Efficient Heuristics for Solving Probabilistic Interval Algebra Networks

Kai Zhang and André Trudel
Jodrey School of Computer Science
Acadia University
Wolfville, Nova Scotia, B4P 2R6, Canada

Abstract

A probabilistic interval algebra (PIA) network is an interval algebra network with probabilities associated with the labels on an edge. The probabilities on each edge sum to 1. A solution is a consistent scenario where the product of the probabilities associated with each unique edge label is maximized. In this paper we investigate previous PIA network solution algorithms, and propose new ones. Our first algorithm is based on best first search and guarantees to output the optimal solution. However, this algorithm is only feasible for toy problems. We augment the algorithm with three heuristics. Although our proposed algorithm does not guarantee an optimal solution, it is very useful in practice. Good solutions can be generated quickly.

1. 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).

We extend IA networks with probabilities. Each relation on an edge is assigned a probability, and the probabilities sum to 1. An example of a probabilistic IA (PIA) network is shown in Figure 1. One interpretation for the numbers is preference. For example, the user may prefer a solution which contains one relation instead of another on a specific edge. 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 PIA network is a consistent labeling which maximizes the product of the probabilities associated with each label in the solution. For example, the solution to the PIA network in Figure 1 is shown in Figure 3.

A scenario of a PIA or IA network has the same edges and nodes as the original network, with the added restriction that each edge has a single label taken from the original network. Figure 2 and Figure 3 show two scenarios of the network in Figure 1. The scenario in Figure 2 is inconsistent (it is impossible to have four intervals each before the other), while the one in Figure 3 is consistent.

In the next section, we give an overview of two previous solutions to PIA networks. Experimental results show that both approaches are unacceptably inefficient in practice. The remainder of the paper presents heuristics which improve the performance of one of the approaches.

2. Previous work

There are two previous approaches for solving PIA networks. The first is a set of algorithms proposed by Ryabov and Trudel [3][4]. The algorithms implement probabilistic versions of the inversion, composition, and addition operations. A standard path-consistency algorithm is modified to deal with uncertain interval relations. A greedy backtracking algorithm with heuristics is also used. The approach locally optimizes the probabilities at the edge level to hopefully generate an optimal or near optimal solution. Note that the algorithms in [3] and [4] are presented without implementations. For this paper, we implemented and tested the algorithms. Test results are given at the end of this paper. Unfortunately, the local greedy heuristic does

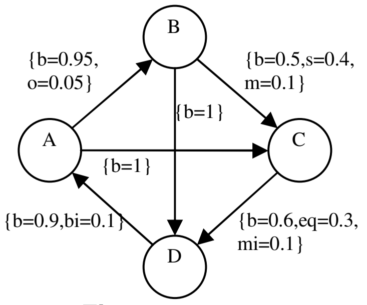

Figure 1. A PIA Network

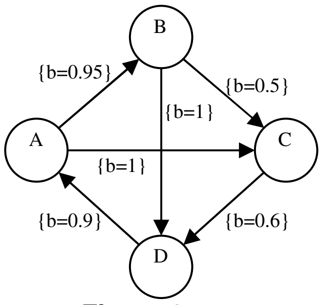

Figure 2. An inconsistent scenario

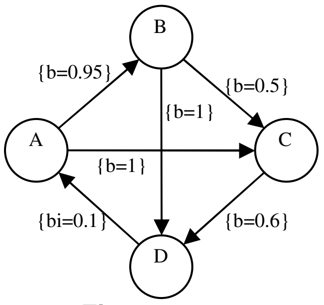

Figure 3. A consistent scenario which is also the solution

not perform well.

Another approach is to generate all the consistent scenarios and then rank them [5]. We first strip off the probabilities on each edge, and then generate all the possible scenarios. Each scenario is checked to see if it is a solution. If yes, it is stored. For each solution, we compute its probability by assigning a probability to each label which is equal to its probability in the original PIA network. We then take their product. The solution with the highest value is the solution to the PIA network. It is trivial to add code to find the least likely, or median solution. As expected, generating all the solutions to an NP complete problem is not a viable approach. In the worst case, an n node network has $13^{(n^2-n)/2}$ scenarios. For example, a 4 node network can have up to 4,826,809 scenarios. Experimental results presented in [5] confirmed that this solution is not feasible for even small networks.

In this paper, we pursue the second approach.

3. Proposed algorithm

Instead of generating all the scenarios, we first generate the one with the highest probability. If it is consistent, we have solved the PIA network. If not, we generate the scenario with the second highest probability and repeat the process. The algorithm continues until a solution is found. Note that if scenarios are generated and tested in descending order of probability value and a solution exists, this approach is guaranteed to find the optimal solution.

One implementation issue arises when we attempt to compute the product of the probabilities on each edge of a scenario. The result is too small to store in a typical arithmetic data type (e.g., the product is in the order of 10^{-200} or smaller for large networks). We use an alternative approach to evaluate the probability of a scenario. We first define the absolute ratio of an edge in a scenario to be the edge label's probability divided by the probability of the largest label on that edge in the PIA network. For example, the absolute ratio of edge DA in Figure 2 is 0.9 / 0.9 = 1 and in Figure 3 it is 0.1 / 0.9 = 0.111. We then define the probability of a scenario as the product of all its edges' absolute ratios. Note that the scenario constructed by taking the largest probability on each edge (e.g., Figure 2) has a probability of 1. All other scenarios will have a probability between 0 and 1 (e.g., Figure 3 has a probability of 1 * 1 * 1 * 1 * 1 * 0.111 = 0.111).

The GetNextScenario function in Algorithm 1 generates the scenarios in descending order of probability. After generating the k-1 largest scenarios, it is guaranteed to generate the kth largest one. The first 8 scenarios generated by the GetNextScenario function for Figure 1 are shown in Figure 4 (edges AC and BD are omitted since their labeling never changes).

For each scenario generated by Algorithm 1, we check if it is a solution by verifying if it is path consistent. If consistent, we have a solution. Otherwise, the next scenario is generated.

Due to a lack of space, we have omitted Algorithm 1's correctness and complexity proofs. It is a polynomial time algorithm (quadratic). Note that path consistency is also a polynomial algorithm. Therefore, the kth largest scenario can be generated and tested in polynomial time. The bottleneck is that in the worst case, an exponential number of scenarios must be generated and tested. Experimental results confirm that this approach is not feasible for large networks. In the remainder of the paper, we present heuristics to speed up the approach.

```
function GetOptiamlSolution(Network)
1. initialize found_scenario_list
2. CurrentScenario=GetNextScenario(found_scenario_list)
3. while(CurrentScenario is not path consistent)
4.     CurrentScenario=GetNextScenario(found_scenario_list)
5.     if CurrentScenario is NULL // no solution
6.         return NULL
7. return CurrentScenario

function GetNextScenario(found_scenario_list)
1. initialize largest_scenarios_list
2. if found_scenario_list is empty        // the first scenario
3.     let every variable of scenario_k be its largest value.
4.     make the item's branch, sort it by descending order, and then set its CurrentItem pointer to the first
       item
5. else
6.     do
7.         for each branch in found_scenario_list
8.             if CurrentItem pointer is not at the end of the branch
9.                 get the item under the CurrentItem pointer
10.                compute its probability and put it to largest_scenarios_list
11.                move the CurrentItem pointer to the next item
12.        if largest_scenarios_list is empty    //all the scenarios have been emulated
13.            return NULL
14.        find the item within largest_scenarios_list that has the largest probability.
15.        make the item's branch, sort it by descending order, and then set its CurrentItem pointer to the
           first item
16.    while the largest scenario found in step 10 is not in found_scenario_list
17.    let scenario_k be the largest scenario
18. push scenario_k into the found_scenario_list.
19. return scenario_k
```

Algorithm 1. Generate and test scenarios in decreasing order of probability

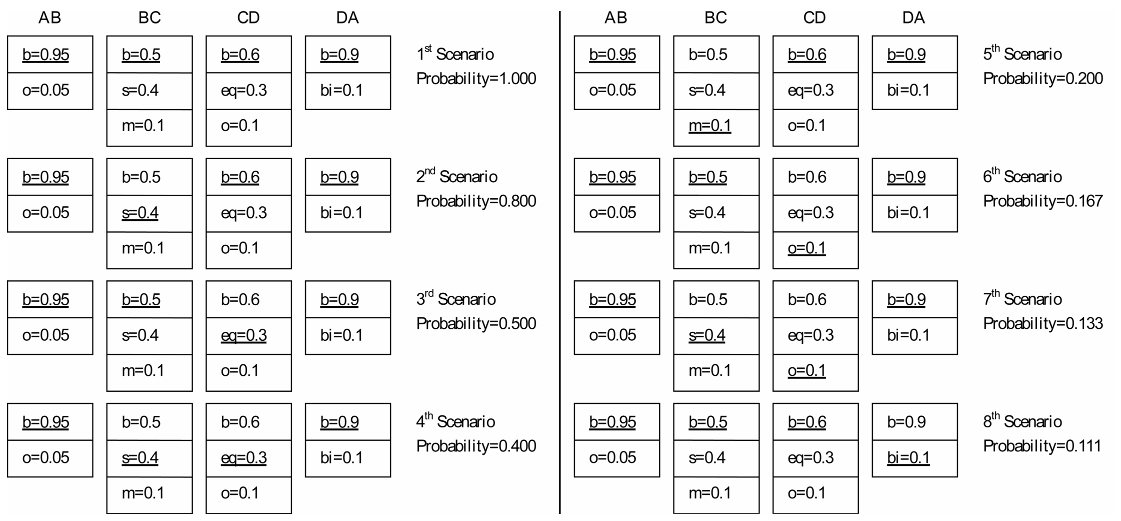
AB
BC
CD
DA
b=0.95
o=0.05
b=0.5
s=0.4
m=0.1
b=0.6
eq=0.3
o=0.1
b=0.9
bi=0.1
1st Scenario Probability=1.000
2nd Scenario Probability=0.800
3rd Scenario Probability=0.500
4th Scenario Probability=0.400
5th Scenario Probability=0.200
6th Scenario Probability=0.167
7th Scenario Probability=0.133
8th Scenario Probability=0.111

Figure 4. The first eight scenarios

4. Heuristic 1: Ignore I edges

Missing edges in the PIA network are assumed to be present with a label of I. Each of the 13 relations is assigned a probability of 1/13. The edge's absolute ratio is always 1 and therefore has no effect on the probability of the final solution. When generating scenarios, we add the heuristic that scenarios that differ only in the particular label assigned to a missing edge are ignored. For example, in Figure 5 we do not consider the AC and BD edges when generating scenarios. In this case, it reduces the number of scenarios from 6084 to 36.

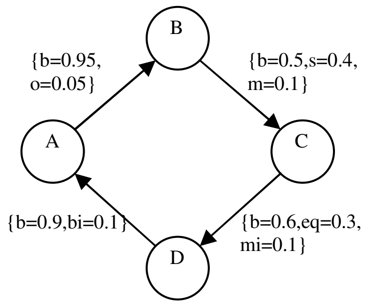

Figure 5. An example of ignoring I edges

5. Heuristic 2: Inconsistent scenario recognition

When the path consistency algorithm identifies three edges whose labels form an inconsistent sub-network, the scenario is inconsistent. In practice, this inconsistent sub-network can be extended to more than three edges. Algorithm 2 is used when path consistency detects an inconsistency at an edge. Algorithm 2 uses backtracking and breadth first search to find the maximal inconsistent sub-network. In the future, we can immediately discard scenarios which contain one of the inconsistent sub-networks.

We explain Algorithm 2 using an example. The scenario in Figure 6 is not consistent. When checking path consistency, suppose we start with the triangle {BC, CD, BD} and then {AB, BD, AD}. There is no edge between B and D in the original network, indicating the label is {I}. We use the symbol "+" to represent the addition operation (which in this case is set intersection) and "•" for the composition operation. After first checking, BD = BD + (BC • CD) = {I} + ({b}•{b}) = {b}, BD is updated to {b}, and the stack for BD has its first item, which is the modification: {{I}→{b}, by BC and CD}. We then check path consistency on the triangle {AB, BD, AD}. Since AD = AD + (AB • BD) = {bi} + ({b}•{b}) = {∅}, the stack for AD has its first item, which is the modification: {{bi}→{∅}, by AB and BD}. Now the path consistency algorithm finds the conflict triangle {AB, BD, AD} (shown in Figure 7 with broken lines). Then we run the GetInconsistentSubnetwork function on the empty edge AD. After steps 1 to 8, the BadSubnetwork is {AD = bi}, and q is {{AD, Pivot=1}}. Then the algorithm calls the BacktrackStack function. From the first item in AD's stack, the algorithm finds that the modification came from AB and BD (step 2 in BacktrackStack function). And then the algorithm finds that their pivots are 0 and 1, respectively (step 3 in BacktrackStack function). The pivots indicate that the conflict came from the original value of AB and the first modification of BD. Then it returns to the GetInconsistentSubnetwork function. AB is set to BadSubnetwork (steps 13 to 16), while BD is set to q (steps 17 to 20), because its pivot is not 0. Now q is {{BD, Pivot=1}}. The loop goes back to step 9 and then backtracks q's last item, which is BD. The BacktrackStack function returns that the modification of BD came from BC and CD, and both of their pivots are 0, indicating the conflict came from their original values. So BC and CD are then added to BadSubnetwork, and the function terminates.

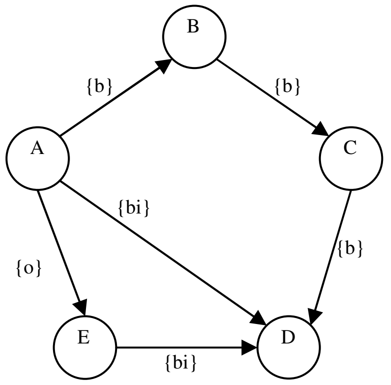

Figure 6. Inconsistent scenario

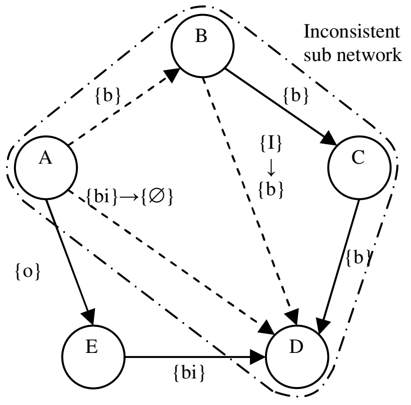

Figure 7. The inconsistent sub-network

Finally, the BadSubnetwork is {AD, AB, BC, CD}, which is shown in Figure 7. In the future, any scenarios generated which contain this inconsistent sub-network can immediately be rejected as inconsistent.

```
function GetInconsistentSubnetwork(BacktrackStacks, BadSubnetwork)
1. initialize breath-first search queue: q; push $R_{i,k}$'s stack to q, the stack's last item as the pivot // for part A
2. find the conflict part B within the $R_{i,k}$'s stack
3. if conflict part B comes from the original value of $R_{i,k}$
4.     push $R_{i,k}$ to BadSubnetwork
5. else
6.     push $R_{i,k}$'s stack to q; set the position of conflict part B as the pivot
7. while q is not empty // do the breadth-first search
8.     item = q's last item
9.     pop q's last item
10.    BacktrackStack(item, edge1, pivot1, edge2, pivot2)
11.    if pivot1 is 0
12.        put edge1 to BadSubnetwork
13.    else
14.        put edge1 to q, pivot1 as pivot
15.    if pivot2 is 0
16.        put edge2 to BadSubnetwork
17.    else
18.        put edge2 to q, pivot2 as pivot

function BacktrackStack(BacktrackItem, var edge1, var pivot1, var edge2, var pivot2)
1. compute the BacktrackItem's difference before and after its pivot and assign it to Diff
2. get the two edges in the BacktrackItem's pivot, assign them to edge1 and edge2 respectively
3. within edge1 and edge2's stacks, search the values that result in the Diff in BacktrackItem's stack, and
   assign the values' positions to pivot1 and pivot2 respectively
```

Algorithm 2. Inconsistent scenario recognition

```
function InitGeneralizedScenario(GroupedNetwork, OriginalNetwork, Threshold)
1. copy the vertices from OriginalNetwork to GroupedNetwork
2.
3. for each edge of the OriginalNetwork //group the entries according to the threshold
4.     sort the entries by descending order of their probabilities
5.     CurrentGroup=create a new group
6.     put entry[1] into CurrentGroup
7.     CurrentGroup's probability= entry[1]'s probability
8.
9.     for i=2 to 13
10.        if (entry[i] / CurrentGroup's probability) $\geq$ Threshold
11.            put entry[i] in current group
12.        else
13.            put CurrentGroup into GroupedNetwork's corresponding edge
14.            CurrentGroup=create a new group
15.            put entry[i] into CurrentGroup
16.            CurrentGroup's probability= entry[i]'s probability

function GetNextGeneralizedScenario(found_scenario_list, GroupedNetwork)
1. call GetNextScenario(found_scenario_list) on the GroupedNetwork
```

Algorithm 3. Generalized scenario algorithm

6. Heuristic 3: Grouping scenarios with similar values

Instead of generating scenarios with one label per edge, we group similar valued labels on the edges. A generalized scenario is defined to be a scenario with the exception that each edge can have one or more *similar* labels. The intuition for using generalized scenarios is that if it is found to be inconsistent using the path consistency algorithm, it can be immediately discarded. This is potentially eliminating a large number of scenarios from the search space. If a generalized scenario is path consistent, it can potentially contain a solution. Note that since edges may contain more then one entry, path consistency does not guarantee global consistency [6].

We use a threshold value to divide an edge's labels into similar groups. For example, suppose there is an edge with labels {o=0.4, eq=0.3, bi=0.2, b=0.05, m=0.05}, and we set the threshold at 0.7. If the ratio between a label and the largest label is greater than the threshold value, they are grouped together. Consider o and eq: 0.3/0.4=0.75 which is larger than 0.7. Therefore, o and eq are grouped together. bi is not put in that group: 0.2/0.4 = 0.5 < 0.7. b is not grouped with bi: 0.05/0.2=0.25 < 0.7. Therefore bi becomes a singleton group. b and m are put in a third group since 0.05/0.05 = 1 > 0.7. The sets of similar labels for this edge are {{o=0.4, eq=0.3}, {bi=0.2}, {b=0.05, m=0.05}}.

The threshold and group size are inversely related. In general, the larger the threshold value the smaller the groups. Conversely, the smaller the threshold, the larger the groups will be. For example, if the threshold in the previous example is set at 0.5 instead of 0.7 the grouping becomes {{o=0.4, eq=0.3, bi=0.2}, {b=0.05, m=0.05}}. At 0.8 it is {{o=0.4}, {eq=0.3}, {bi=0.2}, {b=0.05, m=0.05}}.

We use Algorithm 3 to generate generalized scenarios. This algorithm groups the entries in the network's edges, and then generates generalized scenarios using the same approach as Algorithm 1.

```
function FindSolution(InitialThreshold, Step)
1. Threshold = InitialThreshold
2. define GroupedNetwork, FoundSolutionList, BadSetList
3. define CurrentGeneralzedScenario , PathConsistentGeneralizedScenarioList
4. do
5.    clear FoundSolutionList, BadSetList, GroupedNetwork
6.    InitGeneralizedScenario(GroupedNetwork, Threshold)
7.    do
8.        do
9.            GetNextGeneralizedScenario(FoundSolutionList)
10.       while FoundSolutionList.LastItem is not in BadSetList
11.
12.       CurrentGeneralzedScenario = FoundSolutionList.LastItem
13.       if CurrentGeneralzedScenario is not path-consistent
14.           call GetInconsistentSubnetwork on CurrentGeneralzedScenario
15.           push the inconsistent sub-netwrok to BadSetList
16.   until CurrentGeneralzedScenario is consistent or no generalized scenario left
17.   if Threshold == 1.0
18.       if vanBeek's algorithm [7] finds a solution in the CurrentGeneralzedScenario
19.           goto 25
20.       else // the path-consistent generalized scenario is not consistent
21.           backtrack to the last item in PathConsistentGeneralizedScenarioList
22.           jump to the next generalized scenario and goto 4
23.   push CurrentGeneralzedScenario to PathConsistentGeneralizedScenarioList
24.   Threshold = Threshold + Step ( but keep Threshold less than or equal to 1.0 )
25. fill the probabilities to the solution
26. return the solution
```

Algorithm 4. Combination of all the heuristics

7. Heuristic 4: Combine previous heuristics and recursively apply threshold values

The previous heuristics are combined and called Algorithm 4. Algorithm 4 does not guarantee to output the optimal solution. But, it does generate a consistent scenario if one exists. Algorithm 4 recursively applies Algorithm 3 with increasing threshold values in order to find a path consistent generalized scenario that potentially contains solutions having equal probabilities. We denote the threshold as *h*. During the first iteration, *h* is set to a low value. This generates generalized scenarios with large groupings on the edges. If a generalized scenario is found to be path consistent, we do not generate each of its singleton label scenarios and check each for path consistency. Instead, the generalized scenario is a PIA network. We increase *h* and recursively apply the algorithm to the generalized scenario. Note that the edge label groupings will be smaller. During each iteration, *h* is increased by a constant value. When *h* reaches 1.0, we have a path consistent generalized scenario whose scenarios (with a single label on each edge) have equal probability values. Note that edges that were originally labeled with I may still contain multiple labels. To find a consistent scenario, we first remove the probabilities on the edges of the generalized scenario. We now have a regular (simplified) IA network which can be quickly solved using van Beek's C code. If van Beek's algorithm cannot find a solution, Algorithm 4 backtracks to explore the next best generalized scenario (fortunately, this situation has not arisen during our experiments).

8. Algorithm 4's properties

Soundness: Before terminating, Algorithm 4 verifies the solution using van Beek's IA algorithm. This guarantees that the scenario truly is a solution.

Completeness: Our algorithm does not necessarily generate the optimal solution. But if a consistent scenario exists, it will eventually be generated as part of a generalized scenario. Due to space limitations, we do not provide a formal proof.

Error bound: In the best case, Algorithm 4 finds the solution with the highest probability. In the worst case, it finds a generalized scenario in the first iteration that does not include the optimal solution, and in which the only consistent scenario is the one having the lowest probability among the scenarios in this generalized scenario. The difference between the probability of the optimal and generated solution is between 0 and $1-1/h^m$ where *m* is the number of non-I edges.

9. Experimental results

In addition to the PIA network, Algorithm 4 requires an initial threshold and step value as input. The step value is used as a threshold increment during each iteration of the algorithm. The threshold and step value have a significant effect on performance. For example, we randomly generated a 10 node network. We solved the network using different thresholds while maintaining the step value at a constant 0.01. The initial thresholds used are: 0.10, 0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, 0.48, 0.50, and 0.52. Results are presented in Figure 8 and Figure 9. When the initial threshold is low, the label groupings on the edges are large. This leads to a fast solution (Figure 8) but, the probability of the consistent scenario is generally low (Figure 9). As the initial threshold is increased, the edge label groupings decrease in size. Also, as shown in Figure 8, running time increases.

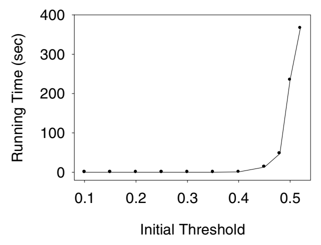

Figure 8
Initial threshold's effect on running time

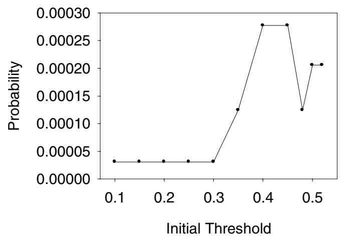

Figure 9
Initial threshold's effect on result's probability

Note that the algorithm ran overnight without terminating when *h* was set initially greater than 0.52. The tradeoff for the increased execution time, are better solutions when the *h* is increased (Figure 9). Note that there is an optimal range for this example. The best solutions were obtained with an initial threshold value of 0.4 and 0.45. These solutions were found quickly. Higher initial threshold values increase running time and decrease the solution's value.

Further work will involve empirically studying the relationship between threshold/step and running time/solution value. One useful heuristic we use for small networks is a higher initial threshold and larger step value. For large networks, we use a lower initial threshold and smaller step value.

To evaluate the performance of Algorithm 4, we implemented the locally greedy algorithms presented in [3][4]. For the comparison, 35 random networks are generated using Nebel's gencsp program. The networks are divided into 7 groups of 5 networks. The 7 groups consist of 5, 10, 15, 20, 25, 30 and 35-node networks. For Algorithm 4, we set the initial threshold and the step value at 0.7 and 0.05 respectively for the 5-node group, 0.3 and 0.03 for the 10 and 15-node groups, 0.3 and 0.02 for the 20 and 25-node groups and, 0.3 and 0.01 for the 30 and 35-node groups.

For all 35 networks, Algorithm 4 was faster in returning a solution. Also, the probability of the solution returned by Algorithm 4 was larger for 34 of the 35 networks. The only case where Algorithm 4 returned an inferior solution was for one of the 10 node networks.

Table 1. Statistical results of the evaluation (time is in seconds)

N	Algorithm 4		Greedy Search		r
	$\bar{t}$	σ	$\bar{t}$	σ	
5	0.030	0.007	0.041	0.011	9.114
10	0.088	0.019	7.082	5.095	12.112
15	2.281	2.648	50.880	14.253	78.769
20	2.928	2.732	389.980	279.200	1,265.144
25	8.281	9.197	1,036.969	543.151	10,152.586
30	20.980	13.325	3,493.414	2,671.562	71,135.595
35	14.456	12.360	8,963.948	7,558.422	3,120,738.689

A summary of the results of running both algorithms on the 35 networks is shown in Table 1. The results are summarized by group. For example, the first line of data is for the first group which consisted of five 5 node networks. For both algorithms, we give the average running time $\bar{t}$ and the running time's standard deviation σ for the 5 networks in the group. For example, on average Algorithm 4 required 0.030 seconds to solve each of the five 5 node networks. To compare the probabilities of the solutions found by each algorithm, we use the following formula:

$$r = \sqrt[5]{\prod_{i=1}^{5} \frac{\Pr(Solution_{\mathrm{A4}i})}{\Pr(Solution_{Greedyi})}}$$

For network *i*, $\Pr(\mathrm{Solution}_{\mathrm{A4}i})$ and $\Pr(\mathrm{Solution}_{\mathrm{Greedy}\ i})$ are the probabilities of the solutions returned by Algorithm 4 and the greedy algorithm respectively. *r* represents the average ratio between the two algorithms' solutions on the networks within a group. On average, Algorithm 4 is superior to the greedy algorithm for a group of networks when $r > 1$.

Based on the results in Table 1, we conclude that Algorithm 4 is much more efficient than the greedy search algorithm. The efficiency difference is drastic for large networks. For the 35-node group, Algorithm 4 takes 6.224 seconds on average, while the greedy search algorithm takes over 2 hours on average, which is about 1440 times slower. In each group, Algorithm 4's average execution time and standard deviation are smaller than the corresponding entries for the greedy algorithm. Also, r is > 1 and increasing for each group.

10. Conclusion and future work

Even for small PIA networks, it is not feasible to generate all the scenarios. Instead, we generated the scenarios in a best first order. We then added three heuristics to this algorithm: ignore I edges, identify maximal inconsistent sub-networks, and group similar valued edge labels. The resulting algorithm is guaranteed to terminate if a solution exists, but does not always return the optimal solution.

There are no benchmark problems in this area. In order to do a comparison, we had to implement previously defined algorithms. Our new algorithm significantly outperformed them.

Future work will involve experiments with larger networks. We will also investigate the relationship between initial threshold and step value on performance. Given a particular PIA network, what threshold and step should be used? What properties of the network should influence the choice? We will also investigate the use of a dynamic step value.

References

[1] J.F. Allen, "Maintaining knowledge about temporal intervals", *Comm. ACM*, 26, 1983, pp. 832-843.

[2] J.F. Allen, "Towards a general model of action and time", *Artificial Intelligence*, 23(2), 1984, pp. 123-154.

[3] V. Ryabov and A. Trudel, "Finding a consistent scenario in a probabilistic temporal interval network", *Artificial Intelligence and Soft Computing (ASC 2004)*, Marbella, Spain, 2004, pp. 347-351.

[4] V. Ryabov and A. Trudel, "Probabilistic Temporal Interval Networks", *11th International Symposium on Temporal Representation and Reasoning (TIME'04)*, 2004, pp. 64-67.

[5] A. Trudel, "Finding all the solutions to an IA network", *Workshop on Spatial and Temporal Reasoning (held during IJCAI 2005),* Edinburgh, Scotland, 2005, pp. 71-76.

[6] P. van Beek and Robin Cohen, "Exact and Approximate Reasoning about Temporal Relations", *Computational Intelligence*, 6, 1990, pp. 132-144.

[7] P. van Beek, "Reasoning about Qualitative Temporal Information", *Artificial Intelligence*, 58, 1992, pp. 297-326.