International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:12, 2007

Finding a Solution, all Solutions, or the Most Probable Solution to a Temporal Interval Algebra Network

André Trudel, and Haiyi Zhang

***Abstract*—**Over the years, many implementations have been proposed for solving IA networks. These implementations are concerned with finding a solution efficiently. The primary goal of our implementation is simplicity and ease of use.

We present an IA network implementation based on finite domain non-binary CSPs, and constraint logic programming. The implementation has a GUI which permits the drawing of arbitrary IA networks. We then show how the implementation can be extended to find all the solutions to an IA network. One application of finding all the solutions, is solving probabilistic IA networks.

***Keywords*—**Constraint logic programming, CSP, logic, temporal reasoning**.**

I. Introduction

ALLEN [1] defines a temporal reasoning approach based on intervals and the 13 possible binary relations between them. The relations are before (b), meets (m), overlaps (o), during (d), starts (s), finishes (f), and equals (=). Each relation has an inverse. The inverse symbol for b is bi and similarly for the others: mi, oi, di, si, and fi. The inverse of equals is equals. A relation between two intervals is restricted to a disjunction of the basic relations, which is represented as a set. For example, (A m B) V (A o B) is written as A {m,o} B. The relation between two intervals is allowed to be any subset of I = {b,bi,m,mi,o,oi,d,di,s,si,f,fi,=} including I itself.

An IA (interval algebra) network is a graph where each node represents an interval. Directed edges in the network are labeled with subsets of I. By convention, edges labeled with I are not shown. An IA network is consistent (or satisfiable) if each interval in the network can be mapped to a real interval such that all the constraints on the edges hold (i.e., one disjunct on each edge is true).

A. Trudel is with the Jodrey School of Computer Science, Acadia University, Wolfville, Nova Scotia, B4P 2R6, Canada (e-mail: Andre.Trudel@acadiau.ca).

H. Zhang is with the Jodrey School of Computer Science, Acadia University, Wolfville, Nova Scotia, B4P 2R6, Canada (e-mail: Haiyi.Zhang@acadiau.ca).

Van Beek has written efficient C code to solve IA networks. His complete package includes approximately 7500 lines of code. The algorithms are described in [6]. To use the code, one must be an expert C programmer, and have a deep understanding of the algorithms. The implementation's emphasis is on efficiency, not on user friendliness.

Van Beek's algorithms have been implemented in the constraint logic programming language Eclipse (http://www.icparc.ic.ac.uk/eclipse/) by Fruhwirth [2]. Another Eclipse implementation [3], uses a meta constraint solving approach. The edges in the IA network are the variables. The domain of each variable is a set of subsets of I. The domains, although finite, can grow large if they include all the subsets of I. The meta qualitative constraint solver requires code for the operations of intersection and composition, and meta-heuristic rules. Note that both [2; 3] also handle quantitative constraints which are not dealt with in this paper.

The implementations mentioned above are non-trivial. The user must be an expert in the implementation language and software. The average user is not capable of making even simple extensions or modifications. We present an implementation which is probably not as efficient as the ones previously mentioned. Our goal is user friendliness and ease of use. The user draws an IA network, and then clicks a button for a solution. The target audience is researchers that need to quickly verify or generate a few solutions, and students entering the temporal area.

II. Theoretical Underpinning

We adopt Tsang's [5] binary CSP definition. A binary CSP of n variables x1,…,xn has a domain Di of possible values associated with each variable xi. Each Di is finite, and it may not necessarily be the case that all the domains are equal. A binary constraint, Rij, between variables xi and xj is a subset of the Cartesian product of their domains. Each Rij is finite. We also require that (a,b) $\in$ Rij if and only if (b,a) $\in$ Rji.

An IA network is a binary CSP with infinite domains. The intervals are the variables. The domain of each variable is the set of pairs of reals of the form (x,y) where x<y. The constraint between two variables i and j is the label on the edge (i,j) in the IA network.

During the past two decades, research on IA networks and finite domain CSPs has progressed relatively independently. The reason is that algorithms specifically designed for finite domains are usually not applicable to infinite domains. It was not widely known that IA networks are indeed finite domain CSPs. For example, van Beek and Manchak [6] write that "two of their heuristics cannot be applied in our context as the heuristics assume a constraint satisfaction problem with finite domains, whereas IA networks are examples of constraint satisfaction problems with infinite domains".

Recently, Thornton et al. [4] show how to convert an IA network into an equivalent non-binary CSP with finite integer domains. They observe that the relative positions of the interval endpoints in an IA network can be used to determine consistency. For example, X=(10,15) and Y=(100.5,110) is a solution to X{b}Y. This solution imposes the ordering X- < X+ < Y- < Y+ on the endpoints where X= (X-,X+) and similarly for Y. A simpler solution is to number the endpoints from left to right which results in X=(1,2) and Y=(3,4).

An IA network with two intervals is consistent if and only if each interval can be mapped to a pair of integers (a, b) where a<b, and a,b∈{1,2,3,4} such that the constraint on the edge holds. Note that it might be the case that endpoints from different intervals get mapped to the same integer (e.g., as in the case of X {=}Y). Thornton et al. [4] generalize the integer mapping to:

Theorem 1: Each interval in an IA network with n intervals can be mapped to a real interval such that all the constraints on the edges hold if and only if each interval in the IA network can be mapped to an interval with integer end-points in the range 1…2n such that all the constraints on the edges hold.

Based on theorem 1, Thornton et al. [4] convert an IA network to a non-binary CSP with finite domains. Each endpoint becomes a variable with domain {1,…,2n}. A label on an edge from X to Y in the IA network imposes a constraint on some or all of the variables X-, X+, Y-, and Y+. For example, X {d} Y generates the constraint (Y- < X-) & (X+ < Y+) which is non-binary since the constraint involves 4 variables. They then apply local search techniques on the non-binary CSP.

We use the finite domain transformation described above. But instead of local search, we use Eclipse and constraint logic programming techniques to solve the IA network.

III. System Overview

An overview of our implementation's components is given in figure 1. The user interacts with the implementation via the graphical user interface (GUI). The GUI, built using jGraph (http://www.jgraph.com/), has 2 windows. The user enters a graph in the top window. There are buttons for drawing nodes and edges. The nodes represent intervals and are each numbered 1, 2, etc. Allen's interval relationships are entered on the edges separated by commas. There are no restrictions on the size or shape of the graph. When the user clicks on the button to request a solution, the solution is drawn in the GUI's bottom window. The graph is re-drawn as entered by the user. Each edge will be assigned a unique label. The entire graph is consistent. If not consistent, a warning message is displayed instead.

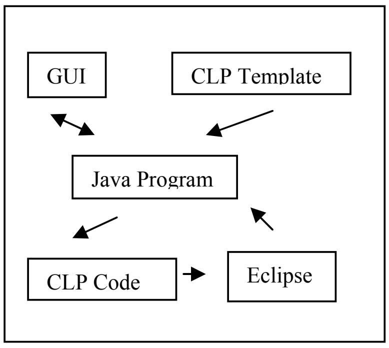

Fig. 1 Implementation

IV. Execution Flow

After the user has entered a graph and requested a solution, control is given to the Java program in figure 1. The program first reads in a constraint logic program template. The template is updated with information from the particular graph to solve. The completed logic program is then passed on to Eclipse. Eclipse will solve the graph and then pass the solution to the Java program. The java program will then display the solution in the GUI.

The CLP template file contains constraint code for Allen's relations. The relations are implemented by placing restrictions on the endpoints. For example, interval (XL,XR) is before (YL,YR) if and only if XR < YL. In Eclipse, we write: b(XL,XR,YL,YR,1) :- XR < YL. The "1" in the last parameter of b is a numeric representation of b and is used to keep track of the relationships on the edges. The relations are numbered from 1 to 13. The after relationship bi is implemented in terms of before: bi(XL,XR,YL,YR,2) :- b(YL,YR,XL,XR,1). The other relations are similarly implemented. The CLP template file also contains clauses to enforce that the left endpoint of each interval precedes its right endpoint.

The Java program copies the contents of the CLP template file to the CLP code file. Graph specific code is then added to the file. Assume we are given an IA network with n intervals (nodes) numbered from 1 to n. The left and right endpoints of the i'th interval are labeled Li and Ri respectively. The set of endpoints is represented in Eclipse as: EndPoints = [L1,R1, L2,R2,…,Ln,Rn]. For example, if n=4 we have: EndPoints = [L1,R1, L2,R2, L3,R3, L4,R4].

The range of each interval endpoint must be explicitly specified and is between 1 and 2n. In Eclipse, this is written in the following format: EndPoints :: 1..2n. For example, if n=4 we write: EndPoints :: 1..8.

The edges are also numbered. For example, if there are 5 edges: Edges = [E1, E2, E3, E4, E5].

Every edge constraint is a disjunction of relationships. We represent the disjunction directly. For example, let the constraint on edge E1 between intervals 1 and 2 be meets or overlaps (i.e., {m,o}). This constraint is represented in Eclipse as: (m(L1,R1, L2,Y2,E1); o(L1,R1, L2,Y2,E1)). Singleton labels are represented directly. For example if instead we have {m} we write: m(L1,R1, L2,Y2,E1).

The problem's constraints have now all been specified and we request Eclipse to generate a solution with the query: finda_solution(Edges). If a solution is found, Edges will be bound to a list of integers. Each integer represents an Allen relation for an edge.

V. One Solution

Figure 2 is a screenshot of the GUI. The top window contains the IA network entered by the user. The solution is shown in the bottom window. The CLP code file generated for this example is shown in figure 3. Note that it is typical that only 1 page of Eclipse code is generated to solve the IA network.

VI. All Solutions

Consider the simple IA network in figure 4. Assume we are only interested in solutions that assign a single label to the edges shown. For example, we don't care about the temporal relationship between the two bottom nodes. Every label can appear in a solution for a total of 8 solutions. The straightforward approach for finding all these solutions is to first find one solution, and then backtrack to find the others. But, we must be careful what we backtrack over:

Labels: Traditional IA network software assumes that each pair of nodes has an edge between them. If an edge is not explicitly shown, it is assumed to have a label of I. If we backtrack over the labels in the network, we must consider 17,576 possible solutions. Notice the combinatorial explosion with a network of only 4 nodes!

Endpoints: If instead, we use software based on Thornton et al's approach [4], we have a network of 4 nodes and must find an assignment of each interval's endpoints to an integer in the range from 1 to 8. Assume we have the label "b" between two nodes X and Y. There are 70 different ways that the endpoints of X and Y can be assigned to integers in the range from 1 to 8 so that X {b} Y holds. For example, one assignment is X = (1,2) and Y = (5,8). For meets, there are 56 different assignments for the endpoints. Therefore, to find all possible solutions in figure 4 by backtracking over the possible endpoint assignments, we must consider over 60 billion possibilities (i.e., 703 x 563).

Note that in the above, the worst case number of possibilities to backtrack over is given. Clever algorithms and heuristics can reduce the number of possibilities.

Another approach is to backtrack over the candidate solutions. We first generate a candidate solution which has one label on each edge. We check if this is a solution. We then generate the next candidate solution and so on. For example, for the network in figure 4 we generate and test 8 candidate solutions. This is the approach we adopt and experiment with in this paper.

The code for finding a single solution was left untouched. We added code to the "CLP Code" file in figure 3 which first generated all the possible solutions. We then apply the original code for finding a single solution to each possible solution to verify if indeed it is a solution. The set of valid solutions are passed back to the "Java Program". The program stores the solutions in a two dimensional array and displays the solutions in the GUI one at a time.

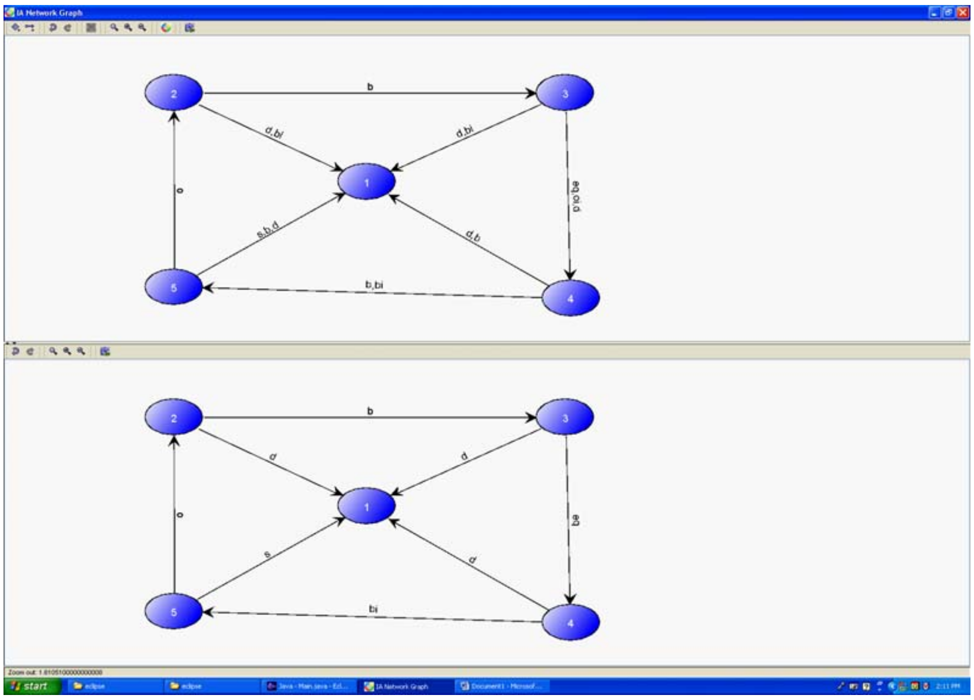

Fig. 2 Example

VII. Most Probable Solution

Let us now add probabilities to Allen's relations on the edges. Each relation is assigned a probability, and the probabilities on an edge sum to 1. One interpretation for the numbers is preference. If there is no edge between two nodes, we assume the label is I, and each relation has equal probability 1/13.

A solution to a probabilistic IA network is a consistent labeling which maximizes the product of the probabilities associated with each label in the solution.

We use the all solutions feature presented in the previous section to solve a probabilistic IA network. We first strip off the probabilities, and then generate and store all the solutions. For each solution, we compute a value by re-assigning a probability to each label in the solution and taking the product. The solution with the highest value is the solution to the probabilistic network. This extra processing is added to the "Java Program" in Figure 1. The Eclipse code was not modified. It is trivial to add code to find the least likely, or median solution.

VIII. What Size Network can be Solved in a Reasonable Amount of Time?

Since finding a single solution to an IA network is an NP complete problem, finding all the solutions is not feasible for large difficult problem instances. Our implementation is targeted at small instances.

The implementation solves the network in Figure 4 almost instantaneously. Some networks of 10 nodes and edges can take overnight to find all the solutions.

```
:-lib(ic).
:-lib(ic_global).
:-lib(propia).

b(_,XR,YL,_,1):- XR < YL.
bi(XL,XR,YL,YR,2):- b(YL,YR,XL,XR,_).
m(_,XR,YL,_,3):- XR = YL.
mi(XL,XR,YL,YR,4):- m(YL,YR,XL,XR,_).
o(XL,XR,YL,YR,5):- XL < YL, XR > YL, XR < YR.
oi(XL,XR,YL,YR,6):- o(YL,YR,XL,XR,_).
d(XL,XR,YL,YR,7):- XL > YL, XR < YR.
di(XL,XR,YL,YR,8):- d(YL,YR,XL,XR,_).
s(XL,XR,YL,YR,9):- XL = YL, XR <YR.
si(XL,XR,YL,YR,10):- s(YL,YR,XL,XR,_).
f(XL,XR,YL,YR,11):- XL > YL, XR = YR.
fi(XL,XR,YL,YR,12):- f(YL,YR,XL,XR,_).
eq(XL,XR,YL,YR,13):- XL = YL, XR = YR.

lrConstraint([]).
lrConstraint([L,R|T]):- L < R,lrConstraint(T).

finda_solution(Edges) :-
    Edges =[E1,E2,E3,E4,E5,E6,E7,E8],
    EndPoints= [L1,R1,L2,R2,L3,R3,L5,R5,L4,R4],
    EndPoints:: 1..10,
    lrConstraint(EndPoints),
    (s(L5, R5, L1, R1,E1);b(L5, R5, L1, R1,E1);
            d(L5, R5, L1, R1,E1)),
    (d(L2, R2, L1, R1,E2);bi(L2, R2, L1, R1,E2)),
    (d(L3, R3, L1, R1,E3);bi(L3, R3, L1, R1,E3)),
    (d(L4, R4, L1, R1,E4);b(L4, R4, L1, R1,E4)),
    b(L2, R2, L3, R3,E5),
    (eq(L3, R3, L4, R4,E6);oi(L3, R3, L4, R4,E6);
            d(L3, R3, L4, R4,E6)),
    (b(L4, R4, L5, R5,E7);bi(L4, R4, L5, R5,E7)),
    o(L5, R5, L2, R2,E8),
    labeling(EndPoints).
```

Fig. 3 CLP code file for the example

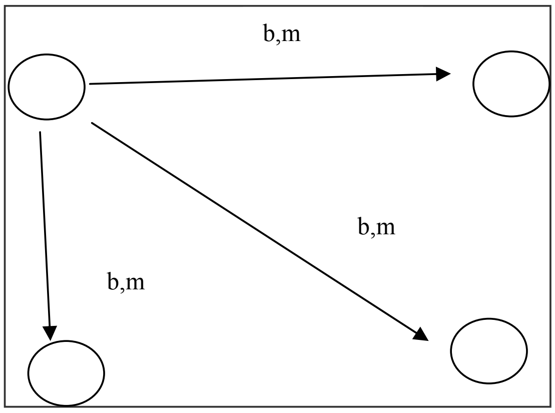

Fig. 4 Simple IA network

IX. Future Work

Future work will involve generating large IA networks and comparing the efficiency of the following algorithms:

- our implementation described in this paper,
- van Beek's C code, and
- off the shelf finite domain binary CSP software.

Before this can be done, we need to add a file I/O feature to the GUI (i.e., allow file input and output to store large test networks).

Future work will also involve optimizing the software. One enhancement we are investigating, is exploiting the presence of "cut edges" or "bridges" in the network. These edges can be found in linear time using a DFS based algorithm. Either of the labels on the bridge can appear in a solution, and they do not influence other labels. If we remove the bridge, we are left with two smaller networks that can be solved quickly and their solutions combined.

X. Conclusion

Our implementation is of benefit to non-technical users. The user does not need to learn specialized software and algorithms. The implementation allows the user to draw any IA network and solve it. Emphasis is on ease of use, not efficiency. We challenge the reader to find a simpler and more direct method for solving qualitative IA networks.

To our knowledge, this is the first time all the solutions to an IA network and probabilistic IA networks have been solved. Unfortunately, only small toy problems can be tackled with the approach described in this paper.

ACKNOWLEDGEMENT

Research of the first author is supported by an NSERC Discovery Grant. We thank Mohammad Masum for coding version 0 of the implementation for his senior undergraduate project.

REFERENCES

[1] J.F. Allen. Towards a general model of action and time, Artificial Intelligence, 23(2), 1984, p. 123-154.
[2] T. Fruhwirth. Temporal reasoning with constraint handling rules, Technical report ECRC-94-05, European computer-industry research centre, Germany, 1994.
[3] E. Lamma, M. Milano, and P. Mello. Temporal reasoning in a meta constraint logic programming architecture, Third international workshop on temporal representation and reasoning (TIME'96), Florida, 1996, p. 128-135.
[4] J. Thornton, M. Beaumont, A. Sattar, and M. Maher. A local search approach to modeling and solving interval algebra problems, The journal of logic and computation, 4(1), 2004, p. 93-112.
[5] E. Tsang. Foundations of constraint satisfaction, Academic Press, 1993.
[6] P. van Beek and D.W. Manchak. The design and experimental analysis of algorithms for temporal reasoning, Journal of Artificial Intelligence Research, 4, 1996, p. 1-18.