UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VINICIUS DE ANTONI

An Asynchronous Algorithm to Improve Scheduling Quality in the Multiagent Simple Temporal Problem

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Álvaro Moreira
Advisor

Porto Alegre, January 2014

CIP – CATALOGING-IN-PUBLICATION

De Antoni, Vinicius

An Asynchronous Algorithm to Improve Scheduling Quality in the Multiagent Simple Temporal Problem / Vinicius De Antoni. – Porto Alegre: PPGC da UFRGS, 2014.

48 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2014. Advisor: Álvaro Moreira.

1. Multiagent Systems. 2. Simple Temporal Problem. I. Moreira, Álvaro.

ACKNOWLEDGMENTS

First of all, I would like to thank my parents Luiz Alberto De Antoni and Rosali Salete Maschio De Antoni for their support throughout the whole process, also I would like to thank my brother Rafael De Antoni for the frequent reviews and discussions. I thank the CS department professors for the mind-opening classes, and my colleagues Gabriel, Samir and Marcelo. Without a doubt, this work would be impossible to finish without the discussions with my colleagues and without the courses.

Special thanks to my former advisor Dr. Rafael H. Bordini for the idea that made this work possible and also to my current advisor Dr. Álvaro Moreira for the patience and effort to make this dissertation happen.

Thank you all.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

STP	Simple temporal problem
MaSTP	Multiagent simple temporal problem
TDP	Temporal decoupling problem
MaTDP	Multiagent temporal decoupling problem
ABT	Asynchronous backtracking
CSP	Constraint satisfaction problem
DCSP	Distributed constraint satisfaction problem
ATF	Asynchronous time finder
DAI	Distributed artificial intelligence
MAS	Multiagent systems
PC	Path consistency
PPC	Partial path consistency
HTTP	Hypertext transfer protocol
TCP	Transmission control protocol

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In order to schedule an activity that depends on other people, we very often end up wasting precious time trying to find compatible times and evaluating if they are accepted by all involved. Even though modeling and solving multiagent scheduling problems seem completely understood and several algorithms can be found in the literature, one limitation still stands up: How to find a compatible time slot for an activity shared by many users without requiring the users themselves to spend time going through their calendar and choosing time slots until everybody agrees. The main contribution of this work is an algorithm called Asynchronous Time Finder (ATF) based on the Asynchronous Backtracking (ABT) that enables applications to find compatible times when scheduling shared activities among several users while requiring minimal user interaction. This dissertation starts by revisiting the Simple Temporal Problem (STP) and its multiagent version (MaSTP), it then shows how they can be used to solve the problem of managing agendas and then finally it presents the ATF giving an experimental evaluation and the analysis of its complexity.

Keywords: Multiagent Systems, Simple Temporal Problem.

Um Algoritmo Asíncrono para aprimorar a Qualidade de Agendamento no Problema Temporal Simples Multiagente

RESUMO

Ao tentar agendar uma atividade que dependa da presença de outras pessoas, geralmente acabamos desperdiçando tempo precioso avaliando os possíveis horários e verificando se os mesmos são aceitos por todos envolvidos. Embora a modelagem e a resolução do problema de agendamento multiagente pareçam estar completamente entendidas e ainda diversos algoritmos possam ser encontrados na literatura, uma questão ainda existe: Como definir horários compatíveis para uma atividade compartilhada sem que os usuários tenham que manualmente escolher horários livres de seus calendários até que todos envolvidos aceitem um horário. A principal contribuição é um algoritmo chamado Descobridor Asíncrono de Horários (ATF) baseado no Rastreamento Asíncrono (ABT) que permite que aplicações encontrem horários compatíveis para atividades compartilhadas requerendo mínima intervenção manual dos usuários. Esta dissertação revisita o Problema Temporal Simples (STP) e a sua versão multiagente (MaSTP), demonstra como eles podem ser utilizados para resolver o problema de agentamentos e ao final apresenta o ATF, a avaliação experimental e a análise de complexidade.

Palavras-chave: Sistemas Multiagentes, Problema Temporal Simples.

1 INTRODUCTION

The study of Multiagent Systems (MAS) focuses on the analysis and development of sophisticated mechanisms to solve problems related to coordination, cooperation, negotiation, privacy, scalability, uncertainty and many other problems inherent to systems that have more than one autonomous entity (agent) pursuing their goals. The problems a multiagent system deals are further extended when the agents's goals are not compatible with each other, resulting in a competitive system. For instance, an agent responsible for John's agenda might want to commit to the least number of activities possible, so John can go home earlier, while the agent responsible for Mary's agenda, according to her will, tends to schedule the most activities possible in each day so Mary can take Friday off. It's obvious that in the case where both agents share one or more activities, without negotiation, the conflicting nature of their goals may render them unable to be met and consequently both John and Mary will perform poorly in the eyes of their manager. Therefore a multiagent system responsible for the week schedule must provide ways through which both agents may reach an agreement so they can have the work done by the end of the week.

Recently, with the advent of intelligent personal assistants capable of managing users's activities, like the one introduced in (MYERS et al., 2007), the attention dedicated to multiagent scheduling problems has increased among researchers of the field. At first the Simple Temporal Problem (STP) (DECHTER; MEIRI; PEARL, 1991) which is a widely accepted representation of the problem of determining if a plan or a schedule is feasible[1], seemed to be a suitable candidate for representing and solving a scheduling problem. There are algorithms, like P3C (PLANKEN; WEERDT; KROGT, 2008), that solve the STP in a centralized fashion, meaning that, in order to evaluate the schedules of a group of agents, a single entity should gather all member's activities and constraints, and solve the corresponding STP. The main issues that arise in the centralized approaches are:

- $Privacy$: Every member would have to reveal their full schedule, which could include information that people want to keep private;
- $Flexibility$: Having to send all their activities and constraints through the network make it costly to change them, since every change must be sent again to the central solver;
- $Scalability$: Relying on a single entity to deal with everyone's schedule might lead to performance issues as the number of agents grows;

[1]A schedule is said feasible if there is an assignment of time points in a way that all constraints are respected, i.e., there is no time conflicting activities.

- *Uncertainty*: Information kept by the central solver might become stale if one or more agents are not able to send their schedules as they change due to network issues, hardware failures, etc...

In (BOERKOEL; DURFEE, 2010), the authors introduce the Multiagent Simple Temporal Problem (MaSTP) which addresses the issues mentioned above that are not covered by the centralized approaches for solving STP. Along with the MaSTP definition, a distributed algorithm called DΔP3C based on P3C is presented. In the algorithm, each agent starts by processing its private schedule. If it is feasible the agents then send and receive constraints regarding shared activities, processing them as they arrive. If the private schedule is not feasible the agent can fix it by removing conflicting activities or changing activities's start time and/or duration before starting sending constraints of shared activities to others. By processing locally the private activities, this approach guarantees privacy since the private information is stored and processed within the agent, and by not relying on a central entity, the algorithm also improves flexibility, reduces uncertainty and becomes highly scalable because there is no bottleneck anymore, the load is distributed between the agents, where each agent is only responsible for its own activities.

One drawback of that solution, as mentioned in (BOERKOEL; DURFEE, 2011), is that since the MaSTP allows activities to have flexible start time and/or duration, e.g., an activity can be scheduled to start anytime from 10 a.m. to 11 a.m. and to last from 30 to 60 minutes, whenever agents agree upon a shared activity, every local change can lead to unnecessary exchange of messages. Every agent involved with a specific shared activity keeps information about the other agents, meaning that, if the shared activity is supposed to start any time between 10 a.m. and 11 a.m. and one of the agents that has a private activity before that, which is supposed to end at 10 a.m., for some reason delays it until 10:30 a.m., this results in the agent sending constraints updates to every other agent, even the delay not affecting the shared activity (it is OK as long as it starts by 11 a.m.). Thus, due to the fact that generally multiagent systems operate in time-critical and highly-dynamic environments, the DΔP3C leaves room for improvement in regards of the exchange of messages.

Hunsberger in (HUNSBERGER, 2002a) defined the Temporal Decoupling Problem (TDP) as being an optimization problem with the objective to find a decoupling of a constraint system. Planken et al. in (PLANKEN; WEERDT; WITTEVEEN, 2010), found out that the TDP could be used to solve the MaSTP with the advantage of also addressing the unnecessary constraints updates. The decoupling is obtained through the addition of extra constraints to each shared activity so that each agent's agenda is decoupled from each other, and once the agents meet the constraints the execution avoids unnecessary coordination. In the previous example, these extra constraints would tell that the shared activity must happen by 11 a.m, so any change that does not conflict with that will not be sent.

Originally, the TDP was specified to find a temporal decoupling in a centralized fashion. Then, the multiagent version called Multiagent Temporal Decoupling Problem (MaTDP) was presented in (BOERKOEL; DURFEE, 2011) along with a fully distributed algorithm for named D-P3C (a newer version of DΔP3C) where agents maintain locally decoupled consistent schedules that, when combined, produce a consistent joint schedule. Nevertheless, when the addition of a new event or a new constraint make the global schedule inconsistent, the agents must exchange messages to accommodate the changes and consequently create a new and consistent decoupling. The MaTDP is obviously an improvement of the MaSTP, but essentially they solve the same problem: given a distributed set of local schedules they both try to find out whether or not the global schedule is feasible.

Even though solving a multiagent scheduling problem seems completely understood, we believe that none of the approaches described above tackle properly the problem of finding a compatible time for every agent involved in a shared activity. In order to find a compatible time for every agent, the current solutions, require several executions with different times for the shared activity until the activity is accepted by every one. For every run, the user must manually specify the time for the activity, when this process clearly could be automated by the scheduling system which has access to all the activities and could automatically pick available time slots (time slots without activities) and find if the others agents have available time slots in common. Hence, the main contribution of this dissertation is a novel algorithm named Asynchronous Time Finder (ATF) that has the goal of finding a compatible time for a new shared activity in a distributed fashion ensuring privacy and performance while requiring minimal user interaction.

Although, there has been a huge improvement recently in the way we can solve the distributed scheduling problem, being through MaSTP or even by seeing the problem as a TDP and using a MaTDP algorithm, the lack of assistance when creating a shared appointment is still out there. Therefore, the purpose of the ATF is not to replace any of the algorithms that solve the MaSTP or the MaTDP but to complement them in a way where users can easily find compatible times for a shared activity, and as soon as they find a time slot for it, the activity then can be added to their schedules and managed by any of the algorithms mentioned above.

The remainder of the dissertation is organized as follows. First, we formally revisit the STP (Section 2.1), Agents and Multiagent Systems (Section 2.2), the MaSTP (Section 2.3) and the MaTDP (Section 2.4). On Section 3 we describe the algorithms we mentioned above and how they are used to solve the scheduling problem. Section 4 exposes the problem that is still open and how this work tries to solve it by utilizing the ATF. We designed and implemented a prototype of a scheduling system that uses ATF and Secion 5 explains how it was done. The evaluation is presented on Section 6 and finally, Section 7 states our conclusion and states the next steps of this work.

2 BACKGROUND

In this chapter we introduce the formalism of the Simple Temporal Problem and the Multiagent Simple Temporal Problem along with a overview of Agents and Multiagent Systems for the reader understanding using the taxonomy and terminology found in the literature.

2.1 Simple Temporal Problem - STP

The Simple Temporal Problem (STP) was first introduced in (DECHTER; MEIRI; PEARL, 1991), being defined as a pair (V, C), where $V = \{v_1, ..., v_n\}$ is a set of time point variables representing events, and $C = \{c_1, ..., c_m\}$ is a set of binary constraints over time points variables, bounding the time distance between two events.

Constraints are linear inequalities that establish an upper bound on the time distance between two time points, represented by $v_j - v_i \leq w_{ij}$ where $w_{ij} \in \mathbb{R}$ is the upper bound of the distance between v_i and v_j. Two constraints c_{ij} and c_{ji} can be combined into a single constraint $-w_{ji} \leq v_j - v_i \leq w_{ij}$, or equivalently , $v_j - v_i \in [-w_{ji}, w_{ij}]$, resulting in both upper and lower bounds. If c_{ij} exists and c_{ji} does not, this is the same as $v_j - v_i \in [-\infty, w_{ij}]$, meaning that there's no lower bound between the two time points. As an example, consider the two events (a) Waking up at 6 a.m and (b) Arriving at work at 8 a.m. Assume that one needs one hour to get ready plus thirty minutes to commute to the office. Thus, we have $v_b - v_a \leq 2$ and $v_a - v_b \leq -1.5$ which leads to $w_{ab} \in [-(-1.5), 2]$, meaning that the time distance between v_a and v_b can be any value between one and a half to two hours.

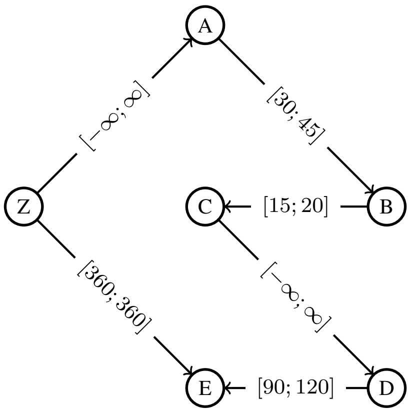


Figure 2.1: John's STP instance

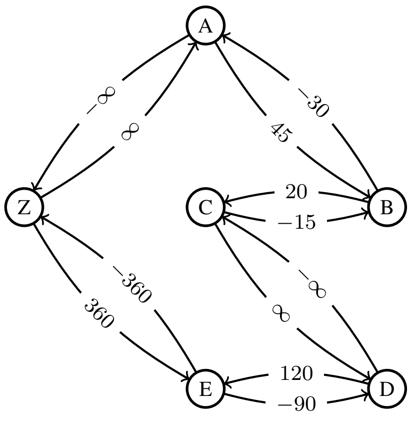


Figure 2.2: The distance graph of Figure 2.1

Given an STP instance, one might be interested in determining its consistency, i.e, if there is an assignment of time points that satisfies all the constraints, or even inquiring about possible time distances between a given pair of time points.

An STP instance can be seen as a graph where the nodes represent the time points and the edges represent the lower and upper bounds of the distance between two time points. A special node Z can be added to represent the "start of time" in the context of the instance, e.g., it can be the first hour of the day (12 a.m.) so other events can stipulate start times relative to that, like waking up at 6 a.m, where the distance to Z would be 6.

The Figure 2.1 illustrates the STP instance described by the following scenario where we assume $Z = 7a.m.$: John wakes up some time after 7 a.m. (A) and takes 30 to 45 minutes to shower, get dressed and have breakfast (B). By car, John takes 15 to 20 minutes to commute to work (C), where he may have some time for doing some paperwork before the meeting with Mary starts (D). The meeting requires from one hour and a half to two hours. Both John and Mary are going to have lunch with their manager at 1 p.m (E). Note that the label $[-\infty, \infty]$ in the edge connecting Z to A means that John can wake up at any time after 7 a.m. and the label $[360, 360]$ in the edge between Z and E tells that the lunch must happen after exactly six hours, which is the 1 p.m deadline.

After modeling this scenario as an STP instance, one can ask questions like:

- Is it possible for John to satisfy all these time constraints?
- What time does John need to set the alarm for in the morning?
- How much time will John have to do the paperwork if he wakes up at 9 a.m?

An STP assignment is an association of specific time values to time point variables and it is considered a solution for the STP if it respects every constraint defined between two time points in a given STP instance. An STP instance is consistent if it has at least one solution.

Every STP instance is associated with a distance graph $G = (V, E)$, which is a weighted and directed graph. Figure 2.2 has the distance graph associated to the STP instance of Figure 2.1, where the set of vertices V is the set of time point variables and E is a set of directed edges, where each edge represents a constraint c_{ij} with the weight w_{ij} and connects the vertices v_i and v_j. Again, a zero time point, $Z \in V$ can be added to express the "start of time".

The consistency of an STP instance can be determined by its distance graph having no negative cycles (a cycle whose sum of edge weights is negative) (XU; CHOUEIRY, 2003). The chapter 3 discusses the algorithms that can tell if the graph has negative cycles and consequently can solve the STP.

2.2 Agents and Multiagent Systems

Multiagent Systems (MAS) is a sub-area of Distributed Artificial Intelligence (DAI) that focuses on the study and research of software that maintains an environment and autonomous entities (agents). These agents interact with the environment basically through perceptions and actions. Usually, each agent has a set of behavioral capabilities, a set of goals and autonomy to use capabilities to achieve goals. Decisions on what actions to take are made by taking into consideration the changes in the environment and the desire to achieve goals.

2.2.1 Multiagent Systems

Historically, the ideas and concepts behind the MAS come from several different areas, not only computer science. We find influences from psychology, philosophy and sociology as well. The modal concepts and modal logic (WRIGHT, 1951) helped define the behaviors and the reasoning process that later would be used to build these software entities called agents. Organizational studies, commonly seen in biology courses about how cells organize themselves to form a complete organism also influenced how agents interact with each other in a MAS.

The purpose of a MAS is to reach a global intelligent behavior from each agent's individual behavior, meaning that, an agent itself does not need to be considered intelligent as long as it is able to reach its goals while respecting the constraints of the environment.

A scheduling system can be easily seen as a MAS. The environment is a calendar, in which days are divided into time slots and time slots can contain activities. The agents represent the users, and their goal is to allocate time slots for both private and shared activities, respecting some predetermined rules, e.g, there cannot be conflicting activities. The global behavior of maintaining a feasible schedule for everyone is reached iff each agent reaches its goals and respects the constraints.

2.2.2 Agents

An agent is an autonomous computational entity, able to perceive, reason and act in its environment to reach its goals (WOOLDRIDGE, 2009).

An agent can be seen as an improved object[1] which maintains information about the environment and other agents in its knowledge base (attributes) and has several different actions (methods) which can change the environment and/or other agents's knowledge base. The main difference is that a regular object follows a thread of execution predetermined by the program while agents autonomously decide what to do based on their knowledge base (WOOLDRIDGE, 2009).

In the context of this work, an agent is a piece of software responsible for helping users to create and maintain a schedule, where the goal is to accommodate the user's activities and the actions are create, change and remove activities, and implicitly coordinate the shared activities with other agents.

2.3 Multiagent Simple Temporal Problem - MaSTP

The algorithms for solving the STP mentioned in the previous section were devised in the context of a single agent. In order to use STP to evaluate the schedules of a group of agents, a central entity should gather all member's activities and constraints, and solve the corresponding STP instance. As already discussed in the introduction, this solution has a series of drawbacks such as: the lack of privacy due to the fact that all the data has to be sent to the central solver and the obvious performance bottleneck. Multiagent Simple Temporal Problem (MaSTP) is a version of STP specifically defined for working with multiple agents.

Informally, we can define the Multiagent Simple Temporal Problem as being composed of n local STP instances, each one assigned to one agent, and a set of constraints that allow relationships between local instances. Formally, Boerkoel and Durfee in (BOERKOEL;

[1]Concept taken from object-oriented programming, that refers to a structure in memory that describes something, e.g, a person, a chair, a data base connection, etc.

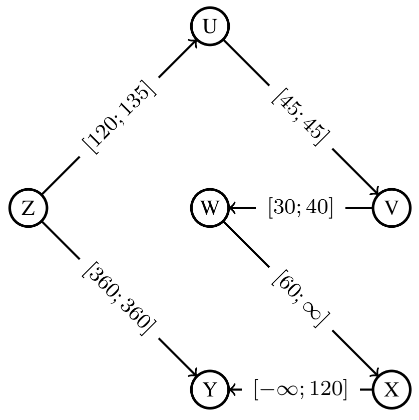


Figure 2.3: Mary's STP instance

DURFEE, 2010) defined the MaSTP's specification as follows:

- $S^i = (V^i, C^i)$ is the agent i's local STP instance;
- V^i is partitioned into V^i_A which is the set of time point variables agent i is responsible for assigning and V^i_X the set of time point variables known to agent i due to its involvement in some inter-agent constraint;
- C^i is partitioned into C^i_A the set of intra-agent constraints, i.e. constraints only between variables in V^i_A, and C^i_X the set of inter-agent constraints, i.e. constraints containing variables in V^i_X;
- $M = (V, C)$ is the MaSTP instance, where $V = \bigcup_i V^i$ and $C = \bigcup_i C^i$.

Furthermore, we can partition V^i_A into two sets: V^i_{AP} are agent i's private time points (which do not appear in V^j_X for any agent j) and V^i_{AS} are agent i's shared time points (which appear in V^j_X for some other agent j). Similarly, we can partition C^i_A into two sets: C^i_{AP} is the set of private constraints, or constraints that have at least one endpoint in the set V^i_{AP}, and C^i_{AS} is the set of shared intra-agent constraints whose endpoints are contained within the set V^i_{AS}.

To illustrate the MaSTP, consider the Mary's scenario (Figure 2.3): Mary wakes up between 9 a.m. and 9:15 a.m. (U), she works out for 45 minutes (V) and takes 30 to 40 minutes to get ready and walk to work (W). She needs at least one hour to finish a report and send it to her manager before the meeting with John (X) which as far as she knows can take up to two hours. After the meeting, Mary will join John and their manager for a business lunch at 1pm (Y). For the sake of simplicity, we also consider the start of time represented by the time point Z being 7 a.m.[2].

The Figure 2.4 shows a MaSTP instance with Mary and John's schedule (the bold constraints are the inter-agent constraints) where we have:

- $V^{John}_A = \{\text{Z1, A, B, C, D, E}\}$
- $V^{John}_{AP} = \{\text{A, B, C}\}$
- $V^{John}_{AS} = \{\text{Z1, D, E}\}$

[2]One can determine a different value for the time point Z as long as when modeling the MaSTP be sure to set the constraint among all the Z time points accordingly.

- $V_X^{John} = \{\text{Z2, X, Y}\}$
- $C_A^{John} = \{c_{\text{Z1 A}}, c_{\text{A B}}, c_{\text{B C}}, c_{\text{C D}}, c_{\text{D E}}, c_{\text{Z1 E}}\}$
- $C_{AP}^{John} = \{c_{\text{Z1 A}}, c_{\text{A B}}, c_{\text{B C}}, c_{\text{C D}}\}$
- $C_{AS}^{John} = \{c_{\text{D E}}, c_{\text{Z1 E}}\}$
- $C_X^{John} = \{c_{\text{Z1 Z2}}, c_{\text{D X}}, c_{\text{E Y}}\}$
- $V_A^{Mary} = \{\text{Z2, U, V, W, X, Y}\}$
- $V_{AP}^{Mary} = \{\text{U, V, W}\}$
- $V_{AS}^{Mary} = \{\text{Z2, X, Y}\}$
- $V_X^{Mary} = \{\text{Z1, D, E}\}$
- $C_A^{Mary} = \{c_{\text{Z2 U}}, c_{\text{U V}}, c_{\text{V W}}, c_{\text{W X}}, c_{\text{X Y}}, c_{\text{Z2 Y}}\}$
- $C_{AP}^{Mary} = \{c_{\text{Z2 U}}, c_{\text{U V}}, c_{\text{V W}}, c_{\text{W X}}\}$
- $C_{AS}^{Mary} = \{c_{\text{X Y}}, c_{\text{Z2 Y}}\}$
- $C_X^{Mary} = \{c_{\text{Z2 Z1}}, c_{\text{X D}}, c_{\text{Y E}}\}$

Once the distributed schedule has been modeled as a MaSTP one can use an algorithm such as D$\triangle$P3C (a more technical overview is presented in Chapter 3) to verify and maintain the schedule consistent through all the agents. Similarly, questions related to the schedule can be made to a MaSTP instance, for example, one could ask the following questions given the instance showed in Figure 2.4:

- Is it possible for both John and Mary to satisfy their time constraints?

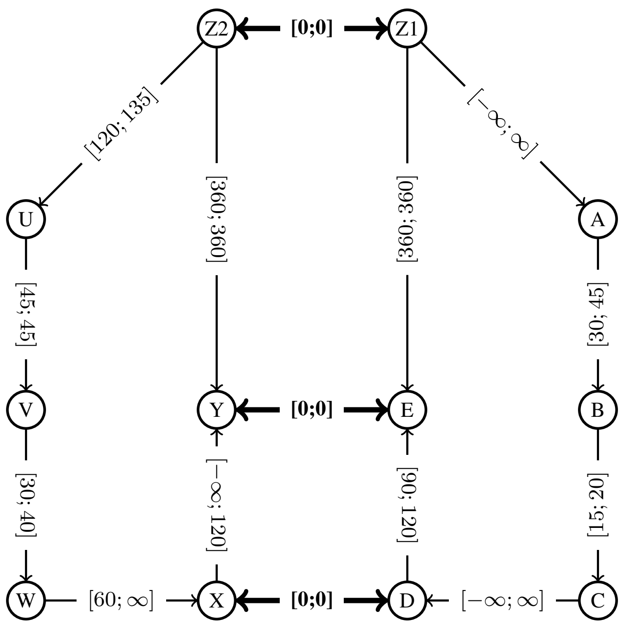


Figure 2.4: MaSTP instance

- How does the time constraint that says that Mary needs at least an hour to work on the report affect the start of the meeting?
- What if the manager changes the lunch time to 11:30 a.m? Is it going to break any constraint?
- What if John gets stuck in a traffic jam for 30 minutes? Does he need to let Mary know about it?

The main differences between the MaSTP and the STP are the new constraints that are added in order to allow the agents to hide their private schedules and coordinate only the shared activities. It is important to note that the MaSTP is also a STP and it could also be solved by one of the STP algorithms but by doing so, no advantage from the partitioning would be taken for improving privacy, flexibility and performance.

Even though the time complexity of D$\triangle$P3C to verify a static MaSTP instance is $O(|\Delta|)$ (Δ is the set of triangles in the graph) it does not perform well in environments where constraints are constantly changing due to the fact that new messages need to be sent every time. The next section talks about the MaTDP which address exactly this issue.

2.4 Multiagent Temporal Decoupling Problem - MaTDP

The Multiagent Temporal Decoupling Problem adds new constraints C_Δ to a MaSTP instance M, guaranteeing that each agent i can execute its local STP instance S independently of the other agents without the risk of running into inconsistency (PLANKEN; WEERDT; WITTEVEEN, 2010). Since the local STPs are completely decoupled due to the new C_Δ constraints, no coordination with other agents is required as long as the C_Δ constraints are satisfied. This is the main difference from the regular MaSTP, where the agents shared constraints and were responsible to maintain them, thus the need to send messages to coordinate every time something changes in the local STPs.

Therefore, the MaTDP is defined as, for each agent i, finding a set of constraints C^i_Δ such that if $S^i_\Delta = (V^i, C^i \cup C^i_\Delta)$, then $\{S'^1, S'^2, ..., S'^n\}$ is a temporal decoupling of MaSTP M (BOERKOEL; DURFEE, 2011).

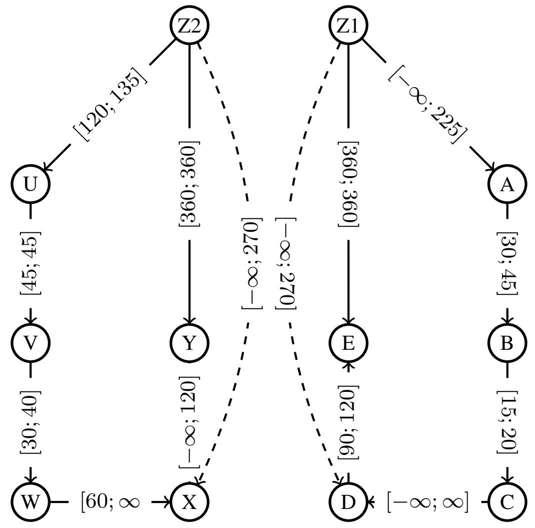


Figure 2.5: MaTDP instance

An example of a temporal decoupling is displayed in Figure 2.5, where John and Mary have agreed to start the meeting at most at 11:30 a.m by adding a new temporal constraint between the meeting and Z (dashed lines). This new constraint allows both agents to process its local STP independently of each other and communication will not be required as long as both local STPs remain consistent.

Finding a temporal decoupling is the key of the MaTDP, mainly because after the decoupling is found the agents need only to solve their own private STP instances with one of the algorithms shown above.

The questions that an instance of the MaTDP can answer are the same as those for the MaSTP, the difference is that due to the new constraints the communication among the agents is reduced, leading to better performance in dynamic environments where changes occur constantly. For instance, a traffic jam or an electric power outage that might prevent users for strictly following the schedule do not necessarily lead to the need of letting all the users know about them through update messages, unless one of the C^i_Δ constraints can not be satisfied anymore. Then, agents may need to reschedule or even cancel the activity.

As mentioned in Chapter 1, the D-P3C is the state-of-the-art algorithm to solve the MaTDP which naturally evolved from the DΔP3C. The authors showed that the D-P3C decreased both the computational effort and the number of messages when compared to the DΔP3C. It is important to note, though, that the MaTDP and the MaSTP are essentially the same problem, with the same input and output, the difference lying on the techniques each one employs to solve the problem.

The problem that remains is that whenever an instance is inconsistent, being a STP, MaSTP or a MaTDP, is up to the user to make changes to the schedule so it can be consistent again. The algorithms themselves are unable to fix inconsistencies like conflicting activities. For example, if Mary had already plans for lunch by the time John invited her, the algorithms would have found an inconsistency in the schedule and would finish its execution, leaving the schedule inconsistent until John removes the lunch activity. Alternatively, John could try adding it again in a different time, which could either work or render the schedule inconsistent one more time. This cycle could happen several times until the activity can be added to the schedule.

The next chapters present the algorithms to solve both the STP and MaSTP and a new algorithm we developed to address inconsistencies requiring minimal user input. With this new algorithm we prevent inconsistencies by finding compatible times beforehand, thus avoiding the try and error scenario described above.

3 ALGORITHMS

In this section we show the algorithms known in the literature to solve the STP and the MaSTP. We present the implementation and discuss the details of each solution, the $\triangle$P3C for the STP and the D$\triangle$P3C for the MaSTP. We choose not to discuss the implementation of the D-P3C because, as highlighted in the previous chapter, both the MaSTP and the MaTDP have the same goal. Hence, to avoid overwhelming the reader we focus only on the MaSTP. For further information on the MaTDP and D-P3C please refer to (BOERKOEL; DURFEE, 2011).

The contribution of this dissertation is focused on addressing the inconsistencies when adding new activities to a multiagent schedule and not on proposing a new algorithm for solving the STP or the MaSTP. The content of this chapter can then be skipped by those who are already familiar with these algorithms.

3.1 Solving the STP

As mentioned in Chapter 2 the STP consistency depends on its associated distance graph having no negative cycles and one way of determining if a distance graph has negative cycles is by calculating its *minimal network*. The minimal network can be calculated through *path consistency* (PC) (DECHTER, 2003), which works by computing the tightest possible path, i.e., the path with the shortest distance possible, between every pair of time points with an *all-pairs-shortest-path algorithm* such as Floyd-Warshall (CORMEN; LEISERSON; RIVEST, 1990). The minimal network obtained is then checked for consistency by validating that there are no negative cycles, that is, ensuring that $w_{ij} + w_{ji} \geq 0 \; \forall i \neq j$. The worst case of a PC algorithm can be $O(|V|^3)$.

In Figure 3.1 we show the result of applying the Floyd-Warshall algorithm on the STP instance of Figure 2.2. It is worth mentioning that with the minimal network we are able to say that John needs to wake up between 7 a.m. and 10:45 a.m., we also know that the meeting with Mary is going to start any time between 11 a.m. and 11:30 a.m., and the earliest John wakes up the more time he will have to do the paperwork.

In (XU; CHOUEIRY, 2003), Xu and Choueiry were the first to recognize that *partial path consistency* (PPC)[1] is sufficient for solving the STP, and they proposed the algorithm $\triangle$STP. PPC is defined for chordal (or triangulated) graphs. A graph is chordal if each cycle of size greater than three contains a chord, i.e. an edge connecting two non-adjacent vertices in the cycle (PLANKEN; WEERDT; KROGT, 2008). One graph can be made chordal by the process of graph triangulation. Depending on constraint struc-

[1]The authors discovered that one need not a complete graph to compute the shortest path between every node, thus the name partial path consistency.

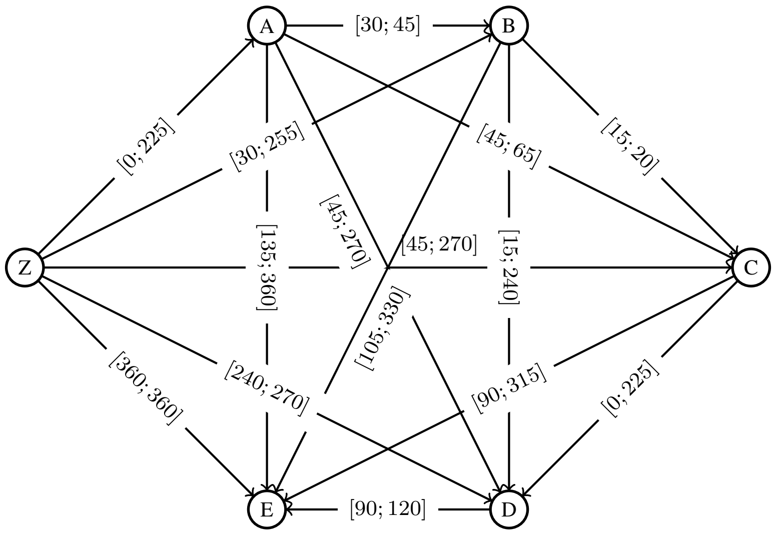


Figure 3.1: The minimal network of the distance graph of Figure 2.2

ture, the number of edges in the triangulated graph may be much smaller than the number of edges in the complete graph (required for PC) (BOERKOEL; DURFEE, 2010). If a user is interested in learning the tightest bounds between two time point variables that are not mutually constrained (and thus have no edge in the distance graph), the scheduling agent can add this edge explicitly prior to triangulation to ensure its inclusion in the STP calculation. The ΔSTP then processes and updates a list of every potential inconsistent triangles from the triangulated graph.

Instead of a simple list, the algorithm P3C (PLANKEN; WEERDT; KROGT, 2008), processes the triangles in a systematic order[2], resulting in a improved performance over ΔSTP. While ΔSTP's worst case is $O(|\Delta^2|)$, the P3C's is $O(|\Delta|)$, with Δ being the set of triangles in the graph.

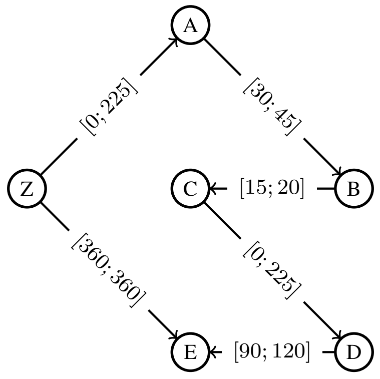


Figure 3.2: PPC applied to the distance graph of Figure 2.2

In (BOERKOEL; DURFEE, 2010) the authors introduced an algorithm for solving the STP in a centralized fashion called ΔP3C as an improvement of the P3C where the input graph does not need to be triangulated, instead, the triangulation will be done on the fly while maintaining the asymptotic behavior of $O(|\Delta|)$. The algorithm takes, as input,

[2]The order in which the triangles are processed is based on the order the triangles are created during the graph triangulation

a STP instance and the subset of time points to process [3], and returns whether or not the instance is consistent. There are two main differences between the ΔP3C and the P3C, (1) the former does not need the STP instance to be triangulated, instead it triangulates the STP on the fly, and (2) the ΔP3C does not take a time point processing order as input, just the subset of time points from where it heuristically chooses the next one to process [4].

Algorithm 1 ΔP3C-1 (BOERKOEL; DURFEE, 2010)

```
 1: procedure ΔP3C-1($S, V_e$)
 2:     $\Delta \leftarrow$ new stack of triangles
 3:     while $V_e \cap V \neq \{\}$ do
 4:         $v_k \leftarrow$ SELECTNEXT($V_e$)
 5:         $V \leftarrow V - v_k$
 6:         for all $v_i, v_j \in N(v_k), i \neq j$ do
 7:             $E \leftarrow E \cup$ JOINNEIGHBORS($v_k, v_i, v_j$)
 8:             return INCONSISTENT if $(w_{ij} + w_{ji} < 0)$
 9:             $\Delta$.push($v_i, v_j, v_k$)
10:         end for
11:     end while
12:     $V \leftarrow V \cup V_e$
13:     return $\Delta$
14: end procedure
```

JOINNEIGHBORS(v_k, v_i, v_j): Creates edges, e_{ij} and e_{ji}, if they do not already exist, initializing the weights to ∞. Then tightens the bounds of these edges using the rule $w_{ij} \leftarrow min(w_{ij}, w_{ik}, w_{kj})$. Returns the set of any edges that are created during the process.

Algorithm 2 ΔP3C-2 (BOERKOEL; DURFEE, 2010)

```
1: procedure ΔP3C-2($\Delta$)
2:     while $\Delta$.size() $> 0$ do
3:         $t \leftarrow \Delta$.pop()
4:         TIGHTENTRIANGLE($t$)
5:     end while
6: end procedure
```

TIGHTENTRIANGLE(v_i, v_j, v_k): Tightens any of the triangle edges that need to be tightened using the rule $w_{ij} \leftarrow min(w_{ij}, w_{ik}, w_{kj})$. Returns the set of any edges that are tightened during the process.

The ΔP3C, like P3C, operates in two stages: ΔP3C-1 (Algorithm 1) and ΔP3C-2 (Algorithm 2). Roughly, ΔP3C-1 heuristically chooses time points through SELECTNEXT procedure (line 4) in order to propagate its temporal constraints to the adjacent time points (line 6). Then, for each triangle, it creates and tightens new fill edges as needed (line 7) and returns a stack containing all the triangles that were added in order to calculate the PPC. With the stack as input, ΔP3C-2 then re-tightens each triangle (line 4).

[3] In the general case the subset is the whole set of time points, meaning that all the time points will be processed.

[4] The heuristic selection of time points must lead to the addition of the fewest fill edges possible. In (BOERKOEL; DURFEE, 2010), the heuristic consist of prioritizing private time points over shared ones.

Figure 3.2 shows the resulting graph after after applying ΔP3C on the STP instance presented in Figure 2.1. Due to the fact that when calculating the PPC all the triangles are tightened the resulted network has also the tightest possible constraints. For instance, John had to wake up sometime after 7:30 a.m., the constraint initially was $[-\infty, \infty]$ and after running the ΔP3C we know that John needs to wake up at most 225 minutes after 7:30 a.m. $[0, 225]$ to be able to complete all his tasks on time.

The ΔP3C could also be used to solve the MaSTP, however, this requires a central agent with access to everyone's schedule in order to process and propagate all the constraints. In the next section, we show that the ΔP3C-1 is part of the DΔP3C, it is used as a subroutine to process the local STPs.

3.2 Solving the MaSTP

The DΔP3C (Algorithm 3) is a fully distributed algorithm that solves the MaSTP. In its implementation each agent i starts by applying the ΔP3C-1 on its local STP and set of private time points V^i_{AP} (line 2). If it is consistent, in order to process their shared time points V^i_{AS}, the agent selects a time point v_k and computes what edges to add and/or update. The agent i then tries to obtain write permission on the set of shared time points and theirs constraints (line 14). First, the agent must confirm that no neighbors of v_k have been processed (line 15). If not, the agent can commit the constraint changes (lines 16-19). Otherwise, it extracts all updates (line 21) to any affected edges to review its local STP and abandons the changes already calculated. After all time points have been processed, the agent sweeps the stack tightening and broadcasting each triangle (line 27 to 34), during this process the agent can also receive notifications from other agents.

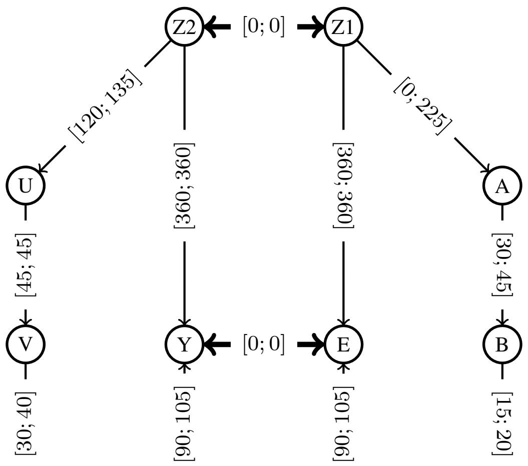


Figure 3.3: PPC applied to the distance graph of Figure 2.4

The result of applying DΔP3C on the MaSTP instance of Figure 2.4 is displayed in the Figure 3.3. Note that, with the new constraints, the meeting between Mary and John (X and D) must finish in at most 105 minutes due to the time Mary needs to finish the report (constraint between W and X).

As mentioned before, users are constantly changing their schedules, adding new private and shared activities, canceling and updating meetings as the days go by, and every time a change happens they need to reevaluate the private STP and send the new constraints to everyone involved in the shared activities so the MaSTP instance can remain consistent. This is tackled by the Algorithm 3 inside the while loop on line 27.

Algorithm 3 DΔP3C (BOERKOEL; DURFEE, 2010)

```
1: procedure DΔP3C($S^i$)
2:     $\Delta^i \leftarrow$ ΔP3C-1($S^i, V^i_{AP}$)
3:     return INCONSISTENT if ΔP3C-1 does
4:     $V^i_E \leftarrow V^i_{AS}$.copy()
5:     while $V^i_E \cap V^i \neq \{\}$ do
6:         $\tilde{E}^i \leftarrow E^i$.copy()
7:         $\tilde{\Delta}^i \leftarrow \Delta^i$.copy()
8:         $v_k \leftarrow$ SELECTNEXT($V^i \cap V^i_E$)
9:         for all $v_i, v_j \in N(v_k), i \neq j$ do
10:            $\tilde{E}^i \leftarrow \tilde{E}^i \cup$ JOINNEIGHBORS($v_k, v_i, v_j$)
11:            return INCONSISTENT if $(w_{ij} + w_{ji} < 0)$
12:            $\tilde{\Delta}^i$.push($v_i, v_j, v_k$)
13:        end for
14:        $o \leftarrow$ REQUESTPROCESSINGORDERLOCK()
15:        if ($o \cap N(v_k) = \emptyset$)) then
16:            $o$.append($v_k$)
17:            $V^i$.remove($v_k$)
18:            $E^i \leftarrow \tilde{E}^i$
19:            $\Delta^i \leftarrow \tilde{\Delta}^i$
20:        else
21:            $S^i \leftarrow$ UPDATEDEDGES($o \cap N(v_k)$)
22:        end if
23:        RELEASEPROCESSINGORDERLOCK($o$)
24:    end while
25:    $V^i \leftarrow V^i \cup V^i_E$
26:    $U \leftarrow$ new updated edge stack
27:    while $\Delta^i$.size() $> 0$ or PENDINGEDGEUPDATES do
28:        $U$.push(RECEIVEEDGEUPDATES())
29:        $\Delta^i$.INSERTADJACENTTRIANGLES($U$)
30:        $U$.clear()
31:        $t \leftarrow \Delta^i$.pop()
32:        $U$.push(TIGHTENTRIANGLE($t$))
33:        BROADCASTANYSHAREDEDGEUPDATES($U$)
34:    end while
35:    return $S^i$
36: end procedure
```

INSERTADJACENTTRIANGLES(U): Updates the triangle stack to include any (externally or internally) updated triangle adjacent to updated edges (except the triangle that caused the update) to its specified location in the triangle stack Δ^i.
N(v_k): Returns the neighbors of v_k.

Now, imagine agent i wants to add a new shared activity with agents j and k. Since agent i does not know anything about agent j's and agent k's private schedule, agent i would have to send as many updates as it takes until the MaSTP is consistent. In other words, agent i would have to request the user to input a new time for the activity every time the schedule is inconsistent and rerun the DΔP3C, wasting processing power, network resources and mainly user’s time. It gets worse as the number of agents involved in the activities grows. In the next chapter we show how we addressed this problem.

4 ASYNCHRONOUS TIME FINDER

In this section we show the Asynchronous Time Finder (ATF) and we explain how it deals with inconsistencies when adding new shared activities, an issue which is not addressed by any algorithm we have discussed so far.

This issue is elucidated in this following scenario: Jane, Matt and Annie are college classmates sharing several classes during the week. They need to meet some time in the next few days to work on a paper they must submit by the end of the month. They don't know about each other's activities after the classes. Matt is responsible for finding a time that works for everybody.

When using one of the previous algorithms, like the D△P3C, Matt would have to try and add the activity several times with different times until nobody's agenda is inconsistent. This is impracticable in a multiagent environment where communication is not cheap and we want to avoid human interaction where it is not really needed. In cases like this it would be much better if some assistance were provided for Matt to easily find compatible time slots instead of just failing.

Our solution consists in Matt sending out his available time slots to the other invitees. As soon as the others receive Matt's request they check if there is an intersection between their time slots and Matt's. When a time slot is available for all participants the meeting can be added as a shared activity to the MaSTP instance. If no compatible time is found, Matt may need to extend the period in which the available times are obtained (e.g from the current week to the whole next month). Even though this requires Matt to manually specify a new time period, we expect that to happen rarely. In other words, the number of times Matt needs to interact with the system is greatly reduced.

Figure 4.1 shows the workflow of the ATF when used in the scenario described above. During the first step (1) the MaSTP is composed of the private STP instances of Matt, Jane and Annie. The next step (2) involves Matt creating the new shared activity. Matt sends the invitation to Jane and Annie and waits for either a confirmation or a decline due to incompatible times (3), in case of the latter (4), Matt chooses a new time and sends new invitations for both Jane and Annie until they all reach an agreement (5). In the last step (6) the new activity is added as a shared activity to each STP instance and it's now managed by the MaSTP algorithm.

Our algorithm is based on the Asynchronous Backtracking (ABT) (YOKOO et al., 1998), that was presented as an alternative for solving the Distributed Constraint Satisfaction Problem (DCSP) (YOKOO et al., 1998) where agents run concurrently and asynchronously exchanging variable values to achieve a stable state where all the known constraints of the environment are satisfied. Our problem can be easily seen as DCSP, where the people involved in a shared activity (agents) have to reach an agreement (respect constraints) regarding the time of the activity (variables).

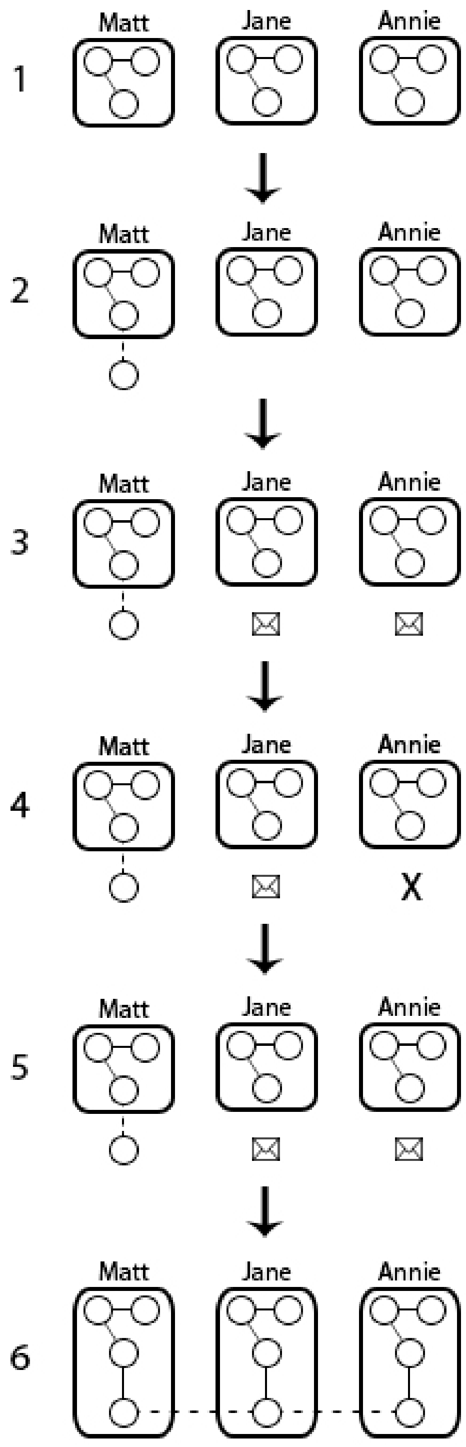


Figure 4.1: ATF workflow

The following sections revisit the Constraint Satisfaction Problem (CSP) and the Asynchronous Backtracking (ABT) in order to give the reader the knowledge necessary to understand how the Asynchronous Time Finder works.

4.1 Constraint Satisfaction Problems

Formally, a Constraint Satisfaction Problem (CSP) is defined as a triple (X, D, C), where X is a set of variables $\{x_1, ..., x_n\}$, D is a domain of values $\{D_1, ..., D_n\}$, and C is a set of constraints $\{c_1, ..., c_n\}$ (YOKOO et al., 1998). A CSP consists of finding an assignment of values to all variables while satisfying all constraints. A constraint is defined by a predicate. That is, the constraint $c_k(x_{k1}, ..., x_{kj})$ is a predicate that is defined on the Cartesian product $D_{k1} \times D_{kj}$. This predicate is true iff the value assignment of these variables satisfies this constraint.

A classic example of a CSP is the eight queens puzzle, where a position on a 8x8 chessboard must be found for each queen, such that none of them threat each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The variable set X contains the position of each queen and the domain of the variables D represents all possible locations on the chessboard, i.e., $D = \{(1, 1), ..., (8, 8)\}$. The constraints set C define the no-threat rule, for example, for subset $X_{12} = (x_1, x_2)$ which contains the variables of the first two queens, the constraint subset would be $C_{12} = \{[(1, 1), (2, 3)], [(1, 1), (2, 4)], ...\}$, and all the other valid assignments of x_1 and x_2 where they are not threating each other.

4.2 Distributed Constraint Satisfaction Problems

A DCSP is a CSP in which the variables and constraints are distributed among automated agents (YOKOO et al., 1998). In order to define the DCSP, the authors assumed:

- Agents communicate by sending messages. An agent can send messages to other agents iff the agent knows the addresses of the agents;
- The delay in delivering a message is finite, though random. For the transmission between any pair of agents, messages are received in the order in which they were sent.

Each agent owns a local variable and it is the agent's responsibility to maintain it. The agent's variable may be constrained by local and shared constraints (inter-agent constraints) and both must be respected. Formally, there exist m agents $1, 2, ..., m$. Each variable x_j belongs exactly to one agent i (represented by $belongs(x_j, i)$). The inter-agent constraint c_k is known to all agents whose variables are influenced by the constraint ($known(c_k, i)$). Thus, the DCSP is solved if the following is true:

- $\forall i$, $\forall x_j$ where $belongs(x_j, i)$, the value of x_j is assigned to d_j, and $\forall l$, $\forall c_k$ where $known(c_k, l)$, c_k is true under the assignment $x_j = d_j$.

4.3 Asynchronous Backtracking

The ABT is an algorithm to solve the DCSP. The DCSP is represented as a directed graph, where the variables are nodes and the constraints are edges. Since each agent has only one variable a node can also be seen as the agent itself. An edge is directed from the value-sending agent to the constraint-evaluating agent. In (YOKOO et al., 1998) an example of DSCP graph is shown and it is illustrated on Figure 4.2, where there are three agents, x_1, x_2, x_3, with variable domains $\{1, 2\}, \{2\}, \{1, 2\}$, respectively, and constraints $x_1 \neq x_3$ and $x_2 \neq x_3$.

The algorithm then consists in each agent instantiating its variables in a concurrent fashion and sending them to other agents that are connected by outgoing edges. After that, each agent waits for incoming messages and responds them according to the procedures described in Algorithm 4.

Each agent has its own set of the other agents's current values (that are connected by incoming edges). This set is defined as the agent's $agentview$, which is composed by pairs (x_k, d_k), where x_k is the agent's identifier and d_k is the agent's current value.

The agents can receive two kinds of messages: One kind is an $ok?$ message, that is sent from an agent willing to check if its variable is compatible with the destination agent's constraints (line 1). Every time an agent receives an $ok?$ message, it adds the source agent's value to its $agentview$ (line 2) and checks whether its $currentvalue$ is consistent with the $agentview$ (line 3).

The other kind of message is a $nogood$ message sent by an agent whose constraints were not satisfied by the destination agent's current value (line 5). As soon as the agent receives a $nogood$ message it adds the $nogood$ set passed on the message to its $nogoodlist$ (line 6). When an $agentview$ is not compatible it is called a $nogood$, and the $nogoodlist$ contains every $nogood$ the agent has received in order to avoid infinite looping [1]. If the $nogood$ contains an agent that is not currently connected to the agent that received the message, a request is sent to this new agent so the connection can be made. This new connection will allow both agents to exchange messages and negotiate their values (line 8).

An assignment is consistent with the $agentview$ if all constraints the agent evaluates are true under the value assignments described in the $agentview$ and $currentvalue$, and also if the following holds for all $nogood$ in the set $nogoodlist$: $nogood \notin (agentview \cup (x_i, currentvalue))$. If the assignment is not consistent with the $agentview$, the agent tries to change the $currentvalue$ so that it will be consistent with the $agentview$ (line 19). If the agent is not able to find a value consistent with its current $agentview$ the backtracking takes place (line 20).

The backtracking happens every time an agent x_i does not find any valid assignment of its $currentvalue$ and then notifies the agent with the lowest priority x_j that is involved in the $nogood$ (line 28). The purpose is that x_j might have other values that will make x_i

[1] By maintaining a $nogoodlist$, the agent can avoid choosing a value that caused an inconsistency before, thus preventing it from infinite looping.

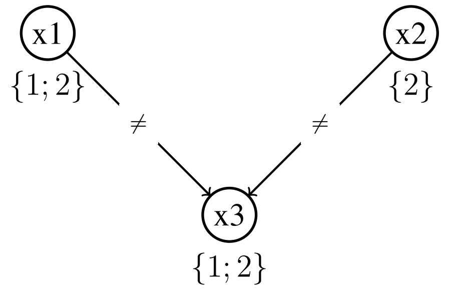


Figure 4.2: Example of a DCSP graph

Algorithm 4 ABT

```
 1: procedure RECEIVE(ok?, $(x_j, d_j)$)
 2:     add $(x_j, d_j)$ to agentview
 3:     CHECKAGENTVIEW
 4: end procedure
 5: procedure RECEIVE(nogood, $x_j$, $nogood$)
 6:     add $nogood$ to nogoodlist
 7:     for all $x_k \in nogood, x_k \notin$ NEIGHBORSIN($x_i$) do
 8:         request $x_k$ to add an edge from $x_k$ to $x_i$
 9:         add $(x_k, d_k)$ to agentview
10:     end for
11:     $oldvalue \leftarrow currentvalue$
12:     CHECKAGENTVIEW
13:     if $oldvalue = currentvalue$ then
14:         send $(ok?, (x_j, currentvalue))$ to $x_j$
15:     end if
16: end procedure
17: procedure CHECKAGENTVIEW
18:     if $agentview$ and $currentvalue$ are not consistent then
19:         if no value in $D_i$ is consistent with $agentview$ then
20:             BACKTRACK
21:         else
22:             select $d \in D_i$ where $agentview$ and $d$ are consistent
23:             $currentvalue \leftarrow d$
24:             send $(ok?, (x_i, currentvalue))$ to NEIGHBORSOUT($x_i$)
25:         end if
26:     end if
27: end procedure
28: procedure BACKTRACK
29:     $nogoods \leftarrow \{ V \mid V =$ inconsistent subset of $agentview\}$
30:     if an empty set is an element of $nogoods$ then
31:         broadcast to other agents there is no solution
32:         terminate
33:     end if
34:     for all $V \in nogoods$ do
35:         select $(x_j, d_j)$ where $x_j$ has the lowest priority in $V$
36:         send $(nogood, x_i, V)$ to $x_j$
37:         remove $(x_j, d_j)$ from $agentview$
38:     end for
39:     CHECKAGENTVIEW
40: end procedure
```

NEIGHBORSOUT(x_i): Returns all agents x_i is connected to.
NEIGHBORSIN(x_i): Returns all agents connected to x_i.

consistent with one of its possible values. If x_j also does not have any valid assignment it notifies the lowest priority agent in its *nogood* and the backtracking goes on until some agent find a new valid assignment. If at some point no valid assignment can be found whatsoever the execution is terminated with a fail state.

The authors of ABT showed that if there exists a solution, the algorithm reaches a stable state where all the variable values satisfy all the constraints and all agents are waiting for an incoming message [2]. Thus, the algorithm is complete, in that it finds a solution if one exists and terminates with failure otherwise.

4.4 Asynchronous Time Finder

The algorithm Asynchronous Time Finder (ATF) presented in this section is based on the ABT plus a few mechanisms to deal with the assumptions we make in order to model the problem described in the previous section as a DCSP. We assume that:

- One single agent is the owner of the activity, meaning that it is responsible for creating the activity and sending the invitations to all agents involved;
- The invitees (agents who are invited to the activity) only exchange their values with the owner of the activity;
- If one invitee can't find any compatible time it notifies only the owner;
- The owner knows when all invitees agreed with a given time or if any or some of them have declined the invitation. Thus, at the end, the owner let the others know if the activity is confirmed or canceled.

We divided the algorithm in three phases: activity creation, time negotiation and activity termination (confirmation or cancellation).

During the first phase the agent who owns the activity sends a $create$ message to every invitee passing the duration d of the activity and a set S with all possible start times.

Upon receiving the invite the agents execute the procedure shown in Algorithm 5 (line 1), where the $owner$ of the activity is set and the available values set D is calculated based on the intersection of the agent's available time D and the times set S sent by the owner of the activity.

The second phase is fundamentally led by the exchange of $ok?$ and $nogood$ messages between the invitees and the owner until everybody agrees with a time or someone give up due to no compatible time available.

The third phase consists of determining when the execution is completed. It can happen as soon as the owner receives a decline from an invitee [3] or when everybody has agreed with the time of the activity. In order to determine if every invitee is synchronized the owner sends a $snapshot$ message to the agents and waits for everyone to respond. The $snapshot$ message asks the agents' current value (line 5) which is store in the owners' $snapshotview$. If everyone responds with the same value the owner then sends a $ok!$ message to confirm the activity, if there is someone with a different value, the owner waits for a random period of time and re-sends the $snapshot$ messages. A maximum number of attempts can be set to avoid infinite waiting.

An example of the ATF execution with the scenario discussed in the beginning of the chapter is shown in Figure 4.3. (1) Matt decided that 3 hours is enough for the meeting, and he is available every Monday and Wednesday from 1pm to 6pm, thus the message $create$ has the values $d = 3$ and $S = \{Mon1pm, Mon2pm, Mon3pm, Wed1pm,$

[2]For our purposes this is too vague because we need strong evidence that the algorithm is finished. We address this in our algorithm where we know when the execution terminated successfully

[3]This behavior can be improved by adding optional invitees to the algorithm where the meeting can happen even if one or more of the optional invitees can't make it.

Algorithm 5 ATF

```
 1: procedure RECEIVE(create, $x_j$, $d$, $S$)
 2:     $owner = x_j$
 3:     $D$ = CALCULATEAVAILABLETIMES($S$, $d$, $D$)
 4: end procedure
 5: procedure RECEIVE(snapshot?)
 6:     send ($snapshot!$, $x_i$, $currentvalue$) to $owner$
 7: end procedure
 8: procedure RECEIVE(snapshot!, $x_j$, $d$)
 9:     add ($x_j$, $d$) to $snapshotview$
10:     if INVITEES() $\subset$ AGENTS($snapshotview$) then
11:         if $\forall(x_j, d_j), (x_k, d_k) \in snapshotview, d_i = d_j$ then
12:             broadcasts ($ok!$) to AGENTS($snapshotview$)
13:         else
14:             wait
15:             CHECKACTIVITY()
16:         end if
17:     end if
18: end procedure
19: procedure RECEIVE(ok!)
20:     send $currentvalue$ to $owner$
21: end procedure
22: procedure CHECKACTIVITY
23:     send ($snapshot?$) to INVITEES()
24: end procedure
```

CALCULATEAVAILABLETIMES(S, d, D): Returns the set of available times based on the intersection of D and S, the duration d of the activity is also taken into account.
INVITEES(): Returns all agents involved in the current activity being negotiated.
AGENTS(view): Returns the set of agents present in the view.
VALUES(view): Returns the set of values present in the view.

$Wed2pm, Wed3pm\}$. Upon receiving the message, the agents responsible for Jane and Annie's schedule calculate the set of available times D. (2) After sending the $create$ message, Matt's agent sends an $ok?$ message to both Jane and Annie along with the provisional start time $Mon1pm$ chosen arbitrarily by Matt. (3) Annie's agent is OK with $Mon1pm$ and sets its current value d to $Mon1pm$. Jane's agent does not have $Mon1pm$ in its D so it sends a $nogood$ message back to Matt. (4) Matt's agent choose another time and send a new $ok?$ message with $d = Mon2pm$. Both Jane and Annie can make it to $Mon2pm$. (5) After some time without receiving a $nogood$ message, Matt's agent sends a $snapshot?$ message to check the current values of Jane and Annie. (6) Jane and Annie's agent reply the $snapshot?$ message with their current values. (7) Matt's agent knows that both are OK with $Mon2pm$ so it sends an $ok!$ message to confirm the activity.

At the end of the ATF execution, the agents can then add the activity to their schedules without breaking the consistency of the MaSTP. Once the activity is added it is managed by the algorithm that is being used by the scheduling system (D$\triangle$3PC for example). It is important to note that the ATF is independent of the algorithm that manages the schedule, its goal is to find a compatible time among several distributed intervals of time maintained

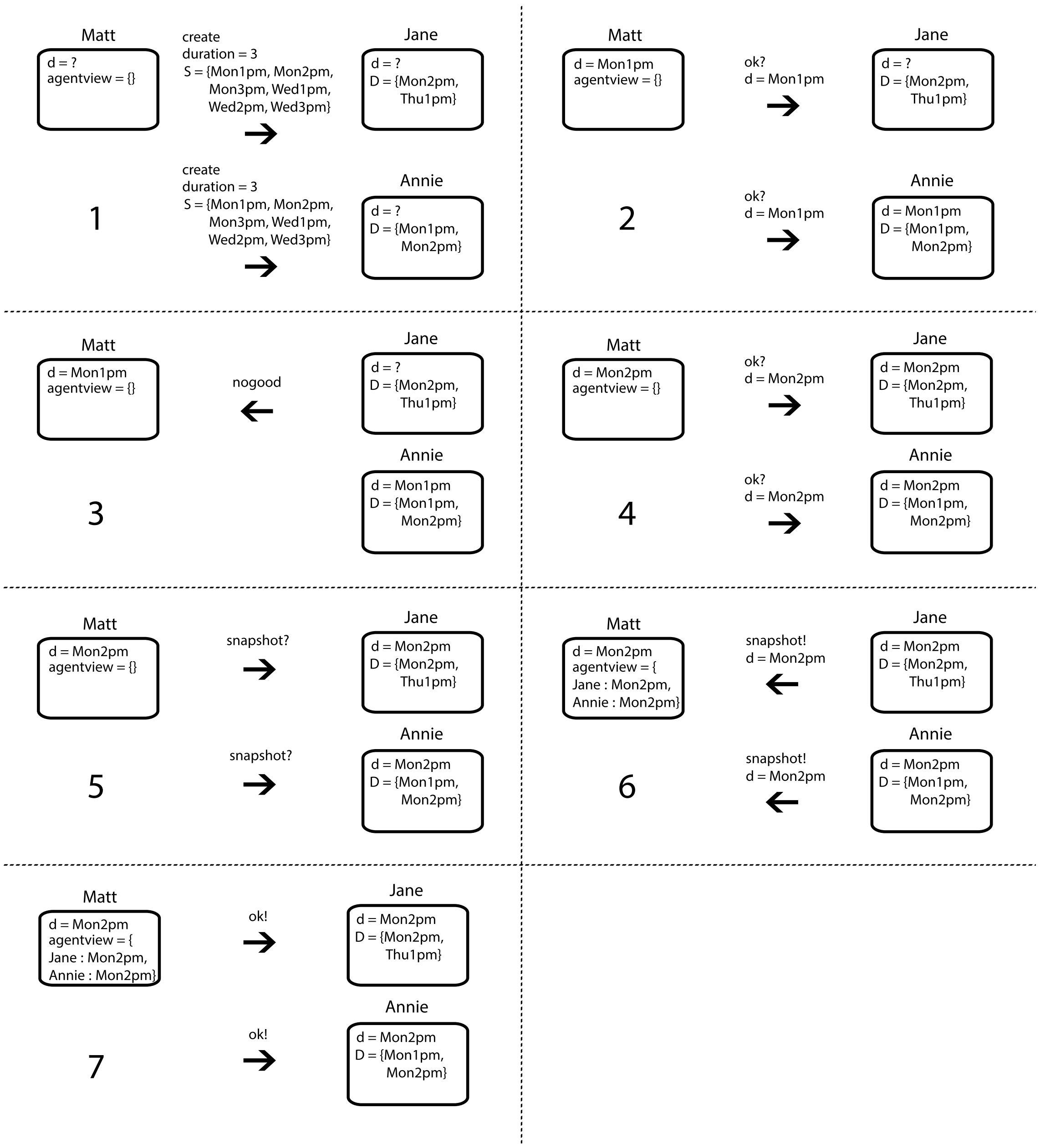


Figure 4.3: ATF's execution flow example

by several different agents and then hand it to the scheduling system. ATF does not know about the MaSTP structure and how the constraints are being coordinated, ATF is, at the end of the day, an extension that can be used to reduce the user interaction when the system already has all the information it needs.

5 A SCHEDULING SYSTEM USING ATF

As a way to evaluate our algorithm performance and usefulness in a scheduling system, we developed a prototype of a system that allows multiple users to manage private and shared activities. This chapter explains the pieces of the system, its design and implementation and discuss the challenges of developing a multiagent system.

5.1 Design

We started the design of our prototype by assuming a few things about how the users would like to use the system:

- Users can be anywhere in the world;
- Users are not always connected, they might miss invitations and schedules changes.
- As soon as users are connected their schedule needs to be updated;

The first challenge was how to design a reliable communication channel that could support these requirements. Mainly, how to deal with the possibly high churn rate, where it is not guaranteed that a given user will be available to receive a message at some given moment. We, then, decided to create a message server that is accessible on the internet through which users would exchange messages. It is important to mention that this server is only responsible for forwarding messages, it will not process or maintain any data about the users's schedule.

This message server can eventually become a bottleneck as the number of agents grow. For the purpose of this prototype one server is enough, but for a real world system more than one message server might be needed so the load of the messages can be distributed.

Each user will run a client program (agent) that is responsible to maintain their local STPs through the DΔP3C and to find time slots for shared activities using the ATF. All the exchange of messages needed by both algorithms must occur via the message server. Whenever a user is online, the agent must establish a connection with the message server to fetch any messages that might have arrived when it was offline in order to keep the local STP up to date. The client program will be the interface to the system and will allow users to add, remove and change activities.

5.2 Implementation

This section explains in a technical level how both the message server and the agent application were implemented.

5.2.1 Messages server

Since our system runs on the internet, HTTP is the protocol that allows all the messages to be exchanged. The message server is essentially an HTTP server that maintains a connection map of agents[1] and websockets (WANG, 2013), which are channels over a TCP connection that allow full-duplex communication between the agent and the message server. Once the agent connects to the system, a websocket connection is established and the agent starts receiving and sending messages.

We chose to implement the message server using node.js (TEIXEIRA, 2012) and the websocket library socket.io (RAI, 2013) due to the short learning curve, mainly because of the amount of documentation and examples that are available, and also due to the fact that node.js uses an event-driven, non-blocking I/O model, that makes it well suited for data-intensive real-time applications, filling the needs of our system.

Algorithm 6 Message Server

```
 1: var io = require('socket.io').listen(80);
 2:
 3: var map = { };
 4:
 5: io.sockets.on('connection', function (socket) {
 6:   socket.on('login', function (data) {
 7:     if (validateAgent(data.id, data.password)) {
 8:       map[data.name].socket = socket;
 9:       sendPendingMessages(data.name, socket);
10:     }
11:   });
12:
13:   socket.on('logout', function (data) {
14:     map[data.name] = undefined;
15:   });
16:
17:   socket.on('message', function (msg) {
18:     if (map[msg.to] != undefined) {
19:       map[msg.to].socket.emit('message', { 'msg' : msg });
20:     } else {
21:       savePendingMessage(msg);
22:     }
23:   });
24: });
```

Algorithm 6 shows a snippet of the message server code. It starts by defining a dependency to socket.io, opening port 80 for incoming connections (line 1), and declaring an empty map (line 3). The callback function declared on line 5 is called every time a new websocket connection is made, inside of it, callbacks for *login* (line 6), *logout* (line 13) and *message* (line 17) are declared in the same context as the websocket. After establishing a new connection, the agent must send a *login* message with its credentials, the server will validate them and add the agent and the websocket to the connection map. At

[1]The agent is identified by a unique id that is assigned when the user is added to system.

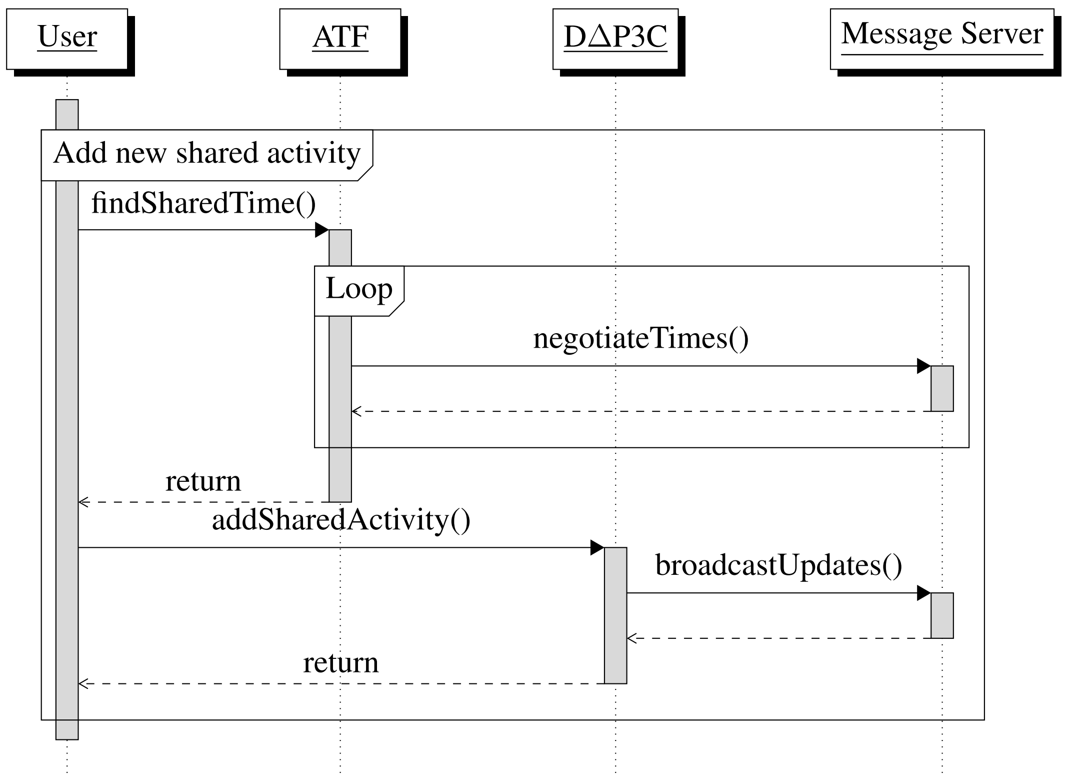


Figure 5.1: Add new shared activity sequence diagram.

this time, the server goes through the messages that could not be delivered before to see if there are pending messages to this agent. Whenever an agent wants to send messages it must send a *message* message with the fields *from*, *to* and *content*, which will be forwarded to the destination agent, if available, or stored as a pending message otherwise. In order to gracefully exit the system the agent can send a *logout* message. To avoid maintaining stale data into the connection map the message server sends health checks every minute and removes agents after three consecutive unsuccessful checks.

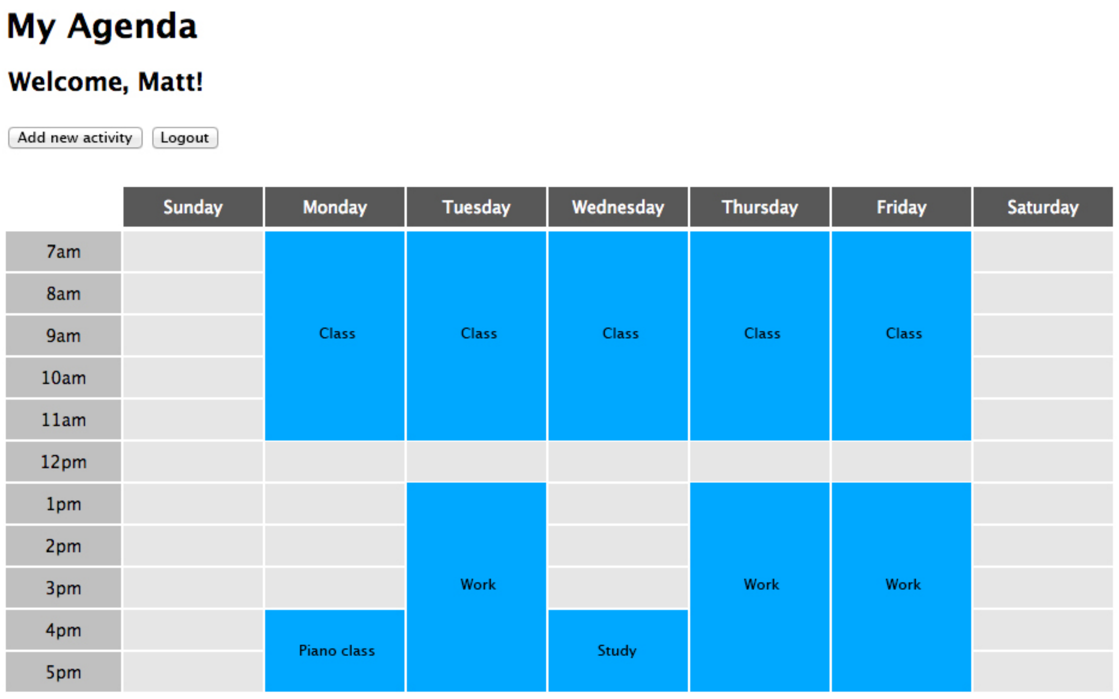
5.2.2 Client application

The client implements both the DΔP3C and the ATF and integrates them with the websocket connection channel, so all the messages are sent via the message server.

The Figure 5.1 is the sequence diagram that shows the client workflow when adding a new shared activity. On the client application, the user selects to add a new shared activity, the client then asks the ATF to find a compatible time between all invitees. ATF starts its execution as described in section 4 and negotiates the times with the other agents through the message server. Once ATF finds a compatible time it returns it to the client that asks the DΔP3C to add the new activity with the time found by the ATF.

The user interface that allows users to interact with the system by managing activities is shown in Figure 5.2 with an example of how Matt's schedule would look like. College classes in the mornings, working three afternoons a week, piano classes on Mondays from 4 pm to 6 pm and studying on Wednesdays from 4 pm to 6 pm. A weekly view of the calendar is presented to the user where the blue boxes represent activities and the grey ones represent available time slots.

In order to add new activities the user has two options: to inform the start and end time or to inform the duration of the activity as shown in Figure 5.3. The system behaves


My Agenda

Welcome, Matt!

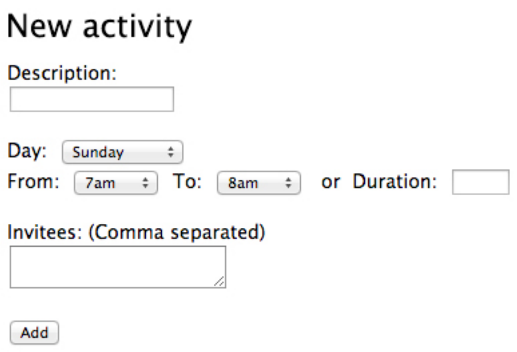
Add new activity Logout

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
7am							
8am							
9am		Class	Class	Class	Class	Class	
10am							
11am							
12pm							
1pm							
2pm							
3pm			Work		Work	Work	
4pm		Piano class		Study			
5pm							

Figure 5.2: User interface: Calendar view

differently based on the option chosen, the former tells the system that the user wants the activity to happen at an specific time, it may fail if there is a conflicting activity already in place. The latter tells the system to find any time slots available, it also can fail if there is no available time slots. The user can also inform a list of invitees to the activity being added, this will trigger the ATF before adding the activity to the local STP through the DΔP3C. If no invitees are listed, the activity can be added directly to the local STP.

The client application was implemented as a web page using HTML and Javascript, taking advantage of the fact that socket.io also provides a client library for websockets. Algorithm 7 shows a snippet of the client code. It starts by specifying where the socket.io library is located (line 1), then it connects to the message server (line 3) and once the connection is made a *login* message is sent. Every time ATF or DΔP3C wants to send a message to another agent the function *sendMessage()* (line 6) is called. The agent can


New activity

Description:

Day: Sunday

From: 7am To: 8am or Duration:

Invitees: (Comma separated)

Add

Figure 5.3: User interface: Add new activity

receive messages related to both ATF or D∆P3C (line 10), consequently, the code needs to check the message type every time it receives a new message so it can redirect the message to the correct handler.

Algorithm 7 Client Application

```
 1: <script src="/socket.io/socket.io.js"> </script>
 2: <script>
 3:   var socket = io.connect('http://messageserver.com');
 4:   socket.emit('login', { 'id' : id, 'password' : password });
 5:
 6:   function sendMessage(agent, content) {
 7:     socket.emit('message', { 'from' : id, 'to' : agent, 'content' : content });
 8:   });
 9:
10:   socket.on('message', function (msg) {
11:     if (msg.content.type == 'ATF') {
12:       ATF.handleMessage(msg);
13:     } else {
14:       D∆P3C.handleMessage(msg);
15:     }
16:   });
17: </script>
```

The system is functional and we were able to test it with some crafted examples and also with some randomly generated problems. In the next section, we show, among other results, how the system performed during our tests.

6 EVALUATION

In this chapter, we discuss the methodology we use to evaluate the performance of the ATF algorithm with respect to the number of messages exchanged, along with the results we collected from testing our system.

6.1 Methodology

In order to evaluate the performance of the ATF we use the random problem generator described in (HUNSBERGER, 2002b) to generate MaSTP instances. The instances are separated in different configurations, each one having a different number of agents varying from 2 to 128. Each agent in a given configuration has 10 to 20 private activities with duration varying from 1 to 4 hours separated from each other by 0 to 4 hours. we define STARTTIME as being the absolute time 0 (when the first activity starts) and the ENDTIME assumes the end time of the last activity.

Our experiments consist in randomly selecting an agent as the owner of 10 new shared activities involving every agent and sending them the invitations. The experiment finishes when every agent has agreed to every activity start time and duration. Each agent calculates the available time it will use to decide whether or not to commit to an activity, based on a set of hours ranging from STARTTIME to ENDTIME+100. In this way, if the agents cannot find times in between their private activities they will have enough time after them, this mechanisms allows every configuration to finish.

Each test was run 25 times and we use the average of the number of exchanged messages as the metric to be compared between configurations. All the tests were run in a single machine with a 4-core CPU of 3.4GHz and 16GB of RAM. The communication channel was emulated by a message queue in memory.

Number of Agents	Total Messages
2	55
4	163
8	378
16	757
32	1605
64	3286
128	6579

Table 6.1: Total messages per number of agents

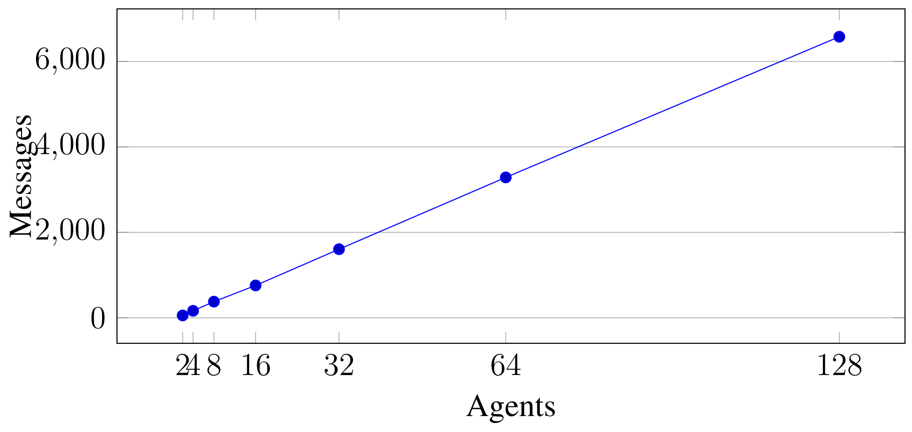


Figure 6.1: Number of messages

We also performed tests with the system described in Chapter 5. The tests consisted in adding one new shared activitie between a few agents and measuring how long it took for the activity to be added to the calendar. The message server was hosted by Amazon Elastic Compute Cloud (Amazon EC2) and was located on the west region of USA. The clients were running in the same machine located on the same region as the message server.

6.2 Results

Table 6.1 introduces the results of our evaluation tests. Each line shows how many messages on average (2nd column) were sent between the agents (1st column) in order to reach an agreement for the 10 shared activities. The results are also illustrated in Figure 6.1.

It is important to note that the complexity of the number of messages is $O(n)$, where n, in this case, is the number of agents. In other words, the number of message grows linearly as the number of agents increase. This is due to the fact that the agents involved in a shared activity exchange messages only with the owner of the activity and not with every other agent (mentioned in Section 4.4). We believe that this is a good thing because the agents should be only concerned about negotiating with the owner, and consequently

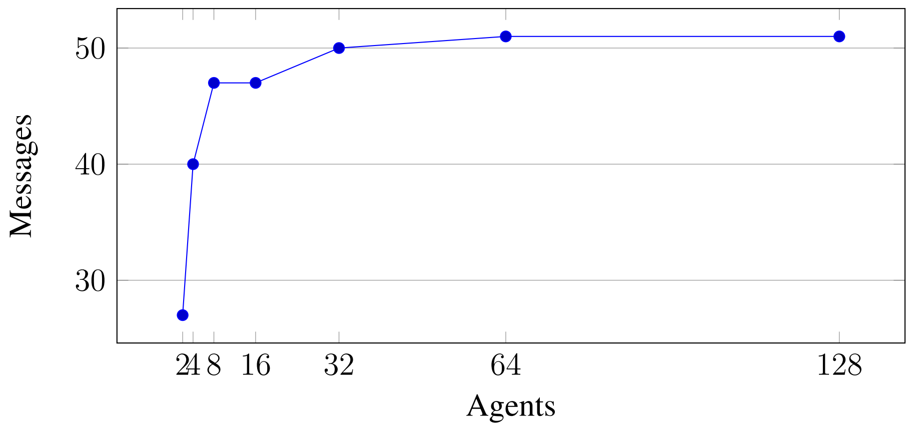


Figure 6.2: Number of messages per agent

Number of Agents	Total Time (ms)
2	442
4	870
8	1525
16	2783

Table 6.2: Total time per number of agents

the owner will negotiate with every other agent.

The Figure 6.2 shows how many messages were sent per each agent during the tests, which technically is constant as does not change as the number of agents grow. We do see a slightly increase and this is due to the conflicts between agents schedules leading to more messages until an agreement is reached. The higher the number of agents the harder is to find a compatible time for everyone.

The Table 6.2 has the time results of the tests we conducted with the system. Each line shows on the second column how much time it took on average to add a new activity between the number of agents in the first column. The time results also show a linear growth as shown in Figure 6.3.

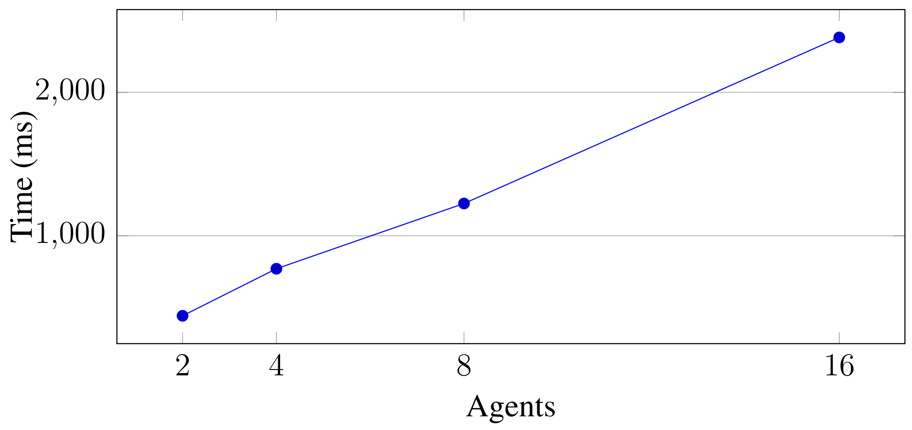


Figure 6.3: Total Time per number of agents

The system tests were executed with all invitees online and ready to send and receive messages, in the real world though, users might be offline for a while and might take a significantly amount of time to respond to invitations.

7 CONCLUSION

In this dissertation, we presented an algorithm to aid the management of shared activities within the MaSTP. We evaluated our algorithm and realized that the number of messages generated in order to negotiate activities' time slots grows linearly with the number of agents, making it scalable and cost efficient. The benefits of the ATF include a reliable way to schedule shared activities with minimal user interaction and a better use of the communication channel since the messages being exchanged have a common goal, i.e, one is not just trying out some possible time slots. We showed, through examples, how current scheduling systems can take advantage of the ATF, giving their users a better experience.

A prototypal, yet functional, scheduling system that uses ATF was shown and discussed. The design and implementation have room for improvement but it worked great as a proof of concept. We currently do not have the tools to test the system with realistic numbers, such as hundreds or even more users managing their agendas through the system, which could give us more data to work with.

In the future, we would like to improve our system by adding more features like offline mode and support to flexible start time and duration (as supported by the MaSTP). We will also be working on a test platform that can simulate hundreds of users and collect data about the system correctness and performance.

APPENDIX A PRINCIPAIS RESULTADOS

Este apêndice tem como objetivo sumarizar em português o que foi apresentado em inglês neste trabalho. O conteúdo aborda a introdução do problema, nossa proposta e os resultados obtidos.

O estudo dos sistemas multiagentes se concentra na análise e desenvolvimento de mecanismos sofisticados para resolver problemas relacionados com coordenação, cooperação, negociação, privacidade, performance, incerteza e muitos outros problemas inerentes aos sistemas que possuem mais de uma entidade autônoma (agente) buscando atingir seus objetivos.

Recentemente, com o advento de assistentes pessoais inteligentes capazes de gerir as atividades dos usuários, como o apresentado em (MYERS et al., 2007), a atenção dedicada a problemas de cronograma multiagentes tem aumentado entre os pesquisadores da área. No início, o Problema Temporal Simples (STP) (DECHTER; MEIRI; PEARL, 1991) que é uma representação amplamente aceita do problema de determinar se um plano ou cronograma é viável[1], parecia ser um candidato adequado para representar e resolver um problema de cronograma. Existem algoritmos que resolvem STP de forma centralizada, o que significa que, a fim de avaliar os horários de um grupo de agentes, uma única entidade deve reunir atividades e restrições de todos os membros e resolver o STP correspondente, pondo em risco a privacidade pois cada membro terá de revelar o seu cronograma, o que pode incluir informações que os usuários não querem compartilhar.

Em (BOERKOEL; DURFEE, 2010), os autores introduzem o Problema Temporal Simples Multiagente (MaSTP), que aborda as questões mencionadas acima, principalmente a privacidade, que não são cobertos pelas abordagens centralizadas para resolver STP. Junto com a definição do MaSTP, um algoritmo distribuído chamado DΔP3C baseado em (PLANKEN; WEERDT; KROGT, 2008) é apresentado. No algoritmo, cada agente começa processando seu cronograma privado, se é viável os agentes enviam e recebem as restrições sobre as atividades compartilhados, processndo-as assim que chegarem. Se o cronograma privado não é viável o agente pode consertá-lo removendo atividades conflitantes ou alterando o tempo e/ou a duração das atividades antes de iniciar o envio de restrições de atividades compartilhadas.

Mesmo que a solução de um problema de cronograma multiagente pareça completamente entendida, acreditamos que nenhuma das abordagens encontradas na literatura ataca o problema de encontrar um tempo compatível para cada agente envolvido em uma atividade compartilhada. A fim de encontrar um horário compatível as soluções atuais exigem vários ciclos de negociação entre os usuários, quando claramente isso pode

[1]o cronograma é dito viável se houver uma atribuição de pontos no tempo de uma forma que todas as restrições sejam respeitadas, ou seja, não há atividades conflitantes.

ser automatizadas pelo sistema, que tem acesso a todas as informações de que necessita para encontrar uma boa solução que satisfaça todos os usuários. Assim, a principal contribuição deste trabalho é um novo algoritmo chamado Asynchronous Time Finder (ATF), que tem o objetivo de encontrar um horário compatível para uma determinada atividade de uma forma distribuída assegurando a privacidade.

O cenário a seguir é adotado para explicar o problema a ser resolvido: Jane, Matt e Annie são colegas que atendem juntos à várias aulas durante a semana. Eles precisam definir algum horário nos próximos dias para trabalhar em um artigo que deve ser entregue até o final do mês. Eles não sabem sobre as atividades de cada um após as aulas. Matt é o responsável por encontrar um horário que seja bom para todos.

A nossa proposta consiste em Matt enviar seus horários disponíveis para os outros convidados e assim que os outros recebem a mensagem de Matt eles verificam se existe uma interseção entre os horários deles e do Matt. Quando um horário está disponível para todos os participantes, a atividade pode ser adicionada na instância do MaSTP. Se não for encontrado um horário viável, Matt pode precisar ampliar o período em que são obtidos os horários disponíveis (por exemplo, ao invés de apenas na semana seguinte, Matt irá procurar horários disponíveis durante todo o próximo mês).

O algoritmo é baseado no Asynchronous Backtracking (ABT) (YOKOO et al., 1998) e é dividido em três fases: criação da atividade, negociação de horários e término da atividade (confirmação ou cancelamento).

Durante a primeira fase, o agente que quer criar a atividade envia uma mensagem de criação para cada convidado passando a duração da atividade e um conjunto com todos os horários disponíveis possíveis. Ao receber o convite, os agentes executam o procedimento onde o criador da atividade é definido e os valores disponíveis são calculados com base na intersecção de horários.

A segunda fase é fundamentalmente conduzida pela troca de mensagens entre os convidados e o criador da atividade até que todos concordem com o horário ou até que alguém desista devido à falta de horários compatíveis.

A terceira fase consiste em determinar quando a execução está concluída. Pode acontecer logo que o criador receba uma recusa de um convidado ou quando todo mundo aceitou o mesmo horário. A fim de determinar se cada convidado concordou com o mesmo horário, o criador envia uma mensagem para os agentes e espera pela resposta de todos. Se todos respondem com o mesmo valor o criador em seguida envia uma mensagem de confirmação, se há alguém com um valor diferente, o criador espera por um período de tempo aleatório e re-envia o pedido. Um número máximo de tentativas deve ser ajustado para evitar a espera infinita.

A fim de avaliar o desempenho do ATF usamos um gerador aleatório descrito em (HUNSBERGER, 2002b) para gerar instâncias do MaSTP. Os exemplos são separados em diferentes configurações, cada uma tendo um número diferente de agentes variando de 2 a 128. Cada agente numa determinada configuração tem de 10 a 20 atividades privadas com duração variando de 1 a 4 horas, separadas umas das outras por 0 a 4 horas. Definimos STARTTIME como sendo o tempo absoluto 0 (quando a primeira atividade começa) e ENDTIME como sendo a hora de término da última atividade.

Nossos experimentos consistem em selecionar aleatoriamente um agente como o criador de 10 novas atividades compartilhadas envolvendo todos os agentes. O experimento termina quando todos agentes concordaram com o horários de todas atividades. Cada agente calcula o tempo disponível que irá utilizar para decidir se deve ou não comprometer-se a uma atividade, com base em um conjunto de horários que vão de

STARTTIME a ENDTIME+100. Desta forma, se os agentes não conseguirem encontrar horários disponíveis entre suas atividades privadas terão tempo suficiente após a última atividade (este mecanismo permite que todas configurações terminem).

Os testes foram realizado 25 vezes, e usamos a média do número de mensagens trocadas como a métrica a ser comparada entre as configurações. Todos os testes foram executados em uma única máquina com um processador de 4 núcleos de 3.4GHz e 16GB de RAM. O canal de comunicação foi emulado por uma fila de mensagens na memória.

A Tabela 6.1 apresenta os resultados de nossos testes de avaliação. Cada linha mostra quantas mensagens em média (2ª coluna) foram enviados entre os agentes (1ª coluna), a fim de chegar a um acordo para as 10 atividades compartilhadas. Os resultados são também ilustrados na Figura 6.1.

É importante observar que a complexidade do número de mensagens é $O(n)$, onde n, neste caso, é o número de agentes. Em outras palavras, o número de mensagem cresce linearmente conforme o número de agentes aumenta. Isto é devido ao fato de que os agentes envolvidos em uma atividade compartilhada só trocam mensagens com o criador da atividade e não com outros convidados. Acreditamos que isso é positivo, pois os agentes devem apenas se preocupar em negociar com o criador.

REFERENCES

BOERKOEL, J. C.; DURFEE, E. H. A Comparison of Algorithms for Solving the Multiagent Simple Temporal Problem. In: INTERNATIONAL CONFERENCE ON AUTOMATED PLANNING AND SCHEDULING, 20., Toronto, Canada. **Proceedings...** [S.l.: s.n.], 2010. p.26–33.

BOERKOEL, J. C.; DURFEE, E. H. Distributed Algorithms for Solving the Multiagent Temporal Decoupling Problem. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 10., Taipei, Taiwan. **Proceedings...** [S.l.: s.n.], 2011. p.141–148.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. **The Floyd-Warshall Algorithm**. [S.l.]: MIT Press and McGraw-Hill, 1990. 558-565p.

DECHTER, R. **Constraint Processing**. 1st.ed. San Francisco, USA: Morgan Kaufmann Publishers, 2003.

DECHTER, R.; MEIRI, I.; PEARL, J. Temporal Constraint Networks. **Artificial Intelligence Magazine**, [S.l.], v.49, n.1-3, May 1991.

HUNSBERGER, L. **Group Decision Making and Temporal Reasoning**. 2002. PhD Thesis — Harvard University, Cambridge, Massachusetts.

HUNSBERGER, L. Algorithms for a Temporal Decoupling Problem in Multi-agent Planning. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 18., Sydney, Australia. **Proceedings...** [S.l.: s.n.], 2002. p.468–475.

MYERS, K. et al. An Intelligent Personal Assistant for Task and Time Management. **Artificial Intelligence Magazine**, [S.l.], v.28, n.2, June/July 2007.

PLANKEN, L. R.; WEERDT, M. M. de; KROGT, R. van der. P3C: a new algorithm for the simple temporal problem. In: INTERNATIONAL CONFERENCE ON AUTOMATED PLANNING AND SCHEDULING, 18., Sydney, Australia. **Proceedings...** [S.l.: s.n.], 2008. p.256–263.

PLANKEN, L. R.; WEERDT, M. M. de; WITTEVEEN, C. Optimal Temporal Decoupling in Multiagent Systems. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 9., Toronto, Canada. **Proceedings...** [S.l.: s.n.], 2010.

RAI, R. **Socket.io Real-time Web Application Development**. Birmingham: Packt Pub, 2013.

TEIXEIRA, P. **Professional Node.js building Javascript based scalable software**. Hoboken, N.J. Chichester: Wiley John Wiley distributor, 2012.

WANG, V. **The definitive guide to HTML5 WebSocket**. Berkeley, Calif. Birmingham: Apress Computer Bookshops distributor, 2013.

WOOLDRIDGE, M. **An Introduction to Multiagent Systems**. 2nd.ed. [S.l.]: John Wiley & Sons, 2009.

WRIGHT, G. H. Deontic Logic. **Oxford Journals, Mind Association**, [S.l.], v.60, n.237, p.1–15, 1951.

XU, L.; CHOUEIRY, B. Y. A New Efficient Algorithm for Solving the Simple Temporal Problem. In: INTERNATIONAL SYMPOSIUM ON TEMPORAL REPRESENTATION AND REASONING. **Proceedings. . .** [S.l.: s.n.], 2003.

YOKOO, M. et al. The Distributed Constraint Satisfaction Problem: formalization and algorithms. **IEEE Transactions on Knowledge and Data Engineering**, [S.l.], v.10, n.5, p.673–685, 1998.