UCLouvain

École polytechnique de Louvain

Solving Disjunctive Temporal Problems

A Constraint Programming Approach

Authors: **Mehdi Ben Haddou, Eliot Hennebo**
Supervisors: **Michael Saint-Guillain, Pierre Schaus, Yves Deville**
Reader: **Augustin Delecluse**
Academic year 2022–2023
Master [120] in Computer Science

Abstract

In our increasingly connected world, the management and coordination of time-dependent tasks is crucial. Disjunctive Temporal Networks (DTNs) play a central role in managing such complex, time-sensitive tasks. Their applications range from planning space exploration missions, where precise sequencing of events is vital, to organizing intricate operations within supply chains. However, the resolution of DTNs, due to their combinatorial nature, remains a long-standing challenge.

This thesis provides a fresh perspective to this issue by transforming the quest for a perfect DTN solution into the pursuit of a practical approximation. It proposes a novel approach using Constraint Programming (CP) and the Boolean Switch relaxation, to turn the DTN problem into a Constrained Optimization Problem (COP). This method aims to find feasible solutions faster and provides flexible way of handling DTNs.

To ease this process and further enhance the practicality of our approach, we developed a user-friendly tool. This tool allows for real-time visualization and tracking of constraint violations during the relaxation process. It integrates various solvers along with the strategies we've developed, thereby creating an interactive platform for DTNs resolution.

Aknowledgments

Our sincere thanks go to Michael Saint-Guillain, whose supervision of the project and consistent feedback throughout the year have been crucial to our progress.

We extend our gratitude to Pr. Pierre Schaus and Pr. Yves Deville for their supervision, valuable feedback, and pertinent advice that significantly contributed to the refinement of our work.

We also extend our appreciation to Augustin Delecluse, for his time and effort in reviewing our work as a member of the jury.

Lastly, we would like to thank our families for their continuous support during our master's journey.

Contents

Introduction

Temporal reasoning is a fundamental aspect of many complex systems, where tasks must be managed and coordinated under time constraints. Disjunctive Temporal Networks (DTNs) provide a formalism for representing these constraints, offering a rich expressivity that can capture a wide range of temporal problems. However, the combinatorial nature of DTNs makes their resolution a challenging task.

Previous research, such as the work by Zavatteri et al. [21], has investigated the use of Constraint Programming (CP) for solving DTNs. Their focus was on finding a model that satisfies the constraints, but they concluded that CP might not be the most suitable approach for this task. This conclusion opens up a new avenue of exploration: what if, instead of seeking a perfect solution, we aim for an approximation?

In this thesis, we propose a novel approach to this problem. We use CP and introduce the concept of relaxation to transform the DTN problem into a Constrained Optimization Problem (COP). Specifically, we explore a Boolean Switch relaxation technique. This technique aim to provide an approximate solution for non-tractable problems, offering a more flexible method for handling DTNs. Moreover, we also aim to find solutions faster for tractable problems.

The thesis begins with a comprehensive understanding of the temporal network problem, including the basics of temporal reasoning and more complex constructs from Simple Temporal Networks (STNs) to DTNs. We then review the state of the art in solving temporal networks, discussing existing approaches for both STNs and DTNs.

Building on this foundation, we introduce the concept of CP, providing definitions and discussing key aspects such as consistency, propagation, and search. We propose a method for transforming DTNs into optimization problems, introducing

the concept of soft constraints and relaxation in DTNs, and discussing different encoding strategies and optimization strategies.

Following this preliminary work, our experiments form the central part of this thesis. They were conducted on DTN datasets and also on well-known NP-Hard problems such as Hamiltonian Cycle, Graph Coloring, and SAT benchmarks that we mapped to DTNs. The results of these experiments provide insights into the performance of different CP solver configurations for various DTN problems.

Finally, we present a DTN CP Solving Tool, providing an overview of its features and discussing future perspectives for the tool. This tool serves as a practical implementation of the theories and methodologies discussed in the thesis.

In summary, this thesis explores a new approach to solving DTNs using CP. By introducing relaxation techniques and transforming the problem into a Constrained Optimization Problem, we aim to provide new insights into this complex area of study.

Chapter 1

The Temporal Network Problem

1.1 Temporal Reasoning

1.1.1 Definition

Temporal reasoning is a fundamental domain in artificial intelligence and computer science as it explores how to reason about and manipulate temporal information. For example, temporal reasoning is widely used in planning contexts to schedule a series of tasks, taking into account the time required for each task and any dependencies or conditions that could exist between the tasks.

In practice, it is often implemented with temporal constraints, which are logical statements that specify the allowed relationships between events and actions in time. For example, a temporal constraint might specify that a certain task must be completed before another task can begin, or that two tasks must be performed concurrently.

1.1.2 Temporal networks

Temporal networks are a type of mathematical representation that can be used to model the temporal relationships between events and actions. In a temporal network, events and actions are represented as nodes, and the temporal constraints between them are represented as edges.

By considering these temporal constraints, we can reason about the possible sequences of events and actions that are allowed by the constraints, and use this information to plan and schedule its actions. The advantage of using temporal networks is that they provide a compact and intuitive way to represent complex temporal relationships.

1.2 Simple Temporal Networks

1.2.1 Problem

Definition

A Simple Temporal Network (STN) is a specific type of temporal network that focuses on the quantitative temporal constraints between pairs of events. These networks are considered "simple" because they deal exclusively with direct, binary relationships, expressing constraints as exact time durations or ranges of permissible durations between two events.

Example

The Mars Rover planning problem we are illustrating in the Figure 1.1 can be summarized as follows: the rover can operate between 8am and 3pm, during which time it must complete a series of tasks while adhering to all constraints.

More precisely, the rover begins its day after 8am and needs to recharge its battery for between 45 and 90 minutes. Then it must drive for 1 hour and 30 minutes to reach the location where it will conduct experiments. The experiment involving filming takes 30 minutes and must be completed before 11:30am. The drilling experiment takes 1 hour, and must be done between 10am and 2pm, but only after the first experiment is finished. Once the experiments are complete, the rover must drive 1 hour and 30 minutes to the data transmission site to send the data to the satellite. The satellite is only in range for a short time, so the rover must transmit the data between 2pm and 3pm.

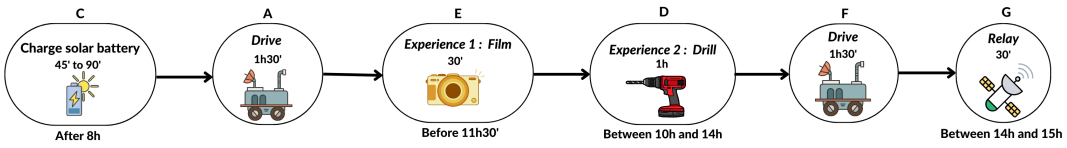

Figure 1.1: Illustrated example : informal representation

Modeling this problem as an STN allows answering the following questions :

- Is it possible to complete all tasks while meeting all constraints?
- What are the earliest and latest times at which the rover can finish its day?

1.2.2 Formalism

Definition

A STN S is mathematically represented by a tuple (T, C) where T is a set of time points variables and C is a set of constraints. A constraint is a binary constraint that creates a relation between a pair of time points variables.

- $\forall t_i \in T$, t_i is a time point variable that generally represents the beginning or the end of an event.
- $\forall C_i \in C$, C_i represents a bound on the elapsed time in the form $t_b - t_a \leq r_{ab}$ between two events t_a and t_b where :
  - $t_a, t_b \in T \subseteq \mathbb{R}$
  - $r_{ab} \in \mathbb{R}$

When two constraints $t_b - t_a \leq r_{ab}$ and $t_a - t_b \leq r_{ba}$ ($\Leftrightarrow -r_{ba} \leq t_b - t_a$) are defined, the more compact notation $-r_{ba} \leq t_b - t_a \leq r_{ab}$ will often be used. (e.g., if $r_{ab} = 5$ and $r_{ba} = -1$, then $1 \leq t_b - t_a$ and $t_b - t_a \leq 5$ are equivalent to $1 \leq t_b - t_a \leq 5$ to express that the duration between A and B has to be included in the interval [1,5]).

In addition, we introduce the *zero time-point* t_0, or sometimes called z, used as a fixed reference point and, most of the time, assigned to the value 0. This allows to explicitly specify the limit of an event t_i, we can then express an *absolute temporal constraint* such as "event a must occur between 10am and 11am" $\Leftrightarrow 10 \leq t_a - t_0 \leq 11$.

Time point variables

When representing STNs, it is common practice to use two time point variables per event: one for the start and one for the end (e.g., In the rover example, the start time point of the event "Charge Solar Battery" is C and the end is C'). This approach has several advantages, such as the ability to express more easily duration constraints and temporal ordering constraints between events.

One of the biggest advantages of using two time points for each event is that it allows for the modeling of temporal relationships between events without forcing an event to start directly after another one.

In other words, with this technique, it is possible to express constraints on the temporal ordering of events, such as "event A must happen before event B" without implying that event B must start immediately after event A. This allows for more flexibility in the representation of temporal information and allows for the modeling of temporal relationships that do not necessarily imply an immediate succession.

To summarize, in a scheduling problem, it is possible to express the constraint that an event A must happen before event B, but also to add a constraint of a time window in which event B can happen after A, this allows a more realistic representation of the problem. (e.g., there should be a minimum time of 5 minutes between the end of event A and the start of event B $\Leftrightarrow A_{end} - B_{start} \leq -5$)

In this example, the start times of events will be represented by letters and the end times of events will be represented by the same letters with an apostrophe. (e.g., $A_{start} \Longrightarrow A$ and $A_{end} \Longrightarrow A'$

Example

The Figure 1.2 demonstrates the process of converting some constraints previously expressed in text form into valid constraints in an STN, by displaying their formal notation.

Each event is represented by two time point variables, one for the start and one for the end. A link represents a constraint, not all constraints are illustrated here, and there are also implicit constraints that are not represented. For example, if C must be before A and A must be before E, then C must be before E. z represents the start time of the schedule, in this case, the start of the day, which allows us to put a constraint on the start of C (start after 8am). C has to last between 45 and 90 minutes, so we have two constraints represented in the interval. E is after A, so we enforce that the difference between the start of E and the end of A is greater than or equal to 0, which is equivalent to the represented less than or equal constraint. D has to start after 10am and finish before 2pm, so we can add two constraints on D and D' relative to Z. Then, we have the example of event F, which has an exact time duration and this is also encoded as two less than or equal constraints.

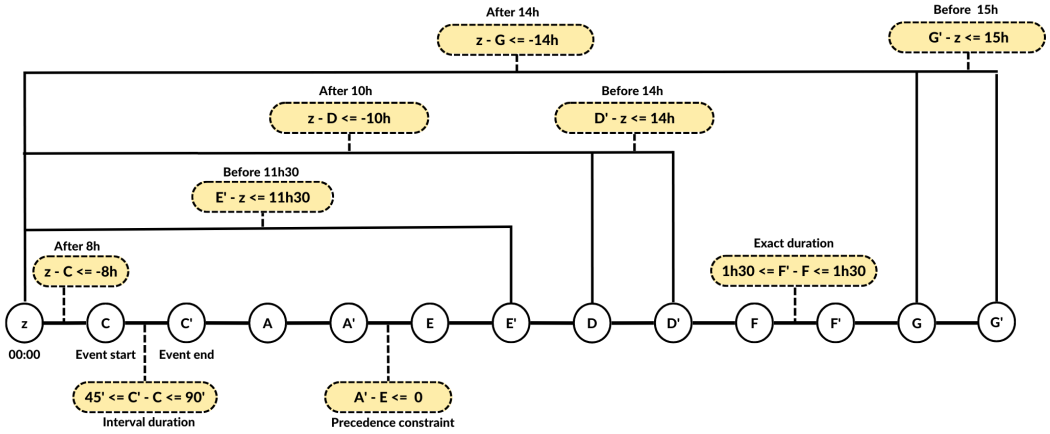

Figure 1.2: Translation of the illustrated example into STN constraints.

1.2.3 Directed constraint graph

Definition

There are several ways to represent an STN, but one common way is to use a directed constraint graph. In this representation, the nodes of the graph represent the events or time points, and the edges represent the constraints between them. The direction of the edge indicates the type of constraint, and the weight of the edge indicates the strength or importance of the constraint.

A STN $S = (T, C)$ can be represented by a similar graph $G_s = (T_s, E_s)$ where :

- T_s is the set of vertices that represent the events $t_i \in T$
- E_s is the set of edges that represent the constraints C such that for every constraints $C_i \in C$ we have the corresponding edge $t_a \xrightarrow{r_{ab}} t_b \iff t_b - t_a \leq r_{ab}$

Constraint(s)	Edge(s)
$C' - C \leq 90$	$C \xrightarrow{90} C'$
$C - C' \leq -45$	$C \xleftarrow[-45]{} C'$
$45 \leq C' - C \leq 90$	$C \underset{-45}{\overset{90}{\rightleftharpoons}} C'$
$D \leq 600$	$z \underset{-600}{\overset{+\infty}{\rightleftharpoons}} D$

Table 1.1: Different types of constraints with their corresponding edge representation

Example

The Figure 1.3 is the directed graph representation of our starting example.

All edges represent minutes starting from 00h, only constraints are represented, any non-existent edge will be considered as having an infinite value, C cannot start before 8am is represented by an edge going from C to z, while the one going from z to C does not exist and will therefore have an infinite value in the following calculations. Having infinity simply allows us not to impose an upper bound on C and express that we don't have an explicit constraint to make C end before a given hour. Just as the edges between C' and C induce that the duration of the event must be included in [45,90], the upper bound on the duration of C is thus 90, while its lower bound is 45.

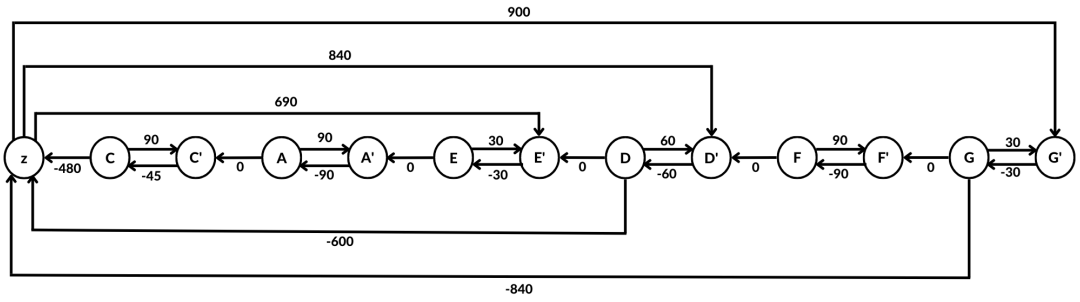

Figure 1.3: Translation of the STN constraints from Figure 1.1 into a directed constraint graph.

1.2.4 Distance matrix

Definition

A STN can also be represented as a distance matrix R of size $n * n$ where n is the number of time points $|T|$ and R_{ab} is equivalent to the constraint $t_b - t_a \leq R_{ab}$. This distance matrix R can directly be induced by the directed constraint graph, and the unknown values will again be represented as $+\infty$.

For instance, to express that event C lasts between 45 and 90 minutes, we have the constraint $45 \leq C' - C \leq 90$ which is decomposed in two constraints, $C' - C \leq 90$ and $C - C' \leq -45$. These constraints will be represented in the distance matrix as $R_{C'C} = -45$ and $R_{CC'} = 90$.

Example

The distance matrix is a representation in the form of a table of the directed constraint graph, by taking all pairs of nodes in our previous example and indicating the costs of the edges that connect them, we simply obtain this distance matrix.

	z	C	C'	A	A'	E	E'	D	D'	F	F'	G	G'
z	0.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	690.0	$+\infty$	840.0	$+\infty$	$+\infty$	$+\infty$	900.0
C	-480.0	0.0	90.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
C'	$+\infty$	-45.0	0.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
A	$+\infty$	$+\infty$	0.0	0.0	90.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
A'	$+\infty$	$+\infty$	$+\infty$	-90.0	0.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
E	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0.0	0.0	30.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
E'	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	-30.0	0.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
D	-600.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0.0	0.0	60.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
D'	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	-60.0	0.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
F	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0.0	0.0	90.0	$+\infty$	$+\infty$
F'	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	-90.0	0.0	$+\infty$	$+\infty$
G	-840.0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0.0	0.0	30.0
G'	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	-30.0	0.0

Table 1.2: Distance matrix of the example

1.2.5 Shortest paths matrix

Definition

The Simple Temporal Network being a Directed Constraint Graph we can find all the shortest paths between every pair of time points in T. Fortunately, this is a polynomial problem that can be solved using a shortest path algorithm such as

Floyd Warshall to find all pairs of shortest paths. This algorithm applied on the initial distance matrix R will give as a result the shortest path distance matrix S of size $n * n$ containing the shortest path between every pair of time points in T.

More formally, for every pair of time points t_i, $t_j \in T$, S_{ij} is the the shortest path length between i and j that can be derived from C. In a similar way, if S contains S_{ij} and S_{ji}, the following constraints $t_j - t_i \leq S_{ij}$ and $t_i - t_j \leq S_{ji}$ are the tightest constraints implied by C between event i and j.

Example

This matrix fully solves the STN. It allows you to calculate the shortest and longest durations between two events. Below the diagonal, the earliest times are shown and above the latest times. The simplest example we want to solve is to retrieve the absolute values of cells $S_{zG'}$ (i.e. upper bound) and $S_{G'z}$ (i.e. lower bound) to know if the STN is consistent. If somewhere in the matrix we see that the lower bound (i.e. earliest time) has a higher value than the upper bound (i.e. latest time), it means that the STN is not consistent. We can also compute some interesting insights between the events, if we want to know what is the lowest duration possible between 2 events for example.

	z	C	C'	A	A'	E	E'	D	D'	F	F'	G	G'
z	0.0	525.0	570.0	570.0	660.0	660.0	690.0	720.0	780.0	780.0	870.0	870.0	900.0
C	-480.0	0.0	90.0	90.0	180.0	180.0	210.0	240.0	300.0	300.0	390.0	390.0	420.0
C'	-525.0	-45.0	0.0	45.0	135.0	135.0	165.0	195.0	255.0	255.0	345.0	345.0	375.0
A	-525.0	-45.0	0.0	0.0	90.0	135.0	165.0	195.0	255.0	255.0	345.0	345.0	375.0
A'	-615.0	-135.0	-90.0	-90.0	0.0	45.0	75.0	105.0	165.0	165.0	255.0	255.0	285.0
E	-615.0	-135.0	-90.0	-90.0	0.0	0.0	30.0	105.0	165.0	165.0	255.0	255.0	285.0
E'	-645.0	-165.0	-120.0	-120.0	-30.0	-30.0	0.0	75.0	135.0	135.0	225.0	225.0	255.0
D	-645.0	-165.0	-120.0	-120.0	-30.0	-30.0	0.0	0.0	60.0	135.0	225.0	225.0	255.0
D'	-705.0	-225.0	-180.0	-180.0	-90.0	-90.0	-60.0	-60.0	0.0	75.0	165.0	165.0	195.0
F	-705.0	-225.0	-180.0	-180.0	-90.0	-90.0	-60.0	-60.0	0.0	0.0	90.0	165.0	195.0
F'	-795.0	-315.0	-270.0	-270.0	-180.0	-180.0	-150.0	-150.0	-90.0	-90.0	0.0	75.0	105.0
G	-840.0	-315.0	-270.0	-270.0	-180.0	-180.0	-150.0	-150.0	-90.0	-90.0	0.0	0.0	30.0
G'	-870.0	-345.0	-300.0	-300.0	-210.0	-210.0	-180.0	-180.0	-120.0	-120.0	-30.0	-30.0	0.0

Table 1.3: Shortest path distance matrix of the rover example

1.2.6 Time windows

Definition

The practical representation of the set of all feasible solutions for a STN can be viewed as a set of time windows representing the earliest and the latest happening time for each event.

Tightest time windows can be easily computed from the shortest path distance matrix S. Indeed, for each event k, S_{zk} contains the latest time from z (zero time points) to t_k and S_{kz} the earliest time from t_k to z. Therefore, with the first column and the first row, we can construct the minimal network by taking the earliest and latest time for each event.

For example, using the shortest path matrix S, we can retrieve the earliest and latest start time of event E by taking S_{zE} and $abs(S_{Ez})$, respectively. These values yield 615 and 660, indicating that event E can start between 10:15 and 11:00.

Example

Event	Earliest (in minutes)	Latest (in minutes)	Earliest (in hours)	Latest (in hours)
z	0.0	0.0	00:00	00:00
C	480.0	525.0	08:00	08:45
C'	525.0	570.0	08:45	09:30
A	525.0	570.0	08:45	09:30
A'	615.0	660.0	10:15	11:00
E	615.0	660.0	10:15	11:00
E'	645.0	690.0	10:45	11:30
D	645.0	720.0	10:45	12:00
D'	705.0	780.0	11:45	13:00
F	705.0	780.0	11:45	13:00
F'	795.0	870.0	13:15	14:30
G	840.0	870.0	14:00	14:30
G'	870.0	900.0	14:30	15:00

Table 1.4: Time windows of the rover example

Interpretation

We can now answer the questions using the shortest path distance matrix and the time windows we computed:

- Is it possible to complete all tasks while meeting all constraints?
  - Yes. $S_{G'z} \leq S_{zG'}$
- What are the earliest and latest times at which the rover can finish its day?
  - Earliest : 14:30 $S_{G'z}$, Latest : 15:00 $S_{zG'}$

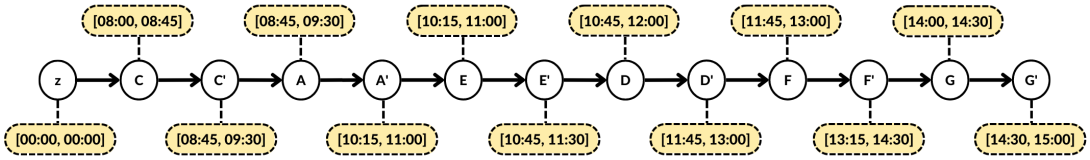

Figure 1.4: Complete example solved with the possible time windows for all events.

1.2.7 Properties

Definition 1 (Schedule)**.** A schedule is a specific assignment to all the time points variables $T_i \in T$. A schedule doesn't necessarily satisfy all the constraints.

Definition 2 (Solution)**.** An assignment to all time points variable $T_i \in T$ is called a solution if it satisfies all the constraints. A value v is called a feasible value for a variable T_i if there exists a solutions in which $T_i = v$. The set of all such feasible values for a variable T_i is the minimal domain. For our example a solution could be the tuple $S = \{C = 480, C' = 525, A = 525, A' = 615, E = 615, E' = 645, D = 645, D' = 705, F = 705, F' = 795, G = 840, G' = 870\}$.

Definition 3 (Consistent)**.** A STN is consistent if it exists at least one *(schedule)* assignment to all the time points variables $T_i \in T$ that satisfies all the constraints $C_i \in C$.

Definition 4 (Implicit constraint)**.** The directed graph representation is very helpful to derive new constraints implied by the set of constraints C. Indeed, since constraints can be seen as a path between two event i and j, we can now express *implicit constraints* that are not initially in C. For example, if C contains two

constraints such as $t_j - t_i \leq 5$ and $t_k - t_j \leq 2$ we can say that a path exists from i to k represented by the implicit constraints $t_k - t_i \leq 7$

Definition 5 (Minimal network)**.** A minimal network is a STN where all the constraints are guaranteed to be as tight as possible. It means that the ranges for constraints and time points contain exactly the values that any valid schedule could have. It is the exact set of all possible solutions for a STN. Therefore, it is a more expressive propriety than consistency.

1.3 Disjunctive Temporal Networks

1.3.1 Problem

Definition

The Disjunctive Temporal Network (DTN) formalism is an extension of STN by incorporating the use of disjunctive temporal constraints. These constraints enable the expression of more complex relationships between events.

For example, a DTN may include constraints such as "either event A must occur before event B, or event C must occur before event D." and non-overlap constraints such as : "the start time of A must be after the end time of B or the start time of B must be after the end time of A".

It is important to note that the use of DTN results in a way more complex network than the standard STN model. This added complexity allows for the reduction to the DTNs of other combinatorial problems such as the Job Shop Scheduling Problem or the Travelling Salesman Problem.

Example

The Mars Rover planning problem that we illustrated before with an STN can be made more complex and summarized as follows: the rover has to choose between different sequences of tasks while respecting different constraints, such as non-overlap constraints (e.g. the rover cannot film a panorama and drill at the same time) or conditional constraints (e.g. the rover must clean it has solar panels only if he spent less than one hour charging its battery).

The rover starts its day at 8am and needs to recharge its battery between 45 and 90 minutes. If it has charged its battery for less than 60 minutes, it can clean its solar panels before the next task, which takes 30 minutes. If not, it can go directly to the next step, which is to drive 1.5 hours to the place of experiments. There, it has two experiments to do in the order it wants: drilling rock and making a panorama of the site. Due to resource constraints, the rover cannot perform both tasks at the same time, and due to light constraints, the rover cannot film after 11:30am. Once the experiments are done, the rover has to drive 1.5 hours to the data transmission site to transmit the data to the satellite. The satellite is only within range of the rover for a short time, so the rover must transmit the data between 2 and 3pm.

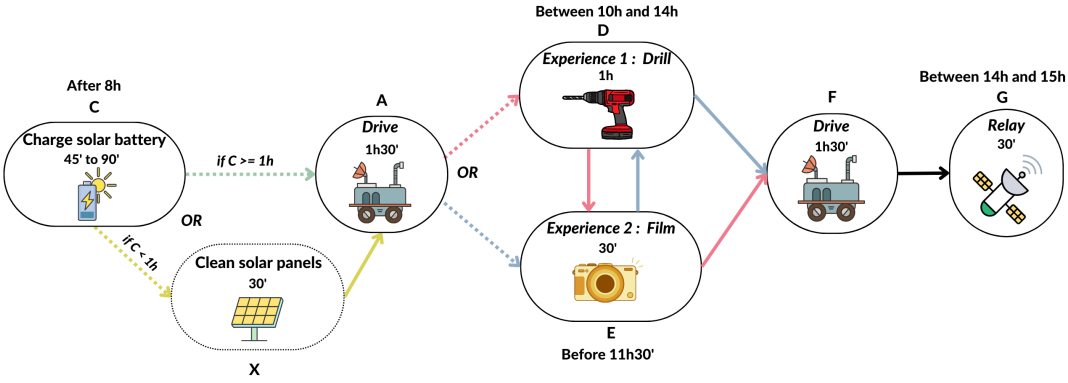

Figure 1.5: Illustrated example

Modeling this problem as a DTN will allow to answer the following questions :

- Is it possible to build a valid schedule such that the Rover arrives in time for the relay?
- Is it possible to build a valid schedule if the rover must drill before filming?

1.3.2 Formalism

Definition

A disjunctive temporal network is mathematically represented by a tuple (T, C) where T is a set of time points variables and C is a set of constraints. A constraint is a disjunction of binary constraints.

- $\forall C_i \in C$, C_i is a disjunction of binary constraints in the form $c_{i1} \vee ... \vee c_{in}$ where :
  - c_{ij} is the j^{th} disjunct of the i^{th} constraint.
  - c_{ij} is in the form $t_b - t_a \leq r_{ab}$ where :
    - $t_a, t_b \in T \subseteq \mathbb{R}$
    - $r_{ab} \in \mathbb{R}$
    - $\forall t_i \in T$, t_i generally represent the beginning or the end of an event.

Example

The events are represented by 2 time point variables, one for the start and one for the end. An arrow represents a constraint, for the choice we have two arrows representing the OR and we can visualize that as two different disjunctions. For the event duration, the arrow simply goes from the start to the end. The precedence encodes that an event has to happen before/after another. The non-overlap allows to encode that two events cannot happen at the same time, meaning that for both orders, the difference between the end time and the start time of the next event must be greater than 0.

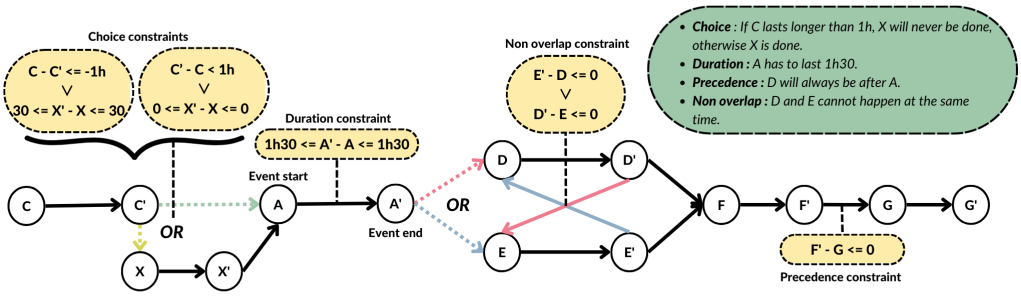

Figure 1.6: Translation of the example into DTN constraints

It is important to note that while the majority of constraints present in STN and DTN examples, such as precedence constraints defined between two events or duration constraints, remain the same, a layer of specificity is added with the incorporation of disjunctive constraints in DTN, which can take many different forms.

1.3.3 Solution

Intuition

One naive approach to solving the problem of DTN is to enumerate all the STNs within it and check if they are consistent. This approach can help to understand the combinatorial explosion of DTNs, as the number of STNs within a DTN increases exponentially with the number of disjunctive constraints between the events. However, this approach is computationally expensive and is not practical for solving large and complex DTNs.

Example

As we have 2 opportunities to make decisions and both have 2 alternatives, in this example, the number of STNs is simply equal to $2^2 = 4$. We can visualize the independent choices in the same color as the arrows in the previous Figure. For example the first STN will encode the state when X is not done because C lasts longer than 1h and when D happens before E.

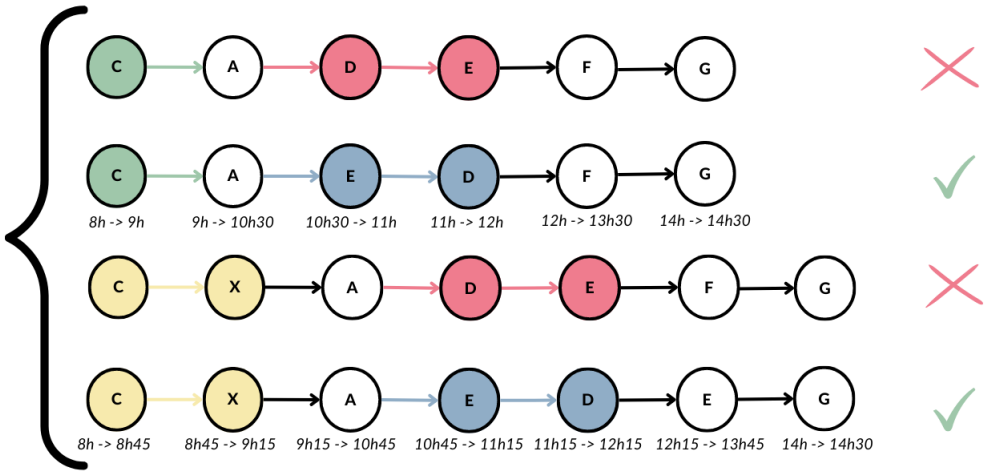

Figure 1.7: Illustration of the enumeration of all the STNs

Interpretation

By solving those STNs, we can answer the different questions:

- Is it possible to build a valid schedule such that the Rover arrives in time for the relay?
  - Yes, as two of the four STNs are consistent.
- Is it possible to build a valid schedule if the rover must drill before filming?
  - No, as none of the STNs that depict this scenario are consistent.

1.3.4 Properties

Definition 6 (Schedule)**.** A schedule is a specific assignment to all the time points variables $T_i \in T$. A schedule doesn't necessarily satisfy all the constraints.

Definition 7 (Solution)**.** An assignment to all time points variable $T_i \in T$ is called a solution if it satisfies all the constraints.

Definition 8 (Consistent)**.** A DTN is consistent if it exists at least one STN within the DTN which is consistent.

1.4 Reducing Combinatorial Problems to DTNs

After the detailed explanation of temporal networks and particularly DTNs, we shift our focus to a distinct characteristic of DTNs, their capacity to represent other combinatorial problems. DTNs, while primarily designed for temporal reasoning tasks, can also serve to model a variety of combinatorial problems due to their flexible structure and properties.

The translation of combinatorial problems to DTN formalism showcases the versatility of DTNs and introduces an alternative approach to understanding and solving these problems. By employing the temporal reasoning and constraint satisfaction methods inherent in DTNs, we can potentially tackle complex, NP-Hard problems that traditionally necessitate specific solutions. Transforming these problems into the DTN framework suggests a possibility of using a general approach to a spectrum of combinatorial challenges.

In the following subsections, we explore this concept further by discussing how some combinatorial problems, such as the Job Shop Scheduling problem, can be formulated and solved as DTN problems. This exploration aims to demonstrate the potential applicability of DTNs in a wider context.

1.4.1 Job Shop Scheduling

The nature of combinatorial problems, particularly those within the NP-Hard class, poses significant computational challenges. A prime example is the job shop scheduling problem, which despite its complexity, can be effectively modelled using the DTN formalism.

The job shop scheduling problem is a type of scheduling problem in which there are a set of jobs that need to be completed, each requiring a specific set of tasks to be performed on a set of machines. The goal of the job shop scheduling problem is to find an optimal schedule for the tasks that minimizes the overall completion time for all the jobs. This involves deciding the order in which the tasks should be performed and which machine should be used to perform each task.

The relationship between temporal networks and job shop scheduling is strong, as temporal networks are used to represent and solve scheduling problems. Therefore a DTN method can be used to solve the job shop scheduling problem by finding a feasible schedule for the tasks that satisfies all the constraints.

Example

Below is a job shop scheduling problem with M = 3 machines and N = 3 jobs. Each task is represented by a pair (m, p) where m is the number of the machine on which the task has to be done and p is the duration of the task.

- **job 0 = [(0, 3), (1, 2), (2, 2)]**
- **job 1 = [(0, 2), (2, 1), (1, 4)]**
- **job 2 = [(1, 4), (2, 3)]**

Solution

A resolution to a job shop problem entails allocating a start time for each task that adheres to the following constraints:

- Only one task can be executed on a machine at a given time.
- Tasks of the same job must be performed sequentially without overlap.
- Tasks must be completed in a single continuous operation.

The Figure 1.8 is a non-optimal solution for the above example.

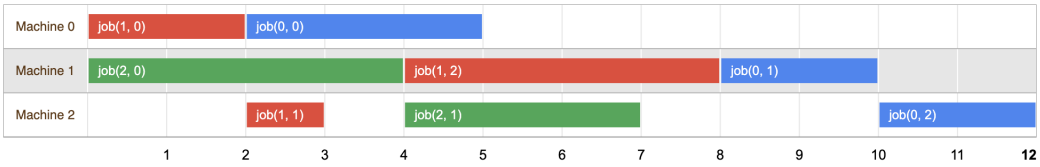

Figure 1.8: Non-optimal solution of the above Job shop example. Each color represents a job, and the pair $job(j, i)$ represents the i^{th} task of job j.

Disjunctive graph

The DTP and the Job Shop Scheduling Problem can both be represented as a disjunctive graph which is a graph that is used to represent constraints and requirements in scheduling problems. It consists of a set of vertices that represent tasks, and a set of edges that connect the vertices and encode the constraints between the tasks.

There are two types of edges in a disjunctive graph: conjunctive edges and disjunctive edges. Conjunctive edges represent dependencies between tasks that must be completed consecutively, while disjunctive edges represent dependencies between tasks that can be completed in parallel as long as they are scheduled on different resources.

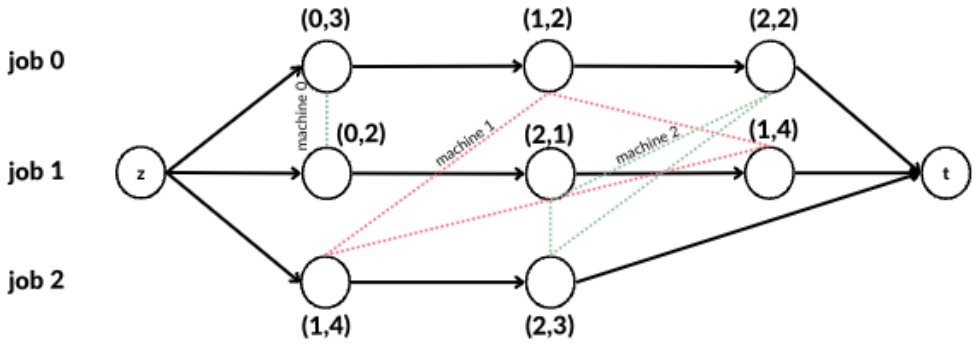

Figure 1.9: Disjunctive graph representation of the job shop example

The disjunctive graph, denoted as $G = (V, C \cup D)$, is defined as follows:

- V is a set of vertices that represent the tasks. In order to represent the start and end times, two fictitious vertices s and t are added. Each vertex in V is assigned a weight, which corresponds to the processing time for the task it represents. Both z and $t1$ have a weight of 0.
- C is a set of conjunctive edges that connect the i^{th} task to the $(i+1)^{th}$ task of a given job. In addition, conjunctive edges are added from s to the first task of each job and from the last task of each job to t. These edges are depicted as plain lines in the figure.
- D is a set of disjunctive edges that connect tasks that are to be processed on the same machine. These edges are depicted as dotted or dashed lines in the figure.

Reducing a job shop problem to a DTN problem

1. Enumerate all sub-tasks from index 0 to i, t_i will contain the start time of the task i.
    - ex: job[0, 0] has i = 0, job[1,0] has i = 3, job[2,1] has i = 7.

2. Transform job shop constraints

    - Job precedence (Binary constraints)
        - ex: $t_0 + 3 \leq t_1$ becomes $t_0 - t_1 \leq -3$
    - Machine overlap (Disjunction constraints)
        - ex: $t_0 + 3 \leq t_3 \vee t_3 + 2 \leq t_0$ becomes $t_0 - t_3 \leq -3 \vee t_3 - t_0 \leq -2$

3. Add source and sink constraints

    - Link source to all the first tasks of each jobs.
    - Link sink to all the last tasks of each jobs.
    - The constraint between source and sink can be used to optimize the objective value as it represents the end time of the problem.

4. Solve the DTN, any consistent STN represents a valid job shop schedule.

Chapter 2

State of the Art: Existing Approaches

2.1 Simple Temporal Networks Resolution

Resolving STNs involves finding their minimal network, which can be achieved by enforcing path consistency. Path consistency ensures that the direct edge between any two nodes reflects the shortest path between them. In this chapter, we focus on two levels of path consistency: Full Path Consistency (FPC) and Partial Path Consistency (PPC). The choice between these depends on the problem's specifics and available computational resources.

FPC considers all possible paths and solves the All-Pairs Shortest Paths (APSP) problem, resulting in a complete constraint graph. We will discuss three algorithms for enforcing FPC: the Floyd-Warshall algorithm, Johnson's algorithm, and the Snowball algorithm.

PPC, on the other hand, considers a subset of all possible paths, striking a balance between computational complexity and solution quality. We will explore the $\triangle$STP and P^3C algorithms that enforce PPC.

2.1.1 Full Path Consistency

Full Path Consistency is a network-wide property that is equivalent to "solving" an STN. This means that enforcing FPC on an STN is equivalent to determining the minimal network. However, it's important to bear in mind that enforcing FPC results in computing a complete constraint graph, which might not be suitable in all scenarios because of the computational complexity involved.

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm, independently introduced by Robert Floyd and Stephen Warshall in 1962 [19], is a classic algorithm for solving the APSP problem. Its simplicity and elegance have made it a popular choice for many applications, particularly those where the algorithm's efficiency isn't a critical aspect.

The Floyd-Warshall algorithm works by continuously updating a matrix that depicts the shortest path distances between all pairs of vertices. It begins with a matrix that includes the direct distances between all pairs of vertices, then progressively updates this matrix to account for the influence of intermediate vertices on the shortest paths.

Algorithm 1 Floyd–Warshall

```
1: Input: A STN S = (V, E)
2: Output: Minimal network of S
3: for k = 1 to n do
4:     for i = 1 to n do
5:         for j = 1 to n do
6:             w_ij = min{w_ij, w_ik + w_kj}
7:             return S
```

The algorithm has a time complexity of $O(n^3)$ and a space complexity of $O(n^2)$, where n is the number of nodes in the network

Johnson's Algorithm

Johnson's algorithm, introduced by Johnson in 1977 [20], is a sophisticated method for enforcing Full Path Consistency (FPC), i.e., computing APSP. This algorithm is particularly suited for sparse graphs.

Johnson's algorithm first reweights the graph to eliminate negative edge weights, a crucial step that allows the use of Dijkstra's algorithm. The reweighting process is carefully designed so that the shortest paths in the reweighted graph remain identical to those in the original graph.

One of the key features of Johnson's algorithm is its use of a heap data structure. This structure is used to implement a priority queue, where the priority is determined by the shortest path found so far to each vertex. The heap allows the algorithm to quickly select the next vertex to visit.

In terms of time complexity, Johnson's algorithm has a complexity time of $O(|V|^2 \log |V| + |V||E|)$, where $|V|$ and $|E|$ represent the number of vertices and edges, respectively [20]. This makes it more efficient than Floyd-Warshall algorithm for sparse graphs, where the number of edges is much less than the square of the number of vertices. However, while more efficient for sparse graphs, Johnson's algorithm is also more challenging to implement than Floyd-Warshall.

Snowball

The Snowball algorithm, proposed by Planken et al. in 2011 [14], represents the state-of-the-art in solving the APSP problem for STNs. It is particularly efficient for graphs with low tree widths.

The idea behind the Snowball algorithm is to gradually build up a group, or "clique", of vertices for which the shortest distances have been calculated. The algorithm starts with a single vertex and progressively adds one vertex at a time to the clique. When a new vertex is added, the algorithm computes the shortest distances between the new vertex and all other vertices within the clique.

The Snowball algorithm's time complexity is $O(n^2wd)$, where n is the number of vertices, w is the tree width of the graph, and d is the maximum degree of any vertex in the graph [14]. This makes it efficient for graphs where the tree width and the maximum degree are relatively small.

2.1.2 Partial Path Consistency

Proposed by Bliek and Sam-Haroud in 1999 [17], Partial Path Consistency is another path consistency concept that applies to the entire network. The introduction of PPC was motivated by the question of whether FPC could be approximated in a manner that preserves, to some extent, the sparsity of the constraint network, and whether this approximation might result in computational efficiency improvements.

The key difference between PPC and FPC lies in the trade-off between computational efficiency and the completeness of the solution. While FPC provides a complete solution, it results in a complete constraint graph, which may be computationally intensive. On the other hand, PPC provides a way to preserve the sparsity of the network and can be enforced more efficiently.

However, there are two special cases of queries for which an answer is available from a (complete) minimal STN, but not from the chordal[1] PPC network: (i) asking for a minimal constraint between an arbitrary pair of time points, not necessarily connected by a constraint; and (ii) assembling a schedule, i.e., dispatching the STN, along an arbitrary ordering of timepoints [12].

△STP

In 2003, Xu and Choueiry were the first to realize that the STP is a convex problem and that PPC therefore computes minimal constraints for it. They published a version of PPC specifically for the STP, called △STP [6].

The △STP algorithm was designed to maintain a queue of triangles, initially containing all triangles in the chordal graph, from which it dequeues triangles to process, one by one [6]. When a constraint edge in a triangle is changed, all other triangles containing that edge are enqueued (if they were not already). This process continues until the queue is empty.

Despite its straightforward and intuitive pseudocode, △STP has two potential weaknesses. First, the memory requirements of the algorithm may be large. The number of triangles in an STN has a tight upper bound of $O(nw_d^2)$, and these triangles must all be stored in memory, at least at the beginning of the algorithm's execution. In the case of, almost, complete graphs, where the treewidth w_d is in the order of the number of nodes n, the memory requirement of △STP becomes $\Omega(n^3)$ [6]. This implies that for large networks or networks with a high degree of interconnectivity, the memory needed to store all triangles can grow cubically with the number of nodes.

Secondly, the queueing process followed by the algorithm may cause triangles to be processed many times. While the authors of △STP did not initially provide a theoretical upper bound on the performance of the algorithm, Planken et al. (2008) showed that in pathological cases, the algorithm may need time quadratic in the number of triangles, leading to a worst-case time complexity of $\Omega(n^4)$ which is worse than Floyd Warshall [13].

[1] A chordal graph, also known as a triangulated graph, is a type of graph in which every cycle of four or more nodes has a chord, which is an edge that is not part of the cycle but connects two nodes of the cycle.

$\mathbf{P^3C}$

In response to the limitations and high computational complexity of $\triangle$STP, Planken and de Weerdt introduced a new algorithm in 2008, called the Partial Path Three Consistency (P^3C) algorithm [13]. The P^3C algorithm is a refinement of the $\triangle$STP algorithm and is designed to address its weaknesses.

Similar to $\triangle$STP, the P^3C algorithm starts from a chordal constraint graph. However, the key difference between the two algorithms lies in their time complexity. While the time complexity of $\triangle$STP is quadratic in the number of triangles in the worst case, the time complexity of P^3C is linear in the number of triangles. This is achieved by introducing a novel approach to managing the queue of triangles, which allows the algorithm to avoid processing the same triangle multiple times.

In terms of spatial complexity, the P^3C algorithm also has a significant advantage over $\triangle$STP. While $\triangle$STP requires memory storage that is cubic in the number of nodes in the worst case, the memory requirements of P^3C are linear in the number of triangles. This leads to a significantly reduced memory footprint for large networks or networks with a high degree of interconnectivity.

Formally, the P^3C algorithm is defined as follows:

In step 6, a triangle is considered inconsistent if there exists a shorter path between two of its vertices through the third vertex. If such a path exists, the edge is updated in step 7 to reflect this shorter path.

The P^3C algorithm has been shown to outperform $\triangle$STP significantly in practice. This makes it a promising choice for solving STPs, especially in scenarios where the constraint graph has a large number of nodes or a high degree of interconnectivity.

Algorithm 2 P3C Algorithm

1: **Input:** An STN $S = (V, E)$
2: **Output:** S P3-consistent
3: Initialize an empty queue Q of triangles
4: **for** each triangle t in S **do**
5: Add t to Q
6: **end for**
7: Order triangles in Q according to a predefined strategy
8: **while** Q is not empty **do**
9: Dequeue a triangle t from Q
10: **for** each edge e in t **do**
11: **if** the edge e is inconsistent with the triangle t **then**
12: Make the edge e consistent
13: **for** each triangle t' containing the updated edge e and not in Q **do**
14: Add the triangle t' to Q
15: **end for**
16: **end if**
17: **end for**
18: **end whilereturn** S =0

2.2 Disjunctive Temporal Networks Resolution

In this section, we elaborate on the evolution of strategies for resolving DTNs. We begin by examining the traditional backtracking method, a simple and effective, but computationally intensive approach for solving DTNs.

Next, we focus on enhanced backtracking methods, where we present advancements such as semantic branching and no-goods that aim to accelerate the search process and improve pruning.

Lastly, the section reviews diverse existing strategies beyond backtracking, including Satisfiability Modulo Theories (SMT), Mixed-Integer Linear Programming (MILP), and SAT-based approaches. These different approaches each offer unique advantages and trade-offs, which are comprehensively compared and discussed.

2.2.1 Simple Backtracking

The initial approaches to solving DTNs were based on a strategy that explored the STN tree using backtracking. This strategy was first introduced in [2], alongside a random generator model designed to standardize the benchmarking strategy. Further research was conducted in [4], which proposed different strategies such as backjumping and various forms of forward-checking.

In the simple backtracking approach, the algorithm starts by choosing a disjunctive constraint from the network. It then begins a search process, adding new constraints (or disjuncts) to the network one at a time. After each new constraint is added, the algorithm checks the network for consistency using a path consistency algorithm[2]. If the network remains consistent, the algorithm continues to the next constraint. If an inconsistency is found, the algorithm tries a different disjunct. If all disjuncts lead to inconsistency, the algorithm backtracks, meaning it goes back to the previous constraint and tries a different disjunct.

Let's consider a simple example with three disjunctive constraints:

- $D(1) : x1 - x2 \leq 3 \vee x1 - x2 \geq 5$
- $D(2) : x2 - x3 \leq 2 \vee x2 - x3 \geq 4$
- $D(3) : x3 - x1 \leq 1 \vee x3 - x1 \geq 3$

[2]See Section 2.1 for STNs path consistency checking algorithms details.

The backtracking algorithm would start by selecting the first disjunct of $D(1)$, i.e., $x1 - x2 \leq 3$. It would then proceed to $D(2)$, selecting its first disjunct, i.e., $x2 - x3 \leq 2$. If this leads to a consistent network, it would then proceed to $D(3)$. If at any point the addition of a new disjunct leads to an inconsistent network, the algorithm would backtrack and try the other disjunct. This process continues until a consistent network is found or all possibilities have been exhausted.

This backtracking approach was initially popular due to its simplicity, it was capable of effectively solving a wide variety of problems. However, it also had its limitations. In particular, the backtracking approach is computationally intensive, especially for larger and more complex networks.

2.2.2 Enhanced Backtracking

The evolution of techniques for solving DTNs has been driven by the need to accelerate the search process and improve pruning. This evolution has led to the development of more efficient algorithms that can handle larger and more complex networks. One of the key advancements in this area is the introduction in [5] of semantic branching and no-goods.

Semantic Branching is a pruning method that relies on the semantics of the constraints in the DTN, i.e., the fact that they encode numeric inequalities. The basic idea of semantic branching is as follows: suppose during the search, the assignment $A \cup \{C_i \leftarrow c_{ij}\}$ is expanded in every possible way but it leads to no solution. That means that in any solution that is an extension of A, if there is any, the constraint c_{ij} does not hold. Thus, the negation of this constraint has to hold in any such solution. In other words, if c_{ij} is the constraint $x - y \leq b$ and we know c_{ij} does not hold, then in any solution that is an extension of assignment A, $\neg c_{ij}$ has to be true, i.e., it must be the case that $y - x < -b$. Thus, when search "branches" after failing to extend $A \cup \{C_i \leftarrow c_{ij}\}$ to a solution, and tries a different value for C_i for the rest of the search under A, we can assume $\neg c_{ij}$ holds. The constraint $\neg c_{ij}$ often tightens the STN that corresponds to the current assignment A; explicitly adding this constraint can lead to values in other variable domains being removed earlier than they otherwise would have been.

No-Goods are essentially assignments that are known to lead to failure, i.e., they cannot result in a solution. The algorithm explores all extensions of a given partial assignment and either returns a solution or a justification for the failure of all extensions of the assignment, known as a no-good. The process of generating no-goods starts at the leaf nodes of the search tree. If the current assignment at

a leaf node violates a constraint, a no-good is returned. This no-good includes the assignment and the variables involved in the violated constraint. The use of no-goods helps to prune the search space by avoiding exploration of assignments that are guaranteed to lead to failure. However, the process of checking the current assignment against all recorded no-goods can be computationally expensive. Therefore, it's important to manage the number of no-goods recorded during the search.

Epilitis integrates various pruning methods such as Conflict Directed Backjumping, Removal of Subsumed Variables, Semantic Branching, and No-good Recording. The algorithm initiates by eliminating subsumed variables, those that have a disjunct already satisfied by the current assignment, thereby negating the need to explore other values in the variable's domain under the assignment. It then assesses whether all variables have been assigned. If so, it declares the assignment as a solution. If not, it selects a variable and inspects all values in the current domain of the chosen variable. The algorithm then conducts forward-checking on each value. If this fails, it records a no-good, a combination of a forbidden assignment and a reason for the failure. If forward-checking is successful, the algorithm recursively calls itself with the new assignment, the remaining unassigned variables, and the updated STN. If all values of the selected variable result in failure, the algorithm adds a semantic branching constraint to the STN and proceeds with the search with the remaining unassigned variables. The Epilitis algorithm is designed to efficiently prune the search space leveraging the no-goods recording, thereby reducing the computational complexity of solving DTNs. Epilitis has been shown to achieve a speed-up of almost two orders-of-magnitude over the previous fastest algorithm.

2.2.3 Different Approaches

Satisfiability Modulo Theories (SMT) is a technique that has been introduced in [22] as a promising approach to solve DTNs. An experimental evaluation [21] confirms that SMT are a state-of-the-art way to solve DTNs. It involves reducing a DTN instance to a formula in Quantifier-Free Real Difference Logic (QF RDL).

MILP approach to solving DTN involves rewriting conditional constraints using binary variables and the big-M technique. This means that each constraint is translated into a system of linear inequalities over real and binary variables. The union of these systems models the original instance. MILP can perform very well on some random instances, but it needs to be handled carefully in order to limit the magnitude of big-M and understand which cuts really help.

SAT approach to solving DTNs was for the first time introduced in [23] and involves translating the temporal constraints into Boolean formulas, which can then be solved using a SAT solver. This involves encoding each time point as a Boolean variable and each constraint as a Boolean formula. The SAT solver then tries to find a satisfying assignment of Boolean values to the variables that satisfies all the constraints. If such an assignment exists, it represents a feasible schedule for the DTN. If not, the DTN is unsolvable.

Conclusion: In the evaluation conducted in [21], a comprehensive comparison was made among a variety of solvers, including six SMT solvers, four MILP solvers, three SAT solvers, and three CP solvers. These were tested across multiple benchmark sets. The key findings from the study can be summarized as follows:

- SMT is the most effective technology for addressing these types of problems.
- MILP also showed promising results. In some random instances, it was up to twice as fast as SMT.
- SAT was found to be less efficient due to the complexity and size of the circuits. It performed reasonably well only on a specific benchmark with a specific encoding.
- CP did not perform as well. The only exception was Gecode, which, when used on random DTN instances, required 31.84% less time compared to SMT.

In the experimental evaluation, CP was tested with basic configurations, which resulted in less than optimal performance. However, in this thesis, we aim to delve deeper into the study of CP to explore potential improvements. This will be discussed in detail in the subsequent sections of this thesis.

Chapter 3

Constraint programming

Constraint Programming (CP) has become very popular in recent years as it provides a powerful approach for problem solving, particularly for combinatorial problems. The beauty of this paradigm lies in its simplicity: problems are represented as a set of variables and constraints, and it's the computer's task to find a solution.

In fact, it has been stated by E. Freuder that "Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."

3.1 Definitions

3.1.1 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined by a set of variables, each associated with a domain of possible values, and a set of constraints restricting the combinations of these variables. The formal definition of a CSP employs the notation (X, D, C) where:

- $X = x_1, x_2, \ldots, x_n$ is the set of variables.
- $D = D(x_1), D(x_2), \ldots, D(x_n)$ is a function that assigns to each variable its domain, or the set of possible values that the variable can assume.
- $C = c_1, c_2, \ldots, c_m$ is a set of constraints, each one being a relation involving some variables and restricting the values that these variables can simultaneously take

In this context, a solution to a CSP is an assignment of values to all variables from their respective domains in such a way that all constraints are satisfied.

3.1.2 Constrained Optimization Problem

A Constrained Optimization Problem (COP) extends a CSP by integrating an objective function that is to be optimized. This function could represent a cost to be minimized or a utility to be maximized. A COP is expressed as a tuple (X, D, C, f), where :

- (X, D, C) forms a CSP
- $f : D(X_1) \times D(X_2) \times \cdots \times D(X_n) \to \mathbb{R}$ is the objective function.

A solution to a COP is a value assignment to the variables that satisfies all constraints and optimizes the objective function [25].

3.2 Consistency

Consistency is a fundamental concept in CP that aids in pruning the search space and hence reducing the computation time. The notion of consistency is related to the level of local satisfaction of the constraints. Two main types of consistency relevant to CP are Domain Consistency and Bound Consistency [24].

Domain Consistency

A constraint is domain consistent if for every variable in its scope, every value in the current domain of that variable can be extended to a solution for the constraint. In other words, for each value in the domain of a variable, there exists a combination of values for the remaining variables such that all these values together satisfy the constraint.

For example, consider a binary constraint $X = Y + 1$ where $D(X) = \{2, 3, 4\}$ and $D(Y) = \{1, 2, 3\}$. This constraint is domain consistent because, for each value in the domain of X, there exists a value in the domain of Y such that $X = Y + 1$ is satisfied, and vice versa.

Bound Consistency

Bound consistency, on the other hand, is a weaker form of domain consistency. A constraint is bound consistent if for every variable in its scope, any value within the bounds of the current domain of that variable can be extended to a solution for the constraint. This does not guarantee that all values can be extended to a solution, only those at the boundaries.

Consider another example: the constraint $X = Y + 1$ where $D(X) = \{2, 3, 4\}$ and $D(Y) = \{1, 3\}$. This is an example of bound consistency. The smallest and the largest values in $D(X)$, which are 2 and 4, respectively, can be extended to a solution for the constraint, as can the smallest and the largest values in $D(Y)$, which are 1 and 3, respectively. Note that we are not considering values inside the bounds, such as $X = 3$, for which there is no corresponding value in $D(Y)$ that satisfies the constraint. In this case, despite not being domain consistent, the constraint is still bound consistent.

In practice, enforcing domain consistency generally provides more pruning of the search space but at a higher computational cost, while bound consistency offers less pruning at a lower cost. The choice between them is an important trade-off based on the specific requirements and constraints of the problem at hand.

3.3 Propagation

Constraint propagation refers to the process of iteratively enforcing consistency on the constraints of a problem until no more changes can be made, i.e., a fix-point is reached. This process narrows the domain of the variables and often leads to a reduction in the size of the search space, making the problem easier to solve. The general principle of constraint propagation can be implemented through various algorithms, among which the fix-point algorithm is a commonly used method.

The pseudo-code of the fix-point algorithm for constraint propagation can be formulated as follows:

Algorithm 3 Fix-point Algorithm

```
1: while there is a constraint c in C that is not consistent do
2:     Enforce consistency on c
3:     if the domain of any variable in c is reduced then
4:         Mark all constraints involving that variable as not consistent
5:     end if
6: end while
```

In practice, constraint propagation does not guarantee finding a solution to the constraint satisfaction problem, but it is an effective way to prune the search space.

3.4 Search

Despite the fact that consistency and propagation can significantly reduce the problem's search space, they often fall short of solving the problem entirely. Therefore, a systematic search strategy is required to explore the remaining search space. The search strategy involves choosing a variable and a value from its domain, and then dividing the problem into smaller subproblems based on this decision. This division process is commonly known as branching.

The success of the search process in CP relies heavily on these three core concepts: branching, variable selection, and value selection. The following subsections detail each of these concepts.

3.4.1 Branching

Branching in constraint programming is the process of making decisions that divide the problem into smaller subproblems, each representing a different potential solution. These subproblems are subsequently solved, typically using a depth-first search approach. Each branching operation creates a node in the search tree, and the solution to the problem (if it exists) is a node in this tree.

Binary Branching A single branch in the search tree is created by selecting a variable and a value from its domain, and then dividing the problem into two subproblems: one in which the selected variable is equal to the chosen value (called the "try" branch), and another where the selected variable cannot take the chosen value (referred to as the "fail" branch).

n-ary Branching Instead of the traditional binary branching, alternative branching strategies can be considered. One such strategy is n-ary branching, where n is the number of values in the selected variable's domain. In this approach, each branch corresponds to assigning a different value from the domain to the variable, effectively dividing the problem into n subproblems. This strategy can prove efficient in certain types of problems where the domain of the variable is small, but it can quickly become unmanageable if the domains are large.

Domain Splitting Another branching strategy is domain splitting, also known as interval splitting, which is especially useful when dealing with continuous or large domains. In this strategy, instead of choosing a specific value from the domain, the domain of the selected variable is split into two subdomains, creating

two branches. Each branch represents a subproblem where the selected variable's domain is restricted to one of the subdomains.

As an example, consider a variable x with a domain $D(x) = [1, 10]$. If we apply domain splitting for branching, we could split the domain into two subdomains, say $[1, 5]$ and $[6, 10]$. This creates two branches: one where $D(x) = [1, 5]$ and another where $D(x) = [6, 10]$. Each branch then represents a different subproblem to be solved.

Choosing a suitable branching strategy is problem-dependent and can significantly influence the efficiency of the search process and the overall performance of the constraint programming approach.

3.4.2 Variable Selection

The variable selection strategy is a key aspect of the search process. It determines the order in which the variables are selected for branching. This order can significantly affect the efficiency of the search process, as it impacts the size of the search tree and thus the computational cost of the search.

First-Fail is a strategy that suggests selecting the variable with the smallest domain, is a popular strategy. The underlying idea of this approach is that a failure is more likely to occur for a variable with fewer choices, and it's generally more efficient to encounter a failure earlier in the search rather than later.

Most Constrained Variable is a heuristic that proposes selecting the variable that appears in the largest number of constraints. The logic behind this approach is that this variable is the most likely to cause a failure, and hence, selecting it early can lead to a more efficient search by failing fast.

In problems with a specific structure or additional knowledge, problem-specific heuristics can be developed for variable selection. These heuristics exploit the problem's special characteristics to guide the variable selection and can often lead to substantial performance improvements.

3.4.3 Value Selection

The value selection strategy is another crucial step in the search process, refers to choosing a value from the domain of the selected variable during branching. The sequence in which these values are chosen can significantly impact the overall

efficiency of the search process, as it directly influences the structure and size of the search tree.

Two commonly employed strategies are the "minimum value first" heuristic and the "most constrained value" heuristic.

Minimum Value First is a heuristic that selects the smallest possible value from the variable's domain first. Beneficial for its simplicity and intuitive approach, its effectiveness can vary depending on the problem. Nonetheless, it is a practical choice in scenarios where smaller values are preferred or a natural value order exists.

Most Constrained Value is a heuristic that chooses the value that seems to be the most constraining for the remaining variables. This heuristic operates on the fail-first principle, i.e., it's more efficient to discover failures sooner rather than later, thereby potentially reducing the size of the search tree.

Example Consider a graph coloring problem with three regions A, B, and C left to be assigned colors from the set $C = \{Red, Green, Blue\}$. The available colors due to existing constraints are as follows:

- $D_A = \{Green, Blue\}$
- $D_B = \{Red, Green, Blue\}$
- $D_C = \{Red, Blue\}$

Using the Most Constrained Value heuristic, we prioritize the value which is applicable to the fewest remaining variables. Here, 'Blue' is the most constrained and would be assigned first. This heuristic aims to reduce the search tree size and increase efficiency by discovering failures early.

3.4.4 Backtracking

Backtracking operates via depth-first traversal of the search tree representative of a constraint satisfaction problem.

The process initiates at the root node, symbolising the problem's initial state, and explores successor nodes corresponding to distinct decision sequences. If the examination of a node reveals adherence to all constraints, thereby denoting a solution, the algorithm terminates, returning this node. Conversely, upon the encounter of a node incapable of developing into a valid solution, termed a 'dead-end', the algorithm reverts or 'backtracks' to the preceding node and inspects its subsequent unvisited successors.

3.5 Solvers

One of the key elements in CP is the notion of models. Models are the formal representations of problems in a manner that computational techniques, including CP, can understand and solve.

In this thesis, we will principally use two tools for the purpose of model construction and problem-solving: the MiniZinc framework and the MiniCP solver library. They will be used individually to solve DTNs, allowing for the comparison of results and the evaluation of different solver's strengths and weaknesses.

3.5.1 MiniZinc Framework

MiniZinc is a versatile medium-level constraint modelling language [26]. Its design centers around providing a solver-agnostic environment, allowing the user to abstract away from the specifics of individual solvers and focus on describing the problem.

Due to its design, MiniZinc is compatible with a multitude of solvers. Through the compiler, MiniZinc models can be translated into FlatZinc, a solver-input language digestible by numerous solvers. Notable solvers compatible with MiniZinc include Gecode, Chuffed, and OR-Tools, all of which offer state-of-the-art performance for various problem classes.

3.5.2 MiniCP Solver Library

In parallel to MiniZinc, we will employ the MiniCP solver library, a compact tool for defining and solving CP models. MiniCP is designed with a focus on providing a simplistic yet robust architecture for research and educational pursuits in CP [8].

Chapter 4

Transforming DTNs into Optimization Problems

4.1 Transition from CSP to COP

Transforming a CSP into a COP requires the introduction of an objective function, which provides a criterion for distinguishing between 'better' and 'worse' solutions. While the goal of a CSP, defined formally as a triple $\langle X, D, C \rangle$, is to find a solution that satisfies all constraints, the COP goes a step further. It not only seeks a solution that meets all constraints but also optimizes an objective function.

$$CSP = \langle X, D, C \rangle \tag{4.1}$$

The objective function f quantifies the quality of a solution, essentially reflecting real-life goals. It thus facilitates a move from a basic CSP, where all satisfying solutions are deemed equal, to a COP, defined as a quadruple $\langle X, D, C, f \rangle$, where satisfying solutions can be ranked based on the value of f.

$$COP = \langle X, D, C, f \rangle \tag{4.2}$$

This transition becomes particularly useful when dealing with DTNs. By converting the disjunctive constraints into an optimizable form, it allows for a more nuanced approach. Our goal transitions from merely finding a feasible solution to locating an optimal or near-optimal solution with the aid of relaxation techniques.

4.2 Soft Constraints and Relaxation in DTNs

Transitioning from CSP to COP entails the introduction of an objective function, with the aim of not just finding a solution, but optimizing it. In this context, it is essential to comprehend the distinction between hard and soft constraints, and the concept of relaxation.

Traditionally in CSP, constraints are viewed as hard constraints, indicating their absolute requirement for satisfaction. A violation of a hard constraint invalidates a solution, rendering it infeasible [25]. However, many practical scenarios may necessitate allowing certain constraints to be violated to amplify flexibility and enhance the chances of finding feasible solutions. Such accommodating constraints are known as soft constraints.

In the context of DTNs, consider a hard constraint such as 'Event A must occur no more than 5 minutes after Event B'. If no solution exists that respects this constraint, the problem is deemed infeasible.

To improve the probability of finding a feasible solution, we can transition this hard constraint into a soft one. The transformation allows for a certain degree of violation of the original constraint. For instance, 'Event A must occur no more than 5 minutes after Event B' could be adjusted to 'Event A should preferably occur no more than 5 minutes after Event B but may occur later if necessary'. The latter formulation incorporates flexibility in the constraint, accommodating solutions where Event A occurs more than 5 minutes after Event B.

With this modification, the objective function now includes a penalty for each violation of the original constraint. While the objective function seeks solutions respecting the original constraint as much as possible, it also allows for scenarios where the constraint may be violated.

This strategy, known as relaxation, serves as a potent tool in addressing intractable problems by making them more manageable. As we dive deeper into this transition, two relaxation strategies stands out for handling the softening of constraints in DTNs, namely the Boolean Switch Strategy and the Slack Variable strategy. These strategies will be explained in the next section.

4.3 Encoding and Relaxation of the COP Model

Building upon the formalism of DTNs and the relaxation strategies discussed in the previous sections, we now set out to encode our DTN into a COP while applying the relaxation strategies of Boolean Switch and Slack Variables.

As a refresher, a DTN is represented by a tuple (T, C), with T standing for a set of time-point variables and C denoting a set of constraints, each of which is a disjunction of binary constraints. Each binary constraint c_{ij} is of the form $t_b - t_a \leq r_{a_b}$ where $t_a, t_b \in T$ and $r_{ab} \in \mathbb{R}$, representing the constraint on the difference of two time points.

4.3.1 Boolean Switch

To incorporate the Boolean Switch strategy, we first enhance our formalism to accommodate the binary variables associated with each disjunction, resulting in a tuple (T, C, B) where B represents a set of binary variables b_i for each constraint C_i.

We assign $b_i = 0$ if the disjunction C_i is respected, and $b_i = 1$ if it is violated. In this context, each constraint C_i becomes a disjunction:

$$b_i = 1 \vee (c_{i_1} \vee c_{i_2} \vee ... \vee c_{i_n}) \tag{4.3}$$

where c_{i_j} represents the j^{th} disjunct of the i^{th} constraint.

The objective function, now updated to accommodate the Boolean Switch strategy, aims to minimize the summation of all binary variables:

$$\min_{\mathbf{T},\mathbf{B}} \sum_{i=1}^{|C|} b_i \tag{4.4}$$

4.3.2 Slack Variable

The incorporation of the Slack Variable strategy demands the introduction of slack variables to our DTN formalism. Our enhanced formalism now becomes a tuple (T, C, S) where S denotes a set of slack variables s_{ij} for each disjunct c_{ij}.

Each binary constraint c_{ij} now transforms into $t_b - t_a \leq r_{ab} + s_{ij}$, where s_{ij} represents the violation extent of the j^{th} disjunct of the i^{th} constraint.

When we transition to the usage of slack variables, the natural question arises as to how to incorporate these into the objective function. An intuitive approach would be to minimize the sum of the slack variables as this would mean minimizing the total extent of constraint violations. A second approach, slightly more nuanced, might be to minimize the sum of the maximum slack variable for each disjunction instead, thereby trying to limit the most severe constraint violation. These two strategies can be represented as:

$$\min_{\mathbf{T},\mathbf{s}} \sum_{i=1}^{|C|} \sum_{j=1}^{|C_i|} s_{ij} \tag{4.5}$$

and

$$\min_{\mathbf{T},\mathbf{s}} \sum_{i=1}^{|C|} \max_{j}^{|C_i|} s_{ij} \tag{4.6}$$

respectively. Here, $\mathbf{T}$ represents the set of time points and $\mathbf{s}$ denotes the slack variables.

However, the naive incorporation of slack variables into the objective function presents a distinct challenge in the context of DTNs. Recall that a DTN is characterized by disjunctions; simply summing the slacks of each disjunct in a disjunction may not be appropriate as some disjuncts are expected to be false.

To address this issue, we enhance our model with a 2D array of boolean variables that capture the status of each disjunct in the network. If we denote this array as B', our DTN formalism becomes (T, C, S, B'), with each boolean variable $b_{ij} \in B'$ associated with a disjunct c_{ij}.

In the extended model, c_{ij} is activated when $b_{i_j} = 1$, and deactivated otherwise. Hence, each constraint C_i transforms into a disjunction of the form :

$$\bigvee_{j=1}^{|C_i|} (b_{ij} \wedge (t_b - t_a \leq r_{ab} + s_{ij})) \tag{4.7}$$

Integrating the boolean variables into the objective function, we aim to minimize the sum of the slack variables associated only with the activated disjuncts:

$$\min_{\mathbf{T},\mathbf{S},\mathbf{B'}} \sum_{i=1}^{|C|} \sum_{j=1}^{|C_i|} b_{i_j} \cdot s_{i_j} \tag{4.8}$$

Thus, by using this strategy, we're able to maintain the balance between enhancing solution feasibility through relaxation and preserving the integrity of the original problem.

4.4 Search and Optimization Strategies

When navigating the search space of our encoded and relaxed DTN, we employ various strategies to find a feasible or nearest feasible solution under the optimization context. In this section, we will delve into the Branch and Bound Depth-First Search (DFS) technique, variable selection heuristics, and value selection heuristics.

4.4.1 Branch and Bound DFS

Branch and Bound DFS is a popular method used to explore the search space efficiently. It performs a depth-first search but prunes branches of the search tree that are guaranteed not to contain a better solution than the best one found so far. This pruning action allows us to avoid exploring unpromising portions of the search space.

Given the objective function $f : \mathbb{T} \to \mathbb{R}$, and the best solution found so far T^* with a value $f(T^*)$, any partial assignment T' that cannot be extended to a complete assignment T such that $f(T) < f(T^*)$ is pruned.

4.4.2 Variable Selection Heuristics

We used different variable heuristics to select which variable to assign next, notably First-Fail, Max-Regret, and Most-Constrained. It's worth noting, however, that the effectiveness of these heuristics can be problem-dependent. Given our objective to remain generic and robust across various benchmarks, we can't necessarily rely on our understanding of a specific problem to hand-craft the optimal heuristic. Instead, we need to choose heuristics that are generally effective across a range of problem instances.

First-Fail The First-Fail heuristic selects the variable with the smallest domain. The reasoning behind this is that variables with smaller domains are more likely to cause failure early in the search, hence should be assigned sooner.

Given a state of the CSP with unassigned variables X', we select the variable $x \in X'$ that minimizes $|D_x|$, where D_x is the domain of the variable x. Formally, the variable to assign next is given by:

$$x^* = \arg\min_{x \in X'} |D_x|$$

Max-Regret The Max-Regret heuristic selects the variable that, if incorrectly assigned, would cause the greatest loss in the quality of the solution. The regret of a variable x, denoted by $R(x)$, is defined as the difference between the smallest value and the second smallest value in D_x that do not violate any constraints in C, mathematically given by:

$$R(x) = \min_{d \in D_x, C \cup \{x=d\}} d - \min_{d \in D_x, d \neq \min_{d \in D_x}, C \cup \{x=d\}} d \tag{4.9}$$

The variable with the maximum regret is then selected as the variable to assign next:

$$x^* = \arg\max_{x \in X} R(x) \tag{4.10}$$

Most-Constrained The Most-Constrained heuristic selects the variable that is involved in the highest number of constraints. This approach is guided by the principle that variables with the most constraints are the hardest to assign.

Given a state of the CSP with unassigned variables X', we select the variable $x \in X'$ that maximizes the count of constraints $c \in C$ that involve x. That is, the variable to assign next is given by:

$$x^* = \arg\max_{x \in X'} |\{c \in C : x \text{ is involved in } c\}| \tag{4.11}$$

4.4.3 Value Selection Heuristics

The value selection heuristic deals with the order in which the values in the domain of the selected variable are tried. Similar to variable selection, the effectiveness of a value heuristic can be problem-specific, making it challenging to identify the best heuristic for all problem instances given our commitment to a generic solution. Despite this, certain strategies like Min Value and Random Value can prove to be beneficial in a wide range of scenarios.

Min Value The Min Value heuristic selects the smallest value in the domain of the variable. The underlying logic is that it is often easier to satisfy constraints with smaller numbers.

Given a variable x with unassigned value and its domain D_x, the Min Value heuristic selects the value $d \in D_x$ that is the smallest. That is, the value to assign to the variable is given by:

$$d^* = \min_{d \in D_x} d \tag{4.12}$$

Random Value The Random Value heuristic, as the name suggests, selects a value randomly from the variable's domain. Although it might not seem strategic, it can often lead to diverse search paths, which may be beneficial in certain problem instances.

Given a variable x with unassigned value and its domain D_x, the Random Value heuristic selects a value $d \in D_x$ uniformly at random. The value to assign to the variable is given by:

$$d^* \sim U(D_x) \tag{4.13}$$

where $U(D_x)$ denotes a uniform random selection from the set D_x.

These search and optimization strategies, along with the relaxation strategies discussed earlier, help to increase the efficiency and robustness of our approach when solving DTNs under the COP framework.

4.5 Metaheuristic and Fine Tuning

To enhance the efficiency of solving DTNs under the COP framework, Large Neighborhood Search (LNS) is introduced as a metaheuristic. In essence, LNS focuses on exploring large portions of the search space, but only modifies a small subset of variables, thereby creating a balance between exploration and exploitation. The process involves two main steps: relaxation and search. This is followed by a discussion of threshold strategies which finely control the exploration and exploitation trade-off.

4.5.1 Large Neighborhood Search

The first step of LNS is relaxation, where a subset of the problem is chosen and the remaining variables are fixed at their current value. This forms a neighborhood around the current solution. Then, search is performed in this neighborhood to possibly find a better solution. LNS iteratively performs these steps, varying the neighborhood, until no improvement can be found or a stopping criterion is met [32].

Given a DTN (T, C), a neighborhood is a subset of T. This neighborhood $\mathcal{N}$ is then used to define a reduced problem $(T \setminus \mathcal{N}, C)$, which is solved to find a new assignment of $T \setminus \mathcal{N}$ that satisfies C. This assignment is then combined with the assignment of $\mathcal{N}$ from the previous solution to form a new solution to the original DTN.

4.5.2 Relaxation and Failures Thresholds

Control of the LNS process is maintained via a pair of thresholds: relaxation threshold and failures threshold. They are important parameters of our generic approach, aimed to be applicable across various DTN benchmarks.

The relaxation threshold determines the size of the neighborhood during relaxation, i.e., the proportion of variables to be kept fixed. It is computed based on a lower bound ($relLB$), an upper bound ($relUB$), and a steepness parameter ($steepness$), all of which control how quickly the threshold changes based on the number of consecutive iterations without finding an improving solution ($lastImprove$):

$$\text{relaxThreshold} = 1 - (\text{relLB} + (\text{relUB} - \text{relLB}) \times \exp(-\text{steepness} \times \text{lastImprove})) \tag{4.14}$$

As $lastImprove$ increases, the function's exponent becomes increasingly negative, causing the relaxation threshold to slowly increase, leading to smaller neighborhoods, and hence a more in-depth, localized search.

The failures threshold determines the number of failures to allow during the search process. Similarly to the relaxation threshold, it is computed using a lower bound ($failLB$), an upper bound ($failUB$), and a steepness parameter ($steepness$):

$$\text{nbFailuresThreshold} = \text{failLB} + (\text{failUB} - \text{failLB}) \times \exp(-\text{steepness} \times \text{lastImprove}) \tag{4.15}$$

Here, as $lastImprove$ grows, the failure threshold decreases, which means the search process becomes less tolerant to failure, thus intensifying the search.

If the search process goes through more than 100 consecutive iterations without finding an improving solution, $lastImprove$ is reset to zero to favor exploration over exploitation.This reset mechanism ensures that if the solution is not improving, the strategy shifts back towards exploration, allowing larger neighborhoods and a higher tolerance for failure.

These thresholds dynamically balance the exploration and exploitation trade-off, allowing the metaheuristic to adapt to the problem's characteristics. The goal of this adaptive mechanism is to be efficient on different benchmarks. Please refer to the figures below for a visualization of the threshold evolution over iterations.

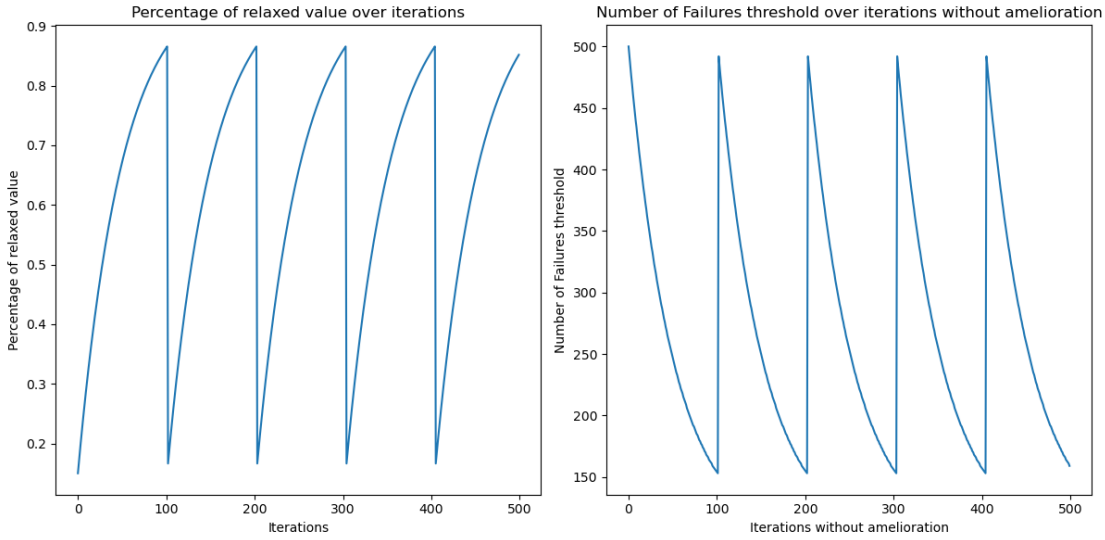

Figure 4.1: Relax and Failure thresholds over iterations

Chapter 5

Experiments

5.1 Experimental Design and Methodology

Our experiments were conducted to assess the efficacy of DTNs relaxation into COP, utilizing an array of configurations comprising different solvers, variable selection heuristics, and satisfaction models. The experimental environment was hosted on a Mac M1 Pro 2021 machine with 16 GB of RAM and 10 threads. The DTN models were encoded using both MiniCP and MiniZinc 2.7.1 and were solved using three different solvers: Gecode 6.3.0, Chuffed 0.11.0, and MiniCP.

In our experiment, we examined a diverse set of configurations, which are summarized in Table 5.1. These configurations are essentially divided into two major categories.

The first category focuses on optimization and uses the Boolean Switch strategy (B.S.). This category is composed of three different solver configurations (MiniCP, Chuffed, Gecode), each paired with two variable selection heuristics, First Fail and Max Regret, making a total of six configurations. In each of these, we employed the min-value heuristic.

The second category is concerned with satisfaction, and it also uses three solver configurations (MiniCP, Chuffed, Gecode). However, unlike the first category, each of these configurations uses the input order variable selection heuristic, using the min-value heuristic for value selection which are the default value of MiniZinc for satisfaction. This results in three configurations for this category.

Thus, combining both categories, we tested a total of nine configurations. These configurations were evaluated using a subset of 100 randomly selected instances for

each of the four benchmarks. Detailed descriptions of these benchmarks will be provided in the subsequent section.

Category	**Solver**	**Var. Heur.**	**Val. Heur.**
Opti. (B.S.)	MiniCP, Chuffed, Gecode	First Fail, Max Regret	Min Val
Sat.	MiniCP, Chuffed, Gecode	Input Order	Min Val

Table 5.1: Experimental Configurations

One crucial aspect to note is that we did not include slack encoding in our experimental configurations. While slack encoding could potentially offer alternative approaches to solve the problem, we encountered substantial challenges when attempting to utilize it. The large search space due to numerous branching, and the difficulty of defining a representative objective function resulted in trouble in obtaining decent solutions. Thus, to maintain the focus and quality of our experiments, slack encoding was not included .

The experimental process was guided by a set of parameters including relaxation and failure thresholds. The value chosen for these parameters is discussed in the previous chapter, *4.5 Metaheuristic and Fine Tuning* where we describe how we ensure adaptability to a variety of benchmarks.

Several metrics were collected during the experimental process to evaluate the performance of the models. For each instance, we noted whether the problem was solved, whether a feasible schedule existed, and the number of relaxed constraints at the time each solution was found. Considering the substantial number of configurations and instances, we set a timeout of 60 seconds for each instance, ensuring the experiments were executable in a reasonable time frame.

These experimental outcomes provide insights into the solvers performance and efficiency in handling DTNs under the COP framework with relaxation. The results, to be discussed in the following sections, assist in identifying the most effective configurations for solving DTNs.

5.2 Benchmarks

5.2.1 industrial-1-dtp

This benchmark set includes a total of 233 instances which correspond to the set CSTNBenchmark2016 from the three sets proposed in [28]. In essence, this benchmark is made of Conditional Simple Temporal Networks (with Uncertainty) which have been adjusted in [21] to fit our context. These adjustments were made by considering all uncontrollable components as controllable, converting these sets into Simple Temporal Networks with Decisions (STNDs). These components of STNDs are then translated into DTN instances.

5.2.2 dimacs-gc-tcsp

This benchmark is made up of 63 instances selected from the renowned DIMACS graph-coloring challenge [29], specifically instances with known optimal coloring. A custom converter was developed in [21] to transform these instances from their decision-problem form into TCSP instances.

5.2.3 orrz-dtp

This benchmark set consists of 200 instances created using the balanced SAT construction detailed in [30]. The instances are generated using two parameters: the number of variables n and a multiplier k, which is used to generate $k * (n - 1)$ clauses. $k = 3.6$ to generate hard instances, with one instance for each n = 11,..., 210. These SAT problems are then converted into DTN instances using a reduction technique [21].

5.2.4 fhcpcs-tcsp

This benchmark set consists of 100 instances taken from the FHCP Challenge [31]. These instances are structurally challenging representations of the Hamiltonian Cycle Problem and range in size from 66 to 9528 vertices, with an average size of just over 3000 vertices. These instances were converted into TCSP instances using a reduction [21]. This set is the most challenging among the benchmarks, with each instance larger than the last and admitting a Hamiltonian Cycle.

5.3 Results

5.3.1 industrial-1-dtp

As observed in Table 5.2 and Figure 5.1, for the industrial-1-dtp benchmark, the satisfaction strategy outperforms the Boolean Switch (B.S.) optimization strategy across all solvers. This could be indicative of the structure of industrial-1-dtp, adapted from Conditional Simple Temporal Networks, which might be more amenable to the satisfaction strategy.

Solver	Strategy	Heuristic	N
gecode	relaxed	max_regret	0
		first_fail	0
	sat	default	34
minicp	relaxed	max_regret	15
		first_fail	3
	sat	default	94
chuffed	relaxed	max_regret	34
		first_fail	6
	sat	default	73

Table 5.2: Solutions found for industrial-1-dtp

Within this context, MiniCP appears to be the most effective solver, significantly excelling under the satisfaction strategy (see Table 5.2). This superiority might be attributable to MiniCP's use of the LNS metaheuristic that facilitates diverse search space exploration.

Gecode, contrastingly, fails to solve any instances when paired with the B.S. optimization strategy, as indicated in Table 5.2. This suggests a possible mismatch between Gecode's methodologies and the B.S. strategy within this benchmark.

When looking at the constraint satisfaction percentages over time (Figure 5.1), it is notable that MiniCP and Chuffed perform better than Gecode, and that the Max Regret heuristic surpasses First Fail. These trends align with the instance-solving performance, hinting at the potential efficacy of Max Regret and LNS within the scope of this benchmark.

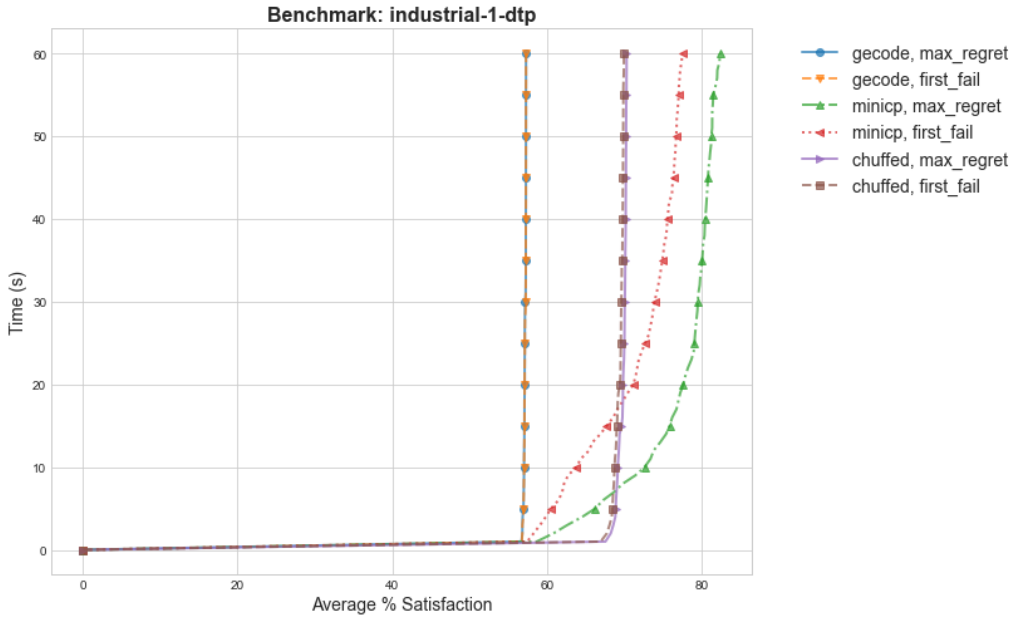

Figure 5.1: Comparison of solvers on industrial-1-dtp

5.3.2 dimacs-gc-tcsp

In the dimacs-gc-tcsp benchmark, we observe that the choice between satisfaction and optimization strategies no longer determines solver performance uniformly. As evidenced in Table 5.3 and Figure 5.2, Gecode solver lightly achieves a higher number of solved instances with the satisfaction strategy compared to both B.S. optimization strategies. In contrast, the differences in performance of MiniCP and Chuffed under different strategies are less pronounced.

Solver	Strategy	Heuristic	N
gecode	relaxed	max_regret	19
		first_fail	24
	sat	default	34
minicp	relaxed	max_regret	25
		first_fail	27
	sat	default	30
chuffed	relaxed	max_regret	19
		first_fail	20
	sat	default	30

Table 5.3: Solutions found for dimacs-gc-tcsp

MiniCP remains a strong performer (see Table 5.3) which is likely a benefit of the LNS metaheuristic, with its effectiveness particularly notable under the First Fail heuristic as showed in Figure 5.2. Despite this, Gecode shows competitive results, particularly when utilizing the First Fail heuristic under the B.S. strategy.

This analysis of the dimacs-gc-tcsp results reveals the sensitivity of solver and strategy performance to the nature of the benchmark problem.

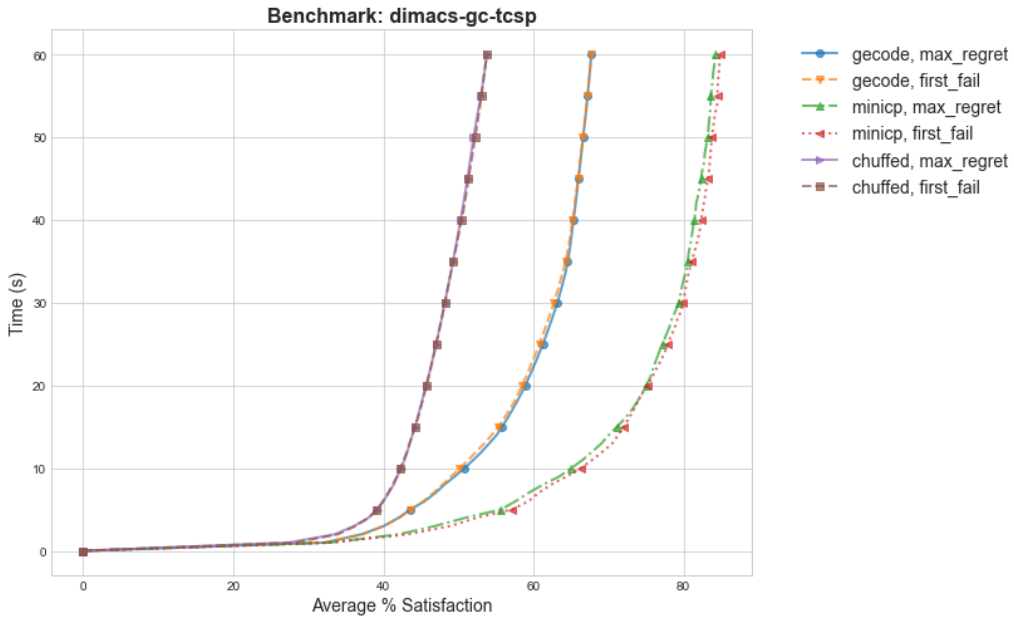

Figure 5.2: Comparison of solvers on dimacs-gc-tcsp

5.3.3 orrz-dtp

The orrz-dtp benchmark presents a different situation from the previous benchmarks. All three solvers demonstrated similar poor performance levels when employing the satisfaction strategy, each solving only a single instance as shown in Table 5.4. Interestingly, no solver could solve any instance under the B.S. optimization strategies, regardless of the heuristic employed.

Solver	Strategy	Heuristic	N
gecode	relaxed	max_regret	0
		first_fail	0
	sat	default	1
minicp	relaxed	max_regret	0
		first_fail	0
	sat	default	1
chuffed	relaxed	max_regret	0
		first_fail	0
	sat	default	1

Table 5.4: Solutions found for orrz-dtp

Considering the percentage of satisfied constraints over time (Figure 5.3), all three solvers demonstrated high levels of satisfaction relatively quickly, with differences in performance becoming more subtle over time. The LNS metaheuristic of MiniCP was again better.

This disparity leads us to the hypothesis that the apparent inconsistency between the number of found solutions and the number of violated constraints might be due to the fact that these instances simply lack a feasible solution. In such scenarios, the relaxed version assists us in approximating a solution that doesn't exist. Therefore, the observation is that the algorithms are relatively efficient at approaching an almost optimal solution, as they all quickly reach a minimum of 90% constraint satisfaction.

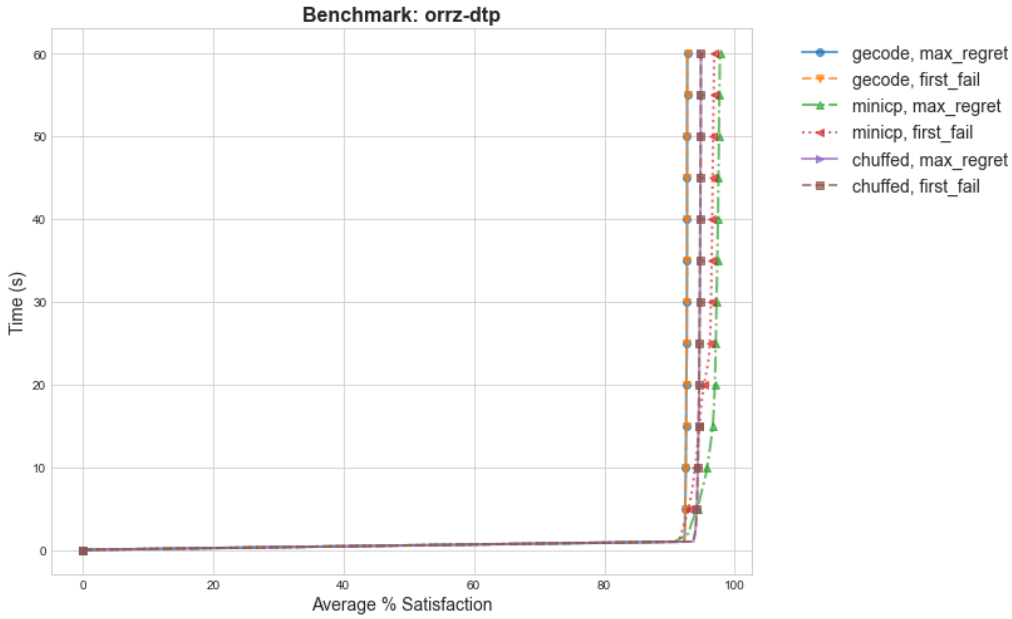

Figure 5.3: Comparison of solvers on orrz-dtp

5.3.4 fhcpcs-tcsp

In the fhcpcs-tcsp benchmark, all three solvers were unable to solve any instances, regardless of the strategy or heuristic applied, as presented in Table 5.5. This is an unusual situation but the benchmark is very challenging and exposes the limitations in the strategies and optimization techniques employed.

Solver	Strategy	Heuristic	N
gecode	relaxed	max_regret	0
		first_fail	0
	sat	default	0
minicp	relaxed	max_regret	0
		first_fail	0
	sat	default	0
chuffed	relaxed	max_regret	0
		first_fail	0
	sat	default	0

Table 5.5: Solutions found for fhcpcs-tcsp

Looking at the plot of satisfaction percentages over time (Figure 5.4), it becomes clear that this benchmark is very difficult. For the most part, the solvers managed to increase the satisfaction levels as time progressed, but at a much slower rate than in the previous benchmarks.

Under the B.S. optimization strategies, the MiniCP solver was the best performer, reaching approximately 21.81% satisfaction with the Max Regret heuristic and 24.78% with First Fail in the 60 seconds cut-off time. The Gecode solver showed slightly lower satisfaction levels, reaching 17.01% and 16.81% with Max Regret and First Fail, respectively.

However, Chuffed solver was only able to reach around 9.54% satisfaction with Max Regret and 9.44% with First Fail. This significant dip in performance further demonstrates the variability of solver performance across different benchmarks.

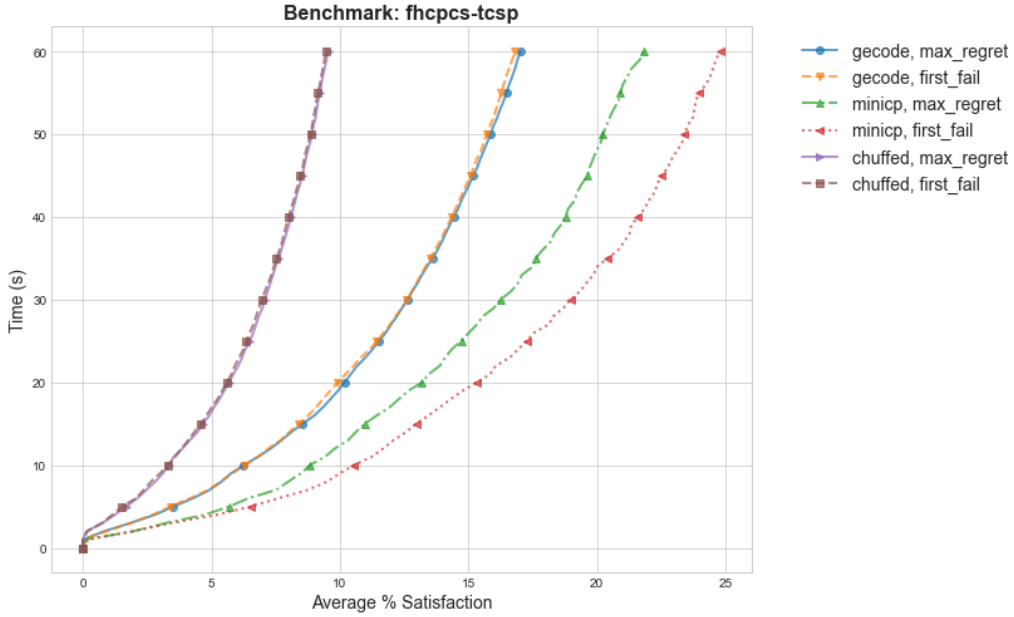

Figure 5.4: Comparison of solvers on fhcpcs-tcsp

5.4 Discussion

Our experiments extend the findings of the work done by [21], providing further insights into the performance and suitability of different CP solver configurations for various DTN benchmarks. Despite the more restricted time limit of 1 minute per instance, as opposed to the 10 or 20 minutes allowed in [21], our results consistently attained similar orders of solutions, indicating efficiency improvements in our approach.

Our methodology also contributes to have a richer understanding of solver performance by considering the degree of constraint satisfaction, rather than just classifying if a solution has been found or if the search timed out. This additional metric presents a deeper understanding of solver performance and reveals how fast and how far a solver can proceed into the search, and then how close it can be from a solution.

For instance, this methodology paid off significantly in our experiments for the fhcpcs-tcsp benchmark. Although [21] were unable to find any solution after a 20-minute search, we were able to indicate that a certain configuration of the MiniCP solver managed to satisfy approximately 25% of the constraints in just a 1-minute search, even if a full solution was not reached.

We also observed that the performance of CP solvers varies widely across different benchmarks, which reflects the diversity of DTN problem structures. The industrial-1-dtp benchmark seemed particularly amenable to the satisfaction strategy, while all the configurations seemed to get very close results for the orrz-dtp.

The effectiveness of a heuristic, such as Max Regret, can be highly context-dependent and potentially less beneficial in certain benchmarks, such as dimacs-gc-tcsp. This underscores the importance of understanding the specific characteristics of a given DTN problem for choosing the most suitable solver configuration.

In essence, the diverse nature of the benchmarks invites us to think more creatively and innovatively about our approaches to problem-solving. Could there be other strategies, heuristics or relaxation techniques yet to be explored that could be more effective? Could a deeper understanding and fine-tuning of the configurations yield improved performance across the varying benchmarks?

Chapter 6

DTN CP Solving Tool

6.1 Overview

This section introduces a tool developed as part of this master's thesis, designed to facilitate the resolution of DTNs using CP. The tool, implemented in Python and utilizing the PyQt5 framework for its graphical user interface, provides a user-friendly platform for experimenting with various solver configurations on DTN instances and receiving real-time feedback about the solving process.

Facilitate the testing of different solver configurations: The tool allows users to select solvers and configurations from those we tested in the previous sections. This includes Chuffed, Gecode, and Mini-CP, each with its strengths and characteristics, and apply them with any configuration to the DTN instances of their choice. This flexibility simplifies the process of identifying the most effective solver and configuration for a given problem.

CSP and COP: The tool offers the capability to approach the problem from two distinct perspectives: as a satisfaction problem and as a minimization problem. The first one focuses on finding a solution that satisfies all constraints, while the second one relaxes the disjunction constraints and tries to minimize the number of violated disjunctions. Along with sometimes being more effective than directly searching for a valid solution, this approach provides flexibility and allows the user to have a "good-enough" solution when the problem is over-constrained or challenging to solve.

Real-Time Feedback: The tool allows tracking the progress of the execution. It displays the constraints violated during the solving process, providing insights into the nature of the problem and the performance of the solver. Additionally, the

tool includes a plot that allows users to visualize the evolution of the minimization process, offering a clear overview of the solver's progress.

6.2 User Workflow

In this section, we will walk through the main steps of using the tool to summarize the experience of a typical user.

6.2.1 Open the tool

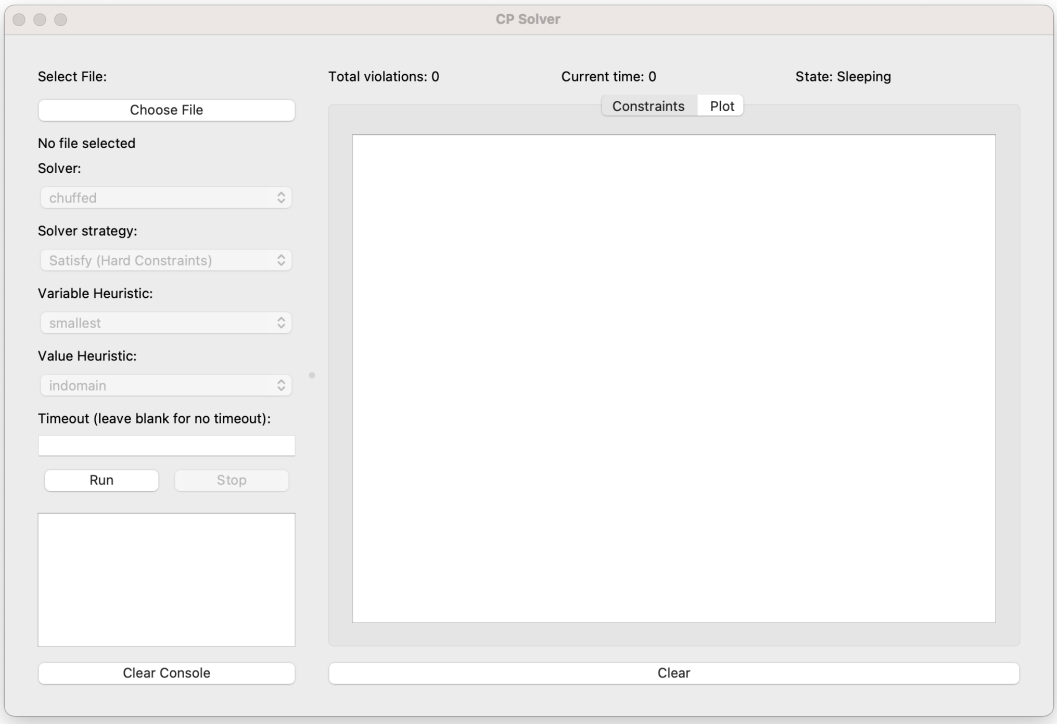

Figure 6.1: Opening view

6.2.2 Upload a file

The tool is designed to accept two types of files: SMT encoded DTN instances as described in [21] and MiniZinc files (.mzn). When a file is uploaded, the disjunctions are shown in the right table, and the configuration panel is enabled.

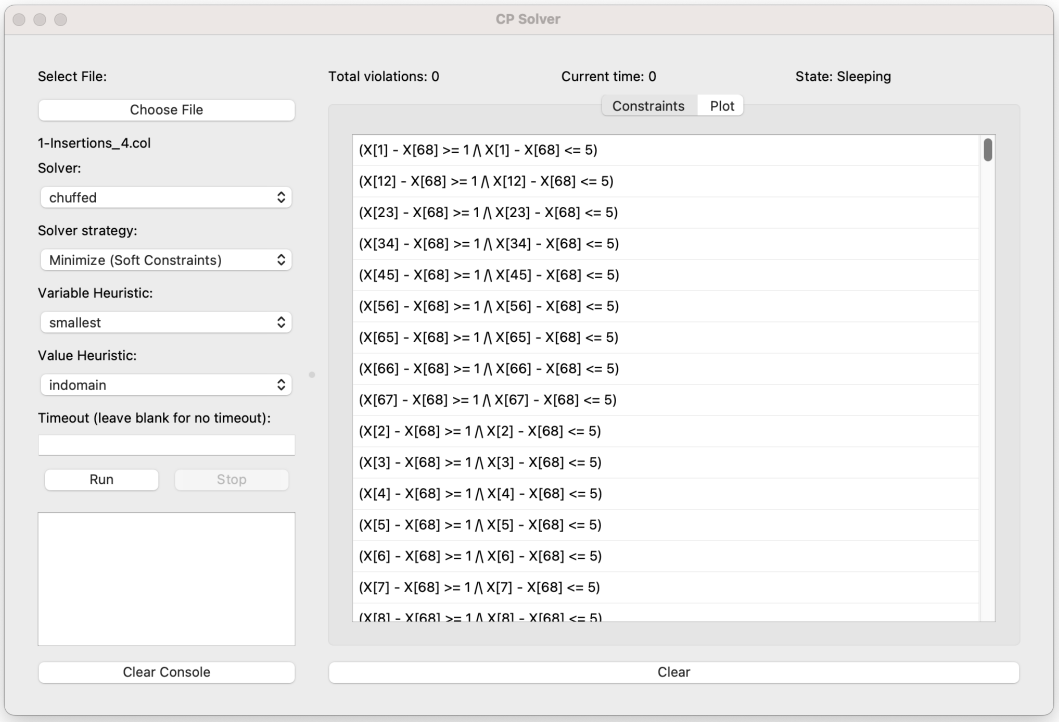

Figure 6.2: File imported

6.2.3 Configure the resolution

1. Select a solver: The available solvers are *Chuffed*, *Gecode*, and *Mini-CP*.

2. Select a strategy: The two available strategies are *Satisfy (Hard constraints)* and *Minimize (Soft constraints)*. If the minimize option is selected, the choice of heuristics for variable and value selection is enabled.

3. Select a variable heuristic *(minimize only)*: The available variable heuristics depend on the selected solver. For Mini-CP, the options are: *first_fail*, *max_regret*, and *most_constrained*. For the other solvers, the list includes: *smallest*, *largest*, *first_fail*, *anti_first_fail*, *most_constrained*, *occurrence*, *input_order*, *max_regret*, *impact*, and *dom_w_deg*.

4. Select a value heuristic *(minimize only)*: The available value heuristics also depend on the selected solver. For Mini-CP, the option is: *indomain_min*. For

the other solvers, the options include: *indomain*, *indomain_min*, *indomain_max*, *indomain_interval*, *indomain_reverse_split*, *indomain_split*, *indomain_split_random*, *outdomain_max*, *outdomain_median*, *outdomain_min*, and *outdomain_random*.

5. Enter a timeout limit: The timeout limit is in seconds and essentially stops the execution of the solver when the time has elapsed.

6.2.4 Execute the solver and analyze the results:

Press the run button, when the solver is running, the tool provides real-time feedback in four different ways, the first three of which are:

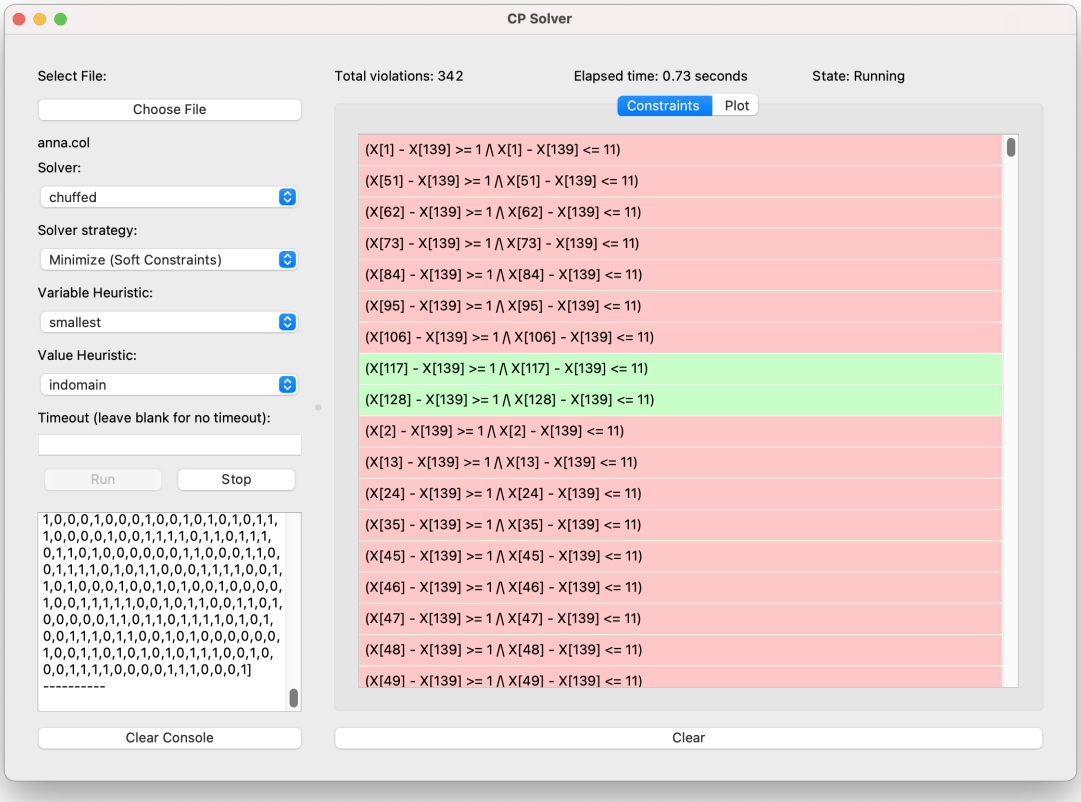

Figure 6.3: During the execution of the solver

- The bottom-left console: It will display information such as the intermediate and final solutions, or the constraints violations array.

- The right panel constraint table: In the case of a minimization, it will show in real-time which constraints are satisfied and which are not.
- The top-right labels: During every execution, the live elapsed time will be displayed along with the total number of violations in case of minimization.

Below the labels on the top-right panel, there are two tabs: *Constraints* and *Plot*.

The *Plot* tab leads to the fourth way of viewing information:

- The plot that displays the number of violations against the elapsed time in real-time. Each time a solution is found, it is represented by a dot at the point in time when it was discovered. This provides insights about the progression of the search and highlights the parts that may be more challenging.

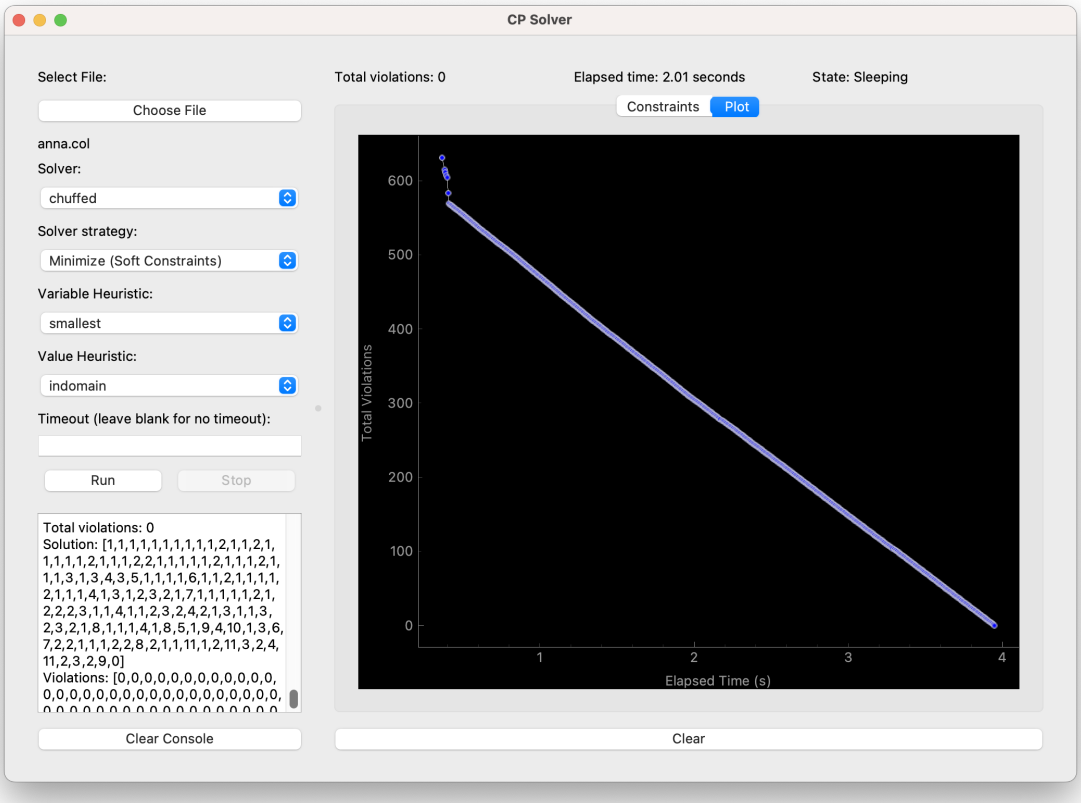

Figure 6.4: Number of violations over elapsed time plot

This concludes the demonstration of the tool's main features, highlighting its ability to solve different DTN problems using various solvers and configurations, and displaying real-time information during the resolution of the problems.

6.3 Future Perspectives

While the tool already implements some features, we have also envisioned some potential enhancements for the future. Here, we will describe additional valuable features that could be implemented into the tool.

Visualizing DTN as a Disjunction Graph: One potential enhancement is the addition of a feature to visualize the DTN as a disjunction graph. This would provide a graphical representation of the problem, making it easier to understand the relationships between different constraints.

Allowing different types of file input: Currently, only two types of files can be inputted into the tool. It would be highly beneficial to accept different types, such as DTNs in pure mathematical notation or in JSON. However, this requires substantial work in accepting various structures and mapping them.

Fine-tuning Large Neighborhood Search: Another potential improvement is the ability to fine-tune an LNS. By adding the ability to fine-tune LNS within the tool, users could experiment with different restart and neighborhood relaxation settings and find the configuration that yields the most optimal solution for their problem.

Tabs for different instances: Allowing users to execute different instances, solvers, or configurations simultaneously could be beneficial. This could be done by allowing users to create tabs and navigate through them. To extract the most value from this feature, it's crucial to ensure that when a tab is not active, its execution is not stopped.

Automated Configuration Testing: The tool could also benefit from a feature that quickly tests different configurations and proposes a potentially optimal configuration for more extended execution. This would save users time and effort in manually testing different configurations and could lead to better solving performance.

Conclusion

In this work, we have delved deeper into the resolution of Disjunctive Temporal Networks problems using Constraint Programming. Building on the experimental evaluations conducted by Zavatteri et al. [21], we have applied specific CP techniques to further this research.

We have transformed DTN problems into Constrained Optimization Problems using a Boolean Switch relaxation. Our experimental evaluations have shown that for all benchmarks, we achieve results of similar magnitudes in significantly less time than the basic CP models presented in [21].

Moreover, we have been able to gain more insights for certain benchmarks where Zavatteri could only achieve a timeout with a classic satisfaction problem. Relaxation allows us to obtain suboptimal solutions and approach a "good enough" solution. For some specific benchmarks, relaxation has led to solutions that satisfy up to 90% of the constraints, a significant improvement over solvers seeking satisfaction that only proposed a timeout.

Our experimental results show that the MiniCP solution, which we have pushed the most, notably with a Large Neighborhood Search, obtains the best results, followed by Chuffed and finally Gecode. However, it's important to note that our results vary across different benchmarks. While we have achieved good results for some benchmarks, others, such as the challenging fhcpcs, have room for improvement. This variation in results underscores the complexity of DTN resolution and the potential for further enhancements in our approach.

We have also developed a tool to help testing various solver configurations easily aiming to find a satisfying solution for an instance. It allows real-time visualization of the constraints that are violated during relaxation and the evolution of the number of violated constraints over time on a plot. The tool integrates the solvers Chuffed, Gecode, and Mini-CP, along with the strategies we have developed, providing a user-friendly platform for DTN resolution.

While we have made some progress, the results also highlight the complexity of the problem and the need for further research. We hope that this work will inspire further exploration in the area of DTNs.

Bibliography

[1] R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks", in Artificial Intelligence, vol. 49, no. 1-3, pp. 61-95, 1991.

[2] K. Stergiou and M. Koubarakis, "Backtracking algorithms for disjunctions of temporal constraints", in Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98/IAAI 98, Madison, Wisconsin, USA, July 26-30, 1998, pp. 248-253.

[3] A. Oddi and A. Cesta, "Incremental Forward Checking for the Disjunctive Temporal Problem", in Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000), Berlin, Germany, August 20-25, 2000, pp. 108-112.

[4] K. Stergiou and M. Koubarakis, "Backtracking algorithms for disjunctions of temporal constraints", Artificial Intelligence, vol. 120, no. 1, pp. 81-117, June 2000.

[5] I. Tsamardinos and M. Pollack, "Efficient solution techniques for disjunctive temporal reasoning problems", Artificial Intelligence, vol. 151, no. 1-2, pp. 43-89, December 2003.

[6] L. Xu and B. Y. Choueiry, "A New Efficient Algorithm for Solving the Simple Temporal Problem", in Proceedings of the CSE Conference and Workshop Papers, Computer Science and Engineering Department, University of Nebraska, 2003.

[7] M. Wilson, T. Klos, C. Witteveen, and B. Huisman, "Flexibility and decoupling in Simple Temporal Networks", Artificial Intelligence, vol. 214, pp. 26-44, September 2014.

[8] L. Michel, P. Schaus, and P. Van Hentenryck, "MINICP: a lightweight solver for constraint programming", Mathematical Programming Computation, vol. 13, pp. 133-184, February 2021.

[9] D. Pisinger and S. Ropke, "Large Neighborhood Search," in Handbook of Metaheuristics, pp. 399-419, September 2010.

[10] P. Schaus, "Variable Objective Large Neighborhood Search: A Practical Approach to Solve Over-Constrained Problems", in Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, 2013, pp. 971-978.

[11] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language", in Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2007), Lecture Notes in Computer Science, vol. 4741, pp. 529-543.

[12] L. Hunsberger and R. Posenato, "Simple Temporal Networks: A Practical Foundation for Temporal Representation and Reasoning", in Proceedings of the 28th International Symposium on Temporal Representation and Reasoning (TIME 2021), vol. 206, pp. 1:1-1:5, Dagstuhl, Germany, September 2021.

[13] LR Planken, MM de Weerdt, and RPJ van der Krogt, "P3C: A New Algorithm for the Simple Temporal Problem", in Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), Menlo Park, CA, USA, American Association for Artificial Intelligence (AAAI), pp. 256-263, 2008.

[14] L. Planken, M. de Weerdt, and R. van der Krogt, "Computing All-Pairs Shortest Paths by Leveraging Low Treewidth", Journal of Artificial Intelligence Research, vol. 43, pp. 353-388, 2012.

[15] L. R. Planken, "Algorithms for Simple Temporal Reasoning", Dissertation Series No. 2013-42, SIKS, the Dutch Research School for Information and Knowledge Systems, ISBN 97890-6562-337-9, published by Delft Academic Press, 2013.

[16] J. O. A. ten Thije, "Towards a Dynamic Algorithm for the Simple Temporal Problem", Algorithmics Group, Department of Software Technology, Faculty EEMCS, Delft University of Technology, Delft, the Netherlands, 2011.

[17] C. Bliek and D. Sam-Haroud, "Path Consistency on Triangulated Constraint Graphs", in Proceedings of the International Joint Conference on Artificial Intelligence, 31 July 1999.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms", 3rd edition, MIT Press, 2009.

[19] R. W. Floyd, "Algorithm 97: Shortest path," Communications of the ACM, vol. 5, no. 6, Jun. 1962.

[20] B. Johnson. Efficient algorithms for shortest paths in sparse networks. Journal of the ACM, 24(1):1–13, 1977. ISSN 0004-5411.

[21] M. Zavatteri, A. Raffaele, D. Ostuni, and R. Rizzi, "An interdisciplinary experimental evaluation on the disjunctive temporal problem", Constraints, vol. 28, Feb. 2023.

[22] A. Cimatti, A. Micheli, and M. Roveri, "Solving Temporal Problems Using SMT: Strong Controllability", in Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2012), Lecture Notes in Computer Science, vol. 7514, pp. 248-264.

[23] A. Armando, C. Castellini, and E. Giunchiglia, "SAT-Based Procedures for Temporal Reasoning," in Lecture Notes in Computer Science, September 1999.

[24] P. Schaus, "Constraint Programming Course (LINFO2365), Louvain-la-Neuve, 2021-2022," Available at: https://uclouvain.be/cours-2021-linfo2365.

[25] F. Rossi, P. van Beek, and T. Walsh, Handbook of Constraint Programming, 1st ed. Elsevier Science, 2006

[26] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language", in Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2007), Lecture Notes in Computer Science, vol. 4741, pp. 529-543.

[27] P. Meseguer, "Towards 40 years of constraint reasoning", Prog. Artif. Intell., vol. 1, pp. 25-43, 2012.

[28] R. Posenato, "CSTNU Tool: A Java library for checking temporal networks," in SoftwareX, vol. 17, 2022, Art. no. 100905.

[29] D. J. Johnson and M. A. Trick, Cliques, coloring, and satisfiability: second DIMACS implementation challenge. Providence, RI: American Mathematical Society, 1996.

[30] I. Spence, "Balanced random SAT Benchmarks," in SAT COMPETITION 2017, 2017, p. 53.

[31] M. Haythorpe, "FHCP challenge set: the first set of structurally difficult instances of the hamiltonian cycle problem," Bull ICA, vol. 83, pp. 98-107, 2018.

[32] P. Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in Lecture Notes in Computer Science, vol. 1520, pp. 417-431, Presented at the Fourth International Conference on Principles and Practice of Constraint Programming (CP-98), 1998.

Appendix

Code repositories

1. DTN CP Solving Tool

The GitHub repository is public and provides all the necessary information for installing and running the tool : https://github.com/mehdi-bh/dtn-tool

2. Experiments

The GitHub repository is public and contains all the scripts and results used and showcased in the experimental evaluations : https://github.com/mehdi-bh/dtn-experiments

3. MiniCP

The Bitbucket repository, associated with the course LINFO2365: Constraint Programming, is not public. If you are interested in consulting it, please request access from either of us:

- mehdi.benhaddou@student.uclouvain.be
- eliot.hennebo@student.uclouvain.be

Once we've secured the necessary permissions from Pierre Schaus, we'll grant you access.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**