-patch algorithms explore consistently fewer nodes than generate-and-test. Generate-test-and-greedy patch with block-preventing patches performed best. For the examined degrees of difficulty, the number of generated nodes increased only linearly, reflecting the fact that block prevention succeeded in totally eliminating backtracking. Although block prevention may not achieve total elimination of backtracking in general, we have demonstrated its potential for significantly reducing backtracking.

5 Contributions and Limitations

Contributions. The approach we have presented is based on viewing satisfaction of a global constraint as an *optimization* process. The hillclimbing process is confined to points in the space of "feasible solutions" (i.e., those which satisfy the incorporated local constraints). The novel contribution of this paper is an approach to automatically compiling a global constraint satisfler (a "patcher") that is based on this hillclimbing viewpoint. To help relate our work to that of others, we can think of the compilation method as involving two major phases: (STEP 1) derive an evaluation function that (implicitly) defines a simple, operational patcher which varies one solution parameter value at a time; and (STEP 2 through STEP 6) optimize the patcher in various ways. We now discuss work related to each of these two phases.

1. Constructing evaluation functions. Much research has focused on constructing evaluation functions for algorithms (e.g., A*) that search *state spaces* (e.g., [Pearl, 1983]). These approaches derive evaluation functions from abstractions of state space operator preconditions. In contrast, our patcher moves in a *parameter value space,* where there are no restrictions on how to move from one state to another; the moves are not restricted by a given set of operators. Thus in some sense, our problem involves not only constructing an evaluation function, but also a set of patching operators to help restrict the search for a satisfactory solution. STEP 1 of our method constructs the evaluation function, based on categorizing the goal-defining conditions (constraints) into one of several generic classes. STEP 2 through STEP 6 can be thought of as constructing implicitly defined patching operators that take restricted forms (e.g., "greedily increase the length of room R"). Note that our approach to constructing evaluation functions could be applied to state space problems like the 8-puzzle. It would involve successfully classifying a problem into a known generic type (e.g., part re-configuration), and having associated with such generic problem types a procedure for constructing an evaluation function (e.g., "number of misplaced parts").

2. Optimizing the hillclimbing algorithm. Our research on constructing hill-climbing patchers is similar to other recent work [Lowry, 1987] on designing efficient optimization algorithms. Lowry illustrates his approach by deriving the simplex method for solving linear optimization problems. His derivation greatly exploits the fact that the space of feasible solutions is *convex.* If that space were not convex (e.g., as in the nonlinear floorplanning problem), many difficulties would arise: the feasible region is not necessarily connected (e.g., constraints such as C5 are disjunctions); it is not always possible to reach an optimal state starting from any feasible state; etc. For such spaces, a more robust patcher must be constructed.

Limitations and future research. We are in the process of implementing this method in the MENDER program, and applying it to global constraints drawn from the house floorplanning domain. The research in this paper has focused on covering constraints; the future research will flesh out the taxonomy of global constraint schemas that our approach can handle. Currently, we have worked out the details of STEP 1 and STEP 6 only for the class of covering global constraints. STEP 2 through STEP 5, however, are expressed in terms independent of a particular type of constraint; they may therefore apply to global constraints generally.

6 Conclusions

We have developed a method for procedurally embedding global constraints (that degrade the performance of generate-and-test algorithms) into a generate-test-and-*patch* algorithm. We have shown how the generate-test-and-patch algorithm can be further improved by allowing "greedy" patches, and by equipping the patcher with a mechanism for heuristically detecting block-preventing moves.

We have empirically shown that generate-test-and-patch algorithms are more efficient than generate-and-test for satisfying a covering constraint in the context of a simple house floorplanning task; we used a manually constructed set of test examples of varying difficulty. Using the same examples, we also demonstrated that further performance improvement is possible using a greedy patching strategy, and block-preventing patches to offset the disadvantages of greed.

Acknowledgements. We are grateful to Lou Steinberg, Jack Mostow, and Don Smith for their helpful comments. We would also like to thank the members of the KBSDE group, the Knowledge Compilation Seminar, and the Rutgers AI/Design Project.

References

[Braudaway, 1988] W. Braudaway. Constraint incorporation using constrained reformulation. Tech. Rpt. LCSR-TR-100, Computer Science Dept., Rutgers University, April 1988.

[Braudaway, 1989] W. Braudaway. Automated synthesis of constrained generators. In *Proceedings IJCAI-89,* Detroit, August 1989.

[Kant and Barstow, 1978] E. Kant and D. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge. *IEEE Transactions on Software Engineering,* 9:287-306, 1978.

[Liew and Tong, 1987] C. Liew and C. Tong. Knowledge compilation: A prototype system and a conceptual framework. AI/Design Project Working Paper No. 47, Computer Science Dept., Rutgers University, February 1987.

[Lowry, 1987] M. Lowry. Algorithm synthesis through problem reformulation. In *Proceedings AAAI87,* Seattle, WA, July 1987.

[Mostow, 1983] D. Mostow. *Machine Transformation of Advise into a Heuristic Search Procedure,* chapter 12. Morgan Kaufmann, Los Altos, California, 1983.

[Pearl, 1983] J. Pearl. On the discovery and generation of certain heuristics. *The AI Magazine,* IV(I):23-33, Winter/Spring 1983.

[Tappel, 1980] S. Tappel. Some algorithm design methods. In *Proceedings of the First National Conference on Artificial Intelligence,* pages 64-67, Stanford University, August 1980.

[Voigt, 1988] K. Voigt. Incorporating global constraints. Tech. Rpt. LCSR-TR-114, Computer Science Dept., Rutgers University, Sept. 1988.