

CONSISTENCY METHODS FOR TEMPORAL REASONING

by

Lin Xu

A THESIS

Presented to the Faculty of
The Graduate College at the University of Nebraska
In Partial Fulfilment of Requirements
For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

May, 2003

CONSISTENCY METHODS FOR TEMPORAL REASONING

Lin Xu, M.S.

University of Nebraska, 2003

Adviser: Berthe Y. Choueiry

Reasoning about time is important in real-life situations and in engineered systems. In this research, we develop new algorithms for solving the Simple Temporal Problems (STP) and the more general Temporal Constraint Satisfaction Problem (TCSP).

First, we propose a new efficient algorithm, the Δ STP-solver, for computing the minimal network of an STP. This algorithm achieves high performance by exploiting a topological property of the constraint graph (i.e., triangulation) and a semantic property of the constraints (i.e., convexity). Importantly, Δ STP-solver implicitly guarantees the decomposition of the constraint graph according to its articulation points. We show empirically that this new algorithm outperforms previously reported algorithms such as the Floyd-Warshall algorithm (F-W), Directed-Path Consistency (DPC), and Partial Path-Consistency (PPC).

Second, we report the integration of three approaches to improve the performance of the exponential-time backtrack search (BT-TCSP) for solving TCSPs. The first approach consists of using our new efficient algorithm (Δ STP) for solving the STP. The second approach exploits the topology of the temporal network. This is accomplished in three ways: finding and exploiting articulation points (AP), checking the graph for new cycles (NewCyc), and using a new heuristic for edge ordering (EdgeOrd). The third approach is a filtering algorithm Δ AC, which is used as a preprocessing step to BT-TCSP and which significantly reduces the size of the TCSP.

Our experiments on randomly generated problems demonstrate significant improvements in the number of nodes visited, constraint checks, and CPU time.

ACKNOWLEDGEMENTS

I would like to thank all those who supported me while I was working on this thesis, especially my academic adviser Dr. Choueiry whose help has been crucial for my progress. Her knowledge and personality always helped me out of the obstacles I encountered in my research.

I am honored to have Dr. Peter Revesz, Dr. Sharad Seth and Dr. Vinodchandran Variyam on my thesis committee. Their feedback has me helped refine this thesis.

I am indebted to Mark Boddy, Paul Morris, Nicola Muscettola and Ioannis Tsamardinos for sharing data and information on the STP. I am grateful to Eddie Schwalb and Rina Dechter for various pointers to TCSPs, and to Deb Derrick for editorial help on various parts of this document.

I would like also to thank everyone in our group, the Constraint Systems Laboratory, especially Daniel Buettner, Eric Moss, Amy Davis, and Hui Zou. I feel really lucky that I am in this group. Everyone is available to offer help when you have any problem, no matter when you need it, no matter how hard it is. You can always count on them. Thank you, my buddies.

At last, I would like to thank my wife. She always stands beside me, helps me, and encourages me. She is my angel and always protects me. I love you, Ping.

This work is supported by NASA-Nebraska grant, NSF CAREER Award #0133568 from the National Science Foundation, and a gift from Honeywell Laboratories.

Lincoln, NE

Lin Xu

Contents

1	Introduction	1
1.1	Qualitative temporal networks	2
1.1.1	Interval Algebra	3
1.1.2	Point Algebra	7
1.2	Quantitative temporal networks	9
1.2.1	The temporal constraint satisfaction problem (TCSP)	10
1.2.2	Solving the Simple Temporal Problem (STP)	13
1.2.3	Solving the TCSP	16
1.3	Questions addressed	19
1.4	Summary of contributions	20
1.5	Guide to thesis	22
2	Random temporal network generators	23
2.1	STP generators	24
2.1.1	GenSTP-0	24
2.1.2	GenSTP-1	24
2.1.3	More STP generators	25
2.2	TCSP generators	26
2.2.1	GenTCSP-0	26
2.2.2	GenTCSP-1	27
3	Solving STP	29
3.1	Introduction	30
3.2	Background	31
3.2.1	Main CSP properties	31
3.2.2	Properties of the STP	33
3.3	STP algorithms	35
3.3.1	F-W and DPC exploiting articulation points	35
3.3.2	PPC algorithm for STPs	36
3.3.3	Δ STP algorithm	37
3.3.4	Features of Δ STP	39
3.4	Empirical evaluations	41
3.4.1	Experiments conducted	42

3.4.2	Observations	50
3.4.3	Significance of our results	52
4	Solving TCSP	53
4.1	Background and motivation	54
4.2	Algorithms for solving the STP	57
4.2.1	Solving the STP using Directional Path Consistency (DPC)	57
4.2.2	Solving the STP using Partial Path Consistency (PPC)	58
4.2.3	Δ STP algorithm used with TCSP algorithm	60
4.3	Exploiting the topology of the constraint network	60
4.3.1	Decomposition using articulation points	61
4.3.2	New cycle check (NewCyc)	61
4.3.3	Ordering heuristic for the meta-CSP	64
4.4	The label filtering algorithm	66
4.4.1	Δ arc-consistency	69
4.4.2	Δ AC algorithm	70
4.5	Experimental results	72
4.5.1	Power of Δ AC	78
4.5.2	Solutions to the TCSP	80
4.5.3	Effects on the size of the search tree	80
4.5.4	Effects on the number of constraints checks (same as CPU time)	82
5	Conclusions and future work	94
5.1	Conclusions for the STP	94
5.2	Conclusions for the TCSP	96
5.3	Directions for future research	98
	Bibliography	100

List of Figures

1.1	<i>Simple relations in the interval algebra.</i>	4
1.2	<i>An example of interval algebra.</i>	5
1.3	<i>Addition and multiplication in the point algebra.</i>	8
1.4	<i>The relation between Interval Algebra and Point Algebra.</i>	9
1.5	<i>An example directed constraint graph for TCSP.</i>	12
1.6	<i>The addition and multiplication operations of TCSP.</i>	14
1.7	<i>The distance graph of Figure 1.5.</i>	15
1.8	<i>Floyd-Warshall's algorithm.</i>	15
1.9	<i>DPC algorithm.</i>	16
1.10	<i>The search tree for solving the meta-CSP.</i>	17
1.11	<i>The backtrack algorithm for solving TCSP.</i>	18
3.1	<i>Left: STP. Right: TCSP.</i>	33
3.2	<i>The PPC algorithm, slightly improved to consider simultaneously all three edges in a triangle.</i>	36
3.3	<i>An example of updating edges. The label of edge BC then that of AC are updated.</i>	37
3.4	<i>The temporal graph as a graph of triangles.</i>	38
3.5	<i>The ΔSTP algorithm.</i>	39
3.6	<i>Constraint Checks for ΔSTP-front, ΔSTP-back and ΔSTP-random.</i>	42
3.7	<i>Constraint Checks (top) and CPU time (bottom) for F-W, F-W+AP, PPC, and ΔSTP.</i>	46
3.8	<i>Constraint Checks (top) and CPU time (bottom) for DPC, DPC+AP, and ΔSTP.</i>	47
3.9	<i>Constraint Checks (top) and CPU time (bottom) for STP solvers, problems generated by GenSTP-2.</i>	48
3.10	<i>Constraint Checks (top) and CPU time (bottom) for STP solvers, problems generated by SPRAND.</i>	49
4.1	<i>A TCSP example (left) and formulate it as meta-CSP (right)</i>	54
4.2	<i>The search tree for the example of Figure 4.1.</i>	55
4.3	<i>Left: List of triangulated subgraphs given an ordering. Right: Inducing a subgraph from the triangulated original graph.</i>	59
4.4	<i>Simple constraint graph.</i>	62
4.5	<i>Comparison of STP checks using different the new-cycle check heuristic.</i>	62
4.6	<i>Only check the consistency of the newly formed biconnected component.</i>	63
4.7	<i>Edge ordering heuristic.</i>	65

4.8	<i>Illustrating the exploration of the edges of a graph by the edge ordering heuristic.</i>	65
4.9	<i>EdgeOrd localize backtracking.</i>	66
4.10	<i>Replacing the global constraint with a polynomial number of ternary constraints.</i>	67
4.11	<i>A consistent triangle.</i>	69
4.12	<i>First-support.</i>	70
4.13	<i>Initialize-support.</i>	71
4.14	ΔAC .	72
4.15	<i>TCSP solvers tested.</i>	72
4.16	<i>Reduction of problem size of TCSP I.</i>	78
4.17	<i>Constraint checks for solving TCSP.</i>	79
4.18	<i>CPU time for solving TCSP I.</i>	79
4.19	<i>The number of solution of TCSP.</i>	80
4.20	<i>Nodes visited by BT-TCSP.</i>	81
4.21	<i>Constraint checks and CPU time for DPC-TCSP without ΔAC (Top: Constraint Checks; Bottom: CPU time [s])</i>	83
4.22	<i>Constraint checks and CPU time for DPC-TCSP after ΔAC (Top: Constraint Checks; Bottom: CPU time [s])</i>	84
4.23	<i>Constraint checks and CPU time for PPC-TCSP using Plan A without ΔAC (Top: Constraint Checks; Bottom: CPU time [s])</i>	85
4.24	<i>Constraint checks and CPU time for PPC-TCSP using Plan A after ΔAC (Top: Constraint Checks; Bottom: CPU time [s])</i>	86
4.25	<i>Constraint checks and CPU time for PPC-TCSP using Plan B without ΔAC (Top: Constraint Checks; Bottom: CPU time [s])</i>	87
4.26	<i>Constraint checks and CPU time for PPC-TCSP using Plan B after ΔAC (Top: Constraint Checks; Bottom: CPU time [s])</i>	88
4.27	<i>Constraint checks and CPU time for ΔSTP-TCSP without ΔAC (Top: Constraint Checks; Bottom: CPU time [s])</i>	89
4.28	<i>Constraint checks and CPU time for ΔSTP-TCSP after ΔAC (Top: Constraint Checks; Bottom: CPU time [s])</i>	90

List of Tables

3.1	<i>Parameters of generated problems.</i>	41
3.2	<i>Experimental results for STP solvers on random STP generated by GenSTP-1.</i>	44
3.3	<i>Experimental results for STP solvers on random STP generated by SPRAND.</i>	45
4.1	<i>Performance of ΔAC</i>	74
4.2	<i>The number of constraint checks for different TCSP solvers.</i>	75
4.3	<i>CPU time [s] for different TCSP solvers.</i>	76
4.4	<i>The number of nodes visited for different TCSP solvers.</i>	77

Chapter 1

Introduction

Space and time have always been important subjects in Science. Thousands years ago, people estimated the time to arrange their schedule by checking the length of the shadow of a pole. With the advent of more modern technologies, people care even more about time. Almost everyone has a watch, so he/she can easily check the time. While people are increasingly aware of time, time limit remains one of the biggest problems. We have exactly 24 hours per day and constantly worry about using these 24 hours more efficiently. For example: Tom wants to serve some tea to his friends. There are a few things he needs to do: clean the pot (5 minutes), clean the tea cups (10 minutes) and boil water (15 minutes). If Tom cleans the pot and tea cups first then boils the water, then he needs 30 minutes to get the tea ready. It is actually easy to find a better schedule. Tom can wash the pot first, and then start to boil water since the pot is clean. While boiling water (15 minutes), Tom can clear the tea cups (10 minutes). After the water is ready, Tom can serve the tea. This takes only 20 minutes. A little reasoning about time can save a lot of time.

The study of time is addressed in almost every area of Science and also in Artificial Intelligence (AI). Reasoning about time is used in almost every area of AI: planning, scheduling, natural language understanding [2], and common-sense reasoning [30]. A typical

temporal reasoning system may include some components, such as a temporal knowledge base, an algorithm to check its consistency, a query-answering mechanism, and an inference mechanism, which yields new information from the one provided [13]. The goal of a temporal reasoning system is that based on the temporal information given in the temporal knowledge base, the system can answer user's questions. The information stored in the temporal database is in the form of propositions, such as "I am watching TV," "Jim has dinner with Jack," with some information corresponding with the temporal intervals representing the duration of these events. The information could be relative (I was watching TV after dinner) or metric (Tom arrived home at least 1 hour earlier than his wife). The information also can be disjunctive, such as "I go to school by bus (30 -45 minutes) or by car (10 -15 minutes)." Using the information stored in the database, the temporal reasoning system may have to answer questions such as "Is it possible that I take the bus to school?" or "If I do not want to be late, When should I get up?".

In this chapter, we will introduce two different types of temporal networks: qualitative temporal networks and quantitative temporal networks. This thesis focuses on solving the quantitative temporal networks.

1.1 Qualitative temporal networks

There are two types of temporal algebra for qualitative temporal networks: Point Algebra (PA) [34] and Interval Algebra (IA) [1]. In Point Algebra, the primary objects are time points, indicating when events occurred or ended. In this algebra, temporal information is expressed by a relation between two time points. For example: *I get up earlier than Jack*. Suppose T_0 is the time point I get up and T_1 is the time point Jack gets up, then this information is expressed as $T_0 < T_1$ in the time point algebra. In Interval Algebra, the primary objects are time periods, during which events occur or propositions hold. For

example: *I meet Rob during lunch (lunch time is 15 min)*, which means my lunch time (time interval I) overlaps with Rob's lunch time (time interval R). This information can also be represented in Point Algebra but it looks more complex: Suppose I_0 and I_1 are the beginning and ending points of I ; R_0 and R_1 are the beginning and ending points of R . Interval I and interval R overlap is expressed as $[(R_0 < I_1) \wedge (R_0 > I_0)] \vee [(I_0 < R_1) \wedge (I_0 > R_0)]$ in the time point algebra.

1.1.1 Interval Algebra

Interval Algebra was first introduced by Allen. It expresses temporal knowledge as qualitative statements relating the positions of two intervals. If we have two intervals A and B , the possible relationships between A and B are: before, meets, overlaps, starts, during, finishes, equal and their inverse ($\{b, m, o, s, d, f, bi, mi, oi, si, di, fi, =\}$). There are 13 simple relations between two intervals.

The simple relations are formally defined in Figure 1.1. With Allen's 13 relations, we can express almost every qualitative temporal network. The relation between two intervals can be expressed by one or more (the disjunction of) simple relations. There are 2^{13} possible relations between two intervals.

For example: *I watch TV AFTER my dinner* and *I finish my dinner BEFORE watching TV* have the same meaning. It means the ending point of my dinner is strictly before the starting point of watching TV. *Tom comes to my house when I am watching TV* means the relation between the interval B (I am watching TV) and interval C (Tom is at my home) is the vector (o, fi, di) . Note that the only information we have is the starting time point of interval C is during the time interval B , we do not know any information about when Tom leaves my house, Figure 1.2.

All relations in interval algebra are defined by vectors. In order to propagate knowledge and constraints, two mathematical operations, addition and multiplication, are defined over


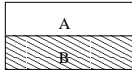

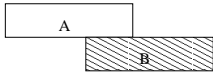
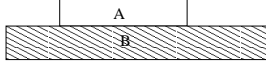
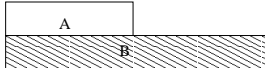
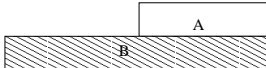
Relation	Symbol	Inverse	Example
A before B	b	bi	
A equal B	=	=	
A meets B	m	mi	
A overlaps B	o	oi	
A during B	d	di	
A starts B	s	si	
A finishes B	f	fi	

Figure 1.1: *Simple relations in the interval algebra.*

vectors of relations.

The operation addition is used in two different vectors describing the same relation of two intervals. The addition intersects these two vectors to provide the relation, which two vectors all allows. For example: *Prof. A tells me I need to read a given paper (b, m, o) my lunch time, Prof. B tells me I need to read the same paper (o, s, d) my lunch time.* If I want to follow their recommendations, I need to read the paper (o) my lunch time. Algorithmically, the sum of two vectors is computed by finding their common constituent simple relations.

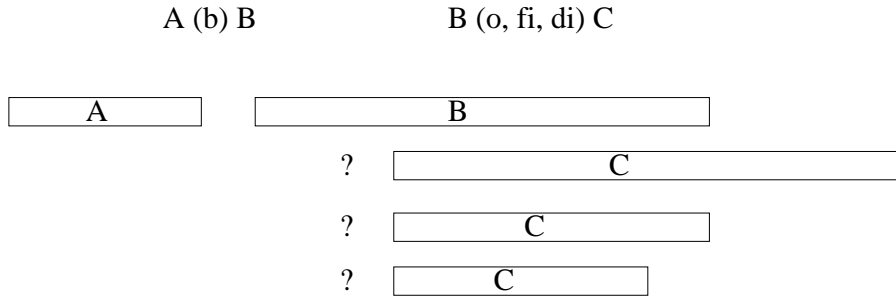
$$V_1 = \{a_1, a_2, \dots, a_n\} \quad (1.1)$$

$$V_2 = \{b_1, b_2, \dots, b_n\} \quad (1.2)$$

$$V_1 \oplus V_2 = \{a_1, a_2, \dots, a_n\} \cap \{b_1, b_2, \dots, b_n\} \quad (1.3)$$

For example:

$$V_1 \oplus V_2 = (b, m, o) \cap (o, s, d) = (o) \quad (1.4)$$



A: I am eating my dinner

B: I am watching TV

C: Tom is visiting me

Figure 1.2: An example of interval algebra.

The operation multiplication is defined when we know the vectors of relations between intervals A and B and between intervals B and C , and we want know the vector of relations of between the intervals A and C . For example: we have vector of relations V_1 between intervals A and B , vector of relations V_2 between intervals B and C . $V_1 \otimes V_2$ gives the vector of relations between intervals A and C by V_1 and V_2 .

$$V_1 = \{a_1, a_2, \dots, a_n\} \quad (1.5)$$

$$V_2 = \{b_1, b_2, \dots, b_n\} \quad (1.6)$$

$$V_1 \otimes V_2 = \{a_1, a_2, \dots, a_n\} \otimes \{b_1, b_2, \dots, b_n\} \quad (1.7)$$

$$= (a_1 \otimes b_1) \vee (a_1 \otimes b_2) \dots \vee (a_2 \otimes b_1) \dots \vee (a_n \otimes b_m) \quad (1.8)$$

For example:

$$V_1 \otimes V_2 = (b, m, o) \otimes (b, m) \quad (1.9)$$

$$= (b \otimes b) \vee (m \otimes b) \vee (o \otimes b) \vee (b \otimes m) \vee (m \otimes m) \vee (o \otimes m) \quad (1.10)$$

$$= (b) \vee (b) \vee (b) \vee (b) \vee (b) \vee (b) = (b) \quad (1.11)$$

The multiplication operation for vectors is the union of multiplication on simple relations. For the example above, the result of multiplication is: $V_1 \otimes V_2 = (b, m, o) \otimes (b, m) = (b)$.

In some cases, computing multiplication can be really complex. For example: *I begin my dinner before I turn on the TV, and Tom comes to my house when I am watching TV. We want know the relation that holds between my having dinner and Tom being in my home.* Suppose, V_1 is the vector of relations between my dinner and TV time, and V_2 is the vector of relations between TV time and Tom is staying in my house. The vector of relations between my dinner and Tom is staying in my house can be obtained as follows:

$$V_1 = \{b, m, o, fi, di\} \quad (1.12)$$

$$V_2 = \{o, fi, di\} \quad (1.13)$$

$$\begin{aligned} V_1 \otimes V_2 &= (b \otimes o) \vee (m \otimes o) \vee (o \otimes o) \vee (fi \otimes o) \vee (di \otimes o) \vee (b \otimes fi) \\ &\quad \vee (m \otimes fi) \vee (o \otimes fi) \vee (fi \otimes fi) \vee (di \otimes fi) \vee (b \otimes di) \\ &\quad \vee (m \otimes di) \vee (o \otimes di) \vee (fi \otimes di) \vee (di \otimes di) \end{aligned} \quad (1.14)$$

$$= (b) \vee (b) \vee (b, o, m) \vee (o) \vee (o, di, fi) \vee (b) \vee (b) \vee (b, o, m) \quad (1.15)$$

$$\vee (fi) \vee (di) \vee (b) \vee (b) \vee (b, fi, di, o, m) \vee (di) \quad (1.16)$$

$$= (b, o, m, di, fi) \quad (1.17)$$

With these two operations, we can model a temporal problem with Interval Algebra, and assert the temporal information in the database of the temporal reasoning system. The system will compute those temporal relations that follow from the user's assertions. This task is executed by a polynomial-time algorithm proposed by Allen: constraint propagation algorithm. This algorithm is sound, in the sense that it never infers an invalid assertion. However, Allen also demonstrates that the algorithm is incomplete. Vilain and Kautz [34]

show that determining the satisfiability of a set of assertions in the interval algebra is an **NP**-hard problem.

Even though Allen's interval algebra is intractable, it still has some usages. First, it can be used to solve some small problems. We can limit the size of our temporal database. Since the size of problem is small, even exponential-time is still acceptable. Unfortunately, this condition does not hold for most real-world problems. Second, we can accept the constraint propagation algorithm's incompleteness. Since the constraint propagation algorithm is polynomial-time and sound, if the user's question only needs very few inferences, then the algorithm may be able to satisfy the user. Of course, for applications that need more temporal reasoning, this option may not find a solution.

1.1.2 Point Algebra

Because no sound and complete polynomial-time algorithm exists for the interval algebra, an alternative approach is to choose a temporal representation other than the full interval algebra. This new representation can be either a fragment of the Allen's interval algebra or a new, less expensive algebra: the point algebra.

In Point Algebra, temporal information is expressed by means of constraints on time points. Obviously, there are only three basic relations between two time points P and Q : before ($P < Q$), after ($P > Q$) and equal ($P = Q$). Hence the number of possible relations between two points is $2^3 = 8$, which is less than that for interval algebra (2^{13}). Reasoning in Point Algebra is a polynomial-time process. In this algebra, temporal networks are represented as variables $\{X_1, X_2, \dots, X_n\}$, where each variable is a time point. The domain of each variable is a set of real numbers, which are the time points the variable may assume. The constraints are one or more relations in $\{before, after, equal\}$. For example, *I began my dinner before I turned on the TV* can be expressed in Point Algebra as $T(dinner) < T(TV)$.

As we discussed for the case of Interval Algebra, we can define the addition operation and multiplication operation for Point Algebra. The definitions of these two operations are the same as the definitions for Interval Algebra. Since there are very limited possible relations between two time points, we can easily build the table for all possible additions and multiplications, Figure 1.3:

Addition:									Multiplication:							
	<	<=	>	>=	=	<>	?			<	<=	>	>=	=	<>	?
<	<	<	F	F	F	<	<	<	<	<	?	?	<	?	?	
<=	<	<=	F	=	=	<	<=	<=	<	<=	?	?	<=	?	?	
>	F	F	>	>	F	>	>	>	?	?	>	>	>	?	?	
>=	F	=	>	>=	=	>	>=	>=	?	?	>	>=	>=	?	?	
=	F	=	F	=	=	F	=	=	<	<=	>	>=	=	<>	?	
<>	<	<	>	>	F	<>	<>	<>	?	?	?	?	?	<>	?	
?	<	<=	>	>=	=	<>	?	?	?	?	?	?	?	?	?	

? is (<, >, =)
 F is (), the null vector

Figure 1.3: Addition and multiplication in the point algebra.

Any temporal network in Point Algebra also can be expressed in Interval Algebra. However, problems expressed by Interval Algebra may not be expressible by Point Algebra. For example, suppose we are given the interval algebra vectors between A B : A (s , d) B which means time interval A starts or is during B . Let x and y are the starting and ending points of A , m and n are the starting and ending points of B . We can express the same information as $x < y$, $m < n$, $m (<, =) x$, $y < n$ in Point Algebra. However, A (b , bi) B in Interval Algebra cannot be expressed in Point Algebra, Figure 1.4.

The constraint propagation algorithm is complete for Point Algebra. The algorithm runs to completion in $O(n^3)$ time. There even exists an algorithm with time complexity $O(n^2)$ for deciding the consistency and for finding a consistent scenario [33]. The minimal

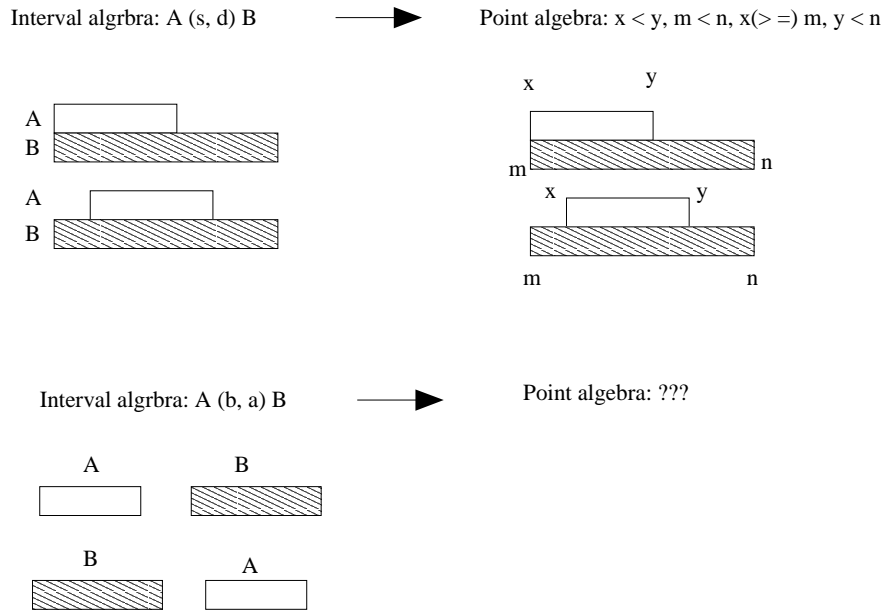


Figure 1.4: *The relation between Interval Algebra and Point Algebra.*

network of a problem in Point Algebra can be obtained using 4-consistency in $O(n^4)$ steps. If we exclude the $(<>)$, the constraints are taken from $\{<, >, =, (>=), (<=)\}$, then this subset of Point Algebra is called Convex Point Algebra. The minimal network of a network modeled in Convex Point Algebra can be obtained by path-consistency algorithm in $O(n^3)$ [13].

1.2 Quantitative temporal networks

In qualitative temporal network, all temporal relations are represented as ‘before’ or ‘after’ relations. In the real-world, this kind of information may not be precise enough. For example, *Tom needs to go to school to attend the first class at 8:00 a.m. He needs to have some breakfast first. Then he either takes a bus or drives himself to school.* Only mention that the breakfast is after he gets up, and driving or taking a bus is after the breakfast obviously are not enough. We need the information such as “when does he get up?”, “How long does it take him to get from home to school by bus?”. To model this information, we

need to enhance the representation with metric information, and model the problem with quantitative temporal networks. Consider the following example:

Tom has class at 8:00 a.m. He can either make breakfast for himself (10–15 minutes), or get something in a local store (less than 5 minutes). After breakfast (5–10 minutes), he goes to school either by car (20–30 minutes) or by bus (at least 45 minutes). Today, Tom gets up between 7:30 and 7:40.

In a quantitative temporal network, the variables are states, which represent snapshots of events. In the example above, let variable P_1 represents the snapshot of Tom getting up, let variable P_2 represents the snapshot of Tom getting his breakfast. The constraints are disjunction of quantitative intervals, such as $C(P_1, P_2)$ is $\{[0, 5], [10, 15]\}$. Based on this temporal information database, we can expect the temporal reasoning system to answer questions such as: “Is it possible that Tom is not late for school?”, “Is it possible for Tom to take the bus?”, “If Tom wants to save money by making breakfast for himself and taking the bus, when should he get up?”, and so on.

There are two kinds of quantitative temporal problems. The first one is temporal constraint satisfaction problem (TCSP), which is general temporal problem and is **NP**-hard. The second one is a restricted, simpler version of TCSP (simple temporal problem STP), which we can solve in polynomial time.

1.2.1 The temporal constraint satisfaction problem (TCSP)

The TCSP problem can be described similarly to the general CSP. A temporal constraint satisfaction problem (TCSP) is composed by a set of variables, each variable has a continuous domain and binary constraints describe the relation between two variables or unary constraints for one variable.

In TCSP model, variables represent time points (snapshot, state). The domain of the

variable is any real number (continuous domain). Each constraint is represented by a set of intervals:

$$\{I_1, I_2, \dots, I_n\} = \{[a_1, b_1], [a_2, b_2], \dots, [a_n, b_n]\} \quad (1.18)$$

There are unary constraints, T_i . They restrict the domain of variable X_i to a disjunction of intervals:

$$(a_1 \leq X_i \leq b_1) \vee \dots \vee (a_n \leq X_i \leq b_n) \quad (1.19)$$

The most popular constraints are binary constraints, $T_{i,j}$, define the allowed values for the distance ($X_j - X_i$). It also allows the disjunction of intervals:

$$(a_1 \leq X_j - X_i \leq b_1) \vee \dots \vee (a_n \leq X_j - X_i \leq b_n) \quad (1.20)$$

In fact, we can easily transfer the unary constraints to binary constraints by adding a reference time point X_0 . Every unary constraint T_i can be transferred to a binary constraint $X_{i,0}$. For example: *Tom gets up between 7:30 a.m and 7:40 a.m* can be expressed as unary constraint $7:30 \leq X_i \leq 7:40$. If we add the initial time point $X_0 = 6:00$, then this information also can be expressed as binary constraint $90 \leq X_i - X_0 \leq 100$.

A binary TCSP is a temporal network that consists of a set of variables, X_1, X_2, \dots, X_n and a set of binary and unary constraints. This network also can be represented as a directed constraint graph, nodes represent variables and the edge $i \rightarrow j$ indicates a constraint between variable i and variable j . The label of edge $i \rightarrow j$ shows the intervals set of the constraint. Figure 1.5 is an example of a directed constraint graph.

Similarly to general CSP, a solution of a TCSP is a set of real values $\{x_1, x_2, \dots, x_n\}$, $\{X_1 = x_1, X_2 = x_2, \dots, X_n = x_n\}$ that satisfies all the temporal constraints. We need

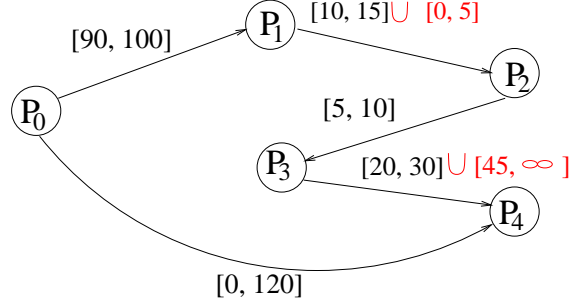


Figure 1.5: An example directed constraint graph for TCSP.

to mention that since the TCSP has continuous domain, the number of solutions usually is infinity. If value v in the domain of X_i appears in a solution of TCSP then this value is a feasible value for variable X_i . The set of all feasible values of variable is called the minimal domain. Obviously, if any minimal domain is empty, then there is no solution for the TCSP and we call this TCSP is inconsistent.

As we defined addition and multiplication for the interval algebra and the point algebra, we can similarly define the addition operation (interval intersection) and multiplication operation (interval composition) for TCSP. Suppose we have constraints $A = \{I_1, I_2, \dots, I_m\}$, $B = \{J_1, J_2, \dots, J_n\}$.

The addition (intersection) of two temporal constraints A and B is the values that are allowed by both A and B .

$$A \oplus B = \{I_1, I_2, \dots, I_m\} \oplus \{J_1, J_2, \dots, J_n\} \quad (1.21)$$

$$= \{I_1 \cap J_1, I_1 \cap J_2, \dots, I_2 \cap J_1, \dots, I_m \cap J_n\} \quad (1.22)$$

$$= \{K_1, K_2, \dots, K_t\} \quad (t < m + n) \quad (1.23)$$

Where $K_s = I_x \cap J_y$ for some x and y . A and B are different sets of intervals for the same constraint and K is the new constraint proposed by A and B . For example, the constraint between X and Y is $A = \{[1, 4], [6, 8]\}$. For some reason we need add a constraint between

X and Y : $B = \{[0, 1], [3, 7]\}$. Since these two constraints apply to the same pair of variables, the allowed values for the new constraint should satisfy both constraints A and B . Hence, the new constraint K is $A \oplus B = \{[1, 1], [3, 4], [6, 7]\}$, see Figure 1.6. The number of intervals in K is up to the sum of the number of intervals in A and B .

The multiplication (composition) of two temporal constraints A and B defines a new constraint K . If constraint K allows value r , there must exist a value $u \in A$ and a value $v \in B$, such that $u + v = r$.

$$A \otimes B = \{I_1, I_2, \dots, I_m\} \otimes \{J_1, J_2, \dots, J_n\} \quad (1.24)$$

$$= \{I_1 \otimes J_1, I_1 \otimes J_2, \dots, I_2 \otimes J_1, \dots, I_m \otimes J_n\} \quad (1.25)$$

$$= \{K_1, K_2, \dots, K_t\} \quad (t < m \times n) \quad (1.26)$$

Where $K_s = [a + c, b + d]$ for some $I_x = [a, b]$ and $J_y = [c, d]$. If A is a constraint between variables X and Y , B is a constraint between variables Y and Z , then the multiplication will propose a new constraint between X and Z . For example: $A = \{[1, 2], [4, 6]\}$, $B = \{[2, 3], [6, 7]\}$, the new constraint $K = \{[3, 5], [6, 9], [10, 13]\}$, see Figure 1.6. The number of intervals in K is up to the product of the number of intervals in A and B .

1.2.2 Solving the Simple Temporal Problem (STP)

The simple temporal problem (STP) is a simple version of TCSP. We notice that the two operations, addition and multiplication, can make the number of intervals per constraint exponentially large. But if there is only one interval per constraint, then the addition and multiplication always output a new constraint with only one interval, which make the STP easy to solve. In an STP, there is only one interval per constraint:

$$a_{i,j} \leq X_j - X_i \leq b_{i,j} \quad (1.27)$$

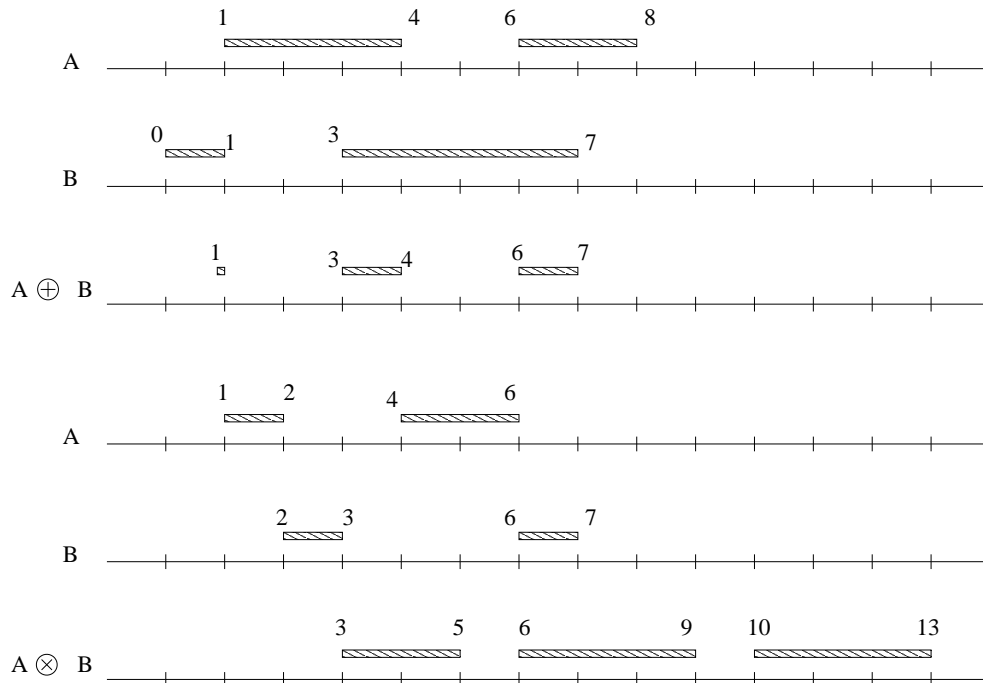


Figure 1.6: *The addition and multiplication operations of TCSP.*

We also can express this inequality as two inequalities:

$$X_j - X_i \leq b_{i,j} \quad (1.28)$$

$$X_i - X_j \leq -a_{i,j} \quad (1.29)$$

Solving an STP amounts then to finding the set of solutions of a system of linear inequalities. We can solve STP by using an all pairs shortest paths algorithm such as Floyd-Warshall algorithm. We reformulate the STP as a distance graph¹, a directed edge-weighted graph $G_d = (V, E_d)$. The difference between a constraint graph and a distance graph is that instead of having a directed edge $i \rightarrow j$ with label of interval $[a_{i,j}, b_{i,j}]$ in a constraint graph, a distance graph labels the edge $i \rightarrow j$ with $b_{i,j}$ and edge $j \rightarrow i$ with $-a_{i,j}$. Figure 1.7 gives the distance graph of the example described in Figure 1.5 (we choose only one interval per edge). Since the multiplication works on two single intervals for STP,

¹Also called a gap-graph [27].

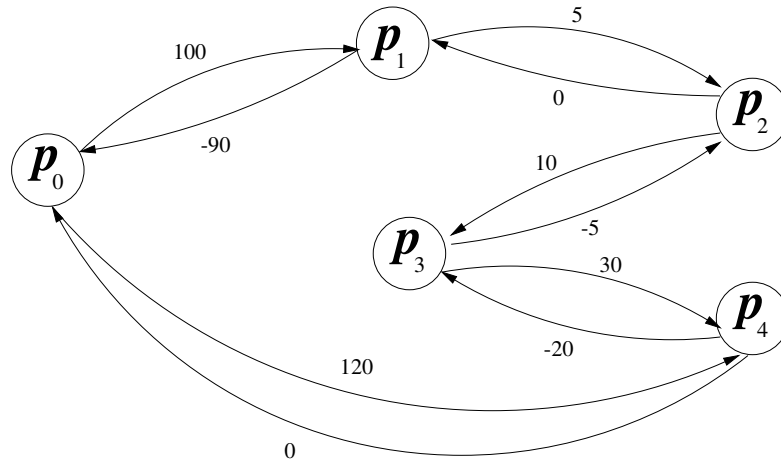


Figure 1.7: The distance graph of Figure 1.5.

we can easily get the constraint between i and j with path $i_0=i, i_1, \dots, i_k=j$ in G_d is $[(a_{0,1} + a_{1,2} + \dots + a_{(k-1),k}), (b_{0,1} + b_{1,2} + \dots + b_{(k-1),k})]$. If there are more than one path from i to j then $x_j - x_i \leq d_{i,j}$, $d_{i,j}$ is the length of the shortest path from i to j .

An STP is consistent if and only if its distance graph has no negative cycles [31]. Any consistent STP is decomposable relative to the constraints in its distance graph.

The complete minimal distance graph can be obtained by using Floyd-Warshall's all pairs shortest paths algorithm. The algorithm finds the minimal network with $\Theta(n^3)$ time. Since this minimal network is decomposable, finding a solution only needs $O(n^2)$ time because decomposability promises backtrack-free.

```

All -pairs-shortest-paths algorithm (STP):
for i = 1 to n do  $d_{i,i} \leftarrow 0$ ;
for i, j = 1 to n do  $d_{i,j} \leftarrow a_{i,j}$ ;
for k = 1 to n do
  for i, j = 1 to n do
     $d_{i,j} \leftarrow \min\{d_{i,j}, d_{i,k} + d_{k,j}\}$ ;
  
```

Figure 1.8: Floyd-Warshall's algorithm.

If we are only interested in the consistency of an STP, we can use directional path consistency algorithm (DPC) [15] instead of Floyd-Warshall's algorithm. DPC is a simple

version of PC-1, the difference between DPC and PC-1 is that DPC is a single pass algorithm. From Figure 1.9, we can find that the set of edges is increasing while the algorithm is executing. At the end of execution of the DPC algorithm, an induced graph is obtained. The number of edges in the induced graph is larger than in the original constraint graph but less than in the complete graph.

```

DPC (STP) :
  for k = n down to 1 by -1 do
    for all i, j | k such that (i, k), (j, k) ∈ E do
       $T_{i,j} \leftarrow T_{i,j} \oplus T_{i,k} \otimes T_{k,j}$ , and
       $E \leftarrow E \cup (i, j)$ , and
      if  $T_{i,j} = nil$  then exit (the network is inconsistent);
```

Figure 1.9: DPC algorithm.

Dechter shows that, given an STP, the algorithm DPC terminates at the final step in Figure 1.9 if and only if the STP network is inconsistent. DPC can be achieved more efficiently than full path consistency. Unlike F-W, which needs $O(n^3)$, DPC can be finished in $O(nW^*(d)^2)$ time for telling the consistency of an STP, where $W^*(d)$ is the maximum number of parents that a node possesses in the induced graph. Notice that $W^*(d)$ is always less than n , if we only want to know the consistency of an STP, we should use DPC instead of F-W due to its lower cost.

There are several methods to solve STPs, we introduce two that guarantee the minimal network with lower cost than F-W, as we discuss in detail in Chapter 3.

1.2.3 Solving the TCSP

The general TCSP has more than one interval per constraint. With the addition and multiplication operation, the number of intervals per constraint can become exponentially large. Davis showed that determining consistency of a general TCSP is **NP**-hard [13].

The general TCSP is modeled as a meta-CSP. The variables are the edges and the domain of variable is the possible intervals. Hence we map a TCSP into a discrete CSP with variables X_1, X_2, \dots, X_n , where $n = |E|$, and X_i with domain of intervals I_1, I_2, \dots, I_m correspond to the label of e_i in constraint graph. Not like other CSPs, the consistency for an assignment, $\{X_1 = I_{1a}, X_2 = I_{2b}, \dots, X_n = I_{nm}\}$ is decided by the consistency of corresponding STP (Figure 1.10).

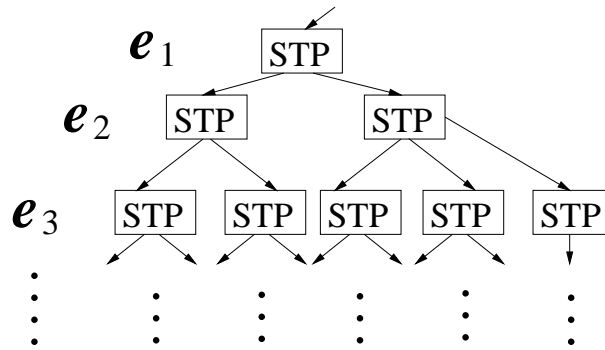


Figure 1.10: *The search tree for solving the meta-CSP.*

A general TCSP problem can be decomposed into $|I_1| \times |I_2| \times \dots \times |I_n|$ STPs. By solving all STP and combining the solutions together, we can get the minimal network for TCSP. The complexity of solving TCSP is the total cost of solving $|I_1| \times |I_2| \times \dots \times |I_n|$ STPs ($O(n^3 k^{|E|})$ [15], k is the maximum number of intervals per edge and $|E|$ is the number of edges). We also can use backtracking search to find all the consistent STPs. With this approach, we can use all techniques to improve the performance of search and find all the solutions.

Figure 1.11 is the backtrack algorithm (BT-TCSP) for solving TCSP [15]. The backtrack algorithm expands a meta-CSP one edge at a time. It has two procedures: **Forward** and **Go-back**. At each step, **Forward** expands one more edge. It assigns a new interval (value) for this edge (variable), extending the current STP. If we can find an interval from the label of the new edge can make the resulting STP consistent, then call **Forward** again. If there is no interval that can make the corresponding STP consistent, then we need to

```

BT (TCSP) :
For ward( $I_1, I_2, \dots, I_i$ )
  if  $i = m$  then
     $M \leftarrow M \cup \text{Solve} - \text{STP}(I_1, I_2, \dots, I_m)$ , and
    Go-back( $I_1, I_2, \dots, I_m$ );
   $C_{i+1} \leftarrow \text{nil}$ ;
  for every  $I_j$  in  $D_{i+1}$  do
    if Consistent-STP ( $I_1, I_2, \dots, I_i, I_j$ ) then
       $C_{i+1} \leftarrow C_{i+1} \cup \{I_j\}$ ;
    if  $C_{i+1} \neq \text{nil}$  then
       $I_{i+1} \leftarrow$  first element in  $C_{i+1}$ , and
      remove  $I_{i+1}$  from  $C_{i+1}$ , and
      Forward ( $I_1, I_2, \dots, I_i, I_j$ )
    else
      Go-back( $I_1, I_2, \dots, I_i$ );
  Go-back( $I_1, I_2, \dots, I_i$ )
  if  $i=0$  then exit
  if  $C_i \neq \text{nil}$  then
     $I_i \leftarrow$  first element in  $C_i$ , and
    remove  $I_i$  from  $C_i$ , and
    Forward ( $I_1, I_2, \dots, I_i$ )
  else
    Go-back( $I_1, I_2, \dots, I_{i-1}$ );

```

Figure 1.11: The backtrack algorithm for solving TCSP.

Go-back. A consistency-check procedure for the STP could be the all-pairs-Shortest-Paths algorithm. The only requirement is that the algorithm be able to determine the consistency of the STP, even if it cannot output the minimal network. **Go-back** goes back to the previous assignment, changes the value of the previous variable, and then calls **Forward**. Although the worst-case complexity of using search is also $O(n^3 k^{|E|})$, it allows us to apply some new techniques into search such as back-jumping, variable ordering, value ordering. They can reduce the complexity much lower than the worst-case complexity.

We already know that the general TCSP is **NP**-hard. There is no way to find a polynomial time algorithm to solve general TCSP. But people may ask questions such as “Is

it possible that there are some TCSPs that are easy to solve?”, “What do those TCSPs look like?”. The answer is positive; some TCSPs are not so hard. If the TCSP is sparse, then there is a good chance we can find articulation points. With articulation points, we can decompose the constraint graph into some non-separable components. Each of these components can be solved independently. For other TCSPs, we may find out most of the intervals in the domain of edges never become a part of a solution. Hence, we can eliminate them first before starting search, which reduces the size of meta-CSP.

1.3 Questions addressed

In this thesis, we address the following questions:

1. Is there a better algorithm than F-W to solve STP?

Answer: We propose two other algorithms to solve STP: PPC and Δ STP. They always perform better than F-W.

2. Can we use the topology of the network to improve the performance of solving STP?

Answer: We show that the exploitation of articulation points helps reducing the number of constraint checks when density is low.

3. Is there an arc-consistency-like algorithm to reduce the size of meta-CSP (TCSP)?

Answer: We propose the Δ AC algorithm for this purpose. We show that dramatically reduces the size of meta-CSP especially when density is high.

4. Can we improve the performance of search by combining better STP solver with BT-TCSP?

Answer: Yes! We provide experimental results, which show significant improvements by applying those better STP solvers.

5. Do we need to check consistency of STP at every node of search tree?

Answer: No. We only need to check the consistency of STP when there exists a new cycle, otherwise a consistent STP necessarily yields another consistent STP at the following level. Further, only the consistency of the newly formed by biconnected component needs to be checked. We use a new cycle check mechanism to detect the existence of new cycle.

6. Does the variable ordering help to improve the performance of search?

Answer: Yes, we propose a new variable ordering heuristic (edge in the TCSP) that reduces the number of nodes visited and the number of constraint checks when searching the meta-CSP.

7. Can we observe the phase transition in solving TCSP?

Answer: The existence of phase transition is uncovered when we apply Δ AC as a preprocessing step for solving TCSP.

1.4 Summary of contributions

Our contributions can be organized into the following categories:

1. *Solving STP:*

- We apply PPC to solve the STP, and develop a new STP solver, Δ STP, which is always better than PPC.
- We show that the articulation points help to improve the performance of solving STP by decomposing the temporal graph.
- The experimental results shows that although DPC may need fewer number of constraint checks than Δ STP when density is high, the performance of the

Δ STP remains superior because it outputs the minimal network, which cannot be obtained by using DPC.

2. *Solving TCSP:*

- We combine different STP solvers with BT-TCSP. The results show that using PPC as STP consistency check algorithm always get better performance than using DPC as STP consistency check algorithm.
- We use articulation points to improve the performance of search.
- A new cycle check mechanism is used to avoid unnecessary consistency check. It reduces the number of constraint checks without change the number of backtracks.
- We find a good variable ordering, which reduces the number of constraint checks and the number of nodes visited.
- A sound and efficient filtering algorithm, Δ AC, is developed to help to solve TCSP by reduce the size of meta-CSP.

3. *Evaluation conditions:*

- We design and implement a few random STP and TCSP generators. Some of them are used to generate random temporal problems for testing our temporal problem solvers.
- A few generators developed by other researchers are also used to testify our STP solvers.

4. Finally, we identify new directions for future research.

1.5 Guide to thesis

This document is structured as follows. Chapter 2 introduces the random temporal problem generators that we designed and used to generate random problems for testing our algorithms. Chapter 3 introduces a number of STP solvers. We compare their performance in terms of the number of constraint checks and CPU time. In Chapter 4, those STP solvers are used as STP consistency check algorithm in BT-TCSP. Some other heuristics are used to improve the performance of search. A special filtering algorithm also introduced in this chapter to reduce the size of meta-CSP. Chapter 5 gives a conclusion reviewing our contributions and stating our directions for future research.

Chapter 2

Random temporal network generators

A temporal constraint network is a constraint graph $G = (V, E)$, where nodes are variables and represent time points, edges are constraints of bounded difference labeled by a disjunction of continuous intervals and represent temporal information. To test our algorithms and heuristics for solving STP and TCSP, some random STP and TCSP generators are developed.

All generators take as input n ($=|V|$), the number of nodes in the temporal constraint graph (TCG) and d , the constraint density. The number of edges in TCG is thus:

$$e = |E| = \frac{(n-2)(n-1)d}{2} + (n-1). \quad (2.1)$$

Another input parameter is an interval of integer values $R = [1, r]$, which is used for value selections as discussed below.

This chapter is organized as follow. In Section 2.1, we state two random STP generators (GenSTP-0, GenSTP-1) developed by us, and two random STP generators (GenSTP-2, SPRAND) from Ioannis Tsamardinos [32] and SPLIB [9]. Based on our STP generators, we also introduce two random TCSP generators (GenTCSP-0, GenTCSP-1) in Section 2.2. A brief summary is given at the end of this chapter.

2.1 STP generators

STP has only one interval per constraint. To generate STP problems, we develop two generators. GenSTP-0 does not guarantee that the generated networks are consistent, and GenSTP-1 guarantees that a user specified percentage (p_c) of the generated problems are consistent (the consistency of the remaining ones is not guaranteed).

2.1.1 GenSTP-0

GenSTP-0 randomly selects $|E|$ edges from all possible edges list, and labels them by one randomly selected intervals within R , given as input. The resulting network is tested for connectivity (an $O(|V| + |E|)$ process) and only connected networks are kept. Since the interval labels of edges are randomly selected, most of generated problems have negative cycles and are thus inconsistent. More formally:

1. Start with a list L of all possible edges (there are $\frac{n(n-1)}{2}$ edges in L).
2. Select a random edge $i \in L$ and remove it from L .
3. Label i with $[l_i, u_i]$, where l_i and u_i are randomly selected in R (with $l_i \leq u_i$).
4. Repeat (2)–(3) until $|E|$ edges have been selected.
5. If the graph is not connected, go to step (1), otherwise save the problem to a file.

2.1.2 GenSTP-1

GenSTP-1 randomly selects $(n - 2)$ distinct points within $R = [1, r]$. These nodes along with the endpoints of R (i.e., 1 and r) constitute the n nodes of the temporal constraint graph (TCG). Edges for TCG are chosen randomly from all possible edges list as in GnSTP-0. However, the lower l_i and upper u_i values of the label of a given edge i between two nodes

a and b (with $b > a$) are chosen such that $l_i \leq (b - a)$ and $u_i \geq (b - a)$. Thus, this process is guaranteed to generate a consistent network.

In order to include some inconsistent instances in a given pool of problems generated using this process, we swap, in a given instance, the labels of two random edges with a probability of $(1 - p_c)$. More formally:

1. Randomly select $(n - 2)$ distinct points within a given interval $R = [1, r]$. Each point corresponds to the ‘position’ of a node in the graph. The first node of the graph has position 1, and the last node in the graph has position r .
2. Start with a list L of all possible edges (there are $\frac{n(n-1)}{2}$ edges in L).
3. Select a random edge $i = (a, b)$ from L and remove it from L .
4. Let δ be the distance between the two points a and b of edge i . Choose two values $0 < \alpha, \beta \leq \delta$, and set the label of the edge i to be $[\delta - \alpha, \delta + \beta]$.
5. Repeat (3)–(4) until $|E|$ edges have been processed.
6. With probability $(1 - p_c)$, swap the labels of two random edges in the graph.
7. If the graph is not connected, go to step (1), otherwise save the problem to a file.

2.1.3 More STP generators

In order to test our algorithms with different types of temporal problems, we introduce two other random STP generators.

The first one (GenSTP-2) is developed by Ioannis Tsamardinos. Similar to GenSTP-1, the randomly generated STPs by GenSTP-2 have no structure constraints. The bound of each edge was chosen randomly from a given interval.

The second random STP generator is `SPLIB`. `SPLIB` is developed by Cherkassky et al. to evaluate shortest path algorithms. It includes several different families of randomly problem generators. We use one of them: `SPRAND`. All random STPs generated by `SPRAND` have a big cycle, which connects all the nodes. It is impossible to find an articulation point in the TCG generated by `SPRAND`, which may makes the STPs harder to solve by some STP solvers.

2.2 TCSP generators

In a TCSP, an edge (representing a constraint) is labeled by a disjunction of non-overlapping intervals. Similar to `GenSTP-0` and `GenSTP-1`, we propose two generators of random TCSP instances. As for the case of the STP, the first generator (`GenTCSP-0`) does not guarantee that its instances are consistent (have a solution), whereas the second one (`GenTCSP-1`) does with probability p_c .

In addition to the inputs specified for the STP generators, TCSP generators take as input k , the maximum number of non-overlapping intervals in the label of an edge.

2.2.1 `GenTCSP-0`

Similar to `GenSTP-0`, `GenTCSP-0` randomly selects $|E|$ edges from all possible edges list (size of $\frac{n(n-1)}{2}$). For each edge, the generator randomly selects up to k intervals within R , given as input. The resulting network is tested for connectivity. Since the interval labels of edges are randomly selected, most of those problems are inconsistent. More formally:

1. Start with a list L of all possible edges.
2. Select a random edge $i \in L$ and remove it from L .
3. Randomly select a number t (number of intervals for an edge) $\leq k$

4. Randomly select $2t$ numbers and sort them. $X_1 \leq X_2 \leq \dots X_{2t-1} \leq X_{2t}$ ($X_i \in R$).
Label i with $([X_1, X_2], [X_3, X_4], \dots, [X_{2t-1}, X_{2t}])$.
5. Repeat (2)–(4) until $|E|$ edges have been selected.
6. If the graph is not connected, go to step (1), otherwise save the problem to a file.

2.2.2 GenTCSP-1

Like GenSTP-1, GenTCSP-1 randomly selects $(n - 2)$ distinct points within $R = [1, r]$. These nodes along with the endpoints of R (i.e., 1 and r) constitute the n nodes of the temporal constraint graph (TCG). Edges for TCG are chosen randomly from all possible edges list as in GnSTP-0. However, to guarantee the resulting TCSP has at least one solution, the lower l_i and upper u_i values of one of the interval in the label of a given edge i between two nodes a and b (with $b > a$) are chosen such that $l_i \leq (b - a)$ and $u_i \geq (b - a)$. To add more labels per edge, the generator also adds up to $k/2$ labels before interval $[l_i, u_i]$, and up to $k/2$ labels after interval $[l_i, u_i]$ (the total number of intervals per label is up to $k + 1$). We also add a parameter H to define the range of all labels. This process is guaranteed to generate a consistent network.

In order to include some inconsistent instances in a given pool of problems generated using this process, we swap the labels of two random edges in the graph with probability of $1 - p_c$. More formally:

1. Randomly select $(n - 2)$ distinct points within a given interval $R = [1, r]$. Each point corresponds to the ‘position’ of a node in the graph. The first node of the graph has position 1, and the last node in the graph has position r .
2. Start with a list L of all possible edges.
3. Select a random edge $i = (a, b)$ from L and remove it from L .

4. Let δ be the distance between the two points a and b of edge i . Choose two values $0 < \alpha, \beta \leq \delta$, and set the label of the edge i to be $[l_i, u_i]$, $l_i = \delta - \alpha$ and $u_i = \delta + \beta$.
5. Randomly select number $p \leq k/2$. Randomly select $2p$ numbers and sort them. $x_1 \leq x_2 \leq \dots x_{2p-1} \leq x_{2p}$ ($x_i \leq H/2$) Push $([l_i - x_1, l_i - x_2], [l_i - x_3, l_i - x_4], \dots, [l_i - x_{2p-1}, l_i - x_{2p}])$ into label i .
6. Randomly select number $q \leq k/2$. Randomly select $2q$ numbers and sort them. $y_1 \leq y_2 \leq \dots y_{2q-1} \leq y_{2q}$ ($y_i \leq H/2$) Append label i with $([u_i + y_1, u_i + y_2], [u_i + y_3, u_i + y_4], \dots, [u_i + y_{2q-1}, u_i + y_{2q}])$.
7. Repeat (3)–(6) until $|E|$ edges have been processed.
8. With probability $(1 - p_c)$, swap the labels of two random edges in the graph.
9. If the graph is not connected, go to step (1), otherwise save the problem to a file.

Summary

We state four random STP generators and two random TCSP generators in this chapter. Since we want compare the performance of STP solvers for finding the consistency of STP or finding the minimal network of STP, we only using GenSTP-1, GenSTP-2 and SPRAND to test our STP solvers. To compare the performance of TCSP solvers for finding all solution of TCSP, we use GenTCSP-1 to generate random TCSPs.

Chapter 3

Solving STP

In this chapter, we propose a new efficient algorithm, the Δ STP-solver, for computing the minimal network of the Simple Temporal Problem (STP). This algorithm achieves high performance by exploiting a topological property of the constraint graph (i.e., triangulation) and a semantic property of the constraints (i.e., convexity) in light of the results reported by Bliet and Sam-Haroud [6], which were presented for general CSPs and have not yet been applied to temporal networks. Importantly, we design the constraint propagation in Δ STP-solver to operate on triangles instead of operating on edges and implicitly guarantee the decomposition of the constraint graph according to its articulation points. We also provide extensive empirical evaluations of all known algorithms for solving the STP on sets of randomly generated problems. Our experiments demonstrate significant improvements of Δ STP-solver, in terms of number of constraint checks and CPU time, over previously reported algorithms such as the Floyd-Warshall algorithm (F-W) [11, 15], Directed-Path Consistency (DPC) [15], and Partial Path-Consistency (PPC) [6].

3.1 Introduction

Many critical applications in planning and scheduling rely on an efficient handling of temporal information represented as a Simple Temporal Problem (STP) [12, 15, 8]. The efficiency of the constraint propagation in such a network is particularly crucial in autonomous space applications as demonstrated by the Deep Space 1 Remote Agent experiment [26]. Further, an efficient STP solver is a crucial component for solving the Temporal Constraint Satisfaction Problem (TCSP) because the search process designed by Dechter et al. [15] for solving the TCSP requires solving an STP at *each* node expansion. Thus, the performance of the overall process depends heavily on the performance of solving an STP. In this chapter, we propose a new algorithm, Δ STP-solver, for solving the STP and demonstrate empirically that it constitutes a dramatic improvement over previously used algorithms.

We achieve this by first combining the results developed by Bliet and Sam-Haroud [6] for general Constraint Satisfaction Problems (CSPs) with a *new* strategy for constraint propagation, which restricts the propagation effort to the triangles of the triangulated constraint network instead of its edges. Then, we apply the resulting mechanism to solve the STP. The triangulation of the graph and the convexity of the constraints in the STP guarantee that Δ STP-solver is complete and sound for proving the consistency of the STP and for finding the minimal (and decomposable) network. This chapter is structured as follows. Section 3.2 recalls the main properties of a CSP and shows how we use them in our study. Section 3.3 discusses the algorithms for solving the STP and explains the advantages of the Δ STP-solver. Section 3.4 describes our experiments and results, and summarizes our observations. At the end of this chapter, a brief summary is given to conclude this chapter.

3.2 Background

A Constraint Satisfaction Problem (CSP) is defined as follows. Given a set of variables, each with a set of possible values defining its domain, and a set of constraints that restrict the combinations of values that the variables can be assigned at the same time, the task is to assign a value to each variable such that all constraints are simultaneously satisfied. Path consistency, as we discuss below, is an important property of a CSP. Recently Bliet and Sam-Haroud [6] proposed the Partial Path Consistency (PPC) algorithm, which determines whether or not a network is path consistent. Since PPC operates on the triangulated constraint graph¹, it realizes significant computational savings over previously known algorithms, especially for sparse networks. In this chapter, we first improve the propagation mechanism of the PPC algorithm by making it operate on triangles instead of individual edges. We then use the improved version to solve the STP. In the next section, we recall the main properties of a CSP and discuss them in light of the STP.

3.2.1 Main CSP properties

The general properties of constraint graphs and the main algorithms for achieving them are outlined below.

- *Path consistency*: This property ensures that given two values for any two variables that satisfy the constraint between these variables, we can find values for variables in any path of any length (possibly infinite) that satisfy the constraints *along* the path [25]. In general CSPs, path-consistency algorithms PC (e.g., PC-1 [25] and PC-2 [22]) are used to enforce path consistency by tightening the binary constraints. (They also tighten the domains, thus enforcing strong path-consistency.) Montanari established that these algorithms, which consider only paths of length two, on a *complete*

¹A graph is triangulated if every cycle of length strictly greater than 3 possesses a chord.

graph² guarantee a path-consistent network [25]. The Directional Path-Consistency (DPC) algorithm, which achieves path consistency *along a given ordering* d of the variables in the search process, was proposed by Dechter [13] as an efficient approximation of PC; it guarantees path consistency only in the direction that matters, which is that of search. Recently Bliet and Sam-Haroud [6] proposed the Partial Path Consistency (PPC) algorithm, which determines whether or not a network is path consistent without necessarily producing as tight network as with PC. Since PPC operates on the edges of the triangulated graph (fewer than those of the complete graph), it realizes significant computational savings, especially in sparse networks.

- *Minimality*: Minimality, the central problem in CSPs, is a property stronger than path consistency. It guarantees that all the binary constraints are as explicit (i.e., tight) as possible [25].
- *Decomposability*: Decomposability is stronger than minimality and guarantees that a solution to the CSP can be found backtrack-free. This is a highly desirable property and guarantees the tractability of the CSP.
- *Consistency*: In contrast to the above, the consistency property guarantees only the existence of a solution. Note that decomposability is a sufficient condition for consistency.
- *Decomposition into biconnected components*: The decomposition of the constraint graph into its biconnected components according to its articulation points³ is a known technique for enhancing the performance of solving a CSP in general. It provides an upper bound, in the size of the largest biconnected component, to the search effort [19]. We establish that the new solver we introduce, Δ STP, implicitly decomposes

²If the graph is not complete, it is made so by adding universal constraints between non-adjacent edges.

³An articulation point of a graph is a vertex whose removal disconnects the graph. A graph with an articulation point is separable, otherwise it is biconnected.

the constraint graph into its biconnected components without using articulation point. This important observation justifies its high performance.

3.2.2 Properties of the STP

A Simple Temporal Problem (STP) is defined by a graph $G = (V, E, I)$ where V is a set of vertices i representing time points; E is a set of edges $e_{i,j}$ representing constraints between two time points i and j ; and I is a set of constraint labels for the edges; see Figure 3.1 (left). A constraint label I_{ij} of edge $e_{i,j}$ is a *unique* interval $[a, b]$, $a, b \in \mathbb{R}$, and denotes a constraint of bounded difference $a \leq (j - i) \leq b$. A Temporal Constraint Satisfaction Problem (TCSP) is defined by a similar graph $G = (V, E, I)$, where each edge label $I_{ij} = \{l_{ij}^{(1)}, l_{ij}^{(2)}, \dots, l_{ij}^{(k)}\}$ is a *set* of disjoint intervals denoting a disjunction of constraints of bounded differences between i and j , see Figure 3.1 (right). We assume that the intervals

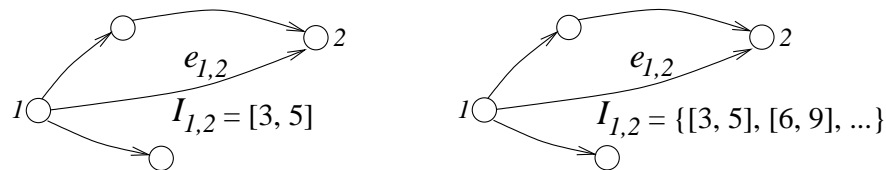


Figure 3.1: *Left: STP. Right: TCSP.*

in a label are ordered in a canonical way. In this section we focus on STPs, but we are integrating our results into an algorithm for solving TCSPs. Below, we show how we exploit the properties of Section 3.2.1 in the context of the STP.

1. *Triangulation of network and convexity constraints.* In addition to proposing PPC, Bliet and Sam-Haroud also showed that when the constraints are *convex*, the PC algorithm (operating on the complete graph) and the PPC algorithm (operating on the triangulated graph) yield the same labeling for the edges common to both graphs. *This important feature of the PPC algorithm has never been exploited before in the context of STPs*, in which the constraints, linear inequalities, are indeed convex. Our

Δ STP-solver exploits this result and yields significant savings of the computational efforts over previously available techniques for establishing path consistency of the STP.

2. *Distribution of composition over intersection.* The two operators on binary constraints for establishing path consistency are constraint composition \otimes and constraint intersection \oplus . Montanari showed that, when constraint composition is distributive over constraint intersection, PC guarantees not only path consistency but also minimality and decomposability [25]. In the case of the STP, constraint composition is interval addition, and constraint intersection is interval intersection, which verify the distributivity as noted by Dechter et al. [15]. Therefore we can deduce that the PPC algorithm and the Δ STP-solver, guarantee the minimality and decomposability of the STP. DPC does not guarantee the path-consistency, minimality or decomposability of the constraint network, however, and this is an important feature, it can be used to determine the consistency of the STP.
3. *Decomposition into biconnected components.* In the special case of the TCSP, and a fortiori the STP, Dechter et al. [15] showed that each biconnected component can be solved independently. If all the components are found to be consistent, then the entire network is consistent. If any of the components is not consistent, then the overall temporal network is not consistent. The minimal network of the original problem is obtained by the union of the minimal networks of the individual biconnected components. When the constraint graph is sparse, this property is particularly attractive. This allows us to process the components in parallel, by independent agents. Thus, decomposition into biconnected components is particularly attractive in the case of STPs, especially for large problems with low density. We show that this decomposition is implicit and automatic in our Δ STP-solver.

3.3 STP algorithms

Here we discuss four different algorithms to solve STPs. The first two solvers, F-W and DPC, have been extensively studied. However, their performance in combination with a decomposition strategy according to articulation points has never been compared before. The third STP solver we study is PPC, which has never before been used on temporal reasoning problems. Finally, we introduce our new solver, Δ STP.

3.3.1 F-W and DPC exploiting articulation points

The Floyd-Warshall (F-W) algorithm for computing all-pairs shortest-paths is a special case of the PC algorithm. F-W is applied to the distance graph of an STP to compute its minimal network in $\Theta(n^3)$. As discussed in Section 3.2, DPC is a single pass algorithm and weaker than PC. It does not necessarily yield a path consistent, minimal, or decomposable network, but it determines if the STP is consistent. DPC is more efficient than F-W; instead of $\Theta(n^3)$, DPC determines the consistency of STP in $\mathcal{O}(nW^*(d)^2)$, where $W^*(d)$ is the maximum number of parents that a node has in the induced graph along the ordering d , which is substantially smaller than n .

We modify the F-W and DPC algorithms to exploit the existence of articulation points in the temporal network. First, we identify the biconnected components [11], then we execute a particular STP solver on each component, independently. This yields two algorithms, F-W+AP and DPC+AP, respectively. It is easy to show that F-W+AP and DPC+AP never check more constraints than F-W and DPC. In fact, for a sparse network, our experiments show that they check substantially less. We also show empirically that, even in the absence of articulation points, F-W+AP and DPC+AP almost never require more CPU time than the original algorithms; when they do, the difference is insignificant due to the overhead for finding the articulation points.

3.3.2 PPC algorithm for STPs

PPC was introduced for general CSPs by Bliet and Sam-Haroud [6] who showed that the path-consistency property can be determined in constraint graphs by triangulating them instead of completing them. They showed a significant improvement in performance in comparison to PC in sparse networks. They also established that, for convex constraints, both PPC and PC compute the same labeling for the edges common to both graphs. Since the constraints in the STP (constraints of bounded difference) are convex, we apply for the first time PPC to solve a continuous domain problem and compute the minimal network of the STP.

As specified in Figure 3.2, the PPC algorithm starts by triangulating the constraint graph G , then iterates over a queue Q_E of all edges, including those edges added to the temporal graph by the triangulation process. It pops an *arbitrary* edge $e_{i,j}$ from the queue, recovers

```

PPC ( $\mathcal{P}$ ):
Begin
consistency  $\leftarrow$  True
 $G \leftarrow$  Triangulate ( $\mathcal{P}$ )
 $Q_E \leftarrow$  edges in  $G$ 
While  $Q_E \wedge$  consistency Do
   $e_{i,j} \leftarrow$  Dequeue( $Q_E$ )
  Forall  $k$  such that  $\langle i, j, k \rangle$  is a subgraph of  $G$  Do
     $I'_{ij} \leftarrow I_{ij} \oplus (I_{ik} \otimes I_{kj})$ 
    When  $I'_{ij} \neq I_{ij}$  Then  $I_{ij} \leftarrow I'_{ij}$  and Enqueue( $e_{i,j}, Q_E$ )
     $I'_{ik} \leftarrow I_{ik} \oplus (I_{ij} \otimes I_{jk})$ 
    When  $I'_{ik} \neq I_{ik}$  Then  $I_{ik} \leftarrow I'_{ik}$  and Enqueue( $e_{i,k}, Q_E$ )
     $I'_{jk} \leftarrow I_{jk} \oplus (I_{ji} \otimes I_{ik})$ 
    When  $I'_{jk} \neq I_{jk}$  Then  $I_{jk} \leftarrow I'_{jk}$  and Enqueue( $e_{j,k}, Q_E$ )
    When  $I_{ij}, I_{ik}$  or  $I_{jk}$  is empty Then consistency  $\leftarrow$  False
  Return consistency
End

```

Figure 3.2: The PPC algorithm, slightly improved to consider simultaneously all three edges in a triangle.

all triangles $\langle i, j, k \rangle$ in which $e_{i,j}$ participates, and updates its label I_{ij} by composing the intervals I_{ik} and I_{kj} and intersecting the result of this composition with the interval I_{ij} . We slightly modify the original algorithm to allow it to update all three edges at once and to terminate when the queue is empty or inconsistency is found. The distributivity property of interval addition over interval intersection guarantees that running PPC on an STP results in the tightest possible labeling (i.e., minimal) of the existing edges.

3.3.3 \triangle STP algorithm

The goal of PPC is to make the labels of the edges of the triangulated constraint graph as tight as possible. When the label of an edge in a triangle is not as tight as it could be, given the labels of the other edges in the triangle, the label is tightened accordingly. This process may require tightening the other edges in the triangle as shown in Figure 3.3. In

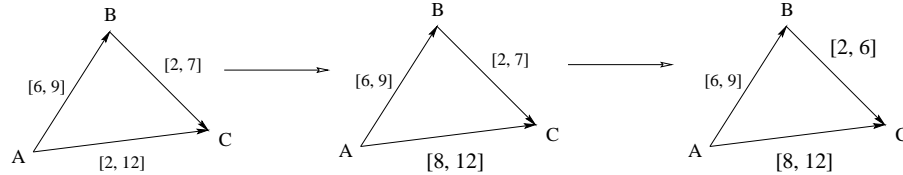


Figure 3.3: An example of updating edges. The label of edge BC then that of AC are updated.

this example we can see that it is worth considering all three edges of a given triangle simultaneously and updating them sequentially. This observation is the basis of our first improvement to PPC, and is already integrated in the algorithm of Figure 3.2.

When the label of an edge in a given triangle is updated, PPC triggers constraint propagation over *all* the triangles in which *any* of the edges of the original triangle participate. This is clearly an overkill since only the triangles in which the updated edges participate need to be considered. This observation was the motivation for our new algorithm.

While all existing methods consider the temporal network as composed of edges, our new algorithm considers the STP as composed of triangles (see Figure 3.4). The graph of

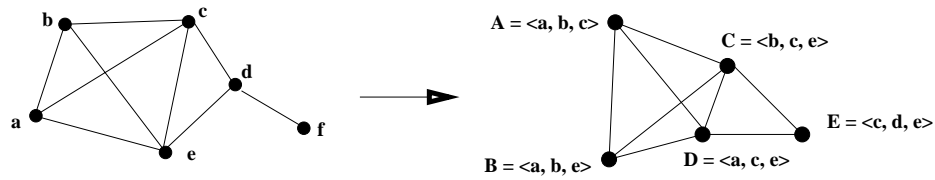


Figure 3.4: *The temporal graph as a graph of triangles.*

the temporal network is replaced by a graph of triangles. Each triangle is represented by a node, and two nodes are connected if and only if the triangles they represent have a common edge. Thus ΔSTP appears as an AC3-like algorithm [22] on this graph of triangles. If an edge of the original constraint graph is not a part of any triangle, it is omitted from the graph of triangles. Indeed, an edge that does not appear in any triangle has no effect on the constraint propagation in the STP and thus can be safely omitted from the graph of triangles. Consequently:

Proposition 3.3.1. *A tree-structured STP is decomposable and consistent, and its edge labels are minimal.*

We call our new algorithm ΔSTP , although it is applicable to general CSPs and would more correctly be called ΔPPC . The new algorithm is shown in Figure 3.5. First, we triangulate the temporal network, using for example the algorithm devised in [28], which may result in new edges. We add these edges to the original constraint graph as universal constraints setting their label to $(-\infty, \infty)$. Then we put all the triangles into a queue, Q_T , of size $O(|E| \text{ degree}(G))$. We check every triangle in the queue. If a given triangle $\langle i, j, k \rangle$ is not minimal, then we update one or more of its edges. We then retrieve all the adjacent triangles that contain any of the updated edges and add them to Q_T if they are not already there. Finally, we remove $\langle i, j, k \rangle$ from the queue, and repeat this process until Q_T is empty or inconsistency is found.

```

 $\Delta$ STP ( $\mathcal{P}$ ):
Begin
consistency  $\leftarrow$  True
 $G \leftarrow$  Triangulate the graph of  $\mathcal{P}$ 
 $Q_T \leftarrow$  all triangles in  $G$ 
While  $Q_T \wedge$  consistency Do
   $Q_E \leftarrow$  empty list
   $\langle i, j, k \rangle \leftarrow$  First( $Q_T$ )
   $I'_{ij} \leftarrow I_{ij} \oplus (I_{ik} \otimes I_{kj})$ 
  When  $I'_{ij} \neq I_{ij}$  Then  $I_{ij} \leftarrow I'_{ij}$  and Enqueue( $e_{i,j}$ ,  $Q_E$ )
   $I'_{ik} \leftarrow I_{ik} \oplus (I_{ij} \otimes I_{jk})$ 
  When  $I'_{ik} \neq I_{ik}$  Then  $I_{ik} \leftarrow I'_{ik}$  and Enqueue( $e_{i,k}$ ,  $Q_E$ )
   $I'_{jk} \leftarrow I_{jk} \oplus (I_{ji} \otimes I_{ik})$ 
  When  $I'_{jk} \neq I_{jk}$  Then  $I_{jk} \leftarrow I'_{jk}$  and Enqueue( $e_{j,k}$ ,  $Q_E$ )
  When  $I_{ij}$ ,  $I_{ik}$  or  $I_{jk}$  is empty Then consistency  $\leftarrow$  False
  When consistency
    For  $e_{m,n} \in Q_E$  Do
       $T_{m,n} \leftarrow$  all triangles containing  $e_{m,n}$ 
      For  $\langle r, t, s \rangle \in T_{m,n}$  Do Unless  $\langle r, t, s \rangle \in Q_T$  Then Enqueue( $\langle r, t, s \rangle$ ,  $Q_T$ )
     $Q_T \leftarrow$  Remove( $\langle i, j, k \rangle$ ,  $Q_T$ )
Return consistency
End

```

Figure 3.5: *The Δ STP algorithm.*

3.3.4 Features of Δ STP

We summarize the features of Δ STP as follows:

- Δ STP has the same pruning power as F-W with less effort. Δ STP achieves minimality on the triangulated graph, without requiring the completion of the graph, which is necessary for F-W. This yields dramatic gains in the computational effort.
- Δ STP automatically decomposes the graph into its biconnected components. The decomposition of the graph into its biconnected components is an effective technique to bind the search effort and enhance the performance of solving a CSP. Our experiments of Figure 3.7 and 3.8 and Table 3.2 and 3.3 show how such a strategy im-

proves the performance of the F-W algorithm, even when the articulation points must be explicitly identified. Because constraints propagate through triangles, PPC and consequently Δ STP implicitly exploit the decomposition into biconnected components. Consider a triangulated temporal network composed of two sets of nodes $X = \{p, x_1, x_2, \dots, x_m\}$ and $Y = \{p, y_1, y_2, \dots, y_n\}$, and p is the articulation point. Suppose that edges exist only between nodes in either X or Y . Since no edges connect these two sets, there obviously are no triangles that connect them. All triangles are either in set X or in set Y . As shown in Figure 3.4, two triangles in the graph of triangles can only be connected by a common edge. Therefore, no triangle in set X is connected to a triangle in set Y . When PPC and consequently Δ STP propagate constraints through neighboring triangles, no updates in set X may affect triangles in set Y . As a result, PPC and Δ STP implicitly guarantee that articulation points in the graph (if any), are exploited, as if the network was decomposed into its biconnected components without actually decomposing it.

- *Δ STP is cheaper than PPC.* Δ STP and PPC use the same idea of Bliet and Sam-Haroud [6]; however, Δ STP is more careful about how updates are propagated and thus exploits triangulation of the graph more effectively than PPC. Although propagation of PPC occurs through triangles, PPC does not have a mechanism to record which triangles really need to be checked. This inability causes some unnecessary constraint checks and a waste of CPU time.
- *Our improvement in solving the STP directly benefits the task of solving the TCSP.* TCSP is NP-hard and is solved with backtrack search. Every node expansion in the search tree needs to solve an STP. Thus a good STP solver is crucial for solving the TCSP. We are currently demonstrating this idea and showing how the decomposition into independent components is particularly useful in this context.

Table 3.1: *Parameters of generated problems.*

Generator	Problem size				Samples per point	Results
	#Nodes	Density		#Edges		
		Range	Step	Range	Step	
GenSTP-1	50, 100	[0.01, 0.1]	0.01			100
	50, 100	[0.2, 0.9]	0.1			100
SPRAND	50	[200, 2000]	200			100
	100	[400, 1400]	200			100
	100	[1600, 2800]	400			100
	257	0.016		768		5
	513	0.008		1536		5
GenSTP-2	256	0.016		$3 \times 256 = 768$		5
	512	0.008		$3 \times 512 = 1536$		5

3.4 Empirical evaluations

We implemented the following six algorithms in Common Lisp. Floyd-Warshall (F-W), Directed-Path Consistency (DPC), and in combination with a mechanism for detecting and exploiting articulation points, F-W+AP and DPC+AP, Partial Path Consistency (PPC), and our new triangle-based solver (Δ STP). We used three generators of random STPs: GenSTP-1, SPRAND, and GenSTP-2. GenSTP-1 is our own generator. We designed it to guarantee that graphs are connected and that at least 80% of the generated instances are consistent. SPRAND is one class of STPs generated by the public domain library SPLIB, [9]. All the problems we generate with SPRAND have a cycle connecting all the nodes (i.e., a structural constraint). This guarantees strong connectivity and the absence of any articulation points. Finally, GenSTP-2 is a generator given to us by Ioannis Tsamardinos and was used in [32]. GenSTP-2 does not enforce the existence of a structural constraint. The density of the temporal network is defined as $Density = \frac{|E| - |E_{min}|}{|E_{max}| - |E_{min}|}$. Table 3.4 summarizes the characteristics of the problems tested, including the size of the instances and the number of samples generated for each measurement point. The results, measured in terms of the number of constraint checks and CPU time, were averaged over the number

of instances and showed a precision of 5%. The detailed data of the above experiments on the instances generated by GenSTP-1 and SPRAND are shown in Table 3.2 and 3.3. The CPU time measurements are made in msec, with a clock resolution of 10 msec.

3.4.1 Experiments conducted

Using the 50-node problems generated by GenSTP-1, we conducted the following experiments:

- *Managing the queue in Δ STP.* The manner in which triangles are inserted in the queue affects the performance of Δ STP. We tested three heuristics for adding the triangles to the queue: at the front of queue (Δ STP-front), at the end of queue (Δ STP-back), and random insertion into the queue (Δ STP-random). All three strategies resulted in the same output (i.e., the same label of the edges). The results in terms of constraint checks are presented in Figure 3.6, The results show that Δ STP-back consistently performs the least number of constraint checks. This can

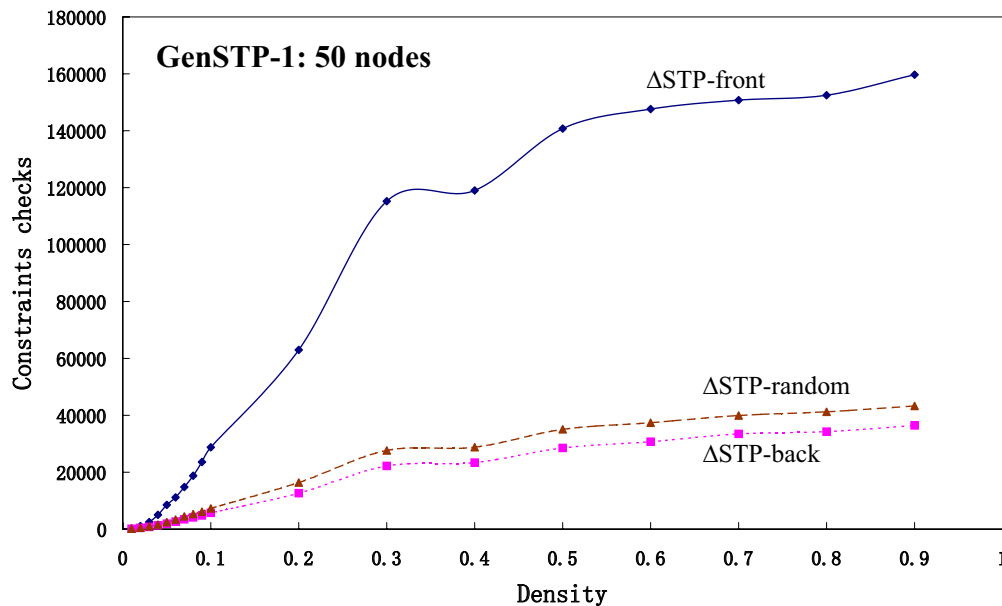


Figure 3.6: Constraint Checks for Δ STP-front, Δ STP-back and Δ STP-random.

be informally interpreted as follows. It is more effective to propagate the constraints as early as possible across the network, in a ‘sweeping’ manner. Interestingly, we noticed that quiescence was consistently reached in 7 or fewer iterations. We use Δ STP-back in the rest of our study.

- *Computing the minimal network.* F-W, F-W+AP, PPC and Δ STP (but not DPC) result in the labels of the common edges, the minimal labels. Figure 3.7 shows that Δ STP clearly and significantly dominates all others, for all values of density.
- *Saving on the constraint checks.*
DPC does not necessarily yield the minimal network, but it can determine whether or not the network is consistent in significantly fewer constraint checks than F-W. Figure 3.8 shows that Δ STP, which is more powerful in terms of pruning power and yields the minimal network, dominates DPC-like strategies when density is less than 50%.
- *Effect of problem size.* In order to compare the performance of these different solvers on larger problems, we tested them on larger problems generated by SPRAND and GenSTP-2. Figure 3.9 and 3.10 show the ratio of the number of constraint checks and that of the CPU time needed for all six strategies tested in reference to the values needed for F-W.

Random STP generated by GenSTP-1 with 50 nodes												
Density	F-W		F-W+AP		DPC		DPC+AP		PPC		ΔSTP	
	CC	CPU (s)	CC	CPU (s)	CC	CPU (s)	CC	CPU (s)	CC	CPU (s)	CC	CPU (s)
0.01	122200.5	0.822	29924.05	0.2091	1777.03	0.1168	744.44	0.0307	273.97	0.0039	125.75	0.0025
0.02	123001.5	0.8347	59091.93	0.4026	3572.7	0.1304	2364.62	0.0683	837.9	0.0109	409.64	0.0045
0.03	120339.99	0.8389	79195.61	0.5297	4769.95	0.1376	3833.36	0.0945	1532.55	0.02	761.71	0.0091
0.04	120044.01	0.8063	90934.63	0.6029	6411.11	0.1547	5525.58	0.1176	2529.68	0.03	1270.41	0.0115
0.05	117382.5	0.7935	99076.94	0.6591	8106.14	0.161	7510.24	0.1394	3766.13	0.0433	1910.97	0.0188
0.06	120075.49	0.8209	108975.06	0.7251	10204.46	0.1804	9746.2	0.1679	5207.57	0.0599	2622.19	0.0269
0.07	120940.51	0.8637	113426.05	0.756	11487.391	0.189	11175.431	0.1818	6679.19	0.0782	3445.79	0.0358
0.08	116800	0.7862	112267.63	0.7598	11715.94	0.1894	11447.12	0.181	7861.92	0.0879	4109	0.0424
0.09	115321.5	0.7778	112951.92	0.7525	13024.311	0.1976	12915.95	0.1986	9240.66	0.1031	4800.74	0.0531
0.1	116336.5	0.7947	114676.23	0.7617	14072.08	0.2115	13975.311	0.207	10857.08	0.1247	5705.62	0.0649
0.2	108926.5	0.7335	108852.99	0.7342	21203.27	0.2717	21203.262	0.2705	23677.2	0.2624	12631.6	0.1533
0.3	120195.99	0.8113	120195.99	0.8019	28912.988	0.347	28912.988	0.3442	41404.09	0.4637	22206.16	0.2676
0.4	106959.5	0.7213	106959.5	0.7147	27121.85	0.3313	27121.85	0.3252	43483.79	0.4958	23388.791	0.291
0.5	108896.5	0.7487	108896.5	0.732	29731.49	0.3506	29731.49	0.3514	53446.668	0.6162	28504.24	0.3553
0.6	109074.99	0.7376	109074.99	0.7314	31533.85	0.3732	31533.85	0.3692	57422.24	0.6662	30716.22	0.4083
0.7	109592	0.7502	109592	0.7294	32002.16	0.3795	32002.16	0.3725	62265.727	0.7224	33464.38	0.4269
0.8	107428.51	0.7298	107428.51	0.7116	32391.83	0.3816	32391.83	0.3719	64625.727	0.7439	34257.42	0.443
0.9	108566.5	0.741	108566.5	0.7207	33249.992	0.3925	33249.992	0.3796	67977.31	0.7931	36429.34	0.4616
Random STP generated by GenSTP-1 with 100 nodes												
0.01	976155.06	8.3611	486223.66	4.088	21574.19	1.0156	14401.68	0.5275	4424.22	0.0586	2225.99	0.0108
0.02	955417	8.2284	737264.25	6.2037	45044.293	1.3432	39022.73	0.9329	14764.17	0.2035	7803.66	0.0772
0.03	944883	7.9927	855073.25	7.142	71363.34	1.3655	67750.06	1.2528	31849.158	0.3818	16698.209	0.1795
0.04	920881.06	7.8254	879589.9	7.3463	89384.945	1.4859	87387.805	1.4347	49463.91	0.5777	26350.969	0.3076
0.05	931483.06	7.8308	918906.56	7.71	115620.83	1.7429	114994.93	1.7121	72491.46	0.8411	38301.637	0.472
0.06	886372.94	7.5324	879934.7	7.3403	116526.336	1.6933	116144.984	1.6616	85443.125	1.0262	45141.34	0.5847
0.07	916842	7.7882	914465.9	7.6159	145073.03	1.9288	144846.11	1.9396	113607.77	1.3013	61303.09	0.8185
0.08	924955.94	7.907	924361.94	7.7039	148479.61	1.9416	148393.72	1.9335	129904.16	1.4633	70892.98	0.9267
0.09	935953	7.9439	935805.6	7.7978	167192.17	2.1092	167192.17	2.1225	161399.25	1.8614	86110.63	1.1857
0.1	895177	7.7186	894583	7.4615	165887.34	2.086	165803.48	2.0561	165634.69	1.9312	90790.92	1.2733
0.2	883597	7.5387	883597	7.3604	218225.31	2.4666	218225.31	2.4527	320976.06	3.7723	175113.86	2.6166
0.3	860400	7.4074	860400	7.1667	232372.25	2.5446	232372.25	2.5384	396075.3	4.7658	219178.31	3.3071
0.4	833850	7.1203	833850	6.9936	240254.4	2.6094	240254.4	2.5553	446748.47	5.4984	247012.77	3.805
0.5	879490.06	7.554	879490.06	7.3287	262964.03	2.8133	262964.03	2.7976	520176.78	6.4435	287163	4.4565
0.6	891914.06	7.6565	891914.06	7.4904	276184.53	2.9108	276184.53	2.8815	564749.56	6.734	309157.75	4.7986
0.7	866636	7.4485	866636	7.3051	267027.4	2.8092	267027.4	2.8233	554381.6	6.5875	303306.12	4.7356
0.8	847892	7.3271	847892	7.2994	258738.61	2.733	258738.61	2.6769	552344.1	6.4986	299997.22	4.8764
0.9	854969	7.3954	854969	7.3383	266861.47	2.8032	266861.47	2.7704	568128.25	6.7406	309514.87	4.9663

Table 3.2: Experimental results for STP solvers on random STP generated by GenSTP-1.

Random STP generated by SPRAND with 50 nodes												
Number of Edges	F-W		F-W+AP		DPC		DPC+AP		PPC		Δ STP	
	CC	CPU (s)	CC	CPU (s)	CC	CPU (s)	CC	CPU (s)	CC	CPU (s)	CC	CPU (s)
200	125000	0.8467	125000	0.8255	21824.031	0.2798	21824.031	0.2847	20247.77	0.236	12111.471	0.1595
400	125000	0.8492	125000	0.8301	30981.5	0.3677	30981.5	0.3732	42313.25	0.4893	25902.35	0.347
600	125000	0.8441	125000	0.8244	34524.73	0.4044	34524.73	0.4035	56231.418	0.6606	34142.043	0.4656
800	125000	0.8467	125000	0.8274	36254.89	0.4255	36254.89	0.4176	64894.547	0.7594	39436.86	0.5334
1000	125000	0.8457	125000	0.8281	37302.24	0.4369	37302.24	0.4318	69790.15	0.825	42623.07	0.5697
1200	125000	0.8521	125000	0.8242	38020.63	0.4473	38020.63	0.4382	73899.914	0.8671	44889.09	0.5796
1400	125000	0.8501	125000	0.8243	38502.508	0.4556	38502.508	0.4442	76743	0.9067	46354.59	0.608
1600	125000	0.8513	125000	0.8331	38902.95	0.4647	38902.95	0.4458	79116.336	0.927	47597.69	0.6287
1800	125000	0.8553	125000	0.8343	39166.152	0.4694	39166.152	0.4532	80540.03	0.9526	48321.05	0.6306
2000	125000	0.8621	125000	0.8363	39381.36	0.4577	39381.36	0.4519	81024.4	0.9536	48789.93	0.6291
Random STP generated by SPRAND with 100 nodes												
400	1000000	8.5076	1000000	8.3707	167877.39	2.1703	167877.39	2.1947	144819.36	1.7659	85055.414	1.4427
600	1000000	8.5019	1000000	8.3572	218599.22	2.5686	218599.22	2.5723	241016.73	2.8585	146966.83	2.5927
800	1000000	8.5177	1000000	8.3523	245378.12	2.775	245378.12	2.7759	318725.3	3.7333	198716.12	3.441
1000	1000000	8.5218	1000000	8.3476	263177.97	2.9213	263177.97	2.9205	380805.94	4.4388	236103.58	4.1202
1200	1000000	8.6507	1000000	8.3242	275036.7	3.0351	275036.7	3.0053	434212.72	5.0349	268235.28	4.6083
1400	1000000	8.6643	1000000	8.3216	283548.44	3.0986	283548.44	3.0367	474789.12	5.4202	292905.87	4.886
1600	1000000	8.7028	1000000	8.3169	289520.4	3.1461	289520.4	3.082	512087.9	6.1406	313113.25	5.4773
2000	1000000	8.7978	1000000	8.3284	298104.53	3.2074	298104.53	3.148	565111.94	6.9515	343748.66	6.0293
2400	1000000	8.5296	1000000	8.3569	303608.8	3.2493	303608.8	3.2088	599295	6.7189	365377.84	5.9462
2800	1000000	8.8195	1000000	8.341	307894.12	3.2842	307894.12	3.2199	631238.44	7.3478	382691	6.1807

Table 3.3: Experimental results for STP solvers on random STP generated by SPRAND.

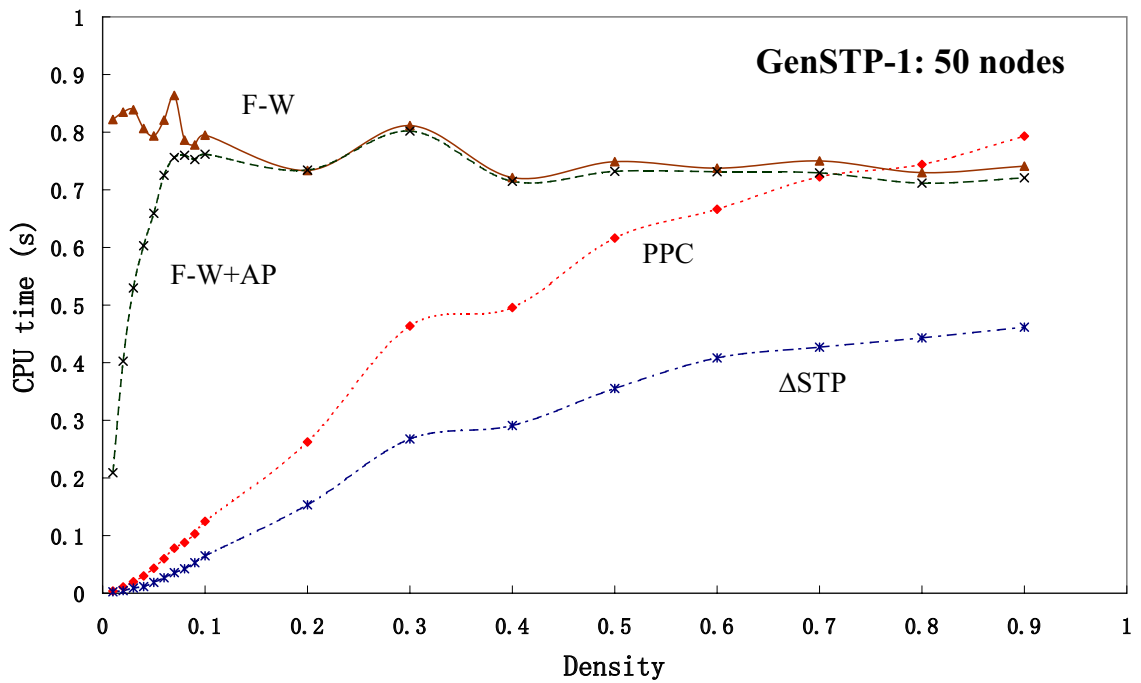
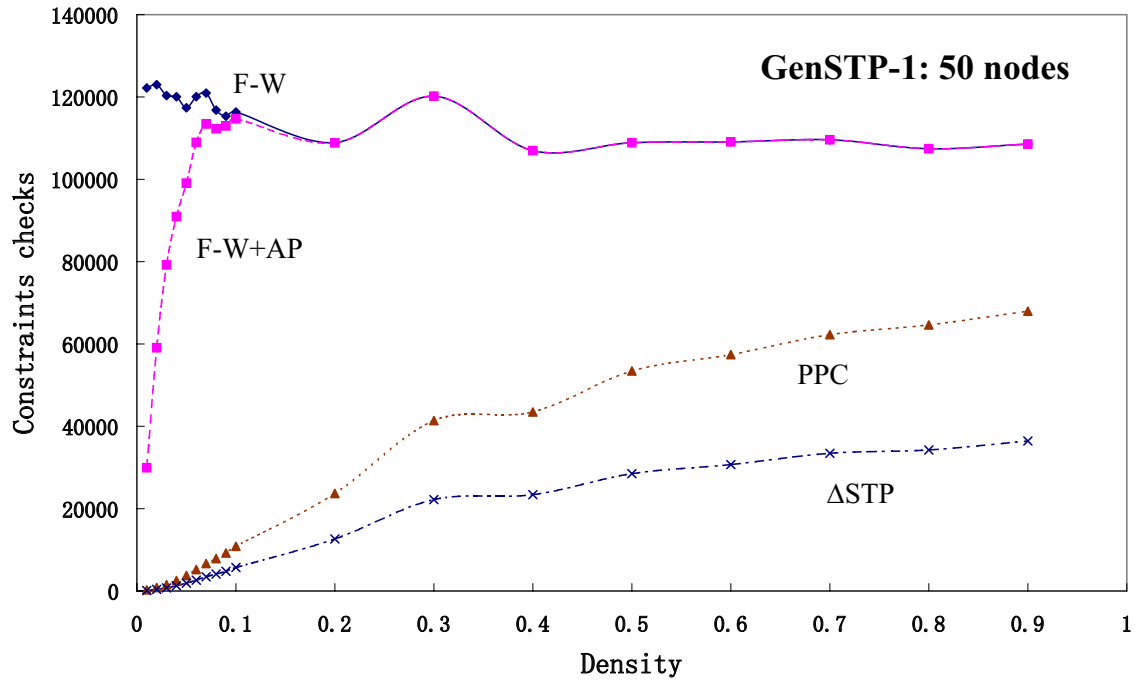


Figure 3.7: Constraint Checks (top) and CPU time (bottom) for F-W, F-W+AP, PPC, and Δ STP.

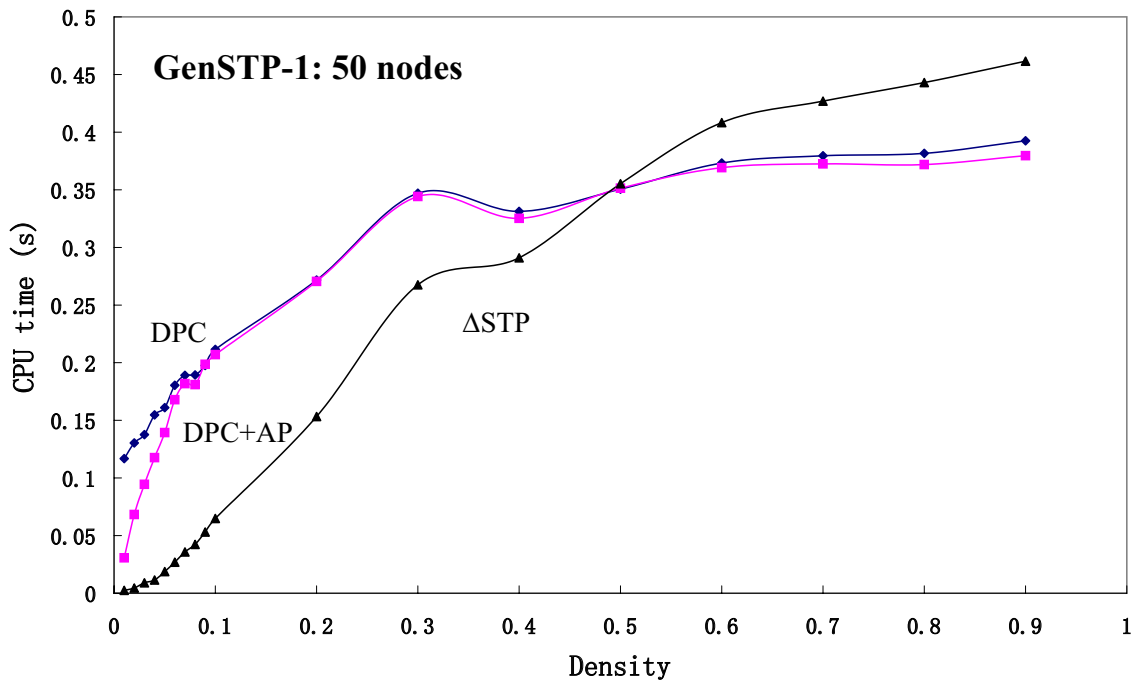
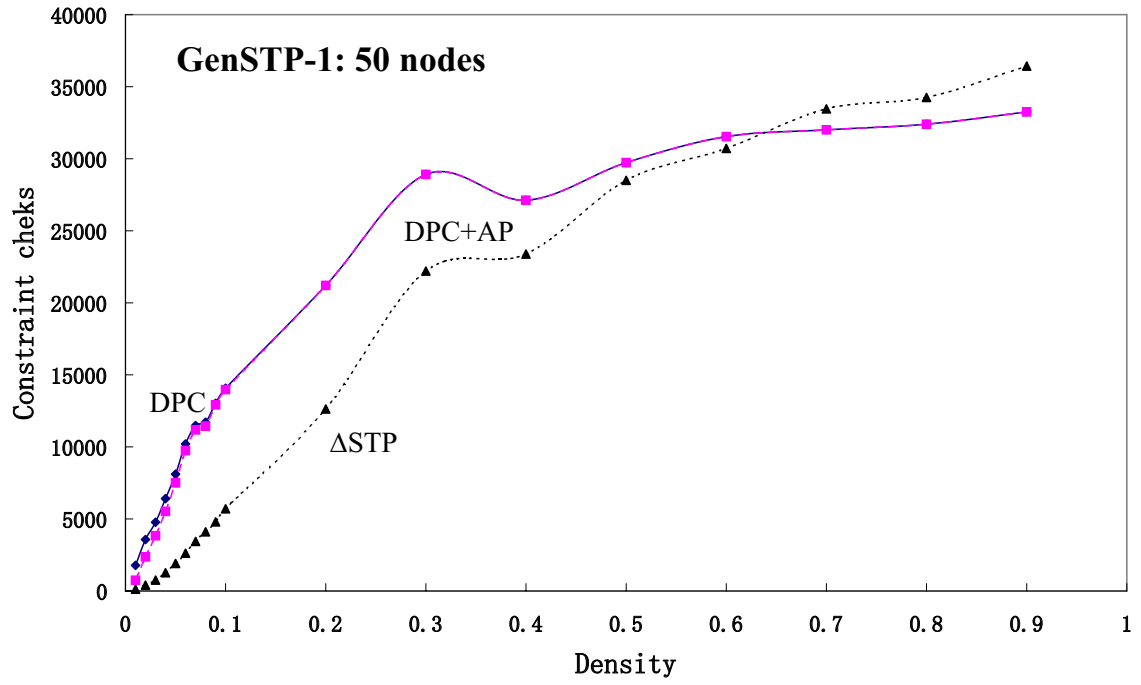


Figure 3.8: Constraint Checks (top) and CPU time (bottom) for DPC, DPC+AP, and Δ STP.

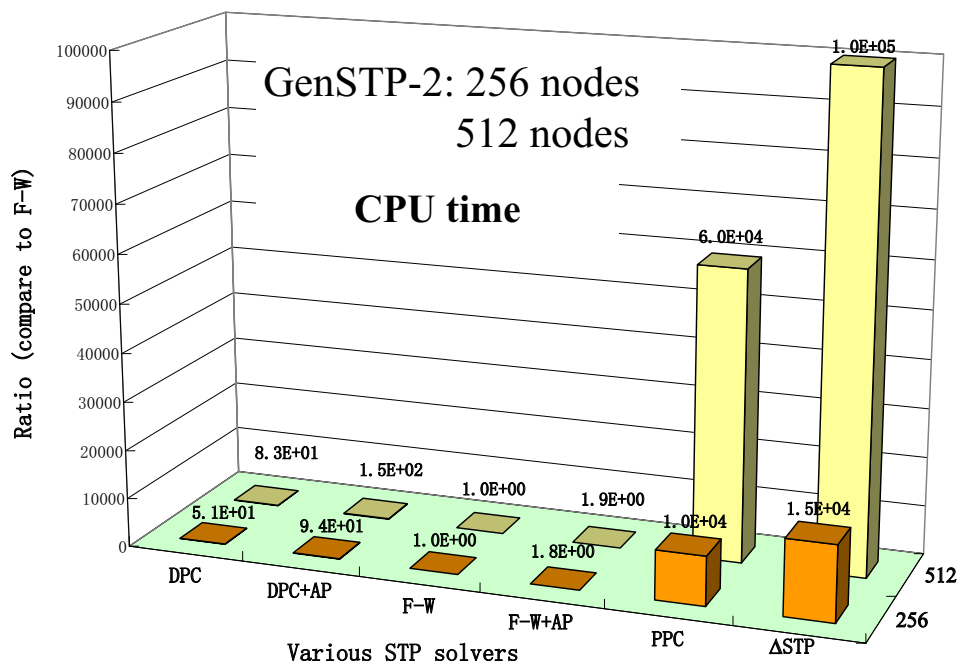
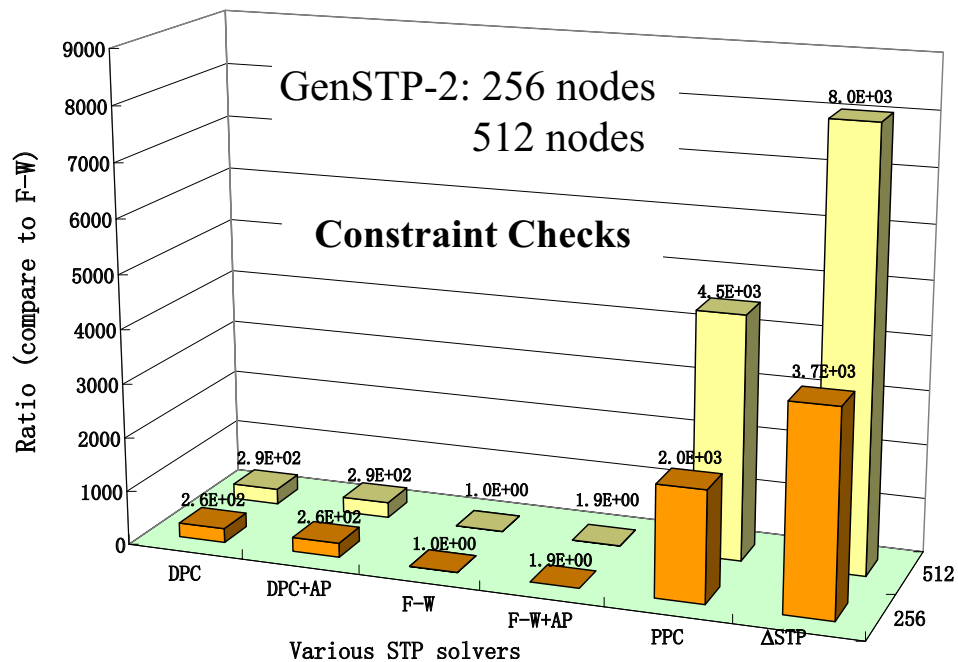


Figure 3.9: Constraint Checks (top) and CPU time (bottom) for STP solvers, problems generated by GenSTP-2.

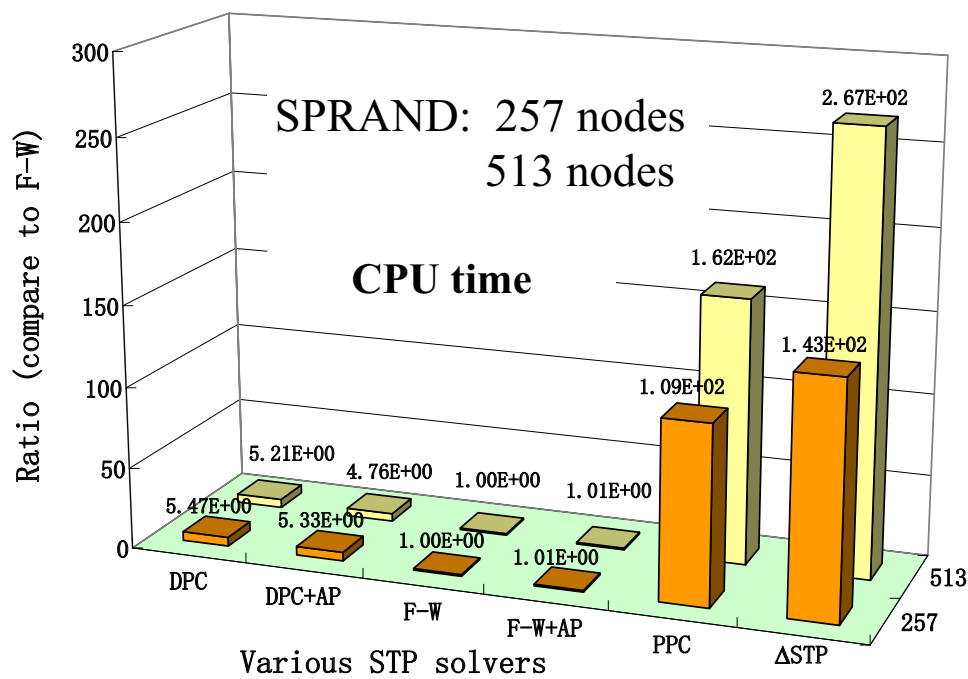
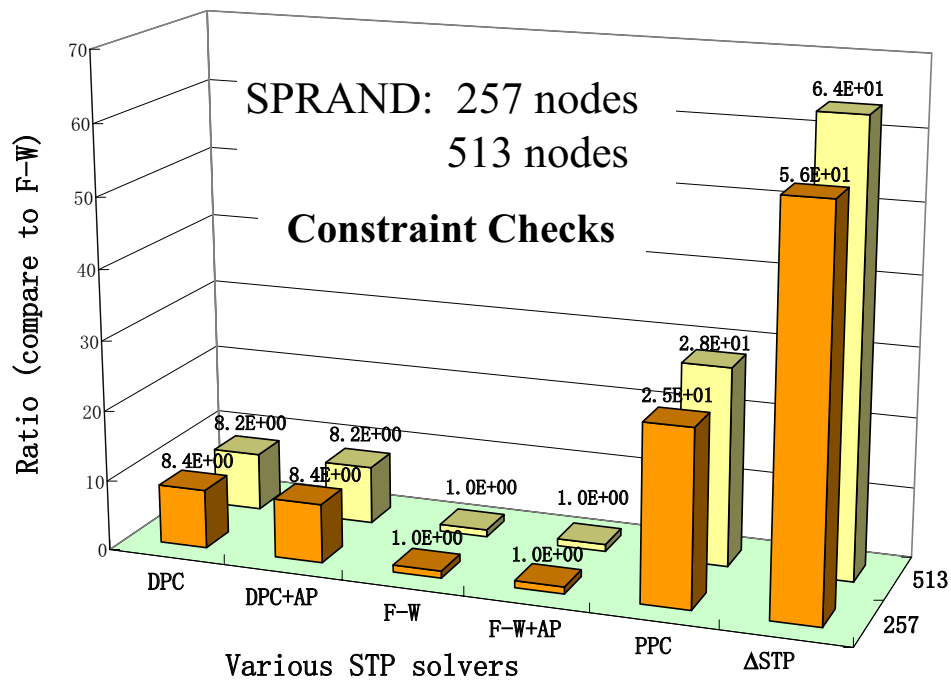


Figure 3.10: Constraint Checks (top) and CPU time (bottom) for STP solvers, problems generated by SPRAND.

3.4.2 Observations

From the above experiments, we draw the following observations:

- *Using articulation points.* Dechter et al. [15] showed that decomposing the temporal network into its biconnected components is particularly effective in enhancing the performance of search. It is worth recalling that this decomposition does not affect the quality of the solution: the same edge labels are found with and without decomposition. Figure 3.7 and 3.8 show that only F-W realizes significant savings when the density is low. In contrast, decomposition into biconnected components does not benefit the DPC solver to the same extent. This can be explained by the fact that the cost of DPC is bounded by $\mathcal{O}(nW^*(d)^2)$, where $W^*(d)$ is the maximum number of parents that a node has in the induced graph. Decomposition does not significantly change the induced width $W^*(d)$; the total cost of solving the subproblems is not significantly smaller than that of solving the original problem. When density is high, the network cannot be decomposed, and F-W+AP and DPC+AP perform almost the same as F-W and DPC, respectively. The problems generated by SPRAND cannot be decomposed because of the existence of a cycle that connects all nodes (i.e., structural constraint). Indeed, Table 3.3 shows the same number of constraint checks for the algorithms with and without articulation points. However, the required effort for finding these articulation points is negligible, since CPU times are the same within the resolution of the clock.
- *Improvements due to PPC.* Given the constraint semantics, PPC is guaranteed to yield the same labels as F-W and F-W+AP on their common edges. Since PPC operates on the triangulated graph it performs significantly better for low density values than F-W, which operates on the complete graph, and even F-W+AP, which exploits the existence of articulation points. When the constraint density increases, the number

of triangles in the graph also increases and so does the cost of PPC. However, the number of constraint checks and, to some extent, the CPU time for PPC remain less than those for F-W and F-W+AP, which quickly reach a stable value, $\Theta(n^3)$. For the larger problems generated by SPRAND and GenSTP-2), Figure 3.9 and 3.10 show the PPC outperforms DPC and DPC+AP, which in turn outperform F-W and F-W+AP. Note, however, that DPC and DPC+AP do not yield the tightest network. A comparison of Figure 3.9 and 3.10 shows that the performance of PPC is better on problems generated by GenSTP-2 than on those generated by SPRAND. This is due to the existence of a cycle connecting all the nodes in problems generated by SPRAND, which prevents decompositions and causes the triangulation process to add relatively more edges.

- *Improvements due to Δ STP.* As a refinement of PPC, Δ STP exploits the benefits of triangulation to a greater degree than PPC does. Experimental results show that Δ STP has always better performance than PPC in all experiments we performed (Figure 3.7 and Table 3.2 and 3.3). For high density values, Δ STP can show a worse performance than DPC (Figure 3.8). However, this slight degradation is misleading since it does not account for the output of these two algorithms. Indeed, Δ STP guarantees the minimal network and DPC does not. Hence, the performance of the former remains superior. The experiments on large problems, shown in Figure 3.9 and 3.10, demonstrate that Δ STP is the absolute winner over all algorithms. A comparison of Figure 3.9 and 3.10 shows that Δ STP, like PPC, is sensitive to the structure of the temporal graph (i.e., the existence of a cycle). It is more effective on problems generated with GenSTP-2 than on those generated with SPRAND.

3.4.3 Significance of our results

In practice, most *real-world* applications exhibit typically STPs with large size and low density [7]. The performance of an STP solver in these situations becomes extremely important. Δ_{STP} is perfect for this kind of job. Its outstanding performance under low density is particularly advantageous and makes it the best algorithm developed to date. Further, when solving a TCSP with search, the STP examined at each node in the search tree is a subgraph of the original TCSP and thus has a lower density than the TCSP. This supports the importance of an efficient STP solver for low density networks. We expect the combination of Δ_{STP} with a TCSP solver to improve dramatically the performance of current TCSP solvers.

Summary

We introduced Δ_{STP} , a new efficient algorithm for solving the STP. Our algorithm advantageously exploits previous results reported in the literature and binds them via a new strategy for constraint propagation based on triangles. We demonstrated that this algorithm outperforms all previous ones in terms of pruning power and performance. More importantly, Δ_{STP} solver provides us with a new perspective on temporal problems as composed by a set of triangles, where two triangles are connected if and only if they have one common edge. Constraint propagation can be carried out according to this new graph of triangles.

Chapter 4

Solving TCSP

In this chapter¹, we address the task of solving the general Temporal Constraint Satisfaction Problem (TCSP). We report the integration of three approaches to improve the performance of the exponential-time, backtrack search (BT-TCSP) proposed by Dechter et al. [15] for this purpose. The first approach consists of using a new efficient algorithm (Δ STP) [35] for solving the Simple Temporal Problem (STP), an operation that must be executed at each node expansion during BT-TCSP. The second approach improves BT-TCSP itself by exploiting the topology of the temporal network. This is accomplished in three ways: finding and exploiting articulation points (AP), checking the graph for new cycles (NewCyc), and using a new heuristic for edge ordering (EdgeOrd). The third approach is a filtering algorithm, Δ AC, which is used as a preprocessing step to BT-TCSP, and which significantly reduces the size of the TCSP [36]. In addition to introducing two new techniques, NewCyc and EdgeOrd, this chapter discusses an extensive evaluation of the merits of the above three approaches. Our experiments on randomly generated problems demonstrate significant improvements in the number of nodes visited, constraint checks, and CPU time.

¹This chapter is the topic of two papers currently submitted for publication [36, 37].

4.1 Background and motivation

A Temporal Constraint Satisfaction Problem (TCSP) is defined by a similar graph $G = (V, E, I)$ as STP, where each edge label $I_{ij} = \{l_{ij}^{(1)}, l_{ij}^{(2)}, \dots, l_{ij}^{(k)}\}$ is a *set* of disjoint intervals denoting a disjunction of constraints of bounded differences between i and j . We assume that the intervals in a label are disjoint and ordered in a canonical way. The following is a typical example:

Tom has class at 8:00 a.m. He can either make breakfast for himself (10-15 minutes), or get something to eat from a local store (less than 5 minutes). After breakfast (5-10 minutes), he goes to school either by car (20-30 minutes) or by bus (at least 45 minutes). Today, Tom gets up between 7:30 and 7:40.

We wish to answer queries such as: “Can Tom arrive at school in time for class?”, “Is it possible for Tom to take the bus?”, “If Tom wanted to save money by making breakfast for himself and taking the bus, when should he get up?”, and so on. This temporal problem can be represented as a temporal graph.

Let P_0 be a reference time-point (e.g., 6:00 am), P_1 the time point Tom gets up, P_2 the time point he starts his breakfast, P_3 the time point he finishes it, and P_4 the time point he arrives at the school. Figure 4.1 shows the temporal graph of this TCSP.

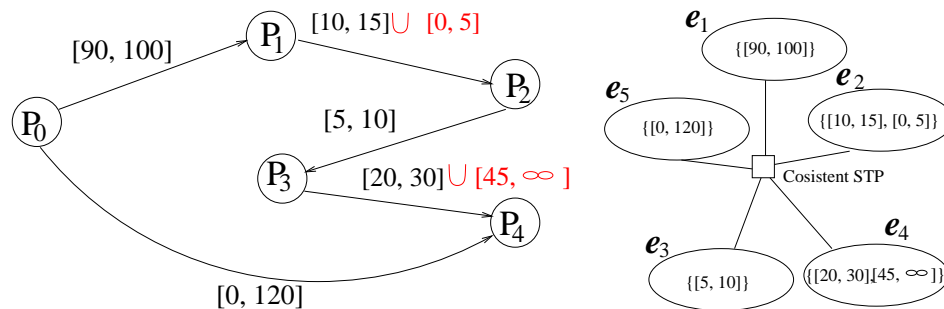


Figure 4.1: A TCSP example (left) and formulate it as meta-CSP (right)

Dechter [13] described a backtrack search procedure (BT-TCSP) for solving a TCSP, which is an NP-hard problem. To this end, the TCSP is expressed as a ‘meta’ Constraint Satisfaction Problem (meta-CSP). The variables of the meta-CSP are the edges $e_{i,j}$ of G . Their number depends on the density of the temporal graph. The domain of a variable $e_{i,j}$ is its label, $I_{i,j} = \{l_{ij}^{(1)}, l_{ij}^{(2)}, \dots, l_{ij}^{(k)}\}$. A partial solution is a set $\{(e_{i,j}, l_{ij}^{(h)})\}$ of variable-value pairs (vvps) that form a consistent STP, which is a global constraint. A complete solution is a consistent STP in which all the edges of G appear. The minimal network of the TCSP is the union of the minimal networks of *all* complete solutions, and solving the TCSP requires finding all the solutions of the meta-CSP. Each node in the tree generated by BT-TCSP is an STP P' that has E' edges, a subset of the edges of the original network ($E' \subseteq E$), each labeled with a unique interval from its domain. When P' is consistent, the node is expanded by adding to P' an edge from $(E - E')$ labeled with an interval from its domain. This yields a new STP that is checked again for consistency. Figure 4.2 illustrates the tree corresponding to the example of Figure 4.1, where edges are considered in their lexicographical order.

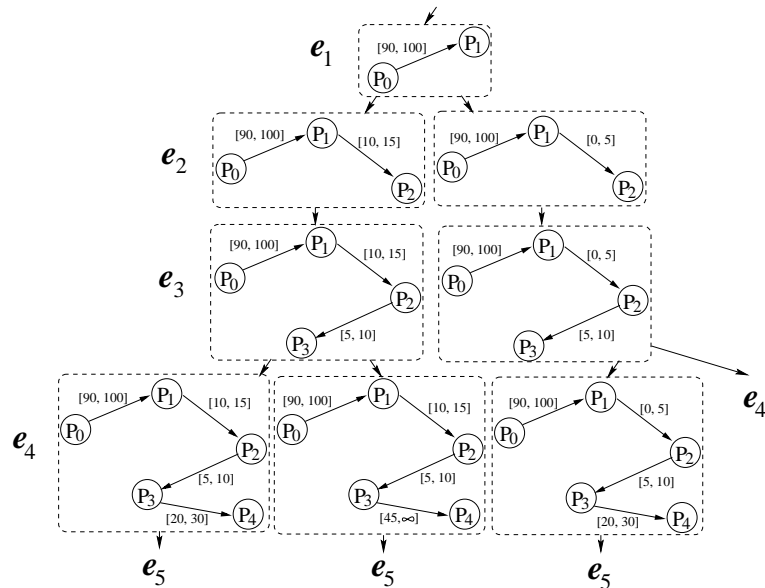


Figure 4.2: The search tree for the example of Figure 4.1.

In this chapter, we combine the following techniques to improve the performance of BT-TCSP, and demonstrate their effectiveness on randomly generated problems:

1. Every node in the tree is an STP that needs to be solved before the search can proceed. Hence, the performance of a TCSP solver depends critically on that of the STP solver. We compare for the first time the performance of various known STP solvers, including a new one, Δ STP, that we proposed in previous chapter. We show that it outperforms all others. Note that the performance of the STP solver does not affect the number of nodes visited in BT-TCSP.
2. One well-known technique to improve the performance of a CSP is to decompose it into sub-problems using its articulation points [17, 19, 15], and to solve the sub-problems independently. We provide for the first time an empirical evaluation of the effectiveness of this technique.
3. Further exploiting the topology of the temporal network, we show how to avoid running an STP-solver by checking the existence of new cycles (NewCyc) in the network as edges are added along a given path in the tree. For the example shown in Figure 4.2, the first four consistency checks are unnecessary because there are no cycles in the respective networks and the corresponding STPs are always consistent.
4. Another way to improve the performance of BT-TCSP is to find a good variable-ordering heuristic for the search. This corresponds to a sequencing of E , the edges of G , as they are added along a given path in the tree. A good sequence reduces unnecessary backtracking and also the number of constraint checks. We introduce a new ordering heuristic (EdgeOrd) that exploits the adjacency of existing triangles in the graph to determine the ordering of their edges in the tree.
5. We reduce the domains of the variables of the meta-CSP by using the efficient filtering algorithm, Δ AC [36].

The contributions of this chapter can be summarized as follows:

1. A new technique for saving constraint checks (NewCyc) and a new ordering heuristic (EdgeOrd).
2. The combination of the above listed techniques (i.e., an STP-solver, AP, NewCyc, EdgeOrd, and $\triangle AC$) to solve the TCSP.
3. Empirical evaluation and analysis of the effectiveness of these techniques and their combinations to demonstrate their significance.

This chapter is structured as follows. Section 4.2 reviews the various STP-solvers we used. Section 4.3 discusses the three improvements that exploit the topology of the temporal network. Section 4.4 addresses a filtering algorithm, which can significantly reduce the size of meta-CSP. Section 4.5 describes our experiments and observations. Finally, a brief summary concludes this chapter.

4.2 Algorithms for solving the STP

TCSP is NP-hard and is solved with backtrack search. Every node expansion in the search tree needs to check the consistency of an STP. Thus a good STP solver is critical for solving the TCSP. We test the following STP solvers: Directed Path Consistency DPC [16], Partial Path Consistency DPC [6], and Triangle-STP $\triangle STP$ [35].

4.2.1 Solving the STP using Directional Path Consistency (DPC)

A basic algorithm to solve an STP is the Floyd-Warshall algorithm (F-W), which computes all-pairs shortest-paths in a distance graph [11]. F-W guarantees consistency, minimality, and decomposability and has a worst-case complexity of $\Theta(n^3)$. Montanari showed that F-W is a special case of the Path Consistency (PC) algorithm [25]. Dechter et al. propose the

Directed-Path Consistency (DPC) algorithm. This algorithm is never more costly than F-W, runs in $O(n^3)$, and can determine the consistency of an STP in $O(nW^*(d)^2)$, where $W^*(d)$ is the induced width of the graph along a given ordering d . DPC determines the consistency of the STP, but does not necessarily yield the minimal and decomposable network. Due to the fact that only the consistency of an STP matters during BT-TCSP, we use DPC instead of F-W because of its lower cost.

4.2.2 Solving the STP using Partial Path Consistency (PPC)

Bliek and Sam-Haroud introduced Partial Path-Consistency (PPC), an algorithm applicable to general CSPs (and not restricted to temporal networks) [6]. PPC works on a triangulated graph, unlike the PC algorithm which requires a complete graph. Further, Bliek and Sam-Haroud showed that when the constraints are *convex*, the PC algorithm (operating on the complete graph) and the PPC algorithm (operating on the triangulated graph) yield equivalent results: the same labeling for the edges common to both graphs and the minimality and decomposability of the STP. PPC never requires more constraint checks than PC, which is advantageous when the (triangulated) graph is sparse. This is particularly attractive in BT-TCSP, which requires solving an STP at each node.

PPC requires that the graph be triangulated, which may result in new edges being added to the graph. We triangulate the temporal network using the algorithm devised in [28]. We represent the new edges as universal constraints in the original constraint graph and set their label to $(-\infty, \infty)$.

In the tree generated by BT-TCSP, each node represents an STP whose graph adds exactly one edge to the graph of the parent of the node (and must be triangulated to be used by PPC). Assuming a static ordering in the tree, the total number of graphs that appear along any given complete path is exactly equal to the number of edges in the original problem. Further, all nodes at a given level of the search tree have the same graph (only

the edge labelings may vary). Thus, under static ordering, the number of possible graphs considered during the BT-TCSP process is exactly equal to the total number of edges in the temporal network.

We devise two methods for accessing the triangulations of the STPs need in given a static ordering, Figure 4.3. In the first method, *Plan A*, we pre-compute all the STPs needed

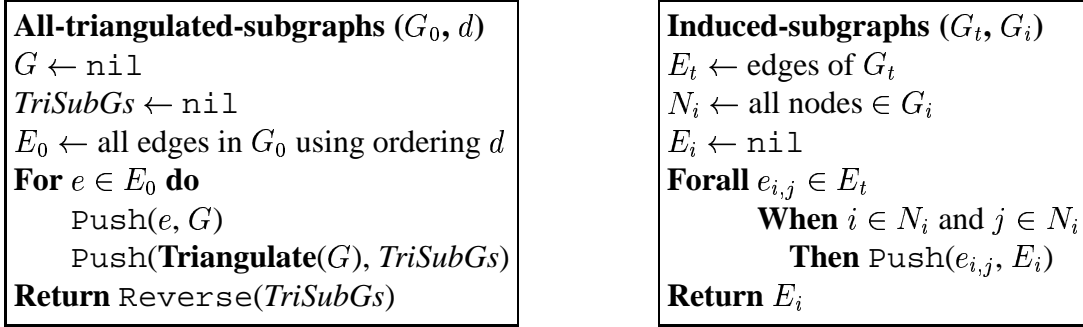


Figure 4.3: *Left*: List of triangulated subgraphs given an ordering. *Right*: Inducing a subgraph from the triangulated original graph.

in search, triangulate them, and store their triangulations for use during search. In the second method, *Plan B*, we triangulate the entire network only once. We, then induce, from the triangulated graph, the subgraph whose vertices form the STP under consideration. Since the original graph is triangulated, each induced subgraph is also triangulated.

- *Plan A*: Given a variable ordering d , the list of the graphs considered during BT-TCSP is generated as shown in Figure 4.3 (left). `Push` adds an item to a list, `Reverse` reverses a list, and `Triangulate` triangulates a graph. We use the i^{th} element of `TriSubGs` list as the triangulated subgraph for the node at the i^{th} level of the tree.
- *Plan B*: Here we compute the triangulated graph only once and induce from it the subgraph needed at every step. Figure 4.3 (right) shows the algorithm where G_t is the triangulated graph of the original network and G_i is the subgraph considered at level $1 \leq i \leq |E|$ in the search. Note that this technique may end up considering denser graphs than necessary, which increases the cost of solving the STP.

Our experimental results show that *Plan A* always outperforms *Plan B* in terms of the number of constraint checks and CPU time. Note that neither of these two plans affects the number of backtracks (the number of nodes visited) in BT-TCSP.

4.2.3 Δ STP algorithm used with TCSP algorithm

Δ STP algorithm can output the same minimal network as F-W and PPC. It uses the idea of triangulation and considers the temporal graph as composed of triangles instead of edges. Constraint propagation is ‘triangle-based’ rather than ‘edge-based.’ As a finer version of PPC, Δ STP can find the minimal network with less cost than F-W and PPC. When density is low, Δ STP is even cheaper than DPC, which does not guarantee the minimal network. Similar to PPC, the pre-requisite condition for Δ STP is to first triangulate the temporal graph. We have introduced above two plans to obtain triangulated subgraphs in the previous subsection. We will use *Plan A* for its lower cost in practice.

When solving a TCSP with search, the STP examined at each node in the search tree is a subgraph of the original TCSP. Thus the STPs we need to check always have lower density than the original TCSP, Since Thus the outstanding performance of Δ STP under low density makes it even more attractive to use for solving the TCSP.

4.3 Exploiting the topology of the constraint network

We propose three techniques topology-based techniques to enhance the performance of search. While the first technique is applied *prior* to search to decompose the problem into independent components, the last two are intertwined with the search process.

4.3.1 Decomposition using articulation points

The existence of articulation points in the graph of the temporal network can be used to decompose the network into its biconnected components, which can be solved independently. Finding the articulation points can be done in $O(|E|)$ [11]. This method provides an upper bound to the search effort in the size of the largest biconnected component [19]. It can effectively reduce the number of constraint checks in BT-TCSP and the number of nodes visited in its tree. A solution to the entire network is a combination of any of the solutions of the biconnected components. The total number of solutions is: $S = \prod_{i=1}^n s_i$, where s_i is the number of solutions for component i . This conjunctive decomposition of the temporal network [20] allows us to solve the sub-problems in parallel, as in a multi-agent system. Articulation points usually appear only when the density is low or when the TCSP has a special topology. Note that even in the absence of articulation points, we could ‘induce’ such decompositions by removing some edges of the graph, in a manner similar to the cycle-cutset method of Dechter and Pearl [14]. We have implemented the mechanism for finding and using existing articulation points but not yet explored how to induce their existence.

4.3.2 New cycle check (NewCyc)

The inconsistency of an STP is detected by the existence of a negative cycle in its distance graph. When the graph of an STP has no cycles, the STP is necessarily consistent².

Proposition 4.3.1. *A tree-structured constraint network is necessarily globally consistent.*

In BT-TCSP, nodes are expanded by adding one edge at a time. When the addition of a new edge does not yield a new cycle in the graph, a consistent STP remains consistent

²Note that is a stronger result than using the tree-structure of the constraint graph, which requires ensuring 2-consistency [18].

regardless of the labeling chosen for the new edge. We exploit this observation to save unnecessary consistency checks.

Corollary 1. *When the addition of an edge to a globally consistent STP yields no new cycles, the resulting STP is globally consistent.*

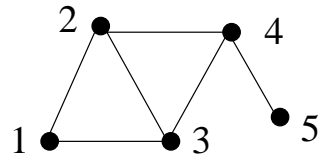


Figure 4.4: Simple constraint graph.

Consider the example of Figure 4.4. Suppose that search adopts the following ordering of the edges: $e_{1,2}$, $e_{2,3}$, $e_{1,3}$, $e_{3,4}$, $e_{2,4}$, and $e_{4,5}$. Figure 4.5 shows the configurations of the STPs checked for consistency at each level in the search.

Search level	1	2	3	4	5	6	Total
STP							
Checking strategy							
Always	✓	✓	✓	✓	✓	✓	6
NewCyc	×	×	✓	×	✓	×	2

Figure 4.5: Comparison of STP checks using different the new-cycle check heuristic.

Along a given path, as the tree generated by search is being explored in a depth-first manner, two strategies can be adopted at a given level: (1) Always check the STP for consistency, and (2) check the consistency of the STP only when a new cycle has been added to the network. At levels 1 and 2, no cycles exist in the graph, and the STP is necessarily consistent, Figure 4.5. At levels 4 and 6, no new cycles have been added to the graph of levels 3 and 5 respectively, and the corresponding STPs remain necessarily consistent regardless of their labeling. As illustrated above, checking for new cycles saves us unnecessary operations.

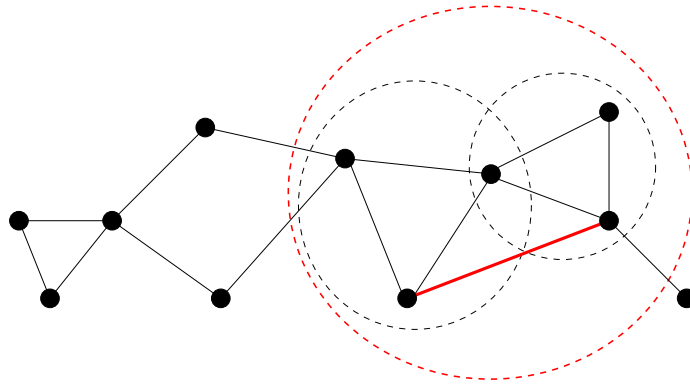


Figure 4.6: *Only check the consistency of the newly formed biconnected component.*

Further, when the addition of a new edge yields a new cycle, two biconnected components of the previous level are necessarily merged into a new biconnected component at the current level. We need to check *only* the consistency of the newly formed biconnected component (Figure 4.6), and we can safely ignore the rest of the temporal network. This allows us to localize the effort of consistency checking to the necessary part of the network.

Corollary 2. *When the addition of an edge to a globally consistent STP yields a new cycle, the resulting STP is globally consistent if and only if the newly formed biconnected component is a consistent STP.*

The application of this new heuristic, NewCyc, significantly enhances the performance of solving the meta-CSP with search. To apply it, we need to identify, between two levels of the search tree, (1) that a new cycle has been introduced and (2) the two biconnected components that were merged as a result. This is done by running the $O(|E|)$ algorithm for finding articulation points at each level, checking whether the number of biconnected components was reduced between two levels, and identifying the component to be checked as that containing the new edge.

4.3.3 Ordering heuristic for the meta-CSP

Variable ordering is an effective heuristic for improving the performance of search. In general, it is governed by the ‘fail first principle.’ The shallower the node pruned in the tree, the larger the pruned subtree, and the larger the cost savings. For the meta-CSP, a node is pruned when it corresponds to an inconsistent STP. Thus, the ordering of the edges (which are the variables of the meta-CSP) affects how quickly an inconsistent STP is found and also the effectiveness of constraint propagation in the STP.

As stated in Corollary 1, along a given path, no inconsistency may occur between one level and the next unless at least one new cycle is formed in the temporal graph. Consequently, a reasonable ordering heuristic is to first consider those edges that form triangles with edges existing in the STP. This may allow us to uncover inconsistencies as early as possible. It also increases the effectiveness of backtracking, because it is more likely to undo an inconsistency by changing the labeling of an edge in the same triangle as the one that yielded the inconsistency than that of a random edge. Our new edge-ordering heuristic orders the edges of the temporal graph in such a way that the network is expanded triangle by triangle ‘around’ the existing edges. The algorithm, given in Figure 4.7, returns the list of edges in the order to be used by the search. It uses basic operations on lists. `Append` concatenates two lists in the order provided. `Pop` removes and returns the first item in a list. It requires that each edge be associated with the number of triangles in which it appears in G , which is bounded by $(n - 1)$, where n is the number of nodes in G (i.e., the time points). We obtain these numbers as a by-product of the implementation of the triangulation algorithm.

Based on the topology of the network, we choose the edge that participates in the largest number of triangles and schedule the edges of those triangles for a priority instantiation during the search. Figure 4.8 illustrates the first steps of the application of the algorithm starting from edge I. First, the triangles in which edge I participates are explored. From


```

EdgeOrd ( $G$ )
 $E_0 \leftarrow$  all edges of  $G$ 
 $E \leftarrow \text{nil}$ 
While  $E_0$  do
   $e_{i,j} \leftarrow$  Edge of  $E_0$  appearing in the largest number of triangles in  $E_0$ 
   $E \leftarrow \text{Append}(E, \{e_{i,j}\})$ 
   $Q \leftarrow \text{nil}$ 
  While  $e_{i,j}$  do
    Forall  $k$  such that  $ijk$  is a subgraph of  $G$  do
       $Q \leftarrow \text{Append}(Q, \{e_{i,k}, e_{j,k}\})$ ,  $E \leftarrow \text{Append}(E, \{e_{i,k}, e_{j,k}\})$ 
       $E_0 \leftarrow E_0 \setminus \{e_{i,j}, e_{i,k}, e_{j,k}\}$ ,  $e_{i,j} \leftarrow \text{Pop}(Q)$ 
Return  $E$ 

```

Figure 4.7: *Edge ordering heuristic.*

there, we reapply iteratively the same process to each of the edges explored, i.e. edges II, III, and IV, gradually covering all the edges in the biconnected component. The modification of the label of any these edges propagates through these triangles. Thus, inconsistencies and deadends are likely to be more quickly detected during search, and backtrack remains locally contained (Figure 4.9).

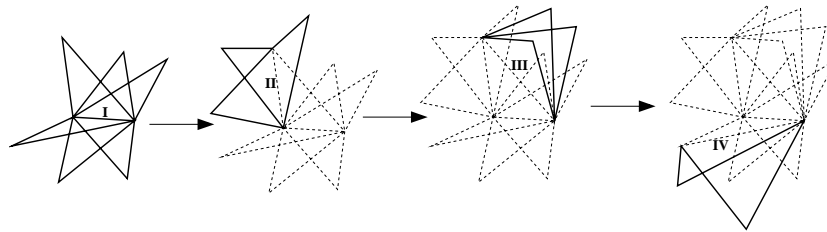


Figure 4.8: *Illustrating the exploration of the edges of a graph by the edge ordering heuristic.*

We can show that this process stops when all the edges in the biconnected component have been visited. Then EdgeOrd restarts from an unvisited edge from the original graph and repeats the process until all edges of the original network have been visited. The function returns a list in which the edges that are in a given biconnected component appear in sequence.

As a result, this ordering heuristic implicitly enables search to examine the biconnected

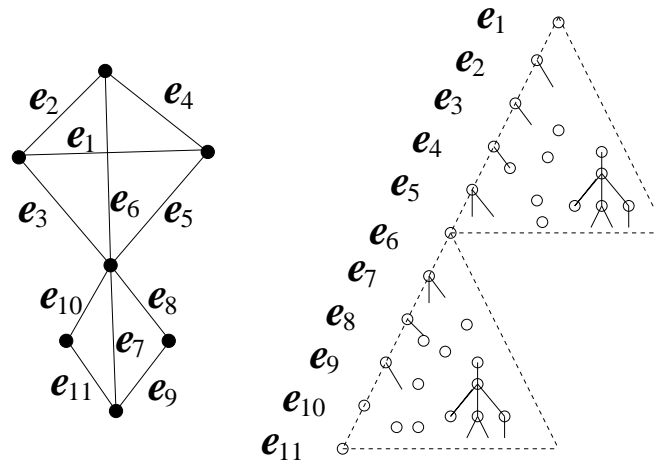


Figure 4.9: *EdgeOrd localize backtracking.*

components of the graph in isolation, and thus decompose the graph automatically. The advantages of this mechanism are:

1. *Localized backtracking*: This heuristic is based on the topology of the temporal graph. Neighboring levels in the search tree are likely to be physically related. When it encounters a deadend, search will backtrack to an edge that is more likely the culprit than another edge taken randomly from the graph.
2. *Automatic decomposition of the graph into its biconnected components*: The decomposition of the graph into its biconnected components is an effective technique to bind the search effort and enhance the performance of solving a TCSP. This ordering heuristic implicitly guarantees that articulation points in the graph (if any), are exploited, as if the network was decomposed into its biconnected components without using the special algorithm necessary for this purpose.

4.4 The label filtering algorithm

When solving a CSP, it is common to run a domain filtering mechanism (such as arc-consistency, AC) as a preprocessing step to search, and to interleave search with a looka-

head strategy (such as forward-checking, FC [21]). The goal of an AC algorithm is to reduce the domain of the variables, thus reducing the size of the CSP and that of the search tree to be explored. Arc-consistency is usually easy to achieve in polynomial time. Quite a few general arc-consistency algorithms exist, such as AC-3 [22], AC-4 [23], AC-6 [4], AC-7 [3], AC-3.1 [38], and AC-2001 [5].

The backtrack search on the meta-CSP requires solving an STP at every node in the search. Its complexity is thus $O(n^3 k^{|E|})$ [13]. Given the definition of the unique global constraint, running a generalized arc-consistency algorithm [24] on the meta-CSP is prohibitively expensive.

Proposition 4.4.1. *Generalized arc-consistency on the meta-CSP is NP-hard.*

Proof: The only constraint in the meta-CSP is a global constraint. Its allowed tuples are all consistent STPs that are solutions to the meta-CSP. Finding its definition to enforce generalized arc-consistency is thus equivalent to solving the meta-CSP, which is NP-hard [13]. □

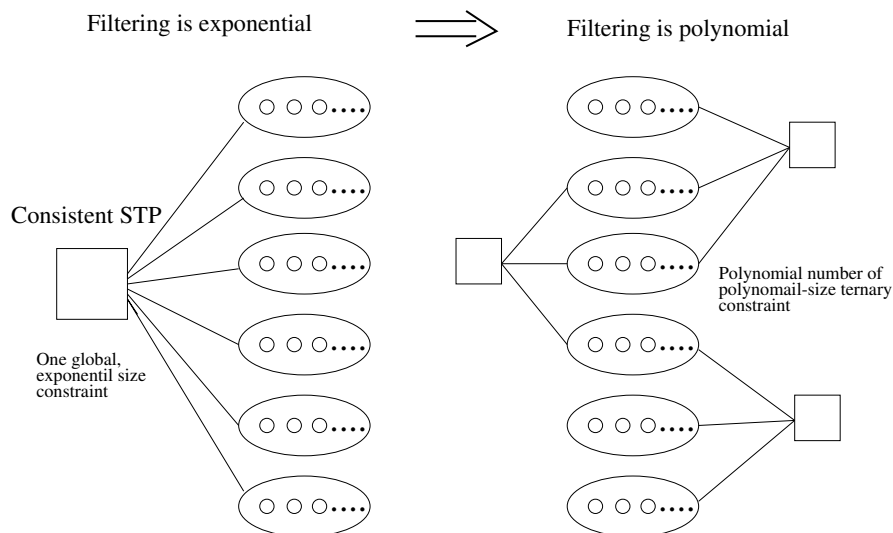


Figure 4.10: Replacing the global constraint with a polynomial number of ternary constraints.

We propose to approximate this problem by replacing the exponential-size global con-

straint in the meta-CSP with a polynomial number of polynomial-size ternary constraints Figure 4.10. We define an efficient generalized arc-consistency algorithm specialized for these ternary constraints, which we call Δ AC. The complexity of Δ AC is $O(\text{degree}(G) \times |E| \times k^3) = O(n|E|k^3)$, resulting in an approximation of the generalized arc-consistency of the meta-CSP. We report the performance improvement of the backtrack search for solving the meta-CSP with and without this preprocessing in terms of CPU time and number of constraint checks CC.

To the best of our knowledge, the only other work reported in the literature on applying consistency algorithms to the meta-CSP is a study by Schwalb and Dechter [29, 13]. They attempt to apply a path consistency algorithm (PC) to the labels of variables of the meta-CSP. Given the disjunctive intervals, this closure algorithm causes a fragmentation problem, which increases the number of intervals per label and makes the resulting meta-CSP even harder to solve by a search algorithm. To avoid this fragmentation problem, Schwalb and Dechter introduced the Upper-Lower Tightening algorithm (ULT) [29]. ULT computes looser networks than those resulting from enforcing full path-consistency, but results in the same upper and lower bounds as PC.

Our approach is neither like path-consistency nor like ULT. We consider each interval as an independent value in the domain of a variable. Our goal is to remove inconsistent individual intervals from the labels, not to tighten these intervals, which may not terminate in the general case and is prohibitively expensive in the integral case.

We reformulate the meta-CSP by replacing its unique global constraint with a ternary constraint $\Delta[e_{i,j}, e_{i,k}, e_{j,k}]$ among every variable $e_{i,j}$, $e_{i,k}$, and $e_{j,k}$ of the meta-CSP that forms an *existing* triangle in the temporal network G . Note that we do not triangulate the temporal network, nor do we make it a complete graph. Below, we define the Δ arc-consistency property as the generalized arc-consistency of this constraint and describe the Δ AC algorithm to achieve it.

4.4.1 Δ arc-consistency

For each triangle ijk in the original temporal network we define a ternary constraint in the meta-CSP $\Delta[e_{i,j}, e_{i,k}, e_{j,k}]$. Given three variable-value pairs $(e_{i,j}, l_{ij})$, $(e_{i,k}, l_{ik})$, and $(e_{j,k}, l_{jk})$ of the meta-CSP, with $i \neq j \neq k$, we say that the labeled triangle $\Delta[(e_{i,j}, l_{ij}), (e_{i,k}, l_{ik}), (e_{j,k}, l_{jk})]$ is a consistent triangle if and only if $(l_{ij} \circ l_{jk}) \cap l_{ik} \neq \emptyset$. Figure 4.11 shows a consistent triangle $\Delta[(e_{i,j}, [3, 5]), (e_{i,k}, [4, 9]), (e_{j,k}, [2, 6])]$. We also say that each

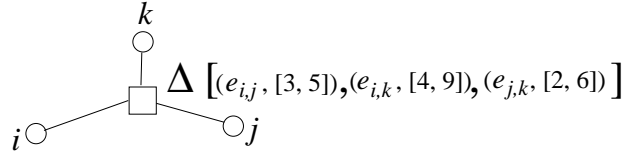


Figure 4.11: A consistent triangle.

variable-value pair in the triangle is supported by the two other variable-value pairs. We introduce the following definitions:

- The ternary constraint $\Delta[e_{i,j}, e_{i,k}, e_{j,k}]$ is Δ AC relative to the meta-CSP variable $e_{i,j}$ if and only if for every interval $l_{ij}^{(x)} \in \text{Domain}(e_{i,j})$ there exist an interval $l_{ik}^{(y)} \in \text{Domain}(e_{i,k})$ and an interval $l_{kj}^{(z)} \in \text{Domain}(e_{k,j})$ such that $(l_{ik}^{(y)} \circ l_{kj}^{(z)}) \cap l_{ij}^{(x)} \neq \emptyset$.
- The ternary constraint $\Delta[e_{i,j}, e_{i,k}, e_{j,k}]$ is Δ AC if and only if it is Δ AC relative to the variables $e_{i,j}$, $e_{i,k}$, and $e_{j,k}$.
- Finally, the meta-CSP is Δ AC if and only if all its ternary constraints are Δ AC.

We identify all the existing triangles in the temporal network and replace each of them by a ternary triangle constraint. The number of these new constraints is in $O(|E| \text{degree}(G)) = O(|E|n)$. The size of each constraint is at most k^3 . Note that we do not add any edges to the temporal network to make it a complete graph or to triangulate it.

4.4.2 Δ AC algorithm

The Δ AC algorithm, shown in Figure 4.14, removes the intervals in the domain of an $e_{i,j}$ that do not have a support in any triangle in which $e_{i,j}$ appears in the temporal graph. It implements mechanisms for consistency checking that are reminiscent of AC-4 [23] and AC-2001 [5] in that it tries to optimize the effort for consistency checking. It uses the procedures `First-support` of Figure 4.12 and `Initialize-support` of Figure 4.13. The `Push` and `Delete` operations we use are destructive stack operations.

```

First-support( $\langle (e_{i,j}, l_{ij}), ijk \rangle$ )
 $t_{ijk} \leftarrow \Delta[(e_{i,j}, l_{ij}), (e_{i,k}, l_{ik}^r), (e_{j,k}, l_{jk}^s)] \leftarrow \text{Supported-by}(\langle (e_{i,j}, l_{ij}), ijk \rangle)$ 
Unless  $t_{ijk}$  Then  $r \leftarrow 1, s \leftarrow 0$ 
For  $m$  from  $(s + 1)$  to  $|\text{Domain}(e_{j,k})|$ 
    Unless  $(l_{ik}^r \circ l_{jk}^m) \cap l_{ij} = \text{nil}$  Return  $\Delta[(e_{i,j}, l_{ij}), (e_{i,k}, l_{ik}^r), (e_{j,k}, l_{jk}^m)]$ 
If  $r = |\text{Domain}(e_{i,k})|$ 
    Then Return nil
    Else For  $n$  from  $(r + 1)$  to  $|\text{Domain}(e_{i,k})|$ 
        For  $t$  from 1 to  $|\text{Domain}(e_{j,k})|$ 
            Unless  $(l_{ik}^n \circ l_{jk}^t) \cap l_{ij} = \text{nil}$  Return  $\Delta[(e_{i,j}, l_{ij}), (e_{i,k}, l_{ik}^n), (e_{j,k}, l_{jk}^t)]$ 
Return nil

```

Figure 4.12: `First-support`.

It operates by looking at every combination of a vvp $(e_{i,j}, l_{ij})$ and the triangles ijk in which it appears, denoted $\langle (e_{i,j}, l_{ij}), ijk \rangle$. The support of $\langle (e_{i,j}, l_{ij}), ijk \rangle$ is the first element in the domains of $e_{i,k}$ and $e_{j,k}$ that yields a consistent triangle (Note that domains and variables are ordered canonically). Intervals in the domain of a variable that are not supported in any triangle are removed from the domain. When an interval is removed, some vvps may lose their support. Δ AC tries to find the next acceptable support. The process is repeated until all vvps have a valid support in every relevant triangle.

We use a hash-table `Supported-by` to keep track of the support of each vvp $(e_{i,j}, l_{ij})$ in a triangle ijk . A key in this hash-table is a tuple $\langle (e_{i,j}, l_{ij}), ijk \rangle$; its value is a consistent

```

Initialize-support( $G$ )
Support-by, Supports: two empty hash-tables
 $Q \leftarrow \{(e_{i,j}, l_{ij})\}$ , set of all vvp in the meta-CSP
 $Q' \leftarrow \text{nil}$ , Consistency  $\leftarrow t$ 
While  $Q \wedge \text{Consistency}$  do
   $(e_{i,j}, l_{ij}) \leftarrow \text{Pop}(Q)$ 
  Forall  $k$  such that  $ijk$  is a subgraph of  $G$  do
     $t_{ijk} \leftarrow \Delta[(e_{i,j}, l_{ij}), (e_{i,k}, l_{ik}), (e_{j,k}, l_{jk})] \leftarrow \text{First-support}(\langle (e_{i,j}, l_{ij}), ijk \rangle)$ 
    If  $t_{ijk}$ , Then Supported-by $[(e_{i,j}, l_{ij}), ijk] \leftarrow t_{ijk}$ 
      Push( $\langle e_{i,j}, l_{ij}, ijk \rangle$ , Supports $[(e_{i,k}, l_{ik})]$ )
      Push( $\langle e_{i,j}, l_{ij}, ijk \rangle$ , Supports $[(e_{j,k}, l_{jk})]$ )
    Else Domain $(e_{i,j}) \leftarrow \text{Domain}(e_{i,j}) \setminus \{l_{ij}\}$ 
      Push( $(e_{i,j}, l_{ij})$ ,  $Q'$ )
      Unless Domain $(e_{i,j})$  Then Consistency  $\leftarrow f$ 
Return  $Q'$ , Supported-by, Supports, Consistency

```

Figure 4.13: Initialize-support.

triangle $\Delta[(e_{i,j}, l_{ij}), (e_{i,k}, l_{ik}), (e_{j,k}, l_{jk})]$. The size of Supported-by is $O(|E|k \text{ degree}(G))$.

We also use a hash-table Supports to keep track of what a given vvp supports in Supported-by. The key is a vvp $(e_{i,j}, l_{ij})$ and the value is a list of the keys of Supported-by that this vvp supports.

The procedure Initialize-support shows how these data-structures are initialized. By construction, Supports has $O(|E|k)$ keys and a total of $O(|E|k \text{ degree}(G))$ elements.

In addition to these hash-tables, Initialize-support returns the list Q' of vvps deleted from the domains of the meta-CSP at the initialization step. ΔAC , shown in Figure 4.14, iterates over the vvps that have been deleted and retracts them from supporting entries in Supported-by.

We can prove that ΔAC terminates, does not remove any consistent intervals (i.e., is sound), and is in $O(\text{degree}(G)|E|k^3) = O(n|E|k^3)$. We can further improve its performance and reduce the number of constraint checks by exploiting the convexity property of

```

 $\Delta AC(G)$ 
 $Q, \text{Supported-by}, \text{Supports}, \text{Consistency} \leftarrow \text{Initialize-support}(G)$ 
While  $Q \wedge \text{Consistency}$  do
   $(e_{i,k}, l_{ik}) \leftarrow \text{Pop}(Q)$ 
  Forall each  $\langle e_{i,j}, l_{ij}, ijk \rangle \in \text{Supports}[(e_{i,k}, l_{ik})]$ 
     $t_{ijk} \leftarrow \Delta[(e_{i,j}, l_{ij}), (e_{i,k}, l_{ik}), (e_{j,k}, l_{jk})] \leftarrow \text{Supported-by}(\langle (e_{i,j}, l_{ij}), ijk \rangle)$ 
    Delete( $\langle (e_{i,j}, l_{ij}), ijk \rangle, \text{Supports}[(e_{i,k}, l_{ik})]$ )
    Delete( $\langle (e_{i,j}, l_{ij}), ijk \rangle, \text{Supports}[(e_{j,k}, l_{jk})]$ )
     $t'_{ijk} \leftarrow \Delta[(e_{i,j}, l_{ij}), (e_{i,k}, l'_{ik}), (e_{j,k}, l'_{jk})] \leftarrow \text{First-support}(\langle (e_{i,j}, l_{ij}), ijk \rangle)$ 
    If  $t'_{ijk}$ 
      Then  $\text{Supported-by}[(e_{i,j}, l_{ij}), ijk] \leftarrow t'_{ijk}$ 
      Push( $\langle e_{i,j}, l_{ij}, ijk \rangle, \text{Supports}[(e_{i,k}, l'_{ik})]$ )
      Push( $\langle e_{i,j}, l_{ij}, ijk \rangle, \text{Supports}[(e_{j,k}, l'_{jk})]$ )
    Else  $\text{Domain}(e_{i,j}) \leftarrow \text{Domain}(e_{i,j}) \setminus \{l_{ij}\}$ 
      Push( $\langle e_{i,j}, l_{ij} \rangle, Q$ )
      Unless  $\text{Domain}(e_{i,j})$  Then  $\text{Consistency} \leftarrow f$ 
Return  $\{\text{Domain}(e_{i,j})\}$ 

```

Figure 4.14: ΔAC .

interval intersection, which we suspect may result in an optimal algorithm.

4.5 Experimental results

Figure 4.15 shows the TCSP solvers we tested, with and without pre-processing by ΔAC .

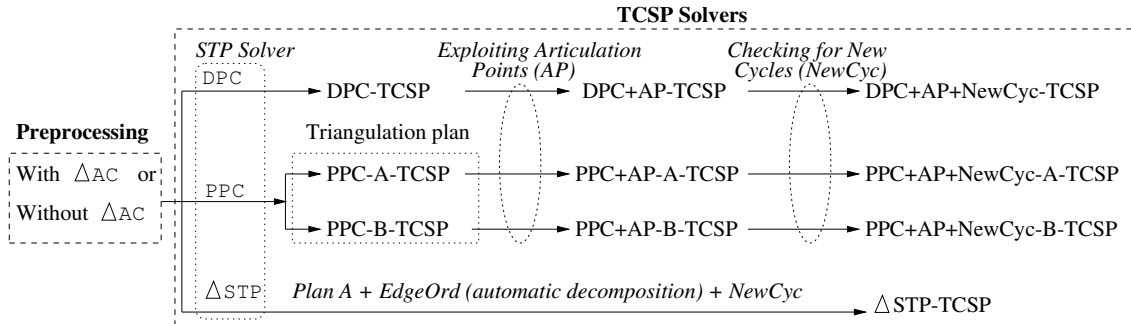


Figure 4.15: TCSP solvers tested.

The STP solvers we used are DPC, PPC, and ΔSTP of Section 4.2. We combined them

with the techniques proposed in Section 4.3 (i.e., AP, NewCyc, and EdgeOrd). We compared their performance in terms of the number of nodes visited NV, constraint checks CC, and CPU time. Note that all CPU time curves have almost exactly the same shapes as the CC curves. We carried out our tests on randomly generated, (guaranteed) connected problems. Our generator, GenTCSP-1, guarantees that at least 80% of these problems have at least one solution. The TCSP instances generated have the following characteristics: $n = 8$, k randomly chosen between 1 and 5, density of the temporal network ($d = \frac{|E| - |E_{min}|}{|E_{max}| - |E_{min}|}$) varies in $[0.02, 0.1]$ with a step of 0.02 and in $[0.2, 0.9]$ with a step of 0.1. The number of variables in the meta-CSP, for which we must find *all solutions*, varies from 7 to 26. The size of the meta-CSP varies on average between 1.6×10^5 and 5.2×10^{15} . We averaged the results of over 100 samples. The goal of our experiments was to study the *effects* on the various solvers of the improvements we proposed ³ (i.e., Δ STP, AP, NewCyc, EdgeOrd, Δ AC), and to establish their effectiveness. It is not our goal here to compare the performance of the various STP solvers, which is already discussed extensively in previous chapter.

Section 4.5.1 demonstrate the filtering power of Δ AC. In order to demonstrate the advantages of Δ AC, we report the cost of solving the meta-CSP with and without this preprocessing. We use the Directional Path-Consistency algorithm DPC also of Dechter [13] to check the consistency of STP at each node. Section 4.5.2 discusses the number of solutions of the problems tested. Naturally, all solvers must find the same solutions. Counting the number of solutions was useful to confirm that all solvers were sound and that our implementation was bug-free. Section 4.5.3 shows the effect of our techniques on the shape of the tree by measuring the number of nodes visited. Section 4.5.4 shows the effect of our techniques on the various TCSP solvers (i.e., DPC, PPC, and Δ STP) on the number of constraint checks. In Sections 4.5.3 and 4.5.4 we also show how filtering the

³Note that although decomposition according to articulation points is a well-known technique, to the best of our knowledge, it has not been yet assessed experimentally.

meta-CSP with ΔAC dramatically improves the performance of search. The effect of this preprocessing is clearly visible in comparisons of the scale of the vertical axis of the charts without and after preprocessing.

The detailed data of the above experiments on the instances generated by GenTCSP-1 are shown in Table 4.5, Table 4.2, Table 4.3 and 4.4.

Table 4.1: Performance of ΔAC

Graph density	Number of variables in meta-CSP	Size of meta-CSP		Number of solutions	Cost of search without ΔAC		Cost of search with ΔAC		Cost of ΔAC	
		Original	Filtered		CPU [s]	CC	CPU [s]	CC	CPU [s]	CC
0.02	7	16701.67	16701.67	16701.67	13.6	518463.66	13.62	518463.66	5.00E-04	0
0.04	8	58448.44	40831.72	4176.91	21.6	843112.7	17.86	712777.75	0.0011	55.53
0.06	8	64780.24	48399.24	4837.69	25.03	965354.3	22.02	868557	0.0012	50.98
0.08	9	282427.3	142638.28	1437.01	24.23	1008288.4	18.14	782634.6	0.0022	122.7
0.1	9	271254.2	132758.27	1331.86	26.08	1103695.6	17.83	793677.7	0.0017	134.14
0.2	11	4257366	653949	105.88	23.95	1105540.5	6.43	335393.7	0.0033	324.44
0.3	13	6.81E+07	2424326.7	20.02	16.32	866010.3	2.1	117963.05	0.005	575.8
0.4	15	1.10E+09	1117395.5	5.97	22.13	1320010.5	0.49	29187.07	0.0075	880.23
0.5	18	6.64E+10	62.07	2.4	26.11	1630835.2	0.07	3654.7	0.0115	1383.8
0.6	20	1.06E+12	33.21	2.35	29.25	1932359.2	0.07	3821	0.015	1711.11
0.7	22	1.61E+13	31.16	2.19	34.87	2297002.5	0.077	3607.89	0.0192	2059.18
0.8	24	2.74E+14	2.41	1.66	57.13	3946315	0.07	3226.7	0.0217	2393.2
0.9	26	5.23E+15	2.48	1.6	74.39	5128653	0.08	3851.71	0.0262	2839.48

TCSP solver without ΔAC										
Density	BT based on DPC			BT based on PPC-A			BT based on PPC-B			BT based on ΔSTP
	AP	AP+NewCyc		AP	AP+NewCyc		AP	AP+NewCyc		NewCyc+EedeOrd
0.02	518463.66	0	0	0	0	0	0	0	0	0
0.04	843112.7	7098.57	5428.29	195478.37	4656.7	3185.11	130208.28	3193.86	3193.86	2732.05
0.06	965354.3	19478.22	15111.471	247673.87	13226.26	8798.33	170087.39	8770.99	8770.99	7070.2
0.08	1008288.4	208692.31	104777.69	329940.2	91361.41	50101.957	206974.66	49796.168	51038.53	17415.29
0.1	1103695.6	146097.48	70758.62	335411.9	69567.13	37837.2	203333.9	37839.12	38099.03	13098.529
0.2	1105540.5	427509.1	132239.03	444156.22	200879.33	70834.13	172033.37	75767.01	67844.07	7640.5903
0.3	866010.3	612256.7	152573.14	419374.37	271919.87	85687.97	129761.72	87368.99	75188.18	13369.98
0.4	1320010.5	1140589.6	231152.44	629796.3	572063	128503.8	139582.05	126608.2	110898.97	13173.97
0.5	1630835.2	1623335.6	253523.25	881600.94	869830.9	140018.53	126249.19	124945.4	112451.02	11500.86
0.6	1932359.2	1932359.2	241397.17	1132821.4	1132821.4	145871.2	124646.92	124646.92	109810.49	13533.8
0.7	2297002.5	2297002.5	203337.03	1221336	1221336	125914.47	106470.74	106470.74	98323.805	11595.09
0.8	3946315	3946315	283957.37	2312516.7	2312516.7	171134.44	136861.39	136861.39	132065.2	13617.9
0.9	5128653	5128653	264101.9	3125162.5	3125162.5	170470.61	142386.44	142386.44	139375.42	13482.91
TCSP solver after ΔAC										
0.02	518463.66	0	0	0	0	0	0	0	0	0
0.04	712777.75	7042.8496	5391.87	180705.23	4627.27	3166.9	122173.29	3175.65	3175.65	2681.83
0.06	868557	19429.16	15079.471	237374.4	13200.36	8782.33	164291.92	8754.99	8754.99	7030.8496
0.08	782634.6	199238.55	100803.305	259604.69	86338.79	47635.937	165931.56	47328.24	48526.64	16875.111
0.1	793677.7	131938.3	64213.543	254822.17	61250.344	34128.92	159538.44	34071.95	34296.9	13686.17
0.2	335393.7	171026.39	59681.01	129256.44	75406.72	31419.69	61471.055	34251.39	29890.4	9583.06
0.3	117963.05	100633.83	39738.96	72655.13	63498.027	26085.182	26988.21	23354.92	20417.04	4034.39
0.4	29187.068	28307.87	10054.83	20862.37	20433.33	6122.45	5851.99	5572.18	4908.52	2710.5
0.5	3654.7	3614.67	2112.71	2872.62	2838.98	1828.04	1384.58	1371.86	1308.73	796.66
0.6	3821	3821	2569.35	3523.51	3523.51	2431.35	1756.16	1756.16	1719.35	1100.83
0.7	3607.89	3607.89	2651.43	3677.83	3677.83	2834.49	1906.57	1906.57	1886.31	1451.6
0.8	3226.7	3226.7	2419.56	3603.56	3603.56	2802.25	1740.82	1740.82	1730.42	1385.8
0.9	3851.71	3851.71	2962.27	4588.59	4588.59	3635.96	2101.98	2101.98	2096.71	1692.36

Table 4.2: The number of constraint checks for different TCSP solvers.

TCSP solver without ΔAC										
Density	BT based on \hat{DPC}			BT based on $PPC-A$			BT based on $PPC-B$			BT based on ΔSTP
	AP	AP+NewCyc		AP	AP+NewCyc		AP	AP+NewCyc		NewCyc+EdgeOrd
0.02	13.6012	0.0012	0.0016	2.9475	8.00E-04	0.0011	2.0418	0.0018	0.002	7.00E-04
0.04	21.5972	0.1454	0.117	8.0523	0.1134	0.089	5.2475	0.0777	0.0743	0.1602
0.06	25.0329	0.4376	0.3413	9.2482	0.3014	0.2303	6.1463	0.1952	0.1888	0.4806
0.08	24.2254	4.3314	2.1852	10.5273	2.1741	1.4013	7.1445	1.2439	1.1838	1.2915
0.1	26.0802	2.9953	1.5577	11.0083	1.6644	1.0835	6.7494	0.9578	0.9003	0.9524
0.2	23.945	8.6494	2.7925	12.1999	5.1939	2.3822	6.2219	2.4927	2.1745	0.5818
0.3	16.3212	10.4973	3.0631	9.7721	6.2291	2.8293	4.3629	2.807	2.4388	1.0811
0.4	22.1312	19.3378	4.7679	13.598	12.2608	4.9896	5.2502	4.7373	4.0686	1.0972
0.5	26.1097	25.7714	5.2948	17.5251	17.4339	5.5902	5.5594	5.4671	4.8505	1.0095
0.6	29.2475	29.2092	5.2324	20.7785	20.7554	6.0247	6.2324	5.9301	5.196	1.2594
0.7	34.8727	34.9366	4.7775	22.7941	22.8446	6.013	5.9852	5.9829	5.4353	1.1629
0.8	57.1265	56.9518	6.8542	39.2782	39.1814	8.8453	9.0351	9.0753	8.0979	1.4266
0.9	74.385	74.5737	7.357	51.7004	52.061	10.4281	11.0689	11.0537	10.0133	1.4812
TCSP solver after ΔAC										
0.02	13.6168	0.0014	0.0013	2.9113	0.0012	0.0026	2.0563	0.0019	0.0015	0.001
0.04	17.8629	0.1464	0.1156	6.6389	0.1139	0.0975	4.281	0.0755	0.0719	0.1612
0.06	22.0242	0.4366	0.3489	8.2771	0.3011	0.2509	5.3853	0.192	0.1837	0.481
0.08	18.1424	4.1696	2.1301	7.7162	2.049	1.4432	4.7949	1.1597	1.1074	1.248
0.1	17.8349	2.6798	1.4	7.6339	1.4468	1.0273	4.7383	0.8468	0.7972	0.9899
0.2	6.4253	3.2855	1.1821	3.2472	1.8491	1.0474	1.7861	0.9873	0.8544	0.776
0.3	2.1025	1.8001	0.7688	1.4603	1.2832	0.735	0.6767	0.5931	0.5272	0.3348
0.4	0.4918	0.479	0.1925	0.3799	0.3713	0.1869	0.1633	0.157	0.1404	0.2406
0.5	0.0654	0.0655	0.0519	0.0562	0.0568	0.0498	0.0422	0.0402	0.0443	0.098
0.6	0.0692	0.0707	0.0621	0.065	0.0665	0.0611	0.0494	0.0491	0.0547	0.1335
0.7	0.0714	0.0727	0.069	0.0701	0.0702	0.0706	0.0564	0.0559	0.0633	0.1906
0.8	0.0655	0.0668	0.065	0.068	0.0689	0.0696	0.057	0.0551	0.0637	0.1765
0.9	0.0781	0.0796	0.077	0.0839	0.085	0.084	0.0681	0.066	0.073	0.228

Table 4.3: CPU time [s] for different TCSP solvers.

TCSP solver without ΔAC										
Density	BT based on DPC			BT based on PPC-A			BT based on PPC-B			BT based on ΔSTP
	AP	AP+NewCyc		AP	AP+NewCyc		AP	AP+NewCyc		NewCyc+EdgeOrd
0.02	22463.73	0	0	22463.73	0	0	22463.73	0	0	0
0.04	22987.63	302.41	302.41	22987.63	302.41	302.41	22987.63	302.41	302.41	318.83
0.06	25003.81	660.84	660.84	25003.81	660.84	660.84	25003.81	660.84	660.84	654.85
0.08	22990.65	3684.61	3684.61	22990.65	3684.61	3684.61	22990.65	3684.61	3684.61	1546.24
0.1	23925.88	3305.01	3309.58	23925.88	3305.01	3305.01	23925.88	3309.58	3309.58	1394.64
0.2	21555.33	8748.92	8749.22	21555.33	8748.92	8748.92	21555.33	8749.22	8749.22	757.95
0.3	14371.13	8992.471	8992.471	14371.13	8992.471	8992.471	14371.13	8992.471	8992.471	1065.07
0.4	17150.84	15490.259	15490.259	17150.84	15490.259	15490.259	17150.84	15490.259	15490.259	951.49
0.5	19171.459	18848.65	18848.65	19171.459	18848.65	18848.65	19171.459	18848.65	18848.65	744.15
0.6	20187.63	20187.63	20187.63	20187.63	20187.63	20187.63	20187.63	20187.63	20187.63	772.57
0.7	20756.48	20756.48	20756.48	20756.48	20756.48	20756.48	20756.48	20756.48	20756.48	643.15
0.8	32091.64	32091.64	32091.64	32091.64	32091.64	32091.64	32091.64	32091.64	32091.64	768.8
0.9	38826.61	38826.61	38826.61	38826.61	38826.61	38826.61	38826.61	38826.61	38826.61	589.11
TCSP solver after ΔAC										
0.02	22463.73	0	0	22463.73	0	0	22463.73	0	0	0
0.04	18735.102	294.05	294.05	18735.102	294.05	294.05	18735.102	294.05	294.05	312.96
0.06	22143.98	653.51	653.51	22143.98	653.51	653.51	22143.98	653.51	653.51	645.45
0.08	16569.23	3396.96	3396.96	16569.23	3396.96	3396.96	16569.23	3396.96	3396.96	1511.45
0.1	16875.42	2864.68	2864.68	16875.42	2864.68	2864.68	16875.42	2864.68	2864.68	1439.16
0.2	5981.71	3390.78	3390.78	5981.71	3390.78	3390.78	5981.71	3390.78	3390.78	854.75
0.3	2101.57	1782.61	1782.61	2101.57	1782.61	1782.61	2101.57	1782.61	1782.61	366.9
0.4	516.44	506.08	506.08	516.44	506.08	506.08	516.44	506.08	506.08	173.95
0.5	49.92	49.19	49.19	49.92	49.19	49.19	49.92	49.19	49.19	38.97
0.6	46.38	46.38	46.38	46.38	46.38	46.38	46.38	46.38	46.38	39.28
0.7	36.48	36.48	36.48	36.48	36.48	36.48	36.48	36.48	36.48	40
0.8	30.23	30.23	30.23	30.23	30.23	30.23	30.23	30.23	30.23	31.34
0.9	33.07	33.07	33.07	33.07	33.07	33.07	33.07	33.07	33.07	33.07

Table 4.4: The number of nodes visited for different TCSP solvers.

4.5.1 Power of Δ AC

The comparison of the average size of the meta-CSP before and after filtering is shown in Figure 4.16. It shows that Δ AC dramatically reduces the size of meta-CSP especially when density is high, which is typical of consistency filtering techniques used as a preprocessing step to search. More importantly, Figure 4.16 shows that the size of meta-CSP obtained after filtering by Δ AC is close to the number of solutions for high-density networks.

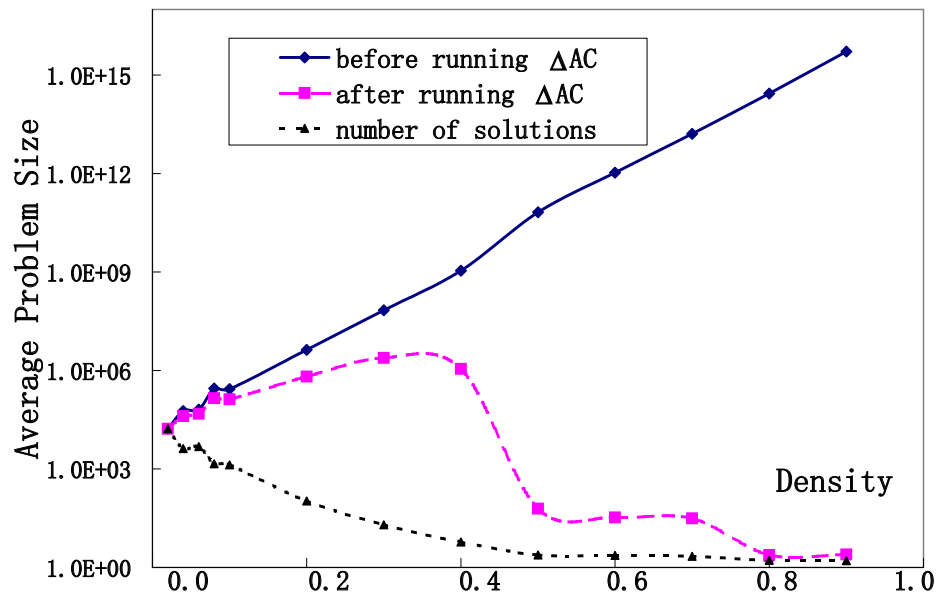


Figure 4.16: *Reduction of problem size of TCSP I.*

The results of solving the meta-CSP in terms of CPU time and constraint checks CC are shown in Figure 4.17 and Figure 4.18, and the numerical values are reported in Table 4.5. In this table, we also report the cost of running Δ AC although it is already included in the cost of search in order to demonstrate that the overhead due to filtering is practically negligible.

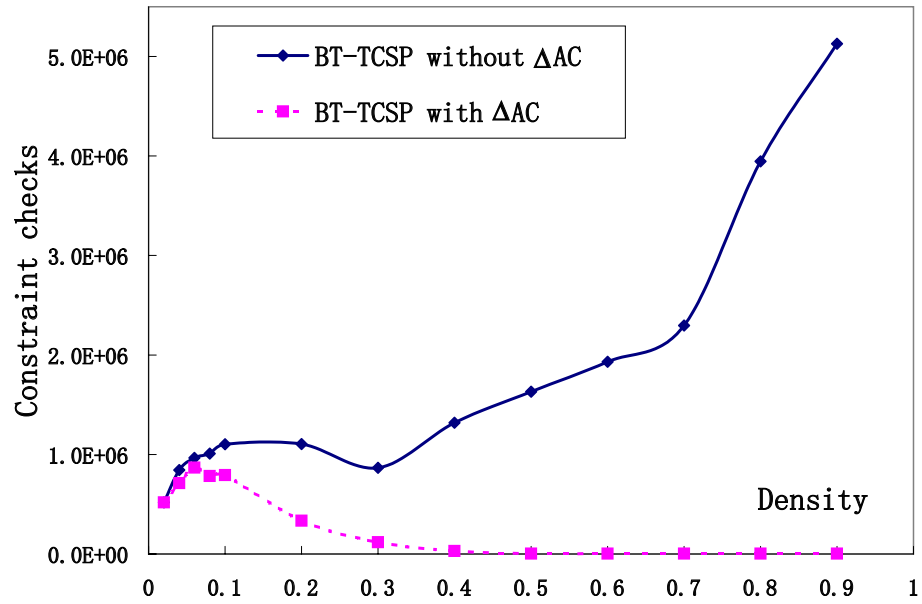


Figure 4.17: Constraint checks for solving TCSP.

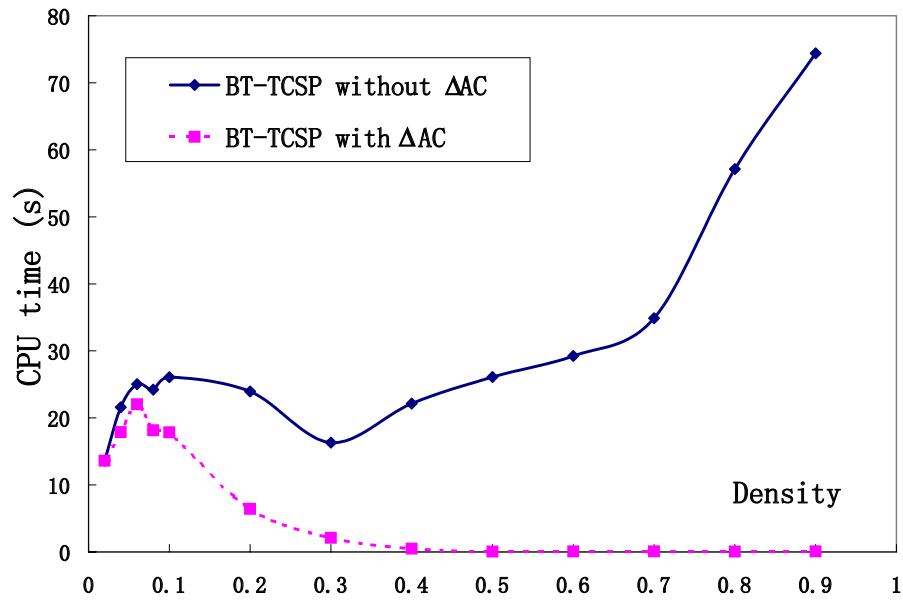


Figure 4.18: CPU time for solving TCSP I.

4.5.2 Solutions to the TCSP

When density is low, there are few constraints, any partial solution is likely to be extended to a global solution, and there are many solutions to the meta-CSP as is seen in Figure 4.19.

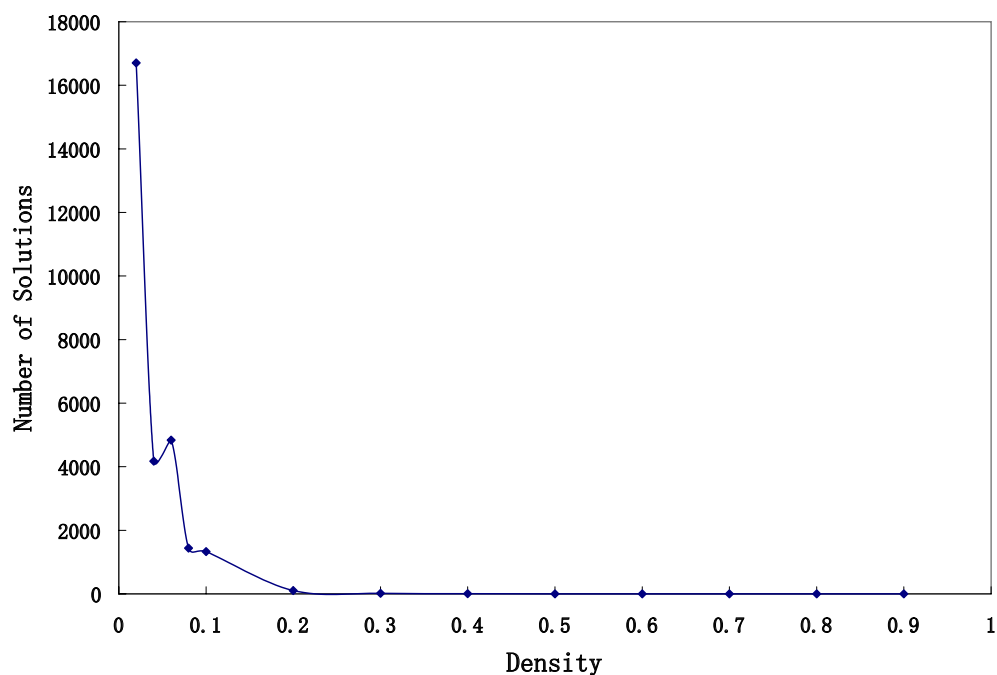


Figure 4.19: *The number of solution of TCSP.*

Indeed, under low density, the temporal network (which is guaranteed connected by construction) has almost no cycles. Thus, almost any combination of intervals in the label of the edges is a solution to the meta-CSP (see Proposition 4.3.1). The number of solutions quickly drops density. When $d=0.9$, there are only one or two solutions, one of which is guaranteed by construction.

4.5.3 Effects on the size of the search tree

The effects of AP and EdgeOrd on the ‘shape’ of the tree can be assessed by the number of nodes visited NV by search. They are shown in Figure 4.20.

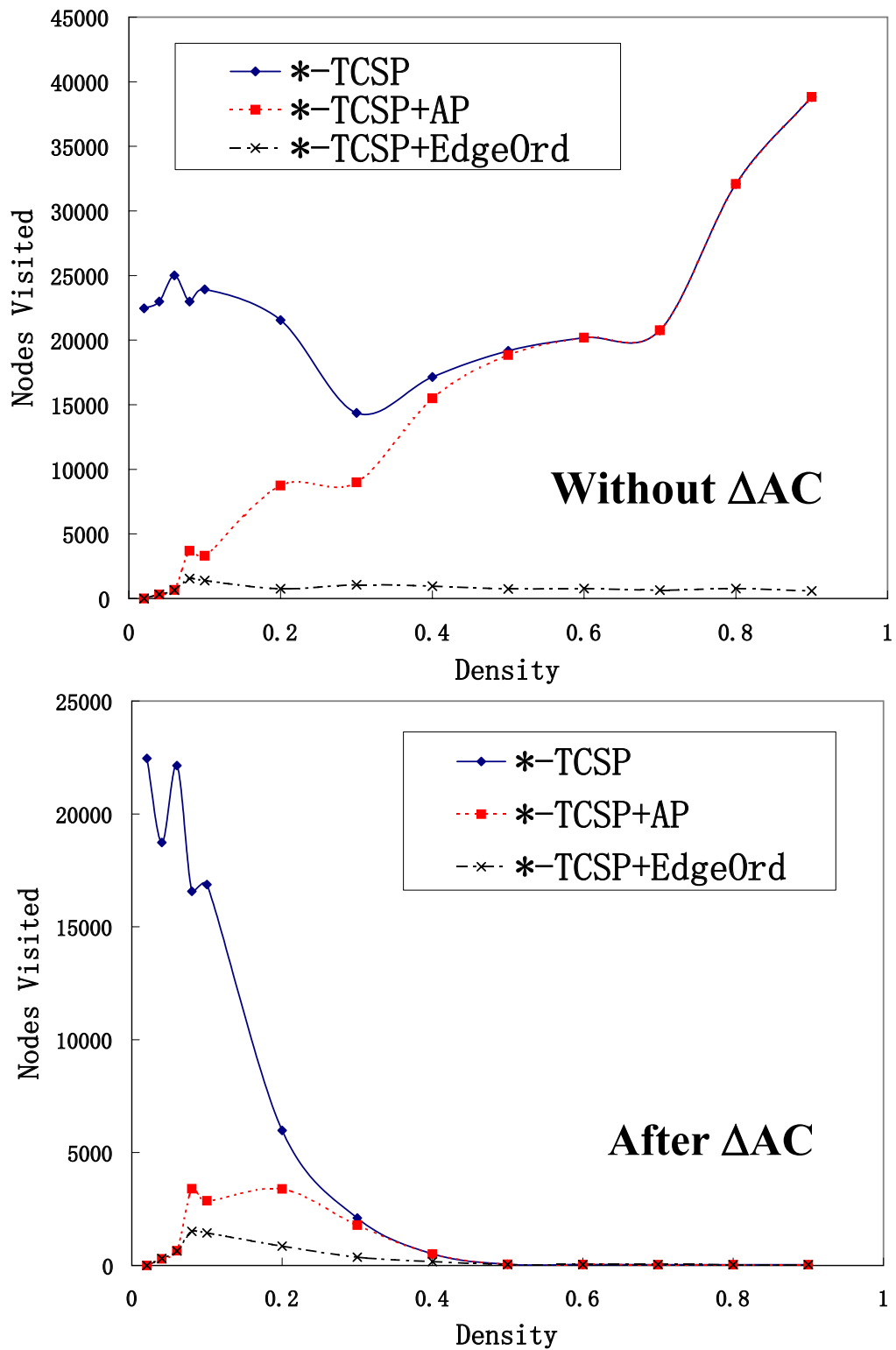


Figure 4.20: Nodes visited by BT-TCSP.

Note that the effects of NewCyc on the various STP solvers (i.e., DPC, PPC, and Δ STP) are irrelevant to this measurement. Indeed, they aim at reducing the cost of checking the consistency of the STP at a node in the tree once search has effectively reached the node. The ‘*’ in the legend of Figure 4.20 indicates that these results hold for all STP solvers tested. Figure 4.20 shows that AP reduces significantly NV when density is low. When density is high, almost no articulation point exists, hence AP does impact NV. The effect of EdgeOrd is quite dramatic across all values for density because it allows BT-TCSP to quickly identify dead-ends, as a good ordering heuristic is supposed to do. Moreover, we find that using Δ AC as a preprocessing step significantly reduce the number of nodes visited especially when density is high, and we start to notice the existence of a phase transition that appears around $d = 0.1$ and becomes increasingly visible as we move toward more effective TCSP solvers.

4.5.4 Effects on the number of constraints checks (same as CPU time)

Here we discuss the effects of our techniques on the various TCSP solvers: DPC, PPC, and Δ STP. We show the benefits of AP and NewCyc on DPC (Figure 4.21 and Figure 4.22). We show the benefits of AP, NewCyc on PPC for both *Plan A* (Figure 4.23 and Figure 4.24) and *Plan B* (Figure 4.25 and Figure 4.26). Finally, we show the benefits of EdgeOrd and NewCyc under *Plan A* on Δ STP (Figure 4.27 and Figure 4.28).

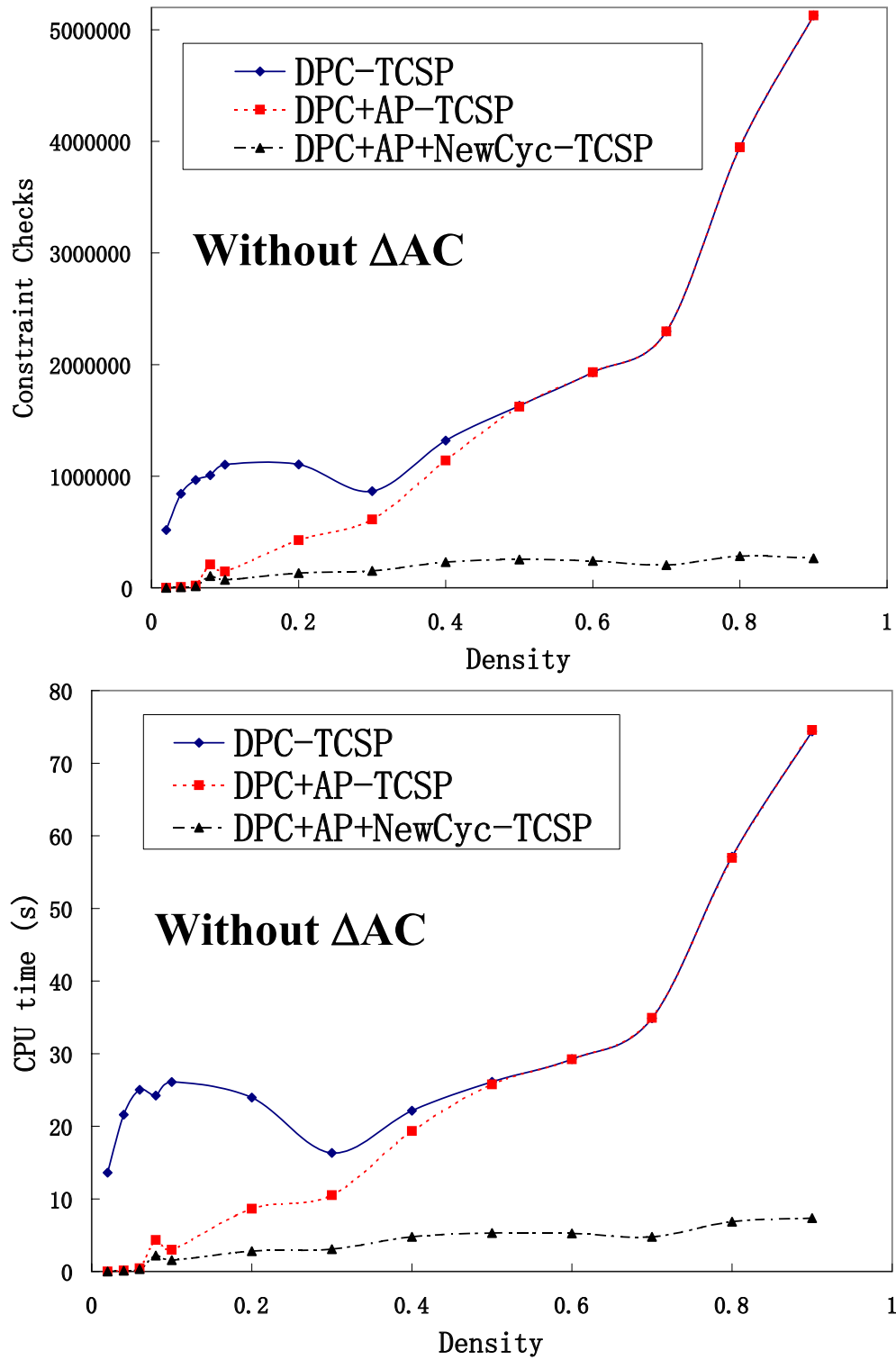


Figure 4.21: Constraint checks and CPU time for DPC-TCSP without ΔAC (Top: Constraint Checks; Bottom: CPU time [s])

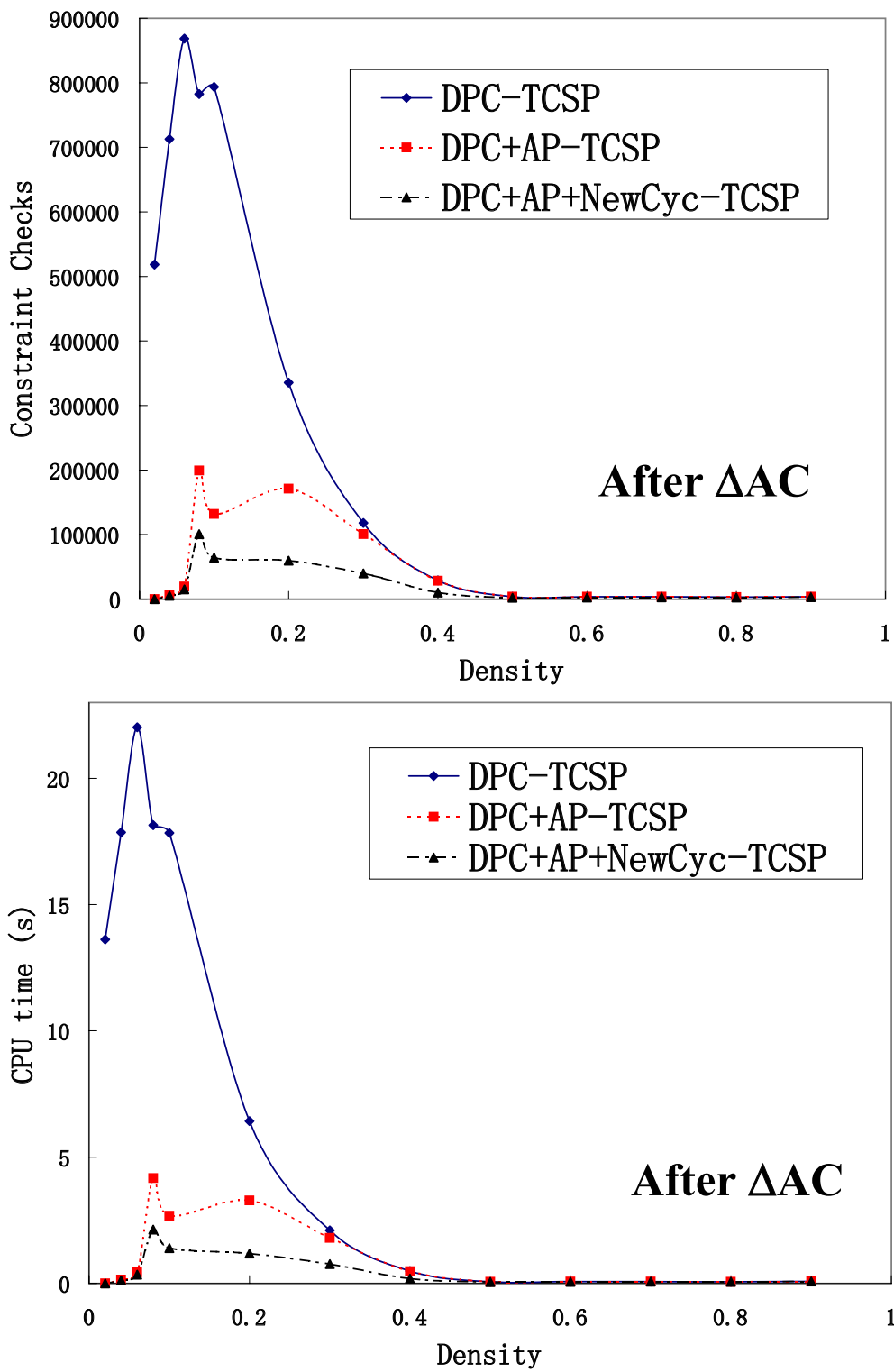


Figure 4.22: Constraint checks and CPU time for DPC-TCSP after ΔAC (Top: Constraint Checks; Bottom: CPU time [s])

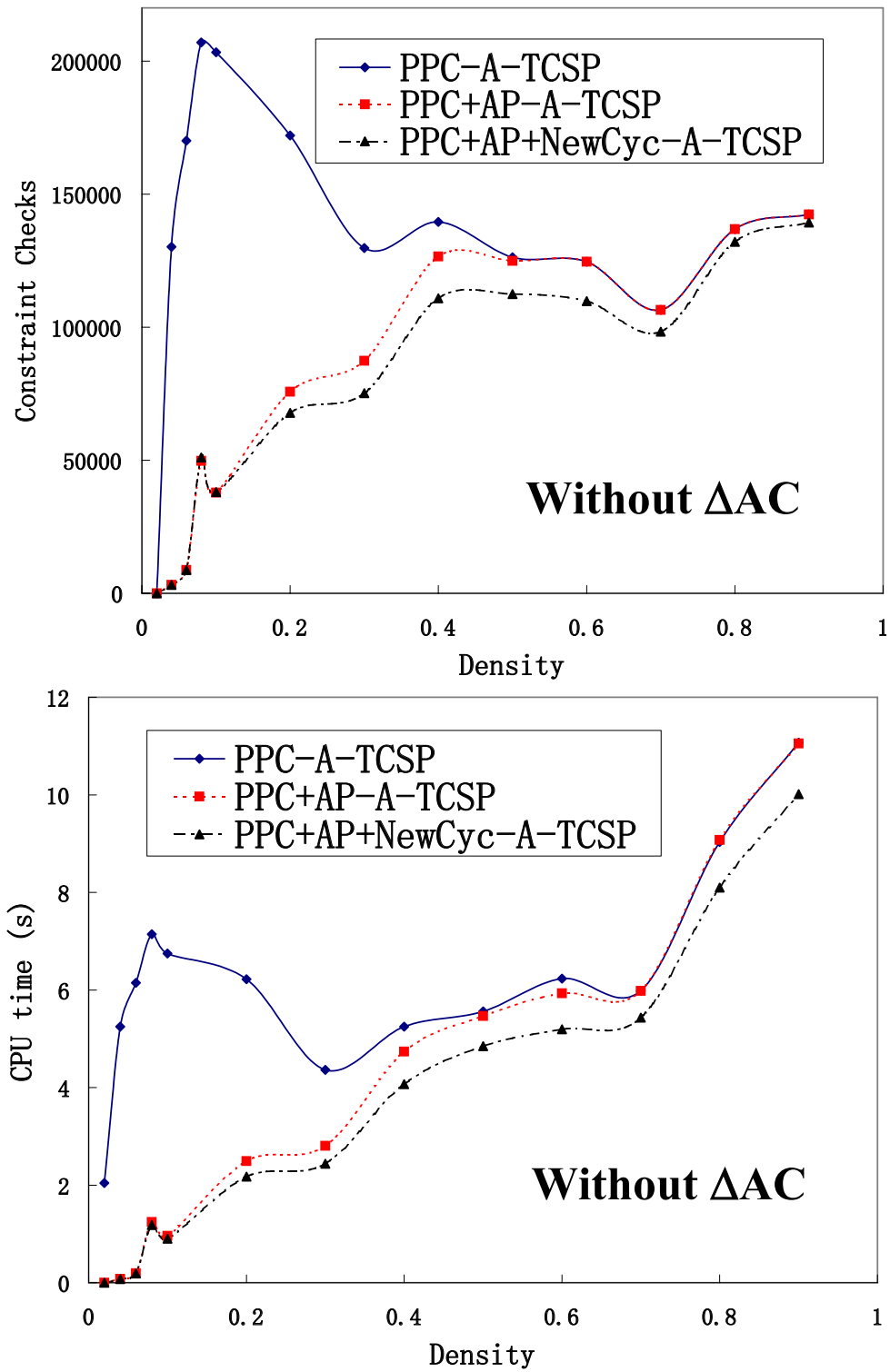


Figure 4.23: Constraint checks and CPU time for PPC-TCSP using Plan A without ΔAC (Top: Constraint Checks; Bottom: CPU time [s])

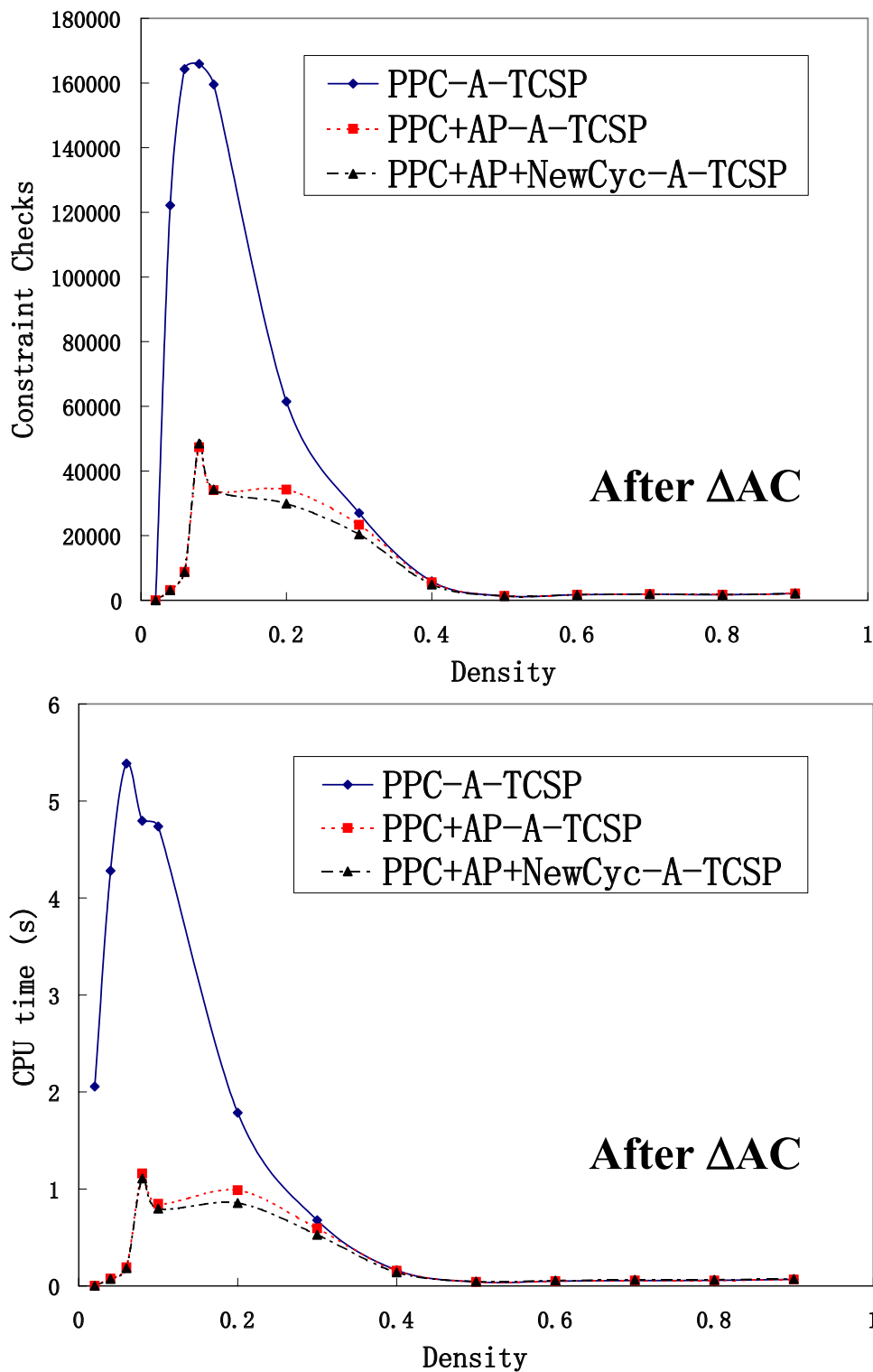


Figure 4.24: Constraint checks and CPU time for PPC-TCSP using Plan A after ΔAC (Top: Constraint Checks; Bottom: CPU time [s])

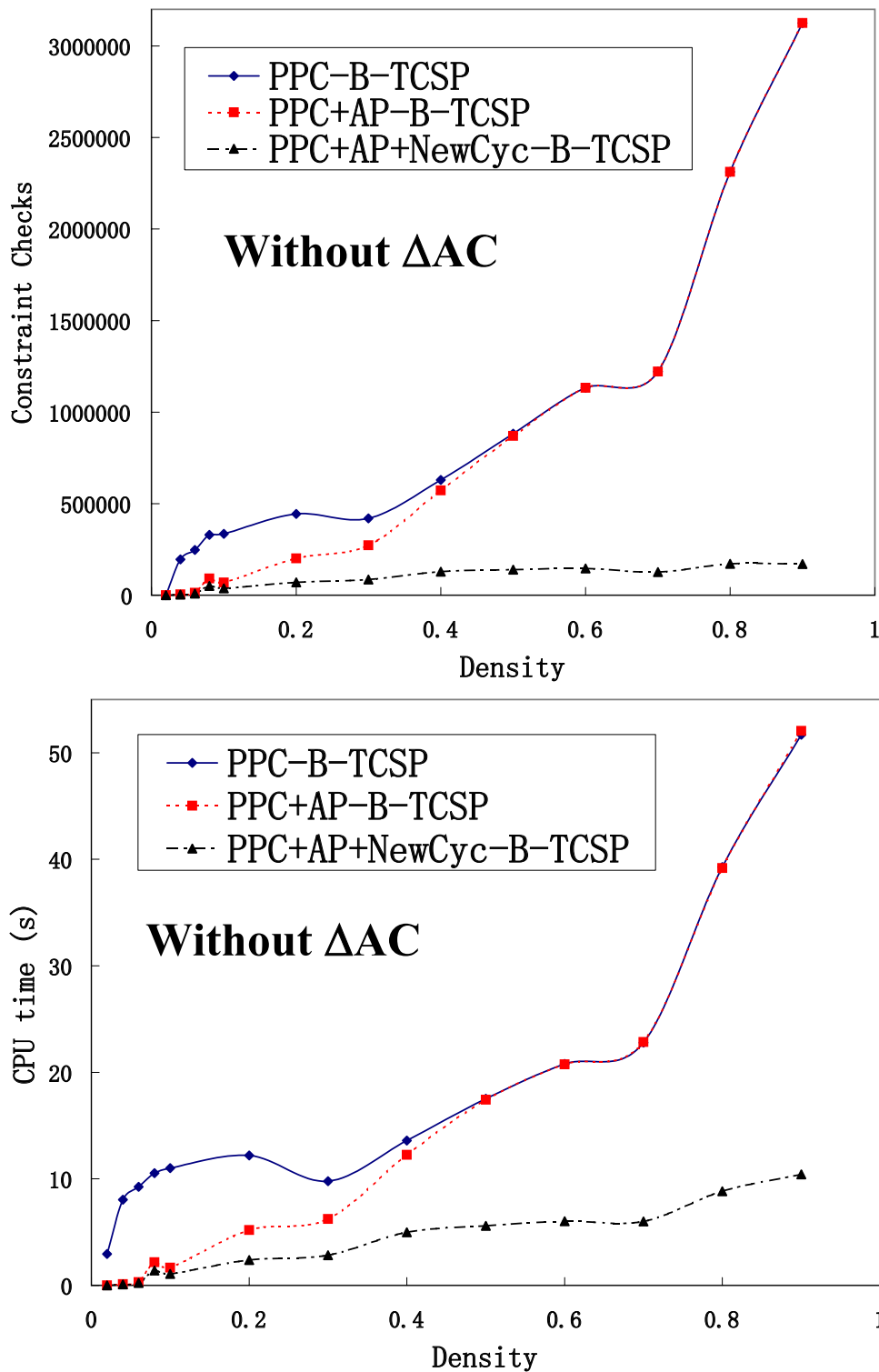


Figure 4.25: Constraint checks and CPU time for PPC-TCSP using Plan B without ΔAC (Top: Constraint Checks; Bottom: CPU time [s])

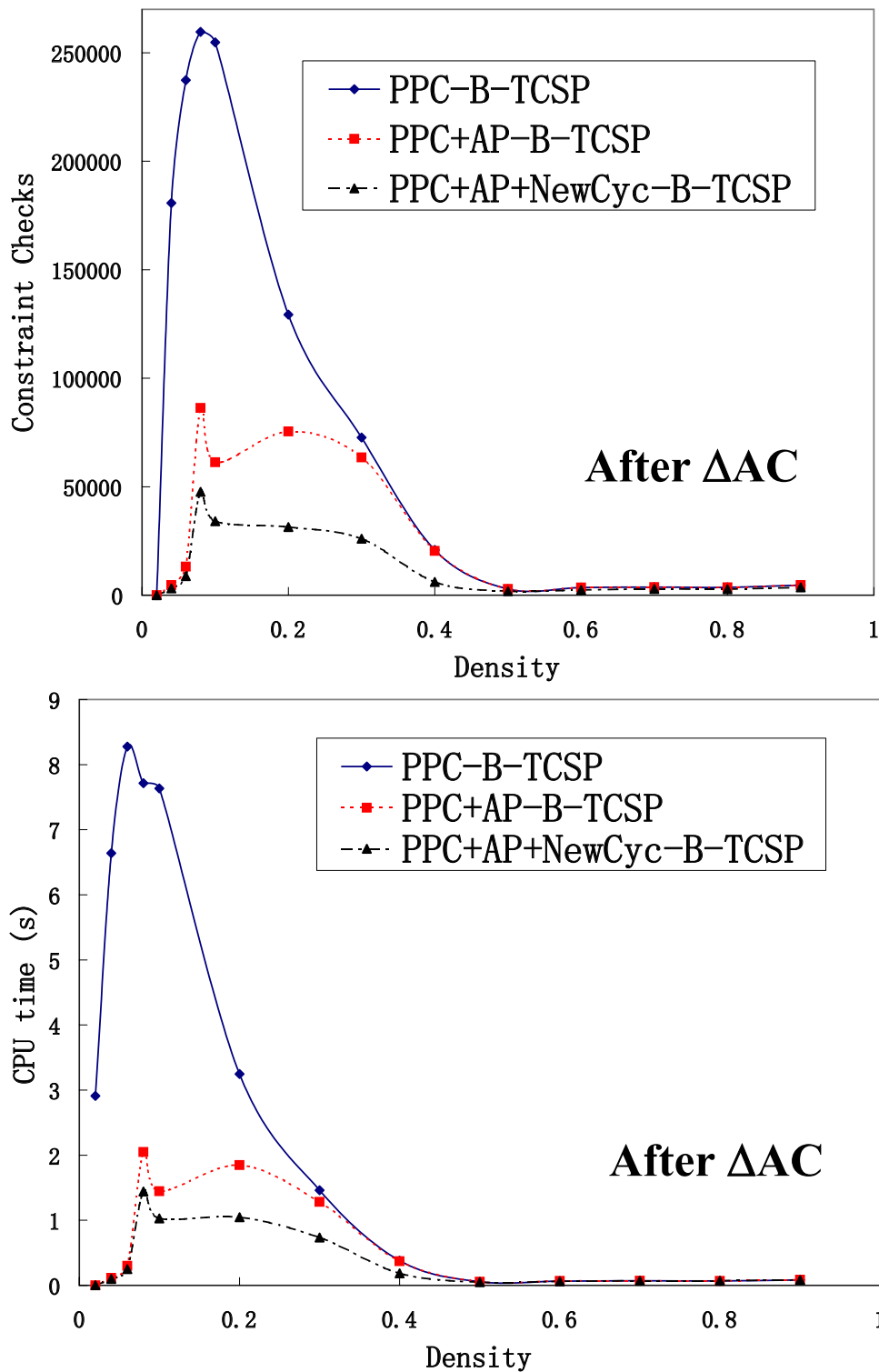


Figure 4.26: Constraint checks and CPU time for PPC-TCSP using Plan B after ΔAC (Top: Constraint Checks; Bottom: CPU time [s])

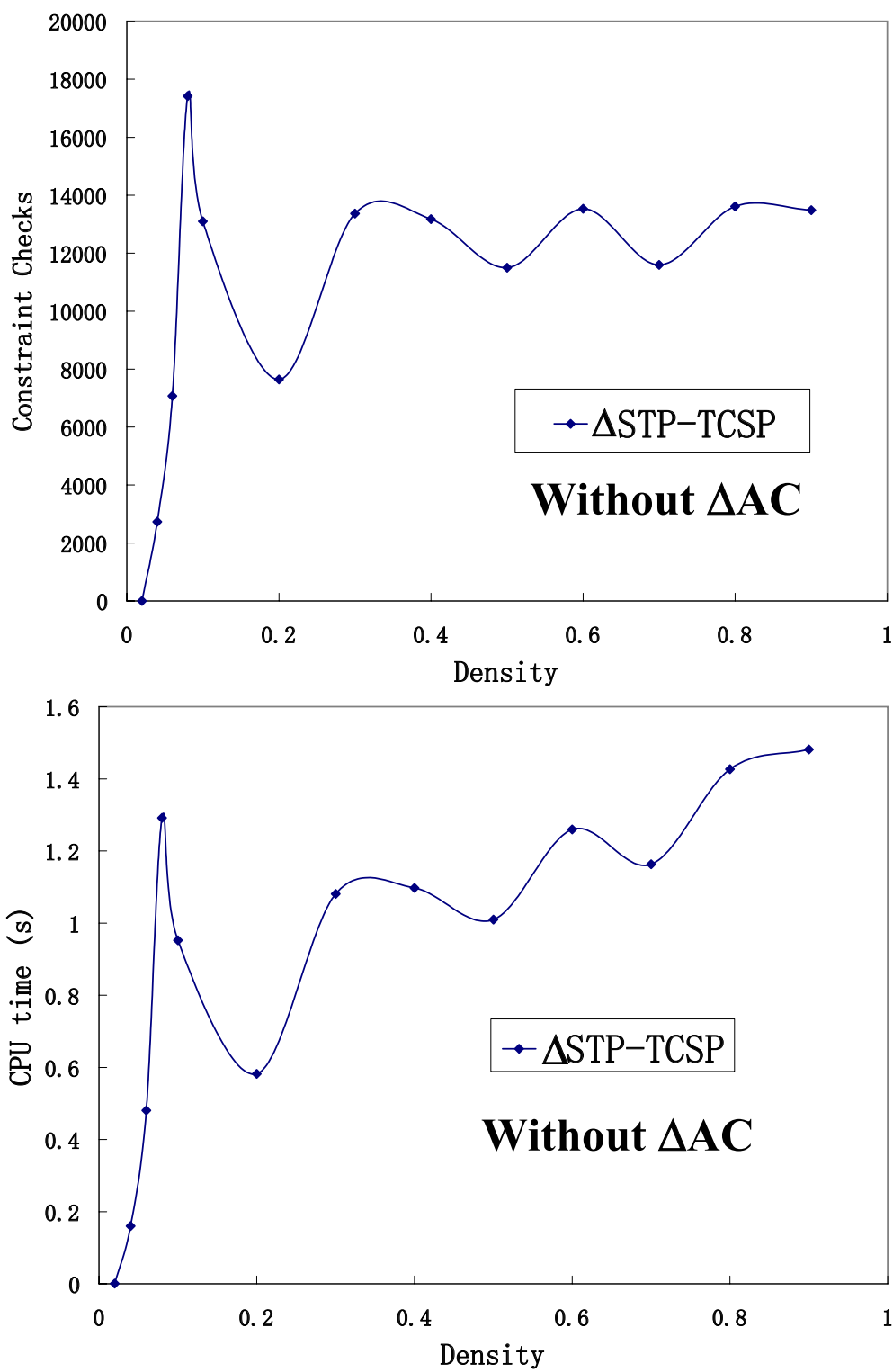


Figure 4.27: Constraint checks and CPU time for Δ STP-TCSP without Δ AC (Top: Constraint Checks; Bottom: CPU time [s])

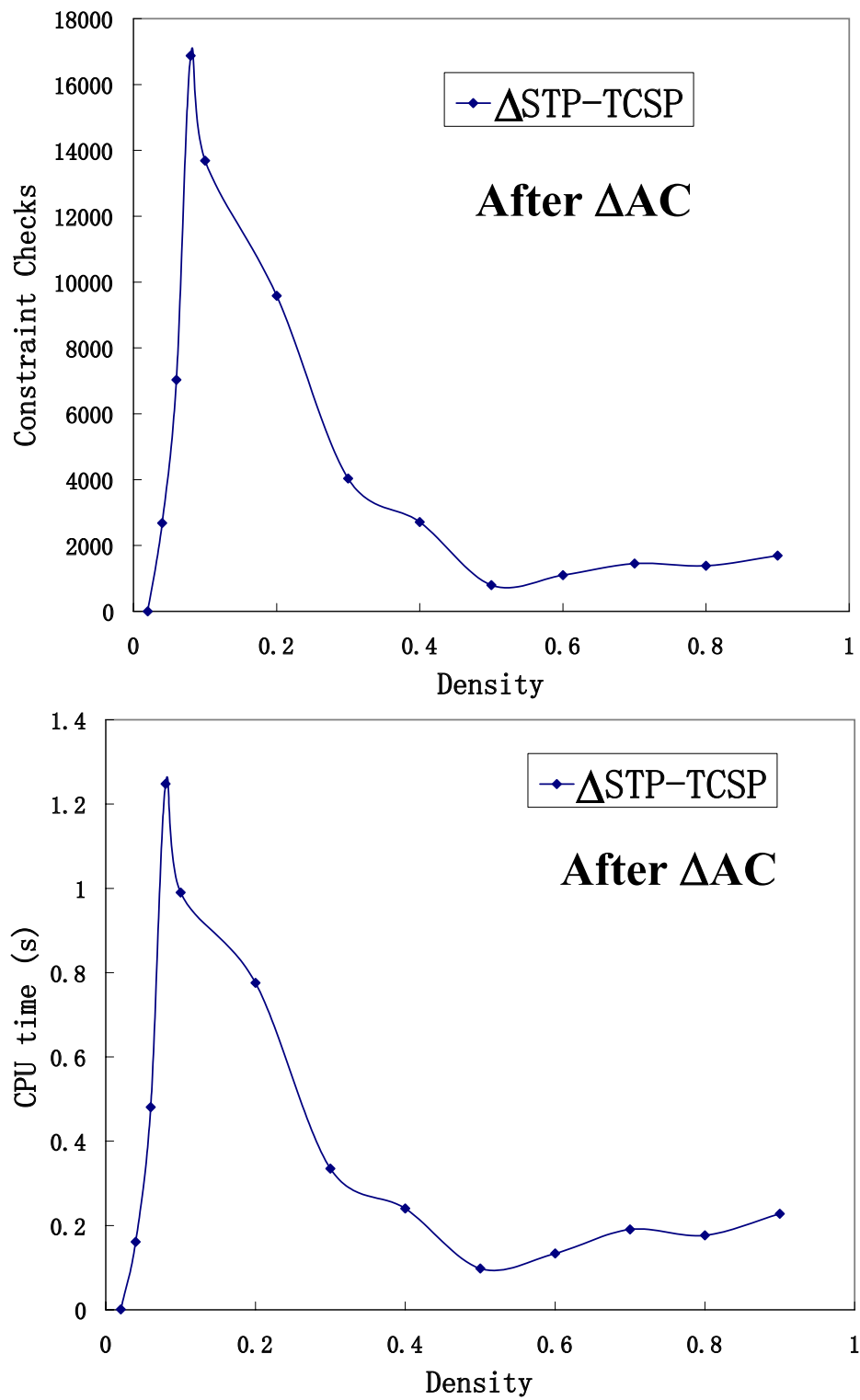


Figure 4.28: Constraint checks and CPU time for Δ STP-TCSP after Δ AC (Top: Constraint Checks; Bottom: CPU time [s])

Exploiting articulation points:

For DPC (Figure 4.21) and PPC (Figure 4.23 and 4.25), AP is again particularly effective for low density graphs but useless for high density ones.

New cycle check:

NewCyc dramatically reduces CC across all density values (even though it has no effect on the number of nodes visited, as stated in Section 4.5.3).

Triangulation plans:

The triangulation of an STP during search, required for PPC solver, is carried out according to *Plan A* (Figure 4.23) and *Plan B* (Figure 4.25) of Section 4.2.2. By comparing the scale of the vertical axis of these two figures, we conclude that *Plan A* is superior to *Plan B*. This can be explained as follows. *Plan A* triangulates, before search, all the networks that will be checked for consistency during search (there are exactly $|E|$ such graphs). *Plan B* finds the triangulation of an STP at a given node during search by inducing a subgraph from the triangulated original STP. Hence, *Plan B* triangulates the network only once, while *Plan A* carries out as many triangulation operations as the number of edges in the network (and levels in the search). However, the induced subgraphs in *Plan B* end up much denser than the ones used by *Plan A*, thus requiring more effort from PPC, the STP solver. Further, the fact that *Plan A* yields no denser graphs than *Plan B* becomes an even more desirable feature when TCSP is dense. This explains the significant differences in behavior between *Plan A* and *Plan B* under high density TCSPs.

 ΔAC :

Compare the cost of solving TCSP with and without using ΔAC , the results show that ΔAC does not negatively affect the cost of search under low density and is tremendously

effective in reducing the total cost under high density. Indeed, the cost of search is almost negligible when density is high. In contrast, search without preprocessing with ΔAC is prohibitively expensive when density is high.

When density is low, the temporal graph has few edges, hence the meta-CSP has relatively few variables and its size is small. When density increases, the number of edges in the temporal graph, and hence the number of variables in the meta-CSP, increase, yield exponentially larger problems. However, this increases the number of triangles in the temporal graph and enhances the filtering power of ΔAC , which removes most intervals. In all cases, the experiments strongly support using ΔAC when solving a TCSP.

The winning combination:

In [35] we compared the performances of F-W, DPC, PPC, and ΔSTP for solving an STP. We found that DPC, PPC, and ΔSTP consistently outperform F-W, the Floyd-Warshall algorithm. Further, ΔSTP consistently outperforms PPC. Indeed, the former is a finer version of the latter. Importantly, when the density of the temporal graph is below 0.4, ΔSTP (which guarantees minimality) outperforms DPC (which does not). For sensibly high densities, we found DPC to be more effective. Since in the search for solving the meta-CSP we consider subgraphs of the original network, the networks at the different levels of the tree are more likely to be sparse than dense. This shows that even when the TCSP is dense, ΔSTP is a good choice for the STP solver. Hence, among the techniques tested, the best combination one could use to solve a TCSP is the one we called ΔSTP -TCSP (Figure 4.15). Indeed ΔSTP outperforms all TCSP solvers including the one based on DPC (compare Figure 4.21 and 4.27).

Summary

At the beginning of our investigations, the best mechanism known to date for solving the meta-CSP⁴ was one based on DPC. We introduced Δ STP, enhanced it with NewCyc and EdgeOrd, and showed empirically that it results in dramatic improvements. Indeed, in comparison to the original DPC, the best combination of our techniques reduces the number of constraint checks by a factor of 500 (median) and 40,000 (average) and that of CPU by a factor of 320 (median) and 1,200 (average).

Further, we showed that our techniques uncover the existence of a phase-transition-like phenomenon for solving the TCSP, which is most visible with Δ STP-TCSP. This observation calls for more detailed investigations in this direction.

⁴Note that we do not include in our comparison algorithms that tighten these intervals in the labels of the edges. Those may not terminate in the general case and are prohibitively expensive in the integral case [13].

Chapter 5

Conclusions and future work

This thesis focuses on solving quantitative temporal problems. It covers aspects of modeling the temporal constraints, solving the temporal problem in general, and exploiting topological and semantic information to improve the solver's performance. We address both the Simple Temporal Problem (STP) and the general Temporal Constraint Satisfaction Problem (TCSP). For this purpose, we combine traditional techniques (e.g., articulation points), the latest results in constraint propagation (e.g., partial path consistency), a new powerful technique for constraint filtering (e.g., ΔAC and ΔSTP), and a set of new search heuristics (e.g., NewCyc and EdgeOrd). We also provide extensive sets of experimental results to compare the performance of the resulting solvers with previously known ones.

5.1 Conclusions for the STP

Simple temporal problem (STP) is a simple version of temporal CSP (TCSP) that can be solved in polynomial time. There are a number of algorithms for this task such as $F-W$ and DPC . We first apply partial path consistency algorithm (PPC) on STP, which is applicable in general CSP. The PC algorithm (operating on the complete graph) and the PPC algorithm (operating on the triangulated graph) yield the same labeling for the edges common to both

graphs, which is the minimal network of an STP. Furthermore, based on PPC, we develop a new STP solver, Δ STP, which is a finer version of PPC. In Δ STP, constraint propagation operates on the set of triangles of the triangulated temporal network instead of operating on its set of edges. It yields the minimal network with a cost always lower than or equal to that of PPC. Note that the topology of temporal graph is very important, we explore the articulation points to decompose the temporal graph into some biconnected components. Experimental results show the following conclusions:

1. Exploiting articulation points reduces the number of constraint checks of F-W and DPC when constraint density is low. The almost non-existence of articulation points under high density results in the same number of constraint checks for both strategies (exploiting articulation points and ignoring them) while not affecting the computational effort. Interestingly, we noticed that the exploitation of articulation points has little effect on the performance of DPC even when constraint density is low.
2. PPC is more efficient than F-W, especially under low density. Δ STP is always superior to PPC in terms of constraint checks and CPU time.
3. F-W, PPC, and Δ STP yield the same minimal labeling of the common edges. DPC only determines the consistency of the STP. It cannot ensure the property of path consistency, or a fortiori that of and minimality, of the temporal network. While it generally needs less constraint checks than Δ STP under high density, the fact that DPC does not guarantee as tight a result as Δ STP does, it is reasonable to consider that Δ STP is superior to DPC in general.
4. F-W is not sensitive to the topology of temporal network. The cost of F-W is always $\Theta(n^3)$. DPC is sensitive to the density of temporal network, because its complexity depends of the induced width. When density is high (induced width is large), the cost of determining the consistency of an STP using DPC increases. PPC and Δ STP are

more sensitive to the structure of temporal network. Problems with big still cycles will increase the cost of finding the minimal network.

5. PPC and Δ STP implicitly guarantee that articulation points in the graph (if any), are exploited, as if the network was decomposed into its biconnected components without actually decomposing it.

5.2 Conclusions for the TCSP

Temporal CSP in general is an **NP**-hard problem. We use backtrack search (BT-TCSP) introduced by Dechter et al. to solve TCSP. The TCSP is considered as a meta-CSP. Every node in the search tree of meta-CSP consists of an STP. The consistency of this STP needs to be checked before the search can proceed or backtrack. The goal is to find all solutions of meta-CSP. By combining all solutions of meta-CSP, we can obtain the minimal network of TCSP.

We introduce a few heuristics to improve the performance of BT-TCSP. Similar to solve CSP, we want save the number of constraint checks (it directly indicates the CPU time for solving CSP) as well as the number of nodes visited (it indicates the number of backtracks). Obviously, Using more efficient STP algorithm can save the number of constraint checks at each node. Consequently, it saves the number of constraint checks of BT-TCSP. Exploring the topology of the temporal graph could be very helpful for solving TCSP. Using articulation points to decompose the temporal problem into some sub-problems provides an upper bound, in the size of the largest biconnected component, to the search effort. But this only works in low density networks. We introduce a new edge check mechanism, which points out the consistency of STP at some level of search tree without actually running the STP consistency check. Using this mechanism saves lots of constraint checks especially under high density. Good variable ordering always helps to improve the efficiency of search. We

introduce an edge (variable) ordering heuristic based on the topology of temporal network. It expands the sub-problem to bigger and bigger set of triangles. It reduces the number of backtracks, constraint checks, and also decompose the graph automatically. At last, a filtering algorithm, ΔAC , is developed to reduce the size of meta-CSP. Experimental results show the following achievements:

1. The use of articulation points to decompose the temporal graph reduces the number of nodes visited and constraint checks in the search tree of the meta-CSP when the temporal network is sparse. It does not affect the performance of search when the network is densely connected.
2. The use of PPC for checking the consistency of STP at each node of the search tree requires fewer constraint checks than DPC does.
3. The different ways for retrieving the triangulated subgraphs needed at every level of the tree significantly impact the overall performance of a PPC-based BT-TCSP. Indeed, *Plan A*, which pre-computes and stores the triangulations of all the subgraphs to be used during search, is more effective than *Plan B*, which induces the triangulated subgraphs from the triangulation of the original graph.
4. The new heuristic NewCyc avoids unnecessary consistency checks at some levels of the search tree. It does not affect the number of nodes visited. Moreover, it checks only the consistency of newly formed biconnected component, which is typically smaller than than the original problem. We noticed that in practice the expansion of graph by addition of one new edge rarely produces a new cycle in the graph. This explains why NewCyc is so powerful in reducing the number of constraint checks for solving the meta-CSP.
5. EdgeOrd, our new variable ordering heuristic for searching the meta-CSP, arranges the edges according to the adjacency property of the temporal network. By this edge

ordering, the network is expanded as bigger and bigger set of triangles. Since the propagation of temporal network is based on triangles, this edge ordering makes the propagation more efficient and quickly determines the inconsistency of an STP. This heuristic reduces the number of nodes visited as well as the number of constraint checks. Applying this heuristic guarantees that articulation points in the graph (if any), are implicitly exploited, as if the network was decomposed into its biconnected components without requiring the use of any special algorithm for this purpose.

6. Δ STP consistently outperforms PPC. We can readily expect that the Δ STP-based BT-TCSP will outperform the one on PPC. To further boost the performance of the BT-TCSP solver, we also apply the new techniques we developed, namely: *Plan A*, *NewCyc*, and *EdgeOrd*. The resulting new TCSP solver, Δ STP-TCSP, is the absolute winner over all TCSP solvers we describe.
7. Δ AC is a sound, cheap, and effective algorithm for constraint propagation in a TCSP. It dramatically reduces the size of TCSP especially when the temporal network is dense. Applying Δ AC as a preprocessing technique for solving the meta-TCSP magnificently improves the performance of search. It also uncovers the potential existence of phase transition, which requires a more thorough investigation.

5.3 Directions for future research

We propose to extend our investigations in the following directions:

1. We can further improve Δ AC's performance and reduce the number of constraint checks by exploiting the convexity property of interval intersection. We suspect that this improvement may result in an optimal algorithm for determining the generalized arc-consistency of the reformulated network.

2. Another interesting direction for future research is to investigate how ΔAC can be used to improve the performance of the ULT algorithm of Schwalb and Dechter [29] since the two approaches are orthogonal.
3. The idea of inducing the decomposition of a graph by ignoring the existence of some edges is particularly attractive to us. With this idea, we can always decompose a large problem into smaller components, as a sufficient approximation for establishing inconsistency. This may dramatically increase the size limit of TCSPs that we are currently able to handle.
4. Investigate how to exploit ΔAC in a lookahead strategy for solving the meta-TCSP.
5. Evaluate empirically how to improve BT-TCSP with dynamic bundling [10], which is particularly attractive in this context since we are looking for all solutions.

The ability to represent time in a flexible way and reason about it effectively is central to the success of Artificial Intelligence and its usefulness in our lives. In this thesis, we have exploited previously known results in a creative way and introduced new techniques to enhance the performance of processing the two representations of temporal networks, the STP and the TCSP. In the future, we plan to pursue our investigations of the basic aspects of temporal reasoning and apply them in specific problem-solving tasks such as planning and scheduling in order to demonstrate their usefulness in practical settings.

Bibliography

- [1] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26:123–154, 1983.
- [2] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 26:123–154, 1984.
- [3] C. Bessière, E. C. Freuder, and J.-C. Régin. Using Constraint Metaknowledge to Reduce Arc Consistency Computation. *Artificial Intelligence*, 107 (1):125–148, 1999.
- [4] Christian Bessière. Arc-Consistency and Arc-Consistency Again. *Artificial Intelligence*, 65:179–190, 1994.
- [5] Christian Bessière and Jean-Charles Régin. Refining the Basic Constraint Propagation Algorithm. In *Proc. of the 17th IJCAI*, pages 309–315, Seattle, WA, 2001.
- [6] Christian Bliet and Djamilla Sam-Haroud. Path Consistency for Triangulated Constraint Graphs. In *Proc. of the 16th IJCAI*, pages 456–461, Stockholm, Sweden, 1999.
- [7] Mark Boddy. Personal communication, 2002.
- [8] Amedeo Cesta, Angelo Oddi, and Stephen Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 8(1):109–136, April 2002.

- [9] Boris V. Cherkasskyn, Andrew V. Goldberg, and Tomasz Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Mathematical Programming*, 73:129–174, 1996.
- [10] Berthe Y. Choueiry and Amy M. Davis. Dynamic Bundling: Less Effort for More Solutions. In Sven Koenig and Robert Holte, editors, *5th International Symposium on Abstraction, Reformulation and Approximation (SARA 2002)*, volume 2371 of *Lecture Notes in Artificial Intelligence*, pages 64–82. Springer Verlag, 2002.
- [11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Co & MIT Press, 2001.
- [12] Thomas Dean and Drew McDermott. Temporal Data Base Management. *Artificial Intelligence*, 32:1–55, 1987.
- [13] Rina Dechter. Constraint Processing. Manuscript, forthcoming, 2003.
- [14] Rina Dechter and Avi Dechter. Belief Maintenance in Dynamic Constraint Networks. In *Proc. of AAAI-88*, pages 37–42, St. Paul, MN, 1988.
- [15] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.
- [16] Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1987.
- [17] Salmon Even. *Graph Algorithm*. Computer Science Press, 1979.
- [18] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1):24–32, 1982.
- [19] Eugene C. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32 (4):755–761, 1985.

- [20] Eugene C. Freuder and Paul D. Hubbe. A Disjunctive Decomposition Control Schema for Constraint Satisfaction. In Vijay Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, pages 319–335. MIT Press, Cambridge, MA, 1995.
- [21] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.
- [22] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.
- [23] Roger Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.
- [24] Roger Mohr and Gérald Masini. Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI-88)*, pages 651–656, Munich, W. Germany, 1988.
- [25] Ugo Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Sciences*, 7:95–132, 1974.
- [26] Nicolas Muscettola, Paul Morris, and Ioannis Tsamardinos. Reformulating Temporal Plans for Efficient Execution. In *Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 444–452, Trento Italy, 1998.
- [27] Peter Revesz. *Introduction to Constraint Databases*. Springer-Verlag, New York, 2001.
- [28] U. Kjærulff. Triangulation of Graphs - Algorithms Giving Small Total State Space. Research Report R-90-09, Aalborg University, Denmark, 1990.

- [29] Eddie Schwalb and Rina Dechter. Processing Disjunctions in Temporal Constraint Networks. *Artificial Intelligence*, 93:29–61, 1997.
- [30] Yoav Shoham. *Reasoning About Change*. MIT Press, Cambridge, MA, 1988.
- [31] Robert Shostak. Deciding Linear Inequalities by Computing Loop Residues. *Journal of the ACM*, 28(4):769–779, 1981.
- [32] Ioannis Tsamardinou. Reformulating Temporal Plans for Efficient Execution. Master’s thesis, Intelligent Systems Program, University of Pittsburgh, 1998.
- [33] Peter van Beek. Approximation Algorithms for Temporal Reasoning. In *Proc. of the 11th IJCAI*, pages 1291–1296, Detroit, MI, 1989.
- [34] M. Vilain and Henry Kautz. Constraint Propagation Algorithms for Temporal Reasoning. In *Proc. of AAAI-86*, pages 377–382, Philadelphia, PA, 1986.
- [35] Lin Xu and Berthe Y. Choueiry. A New Efficient Algorithm for Solving the Simple Temporal Problem. In *10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic (TIME-ICTL 2003)*, Cairns, Queensland, Australia, 2003. IEEE Computer Society Press.
- [36] Lin Xu and Berthe Y. Choueiry. An Efficient Algorithm for Filtering the TCSP. In *Principles and Practice of Constraint Programming*, 2003. Submitted.
- [37] Lin Xu and Berthe Y. Choueiry. Improving Backtrack Search for Solving the TCSP. In *Principles and Practice of Constraint Programming*, 2003. Submitted.
- [38] Yualin Zhang and Roland H.C. Yap. Making AC-3 an Optimal Algorithm. In *Proc. of the 17th IJCAI*, pages 316–321, Seattle, WA, 2001.