An optimum greedy algorithm for choosing minimal set of conflicting constraints in the point sequencing problem

Florent LAUNAY: florent_launay@ifrance.com
Debasis MITRA: dmitra@zach.fit.edu

Tr#: CS-2002-16

Abstract:
In this work, we first have proposed a technique to define the "causes" of inconsistency on an online point based reasoning constraint network. Second, we introduce an algorithm that proposes the user a minimal set of relations to remove when inconsistencies are detected. We have developed and implemented a battery of algorithms for the purpose of this type of reasoning. Some useful theorems and properties are defined for proving the 'minimal' aspect of the algorithm. Finally, we found that our investigation was a polynomially solvable sub problem of the vertex cover problem.

Key Words: Temporal reasoning, Interactive constraint satisfaction problem, Consistency restoration

1 - Introduction

A constraint satisfaction problem (CSP) has typically three objectives. (1) It tries to detect if a problem instance is satisfiable or not, (2) in case the problem instance is satisfiable, it wants to find a solution, or all (or more than one) solutions, and (3) in case there exists some objective function associated to the problem, it tries to find a optimal or sub-optimal solution (constraint optimization problem). Any CSP algorithm would stop after detecting "inconsistency" in a problem instance, depending on the expected level of consistency e.g., the (arc consistency, the k-consistency for integer k>=1, or global-consistency, etc.). However, in a practical situation it is somewhat frustrating for a user, and such a user would often like to know why the input is wrong/inconsistent. Works in the CSP literature have rarely addressed that issue.

One of the reasons researchers avoided addressing the issue of detecting any "cause" for inconsistency is the ambiguity in identifying such a cause. For example, for a set of comparable objects a, b, and c, the information a<b, b<c, and c<a, is inconsistent (Figure 1). There is no preferred constraint here that could be identified as the cause for inconsistency, any one of them could be declared as being responsible. However, if we have {a<b, b<c, b<d, d<e, e<a, c<a}, then the constraint a<b becomes a clear choice as the culprit (Figure 2). In this work, we have started an investigation on such issues of detecting the "reason" behind inconsistency. We have launched our research with a simple domain of point-based temporal reasoning that is tractable and is well understood in the literature. The objective here is to detect a minimal set of constraints that should be eliminated in order to restore consistency in an otherwise inconsistent problem instance. Also, our work picks up

the problem of incremental reasoning (online problem) that is more applicable in real life situations, where information is gradually added to a database.

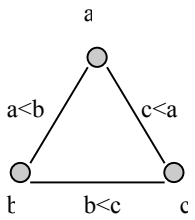


Figure 1

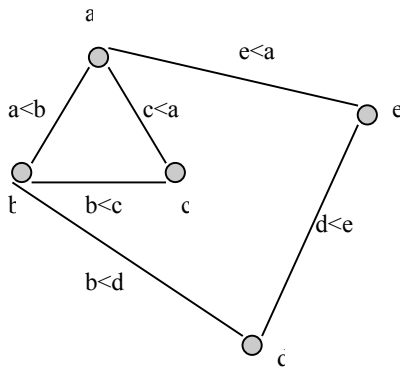


Figure 2

We expect our work on identifying the responsible set of constraints causing inconsistency in a problem instance will be very useful in the diagnosis area, for quite obvious reasons. Adding such a capability would enhance the user-friendliness of any CSP system and thus, enhance its usage.

In the next section we introduce the problem of online interactive reasoning scheme with qualitative constraints between time-points. The algorithms for solving this problem are described in the subsequent section. That section also introduces our approach in defining the "causes" for inconsistency and an algorithm to find it. Following this is an example using our algorithm before turning to some useful theorem and properties used for the next section: the proof of the algorithm. A following section discusses the significance of this work in a more general framework than the incremental problem primarily addressed here and mentions our future directions. We have discussed some related works after that, and then concluded the paper.

2 - Incremental Point-based Reasoning

Point-based temporal reasoning constitutes the simplest form of spatio-temporal reasoning (Vilain and Kautz, 1986). The scheme has three basic relations {<, >, =} and the corresponding relational algebra is comprised of its power set that is closed under the traditional reasoning operators like composition, inverse, set union, and set intersection. Some recent works have shown interests in the point-based reasoning in one dimension with its incremental version (Mitra et al, 1999). In an incremental CSP a new object (a time-point) is inserted into a set of objects already committed in a space (a time-line), i.e., the new point is inserted within a sequence of points, satisfying the binary constraints between the new point and the old ones. The problem could be viewed as a database entry, where a consistency checking needs to be first performed before committing the entry operation.

Mitra et al (1999) have found some interesting results in the point-based incremental reasoning problem. They have observed that a satisfiable region for a new point within a sequence of points would be either a null set of regions (inconsistent) or a contiguous region (an interval) possibly excluding some old points within the interval. They have utilized this property to devise an efficient algorithm for preprocessing before finding the actual valid regions for the new point satisfying the binary constraints.

In this work we have attempted to attack the problem of finding the "cause" behind inconsistency when the latter is detected and have developed an algorithm for the purpose. We have adopted Mitra et al's algorithm and extended for the purpose of first detecting the inconsistency in an incremental problem instance. The following is Mitra et al's algorithm.

Algorithm 1D (Mitra et al, 1999): Scan the sequence of existing points from left to right on the time-line and their relationship/constraints with respect to the "new" node that is to be inserted (within the sequence, satisfying those constraints). Keep a status variable that keeps track of whether the left end (of the list of valid regions for the new point, or as it is called, the "box") has been found, or an equality relation (new = xi) has been found, or the right boundary has been found. The "box" is found when the constraint from the new point to the current point xi changes from >, or >= to < or <=, for i running over all the old points. The *inequality* (<>) and the *tautology* (< = >) are ignored in this scan. A singleton *equality* relation is a hard constraint, making the box converge to that old point. After a "box" is found, if any constraint demands the new point to be outside the "box," then an inconsistency would be detected, otherwise with a second scan over the "box" the algorithm would elicit the exact set of valid regions checking if the old points within the box themselves are valid regions or not. For example, a set of valid region may be {[x5, x5], [x5, x6], [x6, x7], [x7, x7], [x7, x8]}, indicating that the new point may be assigned anywhere on the box [x5, x8) except on the points x6 and x8.

On detection of inconsistency we run a second algorithm to find the conflict set between the constraints. In the next section we describe those issues.

3 - Detecting Constraints Causing Inconsistency

Many sets of constraints together could cause inconsistency, such that removal (or fixing) the constraints in this set would make the system consistent. We call this problem the "consistency restoration" problem and such a set of constraints as the "responsible set." It is quite inconceivable that all the provided constraints need to be removed/fixed to solve the "consistency restoration" problem. Actually this assertion could be easily proved. A related question here is then which particular set we chose as a solution to the problem, and subsequently report to the user. Our proposal is that we choose a *responsible set* that is of minimal cardinality out of all possible *responsible sets*. We call such a minimal cardinality-responsible set as the "minimal set" or *MinSet*. Of course, there could be more than one such minimal set with equal cardinality values, but we would like to find any one of them.

Definition 1: The *degree of conflict* of a given constraint in a CSP is the number of other constraints that it conflicts with.

If a system contains *n* constraints, then the degree of conflict for any constraint will be at most *n-1*, since a relation cannot conflict with itself, and at least 0.

Example 1: Let S be a set of constraints S = {c1, c2, c3, c4, c5, c6}.

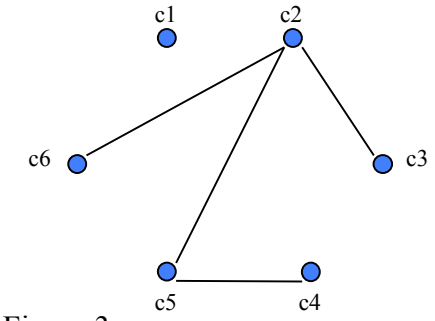


Figure 3

Here, the degree of conflict for c1 is zero, since c1 does not conflict with any other constraint. The degree of conflict for c2 is three, for c3, c4 and c6 each it is one, and for c5 it is two. Removing relation c2 and one of the relations between c5 and c4 would be enough to make the system consistent. Hence, {c2, c4} and {c2, c5} are both *MinSets* here, whereas {c6, c5, c3} is another *responsible set* that is not a *MinSet*.

The following algorithm finds the conflict set for constraints and the corresponding degree of conflict for each constraint.

Algorithm GenerateConflictSet (list of constraints between 'new' and the point-sequence)
ConflictSet = null;
For i =1 to N do DegreeOfConflict[i] = 0; // N number of points in the sequence

For each constraint ci from i =1 to N do // for a relation (new ci xi)
    for each constraint cj from j = i+1 to N do
        if (ci is "<", or "<=", or "=") and (cj is ">", or ">=", or "=") then
            ConflictSet = ConflictSet U {(ci, cj)};
            DegreeOfConflict[i]++;
            DegreeOfConflict[j]++;
        end if;

End Algorithm.

This is obviously an O(N^2) algorithm.

The problem of finding a *minimal set* of constraints removal/fixing of which would restore consistency in a system (constraint network) is solved by the following greedy algorithm.

```
Algorithm FindMinset (ConflictSet, DegreeOfConflict)
  Minset = empty;  // set of minimal nodes to be removed
AggregateDegreeOfConflict = Sum over ConflictSet [DegreeOfConflict];

  While AggregateDegreeOfConflict =/= 0  do                          // O(N)
        Let c = a constraint with the maximum DegreeofConflict;      // O(N log N)
        Minset = Minset U {c};
        for each element (c, ci) in ConflictSet do                   // O(N)
              ConflictSet = ConflictSet  -  (c,ci);
              DegreeOfConflict[c] = DegreeOfConflict[c] -1;
              DegreeOfConflict[ci] = DegreeOfConflict[ci] -1;
              AggregateDegreeOfConflict = AggregateDegreeOfConflict -2;
        end for;
  end while;
  return Minset;
End Algorithm.
```

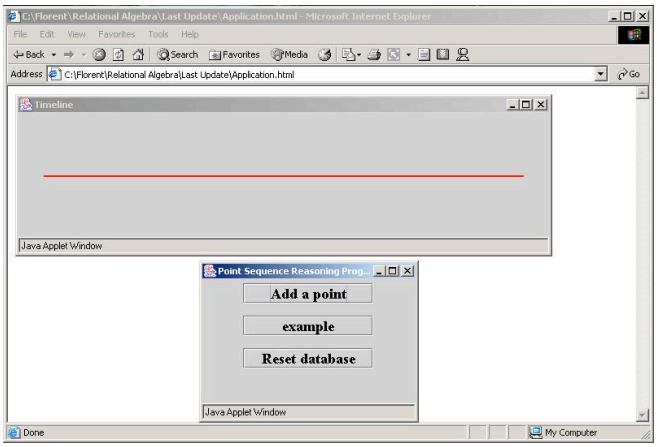
As shown with comments in the algorithm, O (N^2 log N) is the asymptotic time complexity. The problem has a flavor of the well known "vertex cover" problem, and hence a polynomial algorithm is unlikely to be a complete algorithm in general (subject to P =/= NP). However, at least in the point-sequencing problem it could be easily shown that a graph generated over the conflict set (with nodes being the constraints and edges being the conflicting constraints) is a bipartite graph (see properties section). The vertex cover problem is tractable over bipartite graph and the above algorithm is a complete one in this situation. We will show this pattern in one of the following sections.

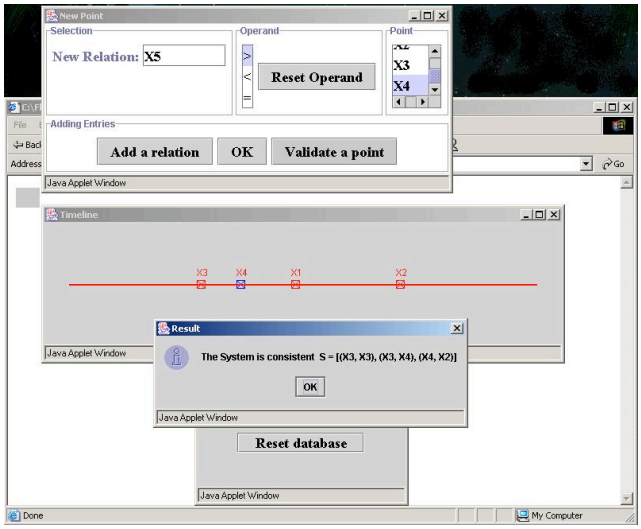
4 - Implementation

All the three algorithms mentioned here have been implemented with a graphical user interface to the software. It displays the valid regions (if consistent) on the time-line with the existing point-sequence, and allows the user to commit the new point on one of the valid regions interactively. Then it goes to the next iteration for accepting the next new point. In case of detecting inconsistency it runs the third algorithm *FindMinset* and dumps the *Minset* for any possible corrective action by the user.

For the sake of our investigation, we developed a program implementing both algorithms described above. A graphical interface allows the user to add points in an online fashion, and at each new point, the system reacts giving the user the informationabout consistency.

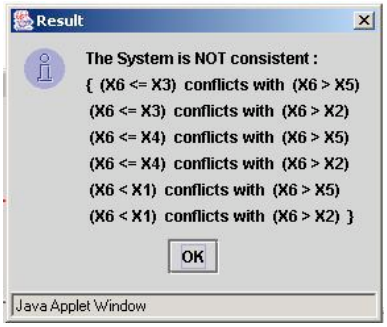
The interface shows the time line basis and a set of buttons allowing the user to Add a new point, load a pre-computed example, or reset the current database.





At each stage, if the system is consistent the program tells the user what is the valid set to insert the new point and highlights this set on the time line base.

In case of inconsistency, the program suggests the user a minimal set of relations to remove from the previous query in order to make the system consistent.



5 - Notations

S = set of all relations making the system inconsistent
n = number of relations in S
$\mathbf{R_i}$ = Relation with index I. Typically, $1 < i < n$
I = a set of minimal relation to remove in order to make the system consistent

m^i = degree of inconsistency of relation Ri
$\sum$ **$m^i(S)$** = Sum of all degree of inconsistency of the set S
$\sum$ **$m^i(I)$** = Sum of all degree of inconsistency of the set I (each $m^i(I)$ is set whenever Ri is added to I)
N = number of pairs of relations conflicting with each other in a set S
R_i * R_j = Relation R_i conflicts with relation R_j.

Given a set of relations generating inconsistency S = { R_1, R_2, …, R_n}. Let's suppose n is the number of relations conflicting with each others.

1) R_1 * R_i
2) R_j * R_k
….
n) R_{n-1}* R_n

*Each one of these relations has its own degree of inconsistency.

*If n = 0, $\sum m^i(S) = 0$, then the relation is consistent.

*Each relation R in the set conflicts with at least one other relation in the set. The number of relation this relation R conflicts with defines the 'degree of inconsistency' of a given relation. Thus, that number will be at most n-1, since a relation cannot conflict with itself, and will be at least one.

6 - Some Theorems and properties

Property 1:
Considering a set S = {R1, .. ,Rn} where Ri is a relation making a system inconsistent, and m^i is the degree of inconsistency of Ri (as defined above). Then the number of conflicting pairs is

$$\mathbf{N = \sum m^i(S)/2}$$

($\sum m^i(S)$ will always be an even number, since inconsistent relations are exists by pairs)

Proof:
Every time a relation appears in a pair of conflicting relation, both relations have their inconsistency degree increased by one, hence the sum of all inconsistency degree is twice the number of conflicting pairs. (Trivial proof)

Property 2:
If

$$\mathbf{\sum m^i(I) >= N, \text{ the system is consistent}}$$

With $m^i(I)$ being the inconsistency degree of relation Ri when it is added to the set I, and N being the number of initial conflicting pairs in S.
Note that $\sum m^i(I) = N$ is a sufficient condition for the system to be consistent. Moreover, in our algorithm, $\sum m^i(I)$ will never be greater than N, since the goal of our investigation is to find a minimal set of conflicting relations.

Proof:
Removing a relation Ri will also remove exactly one instance of each relation that previously conflicted with Ri. So, the sum of inconsistency degree of S will be lowered by $2*m^i$. Recall that when $\sum m^i(S)$ reaches 0, the system is consistent. So, if
$2* \sum m^i(I) = \sum m^i(S)$, the system is consistent; $\sum m^i(I) = \sum m^i(S)/2 = N$

Lemma 1:
From properties 1 and 2, we can conclude that if

$\sum m^i(I) >= \sum m^i(S)/2$, the system is consistent

Inversely, if **$\sum m^i(I) < \sum m^i(S)/2$, then the system is not consistent**.

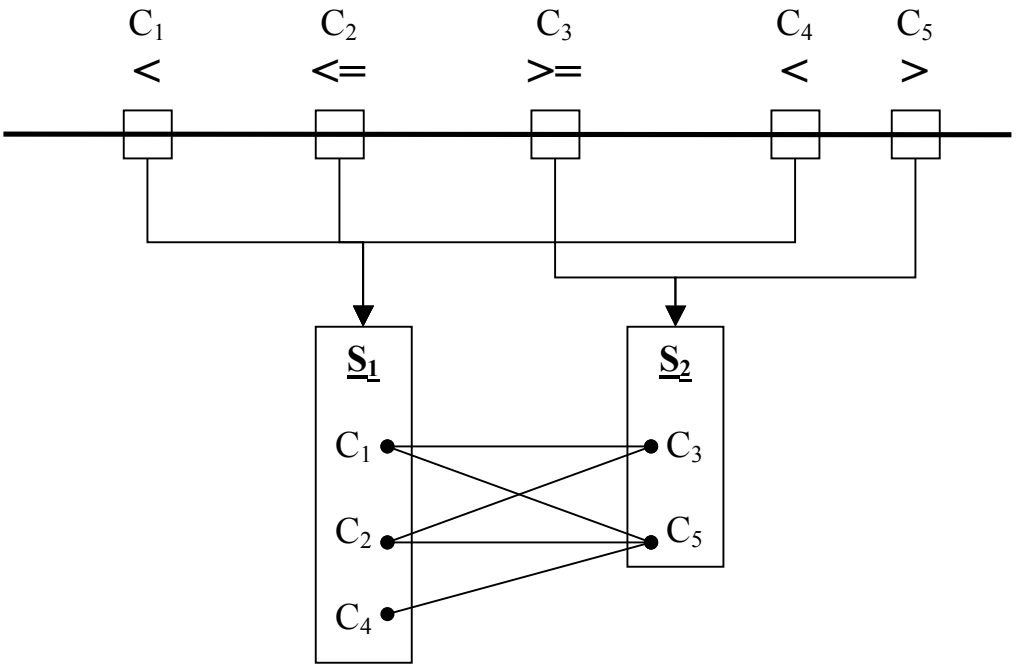
From this point onward in the paper, we will handle inconsistencies caused by equalities as a different case.

Property 3:
The set S is a bipartite graph. Therefore, it can be divided into two sub graphs such that:

S is a bipartite graph with two partitions S_1 and S_2

Where no node of S_k is connected with another node of S_k.

Proof:
Whenever an element is added to S, it is conflicting with another element in S. let us call these two elements V_1 and V_2. Then, either:
V_1 (> or >=) V_2 OR V_1 (< or <=) V_2. According to their relation with each other, V_1 and V_2 will be stored in their respective bipartite sub graph.



Property 4:
If either

$\|S_1\| = 0$ or $\|S_2\| = 0$, then both $\|S_1\| = 0$ and $\|S_2\| = 0$, therefore the system is consistent

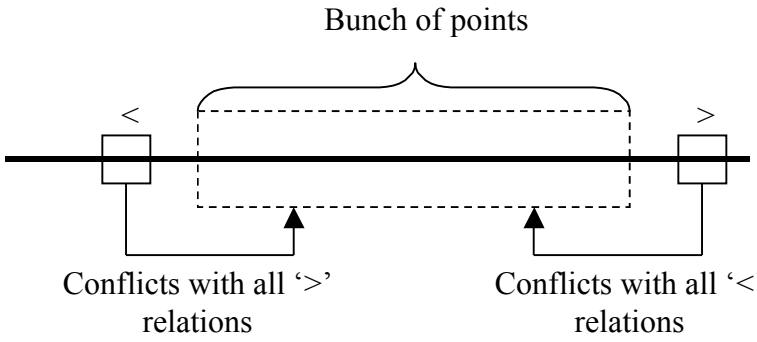
Similarly, $|S_1|$ **or** $|S_2| = 0$ **is a sufficient condition for** $|S| = 0$.

Proof:
If S1 or S2 is empty, then the graph S is totally disconnected and no conflicting constraint remains, hence $|S| = 0$.

Property 5:

The element with the highest number of edges of each bipartite sub graph is connected with all elements of the other sub graph.

Proof:
The element with highest inconsistency degree in each sub graph is on the right most position for the element of the '>' partition (none can be greater) and on the left most position for the '<' partition (none can be smaller).



Theorem 1:

The set of edges of each element in one sub graph is the subset of all the set of edges of each element with degree equal or greater than this element.

Proof:
From property 5, we know that all elements in one sub graph S_1 are connected with the element with the highest number of adjacent edge of the other sub graph S_2. Therefore, any set of edges of any element in S_2 with a degree smaller or equal to the highest one in S_2 will be a subset of the set of edges of the element with highest degree.
If we now remove this element from S_2 (highest number of adjacent edges), the second element with highest number of adjacent edges in S_2 becomes the one with highest number of adjacent edges. Thus, all other set of edges of all other relations will be subsets of the set of edges of the 'new' relation with highest number of edges. This can be applied until there remains only one element in the sub graph. Inversely, all set of edges of any elements in S_1 with degree less than or equal to the element with highest inconsistency degree is a subset of the set of edges of the element with highest inconsistency degree.

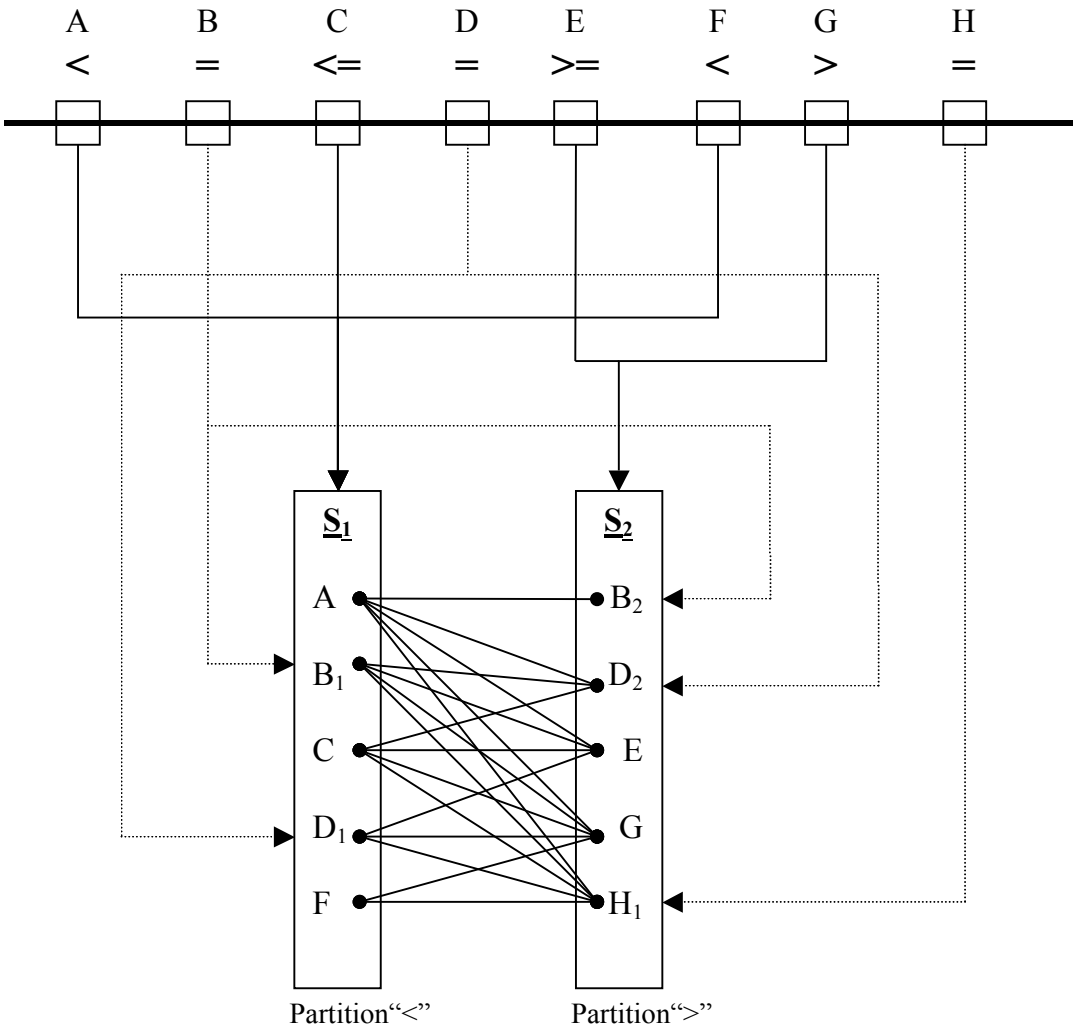
Property 6:

Each time a relation is chosen by Algorithm 1, it has a number of adjacent edges of exactly: Max ($|S_1|$,$|S_2|$)

Proof:
The element with the highest number of adjacent edges is the element in the smaller sub graph and is connected with all elements of the bigger sub graph.

Adding equality

Property 7:



Inconsistencies involving equality can be spitted in two sub relations, each of which appear in its respective sub graph and do not change the nature of the bipartite graph.

In order to add equality, keeping the properties of a bipartite graph, we split equality as if it was two different conflicts in the bipartite graph as well as 2 different points on the same line. In other words, for (new = x_i) we replace it with (new <= x_{i1}) and (new >= x_{i2}) where x_{i1} and x_{i2} are the two names for the same point xi2. The two constraints go to two different partitions of the conflict graph. Each equality relation is composed of two sub relations: <= and >=. Each sub point will be stored in its respective sub graph. Therefore, all relation can be thought in terms of two partitions: "<"and">".
The "<"partition includes the following relations: {<, <=, and the <= part of the equality}.
The ">"partition includes the following relations: {>, >=, and the >= part of the equality}.

If the two partitions of every = relations are thought as an element of one of the two sub graphs formed by the bipartite graph, then all properties hold for all the relations, including equality.
Recall that <=> never creates inconsistency, and since multiple point assignment is not allowed ($x_i = x_{new}$ and $x_i > x_{new}$ is not allowed), <> neither will create inconsistency.

Property 8:

The maximum sufficient number of elements to remove to make the system consistent is <= Min (|S1|, |S2|)

Proof:
At most, removing the entire smallest set will remove inconsistency (bipartite graph)

Lemma 8:

Algorithm 1 finishes with a set |I| <= Min (|S1|, |S2|)

Proof:
From property 6, we know that each time algorithm 1 chooses a relation, it is the one conflicting with Smax = Max(|S1|, |S2|). Then Algorithm 1 chooses a relation in Smin = Min(|S1|, |S2|). After each step, exactly one relation is removed from Smin, and some relation(s) may be removed from Smax. Then, either Smax becomes less than Smin, in which case the next element to be picked up will be in Smax, or Smin still the minimal set, and the next element will be picked up in it. At most, Smin still all along the minimal set, and will be totally removed by successive choices of algorithm 1.

Property 9:

This is not a strong or correct proof, but this fact has to be mentioned since it could become a crucial issue to prove algorithm FindMinSet in the future.

Two different sets I1a and I1b found with algorithm 1 have the same minimal cardinality for a given input:

The goal is to obtain min |I| by removing Max (S1, S2) at each stage until the number of connecting edges reaches 0. Two sets found with algorithm 1 will not necessarily have the same set of elements. If at any stage, a tie occurs between two elements in Max (S1, S2), or if |S1| = |S2|, then the algorithm has the choice among several elements to remove.
Let us consider these two cases separately.
First, if two elements are tied in Max(S1, S2): R1 and R2 (R1, R2 $\in S_i$), then both elements will be picked up by the algorithm in any order.
Second, if at any stage |S1| = |S2|, R1 is element max in S1, and R2 is element max in S2. removing R1 will possibly make |S1| < |S2|, then at the next stage, R2 will be picked up. Otherwise, it could make |S1| > |S2|, meaning that removing R1 removed also at least 2 elements in S2. Then the next element to be picked up will be in S2.

7 - An example running the algorithm *FindMinSet*

Consider the set S = {R1, R2, R3, R4, R5} with the following rule of conflict (note that it contains 7 pairs, so $\sum m^i(S)$ will be 14):

R1 * R2
R1 * R3
R1 * R4
R1 * R5
R2 * R4
R2 * R5
R3 * R4

Then, the corresponding inconsistency degree <Ri> will be:

<R1> = 4
<R2> = 3
<R3> = 2
<R4> = 3
<R5> = 2

Our algorithm chooses R1 and adds it to the set I, $\sum m^i(I) = 4$, and the corresponding inconsistency degree <Ri> becomes:

<R2> = 2
<R3> = 1
<R4> = 2
<R5> = 1

Now, we can either add R2 or R4 to I since they have the same inconsistency degree. Lets choose R4. Now $\sum m^i(I) = 4 + 2 = 6$, we then obtain:

<R2> = 1
<R5> = 1

We now can remove either R2 or R5 since, once again, they have the same degree of inconsistency (choose R2).
$\sum m^i(I) = 6 + 1 = 7$ which is equal to the original number of pairs. The system is now consistent and a minimal set I = {R1, R4, R2} is now found.

8 - Proof of the algorithm *FindMinSet*

Last Update of the proof :(11/19/03)
Proof direction 1:
*Consistent:

When the algorithm finishes, n = 0 and S is empty. When S is empty, every inconsistent relation have been removed, consequently, the system becomes consistent.

* Minimal:

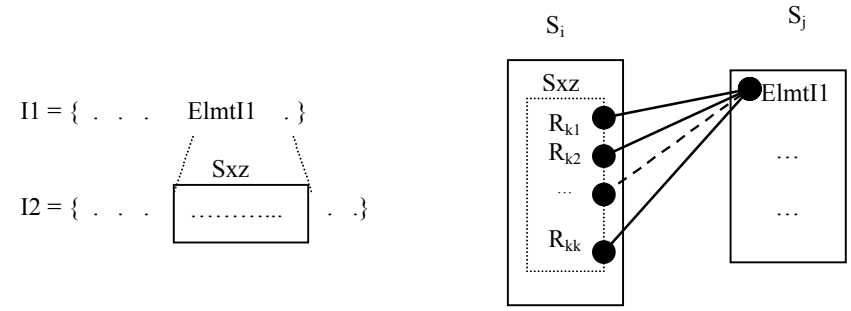
Assumption:

There exist a minimal set of relations I2 within an inconsistent system to remove to regain consistency, which contains less elements than I1.
The set of constraints to remove from an inconsistent system found by Algorithm 1 is not unique. In fact, Algorithm 1 proposes one of theses set if it exists. The choice of this set depends on the ordering of the variables as algorithm one picks the first occurrence of maximal conflict based cardinality within the set of all inconsistent relations. Hence, according to the constraint ordering, Algorithm 1 may find different sets with same cardinality (minimal). Let us call the set of all different sets possibly found by Algorithm 1: GI = {I1a, I1b, I1c, ...}

No proper subset of any set found by Algo 1 can make the system consistent; hence, I2 is not a proper subset of I1i for any i ($1 \leq i \leq |GI|$).
S1, and S2 are the sets held by the two subset of the bipartite set before the algorithms run. (original bipartite sets)
s1, and s2 are the sets held by the two subset of the bipartite set while the algorithms are running. (dynamic bipartite sets)

Proof:

At one time, $|s1| \neq |s2|$ and Algo1 will pick the element with highest inconsistency degree, but I2 will not in order to have a different set from I1. Since the element picked by Algo1 is connected with all elements of the other set of the bipartite graph, Algo2 MUST pick all elements of this last set MAX(s1, s2) (ie, the set with the highest number of elements). But from property 8, removing MIN(s1, s2) is enough to have the bipartite graph totally disconnected. Therefore, the previous assumption is false, leading the proof to a contradiction.



If I2 does not pick ElmtI1, it has to pick the whole set Sxz

9 –Interval Algebra

Discussion about ORD-Horn cases:

Search for inconsistency and for minimal set to remove to get consistency back can be done in polynomial time when the set of constraints being involved belong to the class of ORD-Horn relations.
ORD-Horn algebra is the maximal tractable subset of Allen algebra containing all 13 basic relations. It can always be expressed in a conjunctive normal form where each literal contains at most one relation of the type <= or =, and any number of relations of type /=.

Relation with Allen's algorithm

(A new proof of Tractability for ORD-Horn Relations, Ligozat)
Important clauses about ORD-Horn cases:

1. A constraint network with labels in the ORD-Horn class is consistent if and only if it is path consistent.
2. The class of ORD-Horn relations is the maximal sub-class of Allen's algebra containing the atomic relations, closed by non-empty intersection, conversion and composition, which is tractable.
3. The class of ORD-Horn relations is defined as the set of relations that can be represented by Horn clauses involving beginning and end points.

Simple application of interval algebra problems:

While investigating on a crime that has been committed, inspector Harry gets the following facts from the three witnesses (A, B and C) that were present during the scene:

- A left when C arrived
- B arrived and left before C arrived
- B left after A arrived, and before A left.

Inspector Harry knows that one and only one of the witnesses lies and committed the crime.
Just before giving up, a new witness (D) appears and tells inspector Harry the following facts:

- B left when D arrived
- D arrived before A arrived, and left before A left
- D left when C arrived.

This simple problem can be solved intuitively, but may lead to major errors when solving larger problems. The algorithms presented at the end of this section however would easily solve this kind of problems, and of course much larger problems, by minimizing the number of relations to be taken apart.

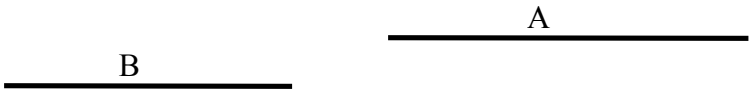
Relation between Point and Interval algebra:

Do point algebra algorithms can be applied to interval relations?
This may be the case when dealing with ORD-Horn relations as far as Interval algebra is concerned. However, the problem of finding consistency for interval algebra is obviously more complex than for point algebra since interval has 13 basic relations against 5 for point algebra. The extra set of relations comes from the fact that the constraints not only deals with a unique discrete point, but with a starting point and an ending point. Some algorithms are already under investigation to serve the purpose of finding consistency, and restoring consistency in case of inconsistency. We will introduce some of them in this report, but first, we need to define interval reasoning, and to build up some graphical conventions to illustrate the different examples we will use.

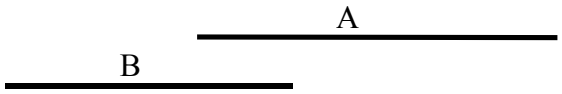
The set of all possible relations for interval relation is defined as follows (Ligozat, A New Proof of Tractability for ORD-Horn Relations).

Among the 13 atomic relations, six are of dimension 2:

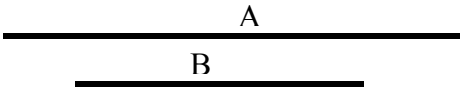
- b (before)
  ex: B b A



- (overlap)
  ex: B o A



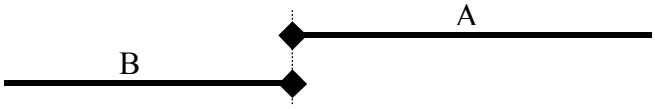
- d (during)
  ex: B d A



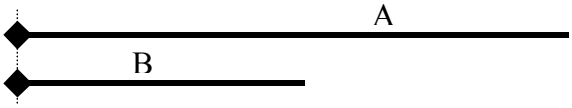
and their respective inverses.

six are of dimension 1:

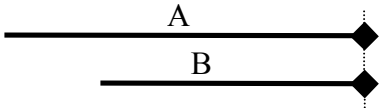
- m (meet)
  ex B m A



- s (start)
  ex B s A

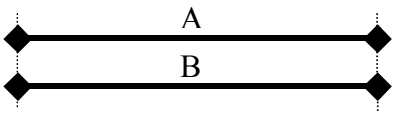


- f (finish)
  ex B f A



One is of dimension 0:

- eq (equal)
  ex B eq A

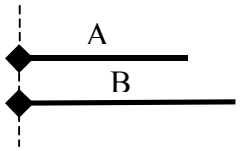


In order to illustrate the resulting relation of two constrained intervals, we will use some symbols for which an explanation is required:

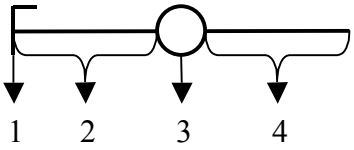
⌈ : Required Starting point

⌉ : Required Ending point

◯ : Forbidden Start/End point



A starts along with B

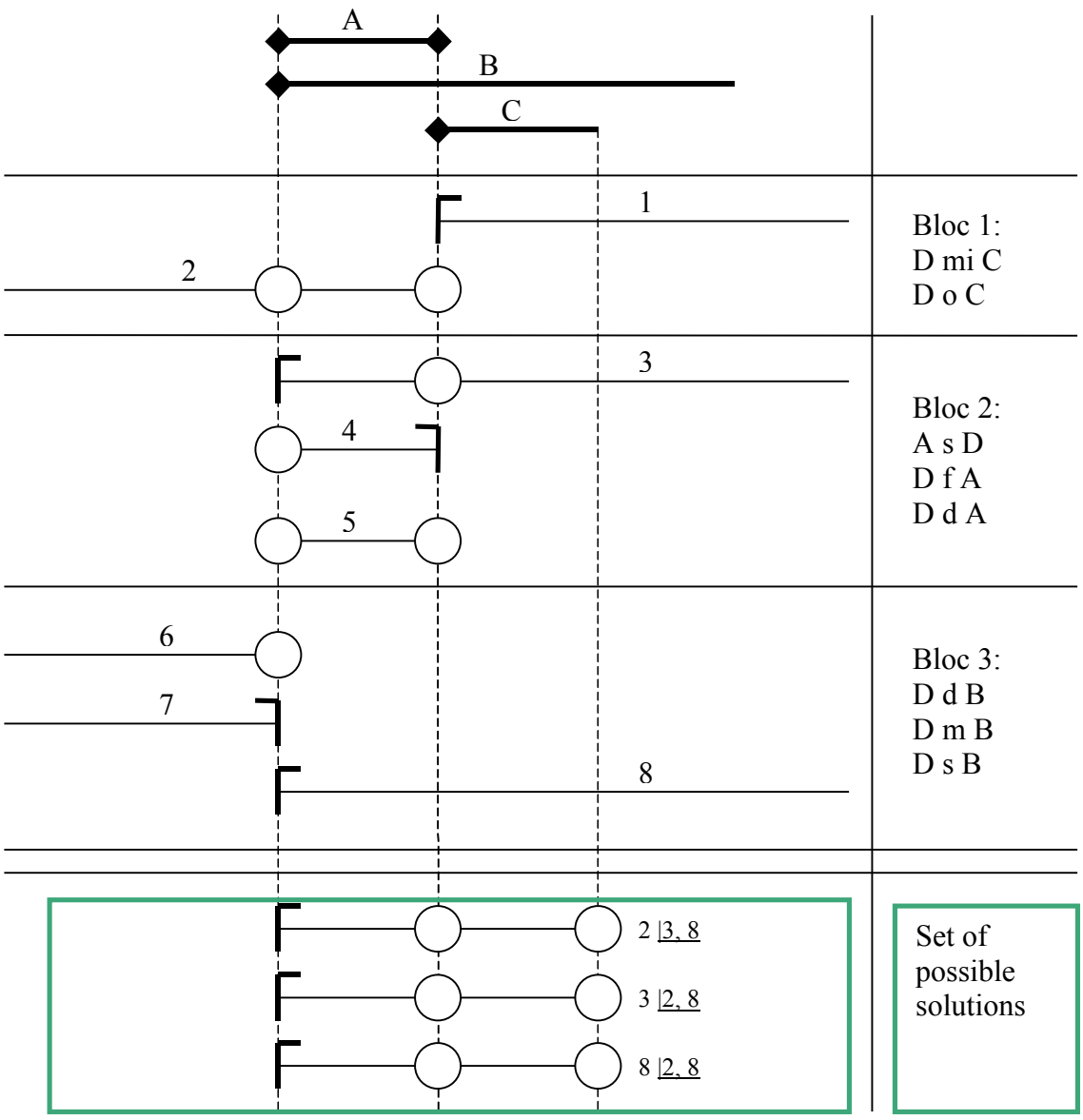


Basic representation of the "Start inverse relation"

1 is the required starting point
2 is a forbidden region for either starting or ending point (it is comprised between _ and O)
3 is a forbidden region for either starting or ending point
4 is a required region for ending point

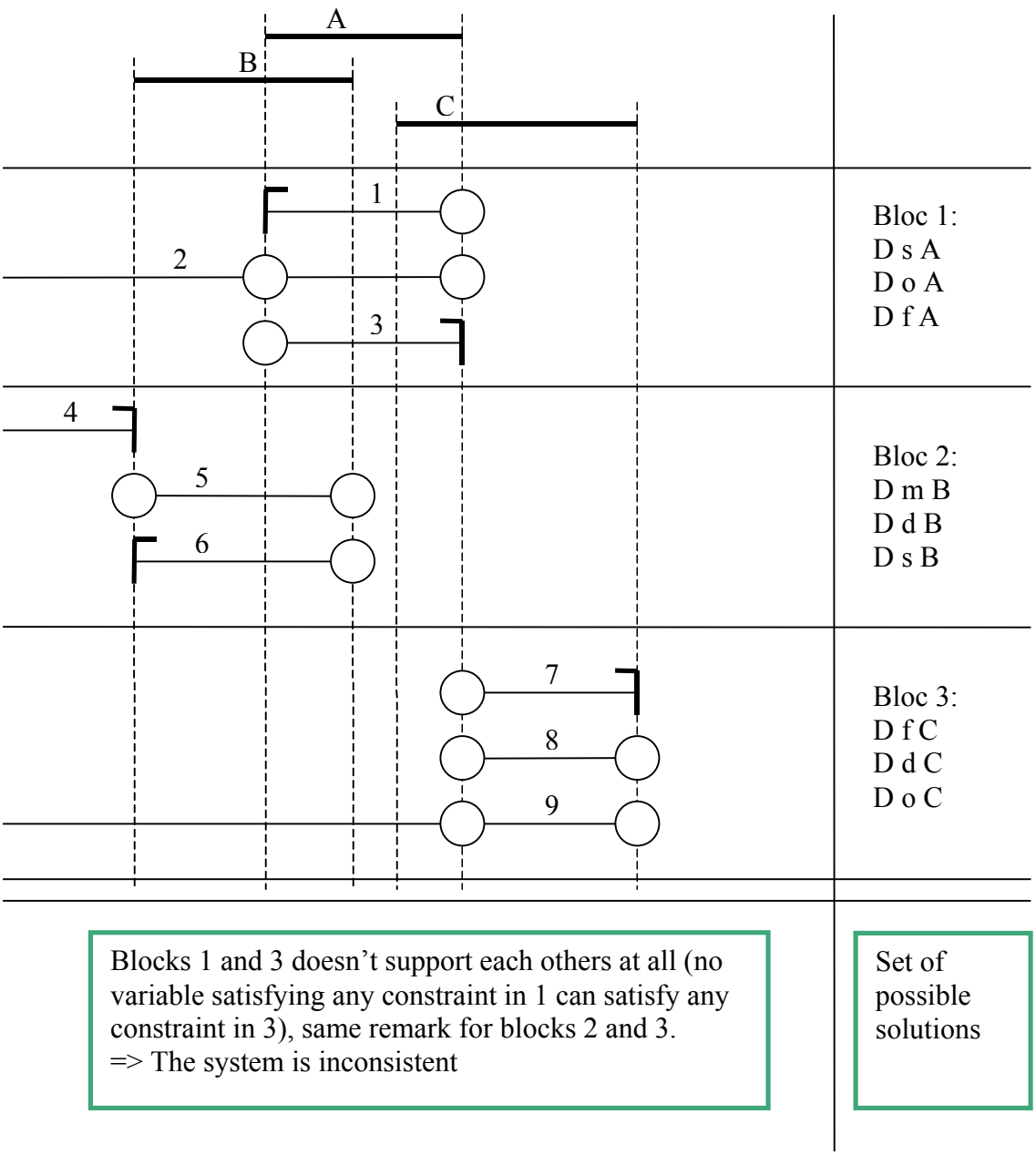
Example one (non ORD-Horn case)

- D mi, o C
- D si, f, d A
- D b, m, s B



Example two (non ORD-Horn case)

- D s, o, f A
- D m, d, s B
- D f, d, o C

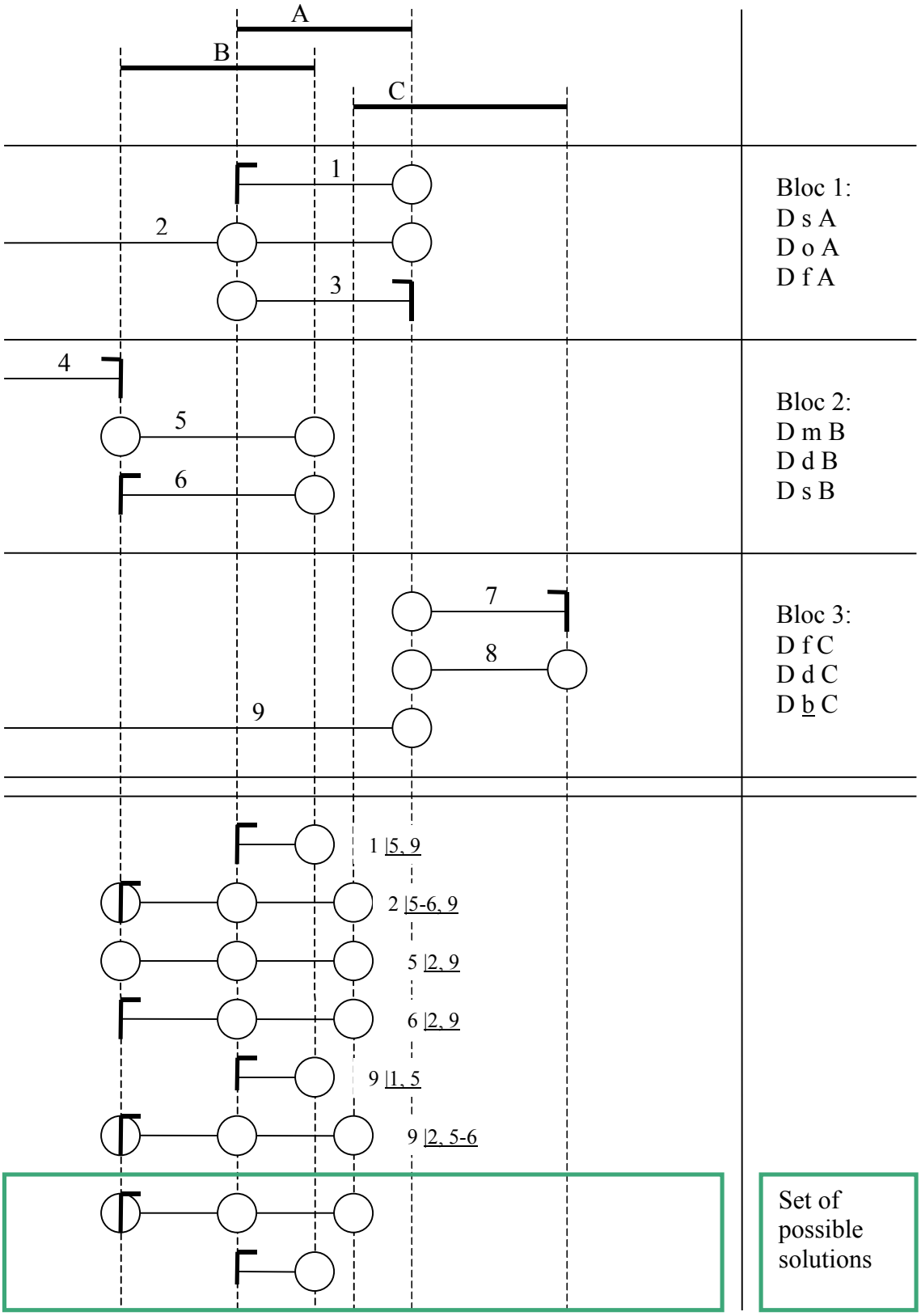


In the next example, we just changed a constraint form the previous block 3 in order to restore consistency

Example three (non ORD-Horn case)

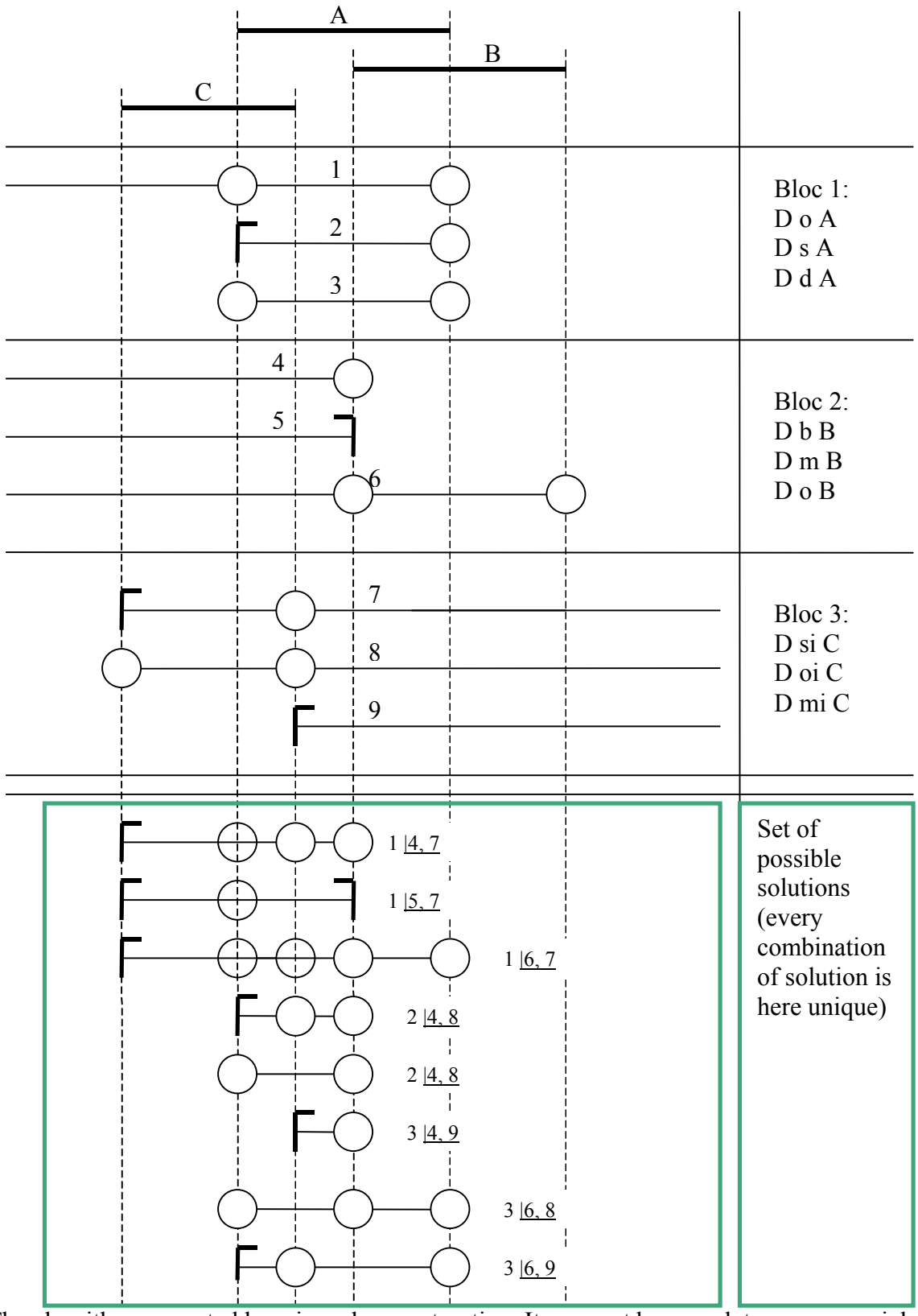
- D s, o, f A

- D m, d, s B
- D f, d, b C


A
B
C
1
2
3
Bloc 1:
D s A
D o A
D f A
4
5
6
Bloc 2:
D m B
D d B
D s B
7
8
9
Bloc 3:
D f C
D d C
D b C
1 |5, 9
2 |5-6, 9
5 |2, 9
6 |2, 9
9 |1, 5
9 |2, 5-6
Set of possible solutions

Example four (ORD-Horn case)

- D o, s, d A
- D b, m, o B
- D si, mi, oi C



The algorithm presented here is under construction. It may not be complete, some special cases may have to be handled, and it may not finish for non ORD-Horn cases.

Algorithm Interval-Consistency

- Split every multiple constraint into unary constraint, and store them in their respective blocks.

- Merge Blocks 2 by 2 by reducing every possible 2 constraint in each block. Prune every constraint that does not have at least one support in the other block. If a block becomes empty at any time, then the system is not consistent.

  Nb: If for some constraint x _ Block 1, y _ Block 2,
  z, _ Block 3, …
  C1x _ C2y _ C3z _ … ≠ Ø, then the system is
  consistent. Otherwise, the system is inconsistent.

- The last resulting block is the set of all possible solutions.

```
FUNCTION interval-Consistency (setOfComposedCstreWithNew) : Blocks
BEGIN
    consistent <- true;
    Blocks[] <- createBlocks (setOfComposedCstreWithNew);
    WHILE (sizeOfBlocks > 1 & consistent)
    DO
            Blocks [last-1] <- mergeBlocks (Blocks [last], Blocks [last-1]);
            consistent <- (Blocks [last-1] = null);
            Blocks [] <- Blocks [] - Blocks [last]; //Just remove the last Block
    END
    return Blocks [1]; //Set of solution if consistent, null otherwise
END
```

```
PROCEDURE createBlocks (setOfComposedCstreWithNew) : []Blocks
    i = 0;
    WHILE ( setOfComposedCstreWithNew != NULL)
    DO
            ComposedConstraint <- any composed constraint from
                                    setOfComposedCstreWithNew;
            Blocks [i+1] <- decompose (ComposedConstraint);
            setOfComposedCstreWithNew =
                    setOfComposedCstreWithNew – ComposedConstraint;
    END
    return Blocks[];
END
```

```
PROCEDURE mergeBlocks (Block_A, Block_B) : Blocks
    Block_C <- null;
    FOR all unary constraint c_A in Block_A DO
            c_C <- null
            FOR all unary constraint c_B in Block_B DO
                    IF compatible (c_A, c_B) DO //   or IF (c_A _ c_B ≠ null)
                            c_C <- c_A _ c_B;
                            Block_C <- Block_C U c_C;
                    END
            END
    END
    return Block_C;
END
```

```
PROCEDURE decompose (ComposedConstraint) : Blocks
    Blocks_R <- null;
    FOR c[k] <- each constraint in ComposedConstraint DO
            Blocks_R <- Blocks_R U c[k];
    END
    return Blocks_R;
END
```

Algorithm build-conflict-set:

Many directions could be investigated to build a conflict set when inconsistency is detected. This will facilitate the regain of consistency for such systems, either by removing the (most) culprit constraints, or by suggesting the smallest changes to do to some constraints to restore consistency.

In a first approach of the discovery of these algorithms, we will only focus on ORD-Horn cases that are proved solvable in polynomial time. We will here explore two of them, even though none of them may be best. This is a first approach of the subject and may be subject to major changes (12/03/2003).

Proposition 1: (detecting inconsistent unary relation)

- Mark every unary relation within each Block each time it does not get support from a whole block.
- Get the minimal mark value within each Block and store it in *MinConflictValue*.
- Remove all constraint of the Block which has highest *MinConflictValue* until the system is consistent

- For each empty block, set (try?) a unary constraint that satisfies all other Blocks.

Proposition 2: (detecting inconsistent Blocks)

- Check two by two and mark every block that does not support each other.
- Run the algorithm *FindMinset* on the set of blocks in order to find the most culprit(s) blocks.
- Remove the culprit(s) block(s), or suggest a (some) consistent block(s).

Proposition 1 would require an algorithm. Although proposition 2 wouldn't, or some minor changes from the point based algorithm *FindMinSet*, the first solution seems to be more flexible (when dealing with non ORD Horn-cases).

Partial Constraint Satisfaction

Our work seems to go the opposite direction from "Partial Constraint Satisfaction" Eugene C. Freuder and Richard J. Wallace, Artificial intelligence 58 (1992) 21 – 70.
In this work, the authors try to solve inconsistent problems by finding a partial solution from the original problem by satisfying the maximum number of constraints. As far as we are concerned, we try to minimize the number of relations to remove to get back to consistency, which seems more convenient when dealing with incremental problem (we know that 'up to this point', the system is consistent)

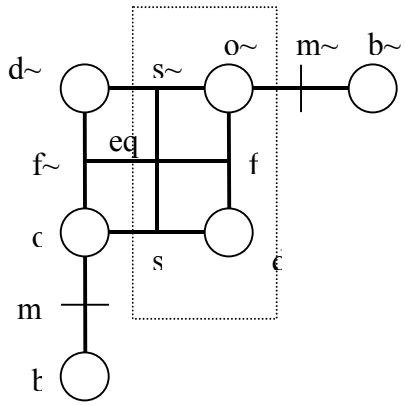
Future works

In future works, we hope to find some tractable cases for general Interval problems, including non ORD-Horn cases. The applications for such algorithm are various and may apply to such diverse domains as plan scheduling, travel itinerary setting, time decision making…

Some examples performed with the algorithm submitted to the flairs workshop 04

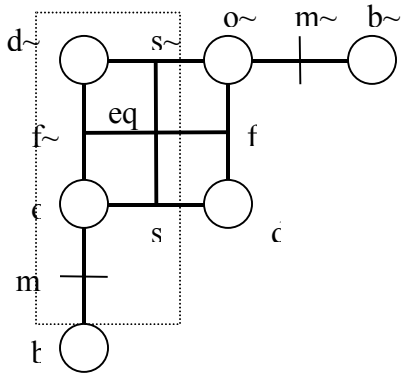
Consistent ORD Horn, non-pointizable case:
Existing set of intervals:

N d,eq,o~ A



N1 ≤ N2, N1 ≠ N2,
A1 ≤ A2, A1 ≠ A2,
A1 ≤ N1, N1 ≤ A2, N1 ≠ A2,
(N1 ≠ A1 V N2 ≤ A2)
(N1 ≠ A1 V A2 ≤ N2)

N m,o,d~,eq B



N1 ≤ N2, N1 ≠ N2,
B1 ≤ B2, B1 ≠ B2,
N1 ≤ B1,
B1 ≤ N2,
(N1 ≠ B1 V N2 ≤ B2),
(N1 ≠ B1 V B2 ≤ N2),
(N2 ≠ B2 V N1 ≤ B1)
(N2 ≠ B2 V B1 ≤ N1)

From the existing intervals, we know that:
B1 ≤ A1, B1 ≠ A1
B2 ≤ A1, B2 ≠ A1
B2 ≤ A2, B2 ≠ A2

Now, let us choose the lowest dimension relations from the disjunction cited above. The set of point relations then becomes:

$N1 \leq N2$, $N1 \neq N2$,
$A1 \leq A2$, $A1 \neq A2$,
$A1 \leq N1$, $N1 \leq A2$, $N1 \neq A2$,
$N1 \neq A1$,
$N1 \leq N2$, $N1 \neq N2$,
$B1 \leq B2$, $B1 \neq B2$,
$N1 \leq B1$,
$B1 \leq N2$,
$N1 \neq B1$,
$N2 \neq B2$

Next step consists in finding the 'boxes' for starting point and ending point individually.

- Grouping all relations for the starting point related to some old points:
  $A1 \leq N1$, $N1 \leq A2$, $N1 \neq A2$,
  $N1 \neq A1$,
  $N1 \leq N2$, $N1 \neq N2$,
  $N1 \leq B1$,
  $N1 \neq B1$
  This gives a valid box for N1:
  $A1<N1<A2$

- Grouping all relations for the ending point related to some old points:
  $B1 \leq N2$,
  $N2 \neq B2$
  Is the valid box for N2.

Since valid boxes have been found for both, starting and ending point, and the start box is before the end box, we are now confident that the system is consistent.

Now, let us recomputed the same example but with the assumption that B is before A.

Inconsistent ORD Horn, non-pointizable cases:
Existing set of intervals:

N d,eq,o~ A

$N1 \leq N2$, $N1 \neq N2$,
$A1 \leq A2$, $A1 \neq A2$,
$A1 \leq N1$, $N1 \leq A2$, $N1 \neq A2$,

$(N1 \neq A1 \text{ V } N2 = A2)$

N m,o,d~,eq B

$N1 \leq N2, N1 \neq N2,$
$B1 \leq B2, B1 \neq B2,$
$N1 \leq B1,$
$B1 \leq N2,$
$(N1 \neq B1 \text{ V } N2 = B2),$
$(N2 \neq B2 \text{ V } N1 = B1)$

From the existing intervals, we know that:
$B1 \leq A1, B1 \neq A1$
$B2 \leq A1, B2 \neq A1$
$B2 \leq A2, B2 \neq A2$

The same lower dimension relation choice leads to the following set of point relations

$N1 \leq N2, N1 \neq N2,$
$A1 \leq A2, A1 \neq A2,$
$A1 \leq N1, N1 \leq A2, N1 \neq A2,$
$N1 \neq A1,$
$N1 \leq N2, N1 \neq N2,$
$B1 \leq B2, B1 \neq B2,$
$N1 \leq B1,$
$B1 \leq N2,$
$N1 \neq B1,$
$N2 \neq B2$

We now look for the 'boxes' for starting point and ending point individually.

- Grouping all relations for the starting point related to some old points:
  $A1 \leq N1, N1 \leq A2, N1 \neq A2,$
  $N1 \neq A1,$
  $N1 \leq N2, N1 \neq N2,$
  $N1 \leq B1,$
  $N1 \neq B1$

But here, since $B1 < A1$, the relations $A1 < N1$ and $N1 < B1$ conflict with each other. No box can be found and the system is hence inconsistent.

1. First, we convert each constraint into conjunctive normal ORD-Horn form as in the example 3 above. Thus, the whole set of constraints *C* becomes a conjunctive normal formula, where each literal is a constraint between one of the boundary points of the new interval *n* to the one of the committed points on the time-line belonging to the *old* intervals in the database. Also, each clause has at most one positive literal. For a consistent problem instance this formula must be true, or each

clause must be true.

2. Then, we pick up the unit clauses (clauses that contain only one literal), which are, by definition the tightest constraints.
3. In the next step we choose a literal involving inequality (the less constraining relations) from each non-unit clause, making sure that no inequality conflicts with an "=" relation picked up the previous step.
4. In the subsequent step, we group the literals into two groups: one involving the *n*- and the other involving *n*+.
5. We then run the *PoSeq* algorithm described before on each group for finding a *Box* for each of the two boundary points of *n,* call them *Box*- and *Box*+ respectively.
    - If none of the Boxes is null, and *Box*+ does not *precede Box*-, then it is feasible to assign *n*- and *n*+ satisfying the constraints, and the system is consistent.
    - If any Box is null, then run the two algorithms, *GenerateConflictSet* and *FindMinSet* described before, in order to find the independent culprits, involving the start point and the end point.

10 - Inconsistency Detection in Other CSPs

A point to note here is that in general CSP ("offline" problem or a full-CSP rather than the incremental one) it may not be possible to detect conflicts between pair of constraints. For instance, (a<b, b<c, and c<a) conflict with each other but one cannot identify any pair here that has mutual conflict independent of the third constraint. However, in the "online" version of the problem that we addressed in the paper some constraints are "committed," or irrevocable, as with the older point-sequence. Only the constraints between the new point and the older points are under scrutiny, and hence, we could create the conflict set as a set of pairs of conflicting constraints. In a full CSP, elements in the conflict set may have to be multi-aried tuples rather than the binary tuples as in our case. The idea of having a minimum set of constraints - removal of which would eliminate all conflicts from the CSP will still be valid in the offline problem as well.

In the discrete CSP the online problem addressed here would appear as committing a value for each variable interactively (by the user), while the system only suggests the satisfiable values for the *new* variable to be added. Alternatively, on detection of inconsistency, it suggests the sources of inconsistency, expecting the user to change some constraints involving that new variable. Hence, there is no backtracking needed. Variables' values once committed becomes unchangeable, on the other hand the constraints (with the "new" variable and only the "old" committed ones) are "soft" and changeable. A future direction of our work will address how to incorporate interactive backtracking by suggesting user with some possible changes in the values of the committed variables as well. In the point-sequencing problem that issue would raise some interesting unexplored question about the topology of the solution space (different from the search space in the sense of problem solving).

Extending the work to the multi-dimensional point-based reasoning as with Cardinal-directions calculus of Ligozat (1998) or with Star-calculi of Mitra (2002) is another obvious future direction for us.

11 - Related Works

The search for "cause" of inconsistency is not new to the CSP community. Identification of such causes for improving the search algorithm is routinely explored over the last two decades. Recent works like that in (Jussien and Lhomme, 2000) proposes optimization for existing consistency search algorithms same as the Tabu search (Glover, 1989 and 1990) and backtracking-based algorithms. For example, *backjumping* and *dynamic backtracking* attempts to find the right constraints at the time of backtracking. Other works using *Dynamic Variables Ordering* (Bacchus and van Run, 1995) tend to ameliorate performance by looking forward in the tree search, applying a method to sort the variables in such a way that the general computation of known algorithms becomes faster and more efficient. The purpose of another algorithm called "Ng learning" (Hirayama and Yokoo, 2000) is to build and maintain a set of No-Good constraints, leading to inconsistencies or for which optimization is not efficient. Hirayama and Yokoo combined this method with asynchronous weak-commitment search algorithm and improved the technique of no-good learning. Similarly, *tabu-search* explores different arrangements of conflicting sets, keeping track of some solutions that should not be explored in the next iterations, rendering them *Tabu*. Several of such 'optimization algorithms' have been attempted within the last few years as it is in the (Klau et al, 2002) where the user, once again is solicited to ameliorate the efficiency of the search. *No-Good backmarking* (Richards et al, 1995) works almost the same way. It processes learning of constraints during search for which a failure occurred and intends to 'repair' some non-optimal paths. A sense of topological relations between the solutions is implicit in this heuristic. However, only a few works have investigated the direction of providing suggestions to the user (as a valid output of the system on detection of inconsistency) for the purpose of his/her interacting with the algorithm by keeping or relaxing some constraints. Amilhastre et al (2002) suggest considering the user's choices as assumptions. This paper proposes an extension of the CSP framework for which interactive decisions involves a method for computing a maximum subset of user's choices to ensure consistency. Thus, many of such "intelligent" backtracking heuristics (or other extensions of classical CSP) may be useful for further investigation on *consistency restoration* in constraint reasoning systems.

In the current framework, we handle all constraints to be of the same importance. But some externally imposed ordering could attach some measures on the constraints. Detecting "minimal cause" for inconsistency will be an optimization problem in that set up. Partial Constraint Satisfaction Problems provide a framework for such a reasoning scheme (Freuder and Wallace, 1992).

12 - Conclusion

In this paper we have described our ongoing experiments with the idea of detecting the causes of inconsistency when the later is found out in a constraint network. We have chosen the *point-sequencing online problem* as our first test bed. A greedy algorithm for the purpose has been developed and implemented. Identification of such causes behind inconsistency might help a user of a constraint reasoning system in "diagnosing" the data/knowledge base. A set of serious questions arises from this work. For instance, the "solutions" to a CSP might form a topological space of their own. Explicit knowledge of this space could possibly help in intelligent backtracking, or in our online framework – helping user to fix the "bug" by providing suggestions. This type of help will enhance the power of any practically deployed CSP system (e.g., a constraint database) significantly.

13 - References

Amilhastre, J., Fargier, H., and Marquis, P., 2002. "Consistency restoration and explanations in dynamic CSPs-Application to configuration" Artificial Intelligence journal, Vol 135, No. 1-2, pp 199-234.

Bacchus, F., van Run, P., 1995. "Dynamic Variable Ordering In CSPs" Principles and Practice of Constraint programming (CP95) pp. 258-275.

Freuder, C. E., Wallace, R. J., 1992. "Partial Constraint Satisfaction" Artificial Intelligence journal, Vol 58, No 1-3, pp 21-70.

Glover, F., 1989. "Tabu search – part I" ORSA Journal on Computing, Vol 1, No 3, 190-206.

Glover, F., 1990. "Tabu search – part II" ORSA Journal on Computing, Vol 2, No 1, 4-32.

Hirayama, K., Yokoo, M., 2000. "The Effect of Nogood Learning in Distributed Constraint Satisfaction" 20th IEEE International Conference on Distributed Computing Systems.

Jussien, N., and Lhomme, O., 2000. "Local search constraint propagation and conflict-based heuristics." Artificial Intelligence journal, Vol. 139, pp 21-45.

Klau, G. W., Lesh, N., Marks, J., and Mitzenmacher, M., 2002. "Human guided Tabu search" To appear in AAAI – 02

Ligozat, G., (1998). "Reasoning about Cardinal directions," Journal of Visual Languages and Computing, Vol. 9, pp. 23-44, Academic Press.

Mitra, D., (2002). "A class of star-algebras for point-based qualitative reasoning in two-dimensional space," Debasis Mitra, accepted to the FLAIRS-2002 Special track on Spatio-temporal reasoning, Pensacola Beach, Florida.

Richards, T., Jang, Y., Richards, B., 1995. "Ng-backmarking – an algorithm for constraint satisfaction" BT technol J Vol 13 No 1 pp. 102-109

Vilain, M., and Kautz, H., 1986. "Constraint propagation algorithms for temporal reasoning." Proceedings of the Fifth *National Conference on Artificial Intelligence (AAAI),* Philadelphia, PA, pp. 377-382.