T04 - Schedule Alignment

Final Report

Universidade do Porto

Faculdade de Engenharia

FEUP

Master in Informatics and Computing Engineering

Agents and Distributed Artificial Intelligence

Group T04_1:
Duarte Duarte - ei11101
Luís Cleto - ei11077

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

December 14, 2014

Summary

The purpose of this project was to develop a tool to solve schedule alignment problems. This type of problems consist of having several agents attempting to organize activities between them. All activities must be scheduled for a time interval where all participants can attend for a solution to be valid. Aditionally, activities can not overlap if they have participants in common since an agent can only attend one activity at a time. A solution can also be better or worse depending on the weak constraints it satisfies. For example, an agent may prefer to have an activity at a given time but also be able to participate if it can't be scheduled for that time interval. In this case, multiple solutions may be acceptable but the ones that satisfy the agent's preference will be better.

To this end, multiple algorithms for solving distributed constraint optimization problems have been implemented and analysed, using the JADE platform to implement the agents. The algorithms chosen to be implemented were ADOPT and a modified version of ABT made to solve this type of problems.

Contents

1 Description

1.1 Scenario

Scheduling events between people is an inherently distributed process, which involves the preferences and restrictions of each of the people involved. Additionally, due to privacy concerns, the solving of this type of problems must be decentralized.

Each event to be scheduled can have several participants, each with their own time restrictions for the event. Additionally, each participant can also have preferences which will contribute to the quality of the solution found when these preferences are satisfied. For example, a participant may only be able to participate in an event from 12:00 to 20:00 but they may prefer to participate only after 15:00. This makes solutions where the event takes place after 15:00 better, but other solutions may also be valid.

For this purpose several algorithms can be used, such as ABT [1], ADOPT [2], DSA [3], MGM [4] and others.

1.2 Objectives

This project aims to implement a program capable of solving distributed constraint optimization problems, applied to the scenario of scheduling common events such as meetings or other group activities. An individual or agent is able to create or accept invitations to events, specify the time periods during which they can attend as well as their preferences. Once the participants are ready, the program then calculates a solution containing the final schedule for all the agents and events.

In order to calculate the solution, different algorithms for solving this type of problem were implemented and compared by analysing their performance in problems of different dimensions as well as the quality of the solutions obtained. The algorithms chosen to be implemented for this project were ADOPT and a modified version of ABT (ABT-opt [10]) that focuses on constraint optimization rather than just constraint satisfaction.

[1] Asynchronous Backtracking
[2] Asynchronous Distributed Constraint Optimization with Quality Guarantees
[3] Distributed Stochastic Algorithm
[4] Maximum Gain Message

2 Specification

Distributed Constraint Optimization Problems (DCOP or DisCOP) are problems in which a group of agents has to distributedly choose values for a set of variables such that the cost of a set of constraints over the variables is either minimized or maximized. The problem has a set of variables divided among a set of agents. Each agent knows only their own domain for the local variables and their constraints. The agents then have to communicate with each other by messages in order to achieve a solution that all can agree with.

2.1 Agents

For solving this problem, it was decided that each Agent acts as a possible participant for the events, acting as an individual trying to organize their own schedule with other individuals. However, several types of behaviours were implemented for each algorithm to be used and analysed. As such, agents will only be able to communicate through the program with other agents using the same behaviours. Each of the behaviours uses its own communication protocol as required by its respective algorithm.

Additionally, a setup behaviour was implemented for allowing users to create events with each other and specify their availabilities and preferences for each event. Once the setup phase is completed, the setup behaviour ends and the algorithm-specific behaviours are initialized. Finally, a subscription behaviour was implemented using JADE's subscription initiator, for detecting when agents join or leave the platform, as well as a leader election behaviour and a Depth-First Tree generation behaviour for the preprocessing phase of the ADOPT behaviour.

It is also worth mentioning that while an agent can have multiple variables (events), the algorithms used are made to handle only one variable per agent. As such, after the setup phase, each agent has virtual agents which contain only one of the master agent's events. The core agent serves as a multiplexer, redirecting messages to its virtual agents in accordance to the event specified in the incoming message. It is automatically assumed that there is a constraint between all virtual agents of an agent that each must have a different value. In order to simplify the explanation of the algorithms, only the virtual agents will be taken into account and will be referred to as agents, with the exception of the algorithm used for the Setup phase which does not require virtual agents.

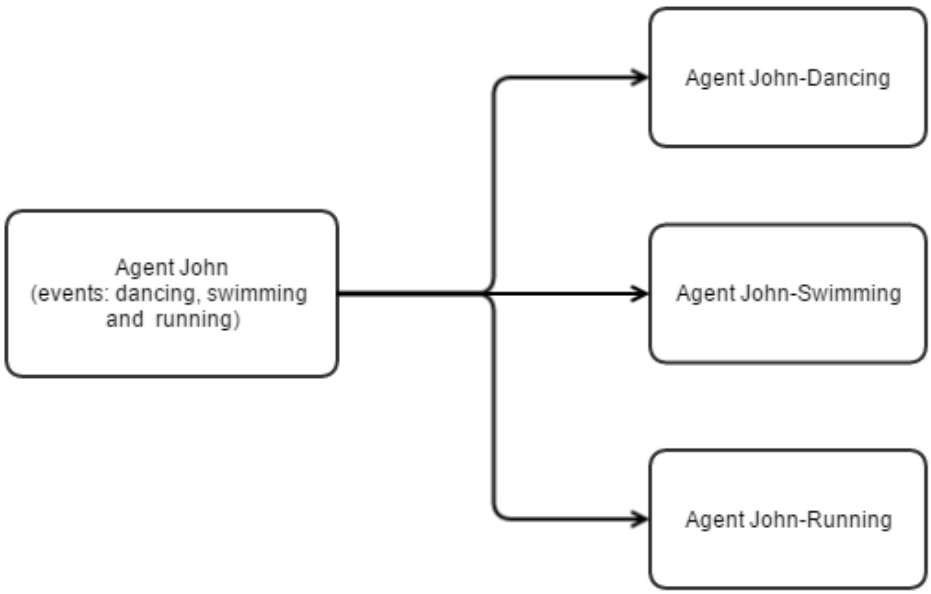

Figure 1: A visual representation of an agent's virtual agents.

2.1.1 Setup Behaviour

This behaviour is used to allow a user to initialize the problem to be solved and, as such, it was designed to be controlled by a graphical user interface. When the agent is created, a subscription behaviour is added using JADE's prototypes. This allows an agent to check for other agents with the same service description already in the platform and be notified whenever one joins or leaves. This simultaneously allows agents to have an updated list of other possible event participants and also to filter them, only being notified for agents of the same type (using the same algorithm).

Whenever an agent creates a new event, a INVITATION message is sent to all of the specified participants containing the event's information. If an agent leaves, all agents will remove that agent from the event's list of participants. Upon creating an event or receiving an invitation, the event is added to the agent's pending list. Before attempting to start the algorithm, the agent is required to answer all the invitations in their pending list. An agent can decline an event, sending a LEAVING message to all other participants, or accept it. In the latter case, they will have to specify their availabilities for attending the event. Additionally, after an agent has accepted an event, they can invite other agents to join that event, as long as they are not already participating. This sends an INVITATION message to that agent, containing the updated event's information, as well as an ADDING message to all the previous participants, informing them that the agent has been added to the event.

Once an agent has replied to all their invitations, they can declare themselves ready to begin the algorithm. When this happens, a READY message is sent to all agents. If an agent cancels the ready phase or receives an invitation while in this phase, they are removed from the ready phase and a CANCEL_READY message is sent to all agents. Once all agents are ready, the setup behaviour ends and the algorithm-specific behaviour is initiated.

2.1.2 ABT-opt Behaviour

This behaviour consists of an implementation of a variant of the Asynchronous Backtracking Algorithm. Since the original ABT focused on constraint satisfaction rather than constraint optimization, it was chosen to implement ABT-opt [10] instead as it is more appropriate for solving optimization problems.

In ABT-opt, there needs to be an ordering of agents in order to identify higher and lower priority agents. For this, JADE's unique agent identifiers were used. This way, the highest priority agent is the one is the one with the lowest identifier.

After the initialization phase, the algorithm begins and each agent looks for the most promising value in their domain. Since they don't yet have any information from other agents, this will be the one with the lowest cost for them. After choosing a value, they will send an OK? message to all lower priority agents. When an agent receives an OK? message, they will check their received NOGOOD's to detect obsolete ones and delete them and then update their agent view with the information of OK? message. They will then run the same cycle as before to check for the most promising value, this time taking into account the information relative to other agents in the agent's view and information from any received NOGOOD's. If the current chosen value changes during this update cycle, a new OK? message is sent to the lower priority agents.

When running the update cycle, if the current cost is higher than zero or the obtained cost is the exact value and not just a lower bound, the agent will terminate and send a TERMINATE message to all lower priority agents with the current cost if it is the highest priority agent, or send a NOGOOD to the agent immediately above it in the priority ranking if it has any.

When an agent receives a NOGOOD, they will update their agent view if there is any unknown agent in the NOGOOD message and send a LINK message to each of those agents. If the NOGOOD is consistent with the current view, the previous NOGOOD's will be re-evaluated to detect obsolete NOGOOD's and the new NOGOOD will be added to the NOGOOD store. The receiving agent will then once more run the update cycle for choosing a new value. Additionally, when receiving a LINK message, an agent will add the sender to the lower priority agents and send them an OK? message with their current chosen value.

Finally, when an agent receives a TERMINATE message, it will update its current cost and send another TERMINATE message to all lower priority agents, if it has any, and terminate its execution.

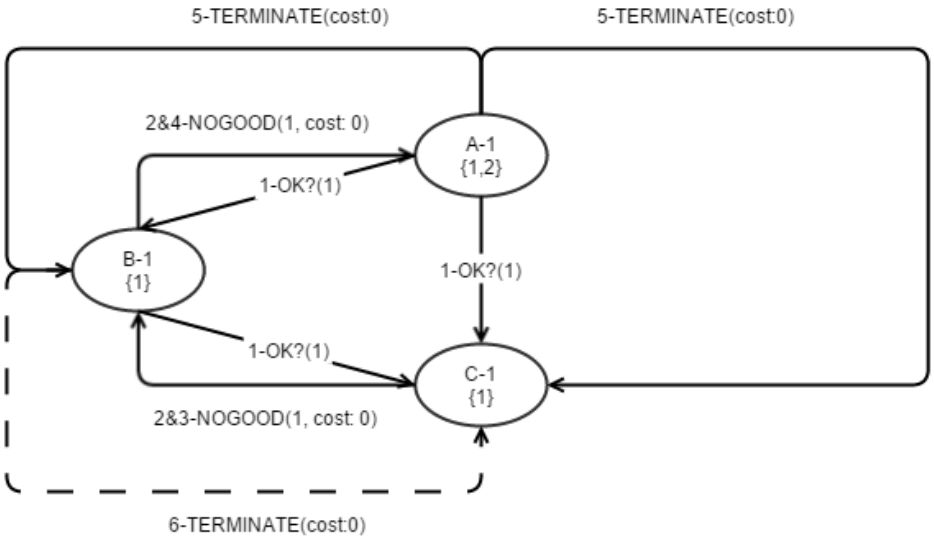

Figure 2: Messages sent in one execution of ABT-opt. All the agents share equality constraints with each other. The message represented by the dotted arrow is not actually delivered and processed since C-1 will have already terminated.

2.1.3 Leader Election Behaviour

This behaviour is used when it is necessary to generate a Depth-First Tree. As long as there is a path from one agent to the other following constraints between variables (virtual agents), they will belong to the same graph and, as such, have the same leader. There will be as many leaders as there are isolated constraint graphs.

In the initialization phase, each agent adds all agents with whom they share constraints as a neighbour and divides them into higher priority neighbours and lower priority neighbours. Each agent then checks if they have any higher priority neighbours. If they do, they wait to receive a LEADER_PROPOSAL message from all of those neighbours, if they do not, they assume they are the leader and send a LEADER_PROPOSAL message to all lower priority neighbours, identifying themselves as the leader. When an agent receives a LEADER_PROPOSAL from all higher priority neighbours, they accept the proposal with the highest priority candidate and forward it to all lower priority neighbours.

If an agent does not have any lower priority neighbours, they still accept the best proposal and inform higher priority neighbours of their choice with a LEADER_CHOICE message and terminates the algorithm. Once an agent has received a LEADER_CHOICE message, it sets the leader to the candidate identified in the message and forwards the LEADER_CHOICE to higher priority neighbours if it has any then terminates. Once an agent terminates, the behaviour ends and the DFS Tree Generation algorithm is initiated.

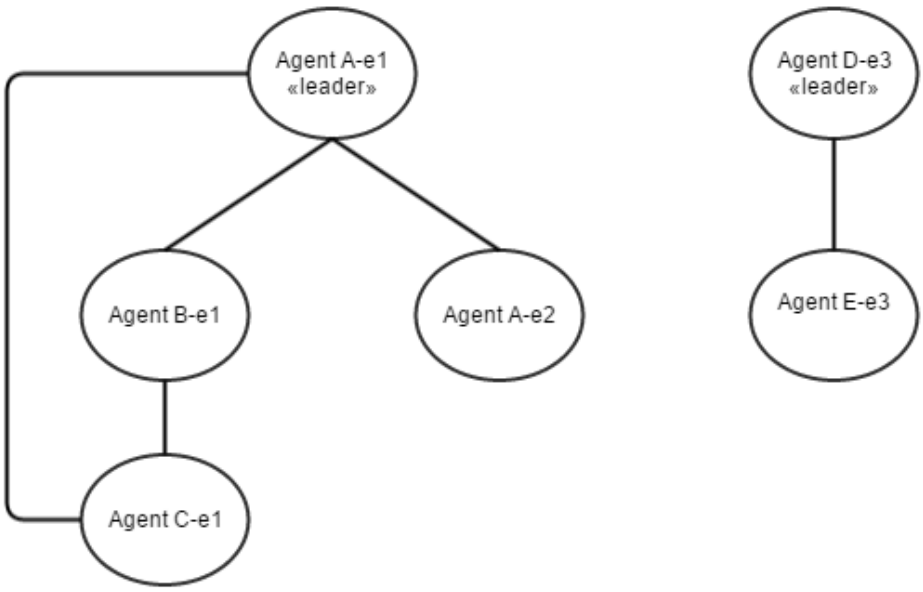

Figure 3: A visual representation of constraint graphs and elected leaders. Edges represent constraints. The first part of an agent's name represents the master agent, or user, and the second part represents the event that that agent is handling.

2.1.4 DFS Tree Generation Behaviour

The ADOPT algorithm requires agents to be organized in a pseudo-tree with each binary constraint representing a child-parent edge or pseudo-parent to pseudo-child edge, since each child can only have one parent to maintain the tree structure. As such, this behaviour was implemented to handle the pre-processing phase of ADOPT. Once the leader election algorithm terminates, this behaviour is initiated and the agents begin organizing themselves into DFS trees where the links are constraints. The DFS generation algorithm used was an adaptation of the pseudo-tree generation algorithm proposed by Thomas Lauété and Boi Faltings [11]. Only minor changes were made to the original algorithm, in order to properly detect when the first CHILD message is received.

When the algorithm begins, each agent will add all of the agents with which they share constraints as open neighbours. Additionally, the agents elected as leaders (one per isolated constraint graph) will choose one of their open neighbours, change their status from open to child and send them a CHILD message. Upon reception of a CHILD message, if an agent does not yet have a parent and is not a leader, they will mark the sender of the message as their parent, removing them from the set of open neighbours. After this, the receiver agent will choose an open neighbour, move them to their set of children and send them a CHILD message. If they have no open neighbours, they will send a CHILD message to their parent, if they are not a leader, and terminate. If an agent already has a parent when they receive a CHILD message, they will move the sender from the set of open neighbours to the set of pseudo-children and send them a PSEUDO message. Upon reception of a PSEUDO message, an agent moves the sender from their set of children to their set of pseudo-parents

and once more check for open neighbours to add as children, terminating if there aren't any.

When the algorithm terminates, the behaviour ends and the ADOPT behaviour is initiated so that the agents may begin calculating a solution.

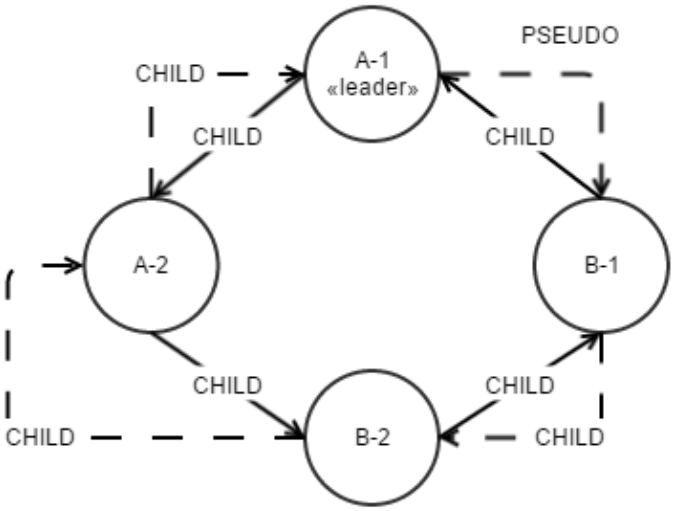

Figure 4: Example of an execution of the DFS generation algorithm. A-1 sends the first CHILD message to A-2 and then the messages follow the order of the arrows, then travel back up the tree following the dotted arrows.

2.1.5 ADOPT Behaviour

This behaviour is initiated once the DFS tree is generated and implements the ADOPT algorithm as proposed by Pragnesh Jay Modi [4].

Each agent starts the algorithm by initializing the lower bound for each value in their domain to zero and choosing the first value as the current value. The agents then send their current values to each of their children and pseudo-children through VALUE messages. When an agent receives a VALUE message from a neighbour, they evaluate the constraint between them and the neighbour for each possible value from their domain and add the cost of the constraint to the lower bound for each of their values. If the lower bound for the current value is higher than the lower bound for another possible value in the domain, a new value is chosen. The agent then sends the lower bound for their value with least lower bound to their parent in the pseudo-tree in a COST message that also contains the current context so as to allow the parent to detect obsolete messages.

When an agent receives a COST message from a child with a compatible context, they add the reported lower bound to the lower bound for their chosen value. If the lower bound for the current value becomes higher than the lower bound for another value in their domain, another value switch occurs in order to minimize the lower bound. Whenever a higher neighbour changers their variable value, the agents will re-initialize the lower bound for each value in their domain to zero.

The message exchanges and value swaps will continue occurring as mentioned until the lower bound for a value in the domain is also an upper bound and the

lower bound for all other values is higher than its lower bound. When this condition is true, the value for which the condition is verified is globally optimal for the agent until and unless a higher neighbour changed value. Once this condition is true at the root agent, they will send a TERMINATE message to its children and terminate itself. After receiving a TERMINATE message, an agent knows that all of its higher neighbours have terminated and once the termination condition is true for that agent, they will send TERMINATE messages to all of its children and terminate execution.

Additionally, the original implementation was extended to include THRESHOLD messages that are sent from parent to child containing a value for the current threshold when it is updated. When a child receives a THRESHOLD message with a compatible context (therefore not obsolete) it will update its threshold value which allows for a more efficient search of values within the domain

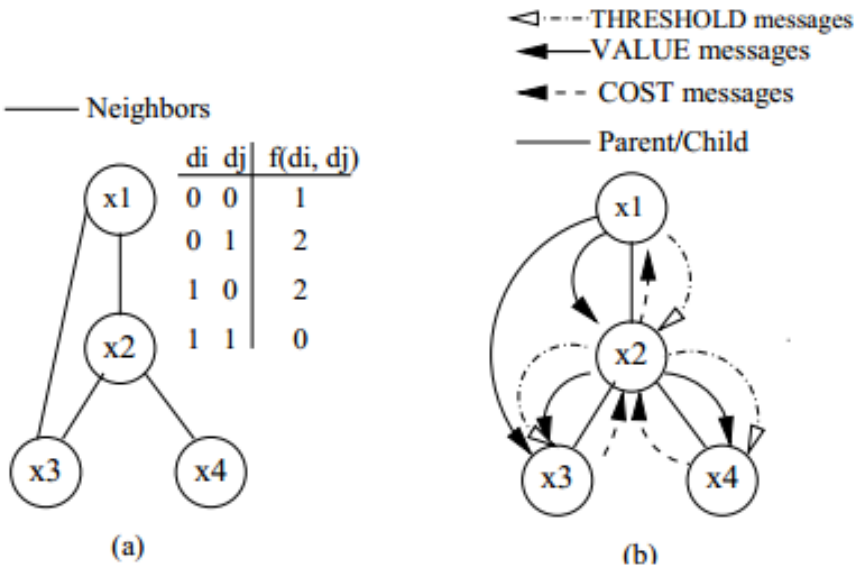

Figure 5: (a) Constraint graph. (b) Communication graph.

3 Development

3.1 Tools and Environment

3.1.1 JADE

This project was developed using the JADE [5] platform, which is a FIPA[6] compliant Java framework that enables the implementation of multi-agent systems. A JADE-based system can be distributed across multiple machines and multiple operative systems. The JADE software comes with multiple tooling that support debugging and run-time configuration. JADE is free software and is distributed by Telecom Italia in open source under the terms and conditions of the LGPL [7] license.

3.1.2 Other Tools

In addition to the JADE platform, this project was implemented in a Windows 8.1 environment, using the IDE [8] IntelliJ IDEA, developed by JetBrains [12]. The programming language in which the project was developed was Java 8, using JDK 1.8.0_25 [14]. This allowed the use of powerful programming expressions such as lambdas [9]. In addition, this version of Java already includes JavaFX [13] which was used to develop the graphical user interface. The Scene Builder 2 tool was also used in the development of the interface, making the design of the application easier and providing a simple method to implement an MVC[10] structure as the views were saved using an XML-based declarative markup language and bound to controllers in Java which in turn interacted with the application itself. Additionally, the ControlsFX library was used to aid in developing the interface as it provides several helpful UI features such as notifications and dialogs. Finally, the JSON-java library was used to simplify the process of serializing and parsing messages sent between agents.

3.2 Structure

3.2.1 Modules

The developed project can be divided into two main modules, the interface and the application. The interface consists of several views and controllers which allow the user to interact with the application itself. The application itself contains a generic Scheduler Agent with specific behaviours for each algorithm, as well as data structures for storing information relative to an event to be scheduled. Additionally, it also contains several helper classes and methods such as a Serializer, for converting messages into a JSON [11] format and parsing them, and a Statistics class to store information about the algorithms' performances.

[5] Java Agent DEvelopment Framework

[6] IEEE Foundation for Intelligent Physical Agents

[7] Lesser General Public License Version 2

[8] Integrated Development Environment

[9] Anonymous functions - A lambda expression in Java consists of a comma separated list of the formal parameters enclosed in parentheses, an arrow token (->), and a body. The data types of the parameters can always be omitted, as can the parentheses if there is only one parameter. The body can consist of a single statement or a statement block

[10] Model View Controller design pattern

[11] JavaScript Object Notation - It's a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate

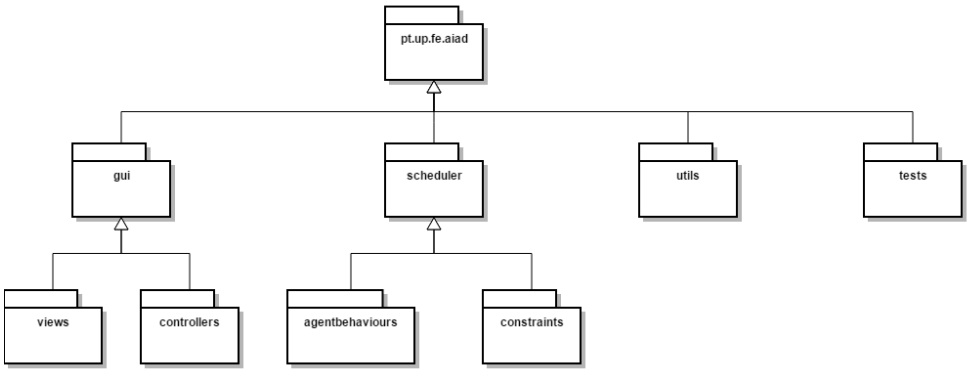

Figure 6: Simple overview of the UML package diagram.

3.2.2 UML Class Diagram

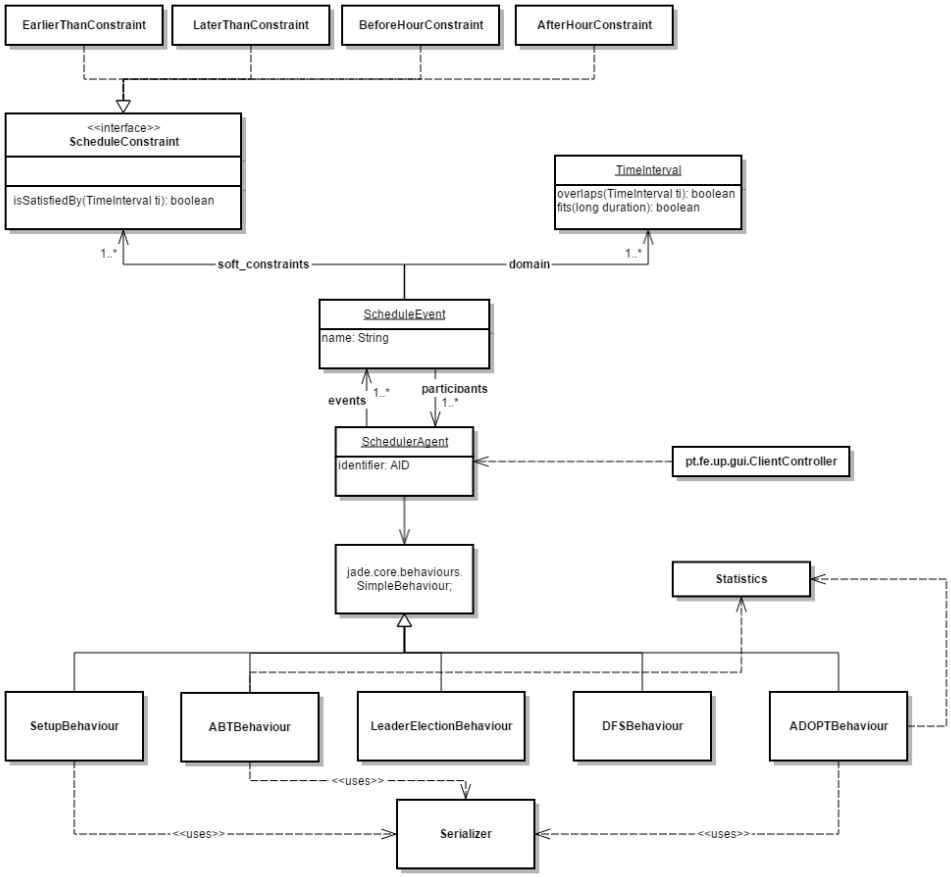

Figure 7: Simplified overview of the UML class diagram.

3.3 Relevant Implementation Details

It is worth mentioning that the implemented constraints allow a user to specify that they would like for an event to take place after a certain time and date (EarlierThanConstraint and LaterThanConstraint) and that they would prefer if an event took place after or before a certain hour, on any available day (AfterHourConstraint and BeforeHourConstraint).

Over the course of the development of the project multiple unit tests were created to validate the code.

4 Experiments

In order to analyse the performance of the implemented algorithms, the total number of messages sent during the execution was used as a metric for comparing ABT-opt and ADOPT since both of these algorithms are optimal. This was also tested and every solution obtained had the optimal cost. However, when comparing the complexity of the algorithms, while changes in the size of the domain and soft constraints affected both algorithms similarly, it was observed that an increase in the number of variables had very different consequences in each algorithm.

In order to study this difference, several experiments were conducted maintaining the size of the domain for each variable, which had size 2 with both options optimal, but progressively increasing the number of variables, which all shared equality constraints as they represented the same event but for different participants. The results obtained show that the number of messages required for ABT-opt to reach a solution increases polynomially with the number of agents, with an approximately quadratic scale factor. On the other hand, for the ADOPT algorithm, the number of messages increased exponentially, making it so that the algorithm soon stopped terminating in acceptable time periods.

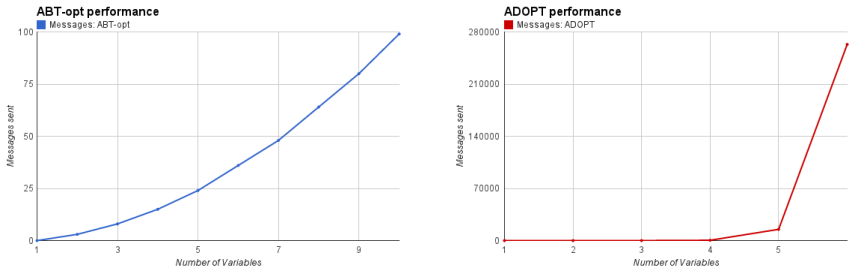

Figure 8: Performance results of the ABT-opt algorithm versus those of the ADOPT algorithm. As the charts show, ABT-opt's message count increases polynomially with the number of variables (almost quadratically) while ADOPT's increases exponentially.

It is also worth noting that the performance of ADOPT is greatly increased if more constraints are added, leading to more values of the domain not being optimal. However, ABT-opt still shows better results.

5 Conclusions

The experiments that were performed helped to analyse the complexity of the implemented algorithms. It was verified that using the ADOPT algorithm leads to the number of messages being sent growing exponentially with the number of variables that are being calculated. On the other hand, if the ABT-opt algorithm is used, this growth is polynomial. Both algorithms are complete and optimal [4, 10]. Taking this information into account, it was concluded that the ABT-opt algorithm would be more ideal for any real world applications since it scales better with the number of variables.

While developing this project it was also concluded that if privacy is not an issue a centralized algorithm would be more ideal for solving this type of problem as it would avoid communication delays and scale better with the number of variables to calculate. Therefore, the main advantage of using an M.A.S. [12] for solving schedule alignment problems is that it allows the algorithms to protect the privacy of the participants. While certain conclusions about a user's availability may still be deduced from the messages they send, most of the information and specific details are kept private. For example, with the implemented distributed algorithms, a user can not directly infer what events another user is participating in unless they are also participating in that same event.

6 Possible Improvements

Future work for this project could involve the implementation of additional algorithms for solving distributed constraint optimization problems, such as the Distributed Stochastic Algorithm and the Maximum Gain Message algorithm. Other algorithms for solving distributed meeting scheduling problems, such as the RR algorithm [13] and SCBJ [14], were also considered. Additionally, several improvements were considered for this project during its development in order to reduce the number of messages necessary to complete the execution of the implemented algorithms:

- Aggregating values to reduce message traffic since several values of the domain behave the same in respect to the constraints.
- Store obsolete NoGood's in ABT-opt to prevent agents from deriving the same information over and over again. This would greatly improve the algorithm's performance for exploring large domains.[10]
- Bounded error: by allowing agents to specify a bounded error for the solution, the performance of the algorithms can be increased in heavily constrained problems by obtaining possibly suboptimal solutions within the specified error bounds of the optimal cost.

[12] Multi-agent system

[13] Algorithm for solving distributed meeting scheduling problems which follows the Round Robin strategy

[14] Synchronous Conflict-based Backjumping

Bibliography

[1] Optimization in Multi-Agent Systems, `http://videolectures.net/ijcai2011_t3_optimization/`, 12 12 2014.

[2] Jade Site — Java Agent DEvelopment Framework, `http://jade.tilab.com/`, 12 12 2014.

[3] Multi-Agent Constraint Problem Solving, `http://www.emse.fr/~picard/cours/3A/masterWI/dcsp/lecture-DCSP.pdf`, 12 12 2014.

[4] Modi, Pragnesh Jay, P. 2003. *Distributed Constraint Optimization for Multiagent Systems*, University of Southern California

[5] Maheswaram, R. T.; Tambe, M.; Bowring, E.; Pearce, J. P.; Varakantham, P., P. 2004, *Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling*, University of Southern California

[6] Rodriguez, Ismel Brito, P. 2007. *Distributed Constraint Satisfaction*, Universitat Politècnica de Catalunya

[7] Yokoo, Makoto; Durfee, Edmund H.; Ishida, Toru; Kuwabara, Kazuhiro, *The Distributed Constraint Satisfaction Problem: Formalization and Algorithms*, IEEE Trans. on Knowledge and DATA Engineering, vol.10, No.5 September 1998

[8] Meisels, Amnon, *Distributed Search by Constrained Agents*, Springer-Verlag London Limited, 2008

[9] Yeoh, William; Felner, Ariel; Koenig, Sven, P. 2010, *BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm*, Journal of Artificial Intelligence Research 38

[10] Rossi, Francesca; Beek, Peter; Walsh, Toby, P. 2006 *Handbook of Constraint Programming*, Elsevier

[11] Léauté, Thomas; Faltings, Boi, P. 2013 *Protecting Privacy through Distributed Computation in Multi-agent Decision Making: Online Appendix 2: DFS Tree Generation Algorithm*, AI Access Foundation

[12] IntelliJ IDEA, `https://www.jetbrains.com/idea/`, 12 12 2014

[13] JavaFX and Scene Builder documentation, `http://docs.oracle.com/javase/8/javase-clienttechnologies.htm`, 12 12 2014

[14] Java8 documentation and tutorials, `http://docs.oracle.com/javase/8/docs/`, 12 12 2014

[15] ControlsFX site, `http://fxexperience.com/controlsfx/`, 12 12 2014

[16] JSON-java project, `https://github.com/douglascrockford/JSON-java`, 12 12 2014

Software

- IntelliJ IDEA - JAVA Integrated Development Environment
- JADE - JAVA Agent Development Framework
- JavaFX - The Rich Client Platform
- Scene Builder 2 - A Visual Layout Tool for JavaFX Applications
- ControlsFX - Open source project for JavaFX that aims to provide really high quality UI controls and other tools to complement the core JavaFX distribution
- JSON-java - An open source implementation of JSON encoders/decoders in Java

Work Division

- Duarte Duarte - 50%
- Luís Cleto - 50%

Appendix A: User Manual

How to Compile

The program can be compiled like any other Java code however JDK 1.8 is required. Main class is *pt.up.fe.aiad.gui.MainFrame*. JUnit 4 unit tests can be found in the package *pt.up.fe.aiad.tests*. All the dependencies are shipped with the submitted code.

Next to the submitted code there is an IntelliJ project ready to be used (e.g Execute the *iScheduler Run/Debug Configuration*).

How to Use

The created user interface is intended to be simple and intuitive to use without sacrificing features in the schedule alignment problem. An annotated list of the most important screens of the program follows.

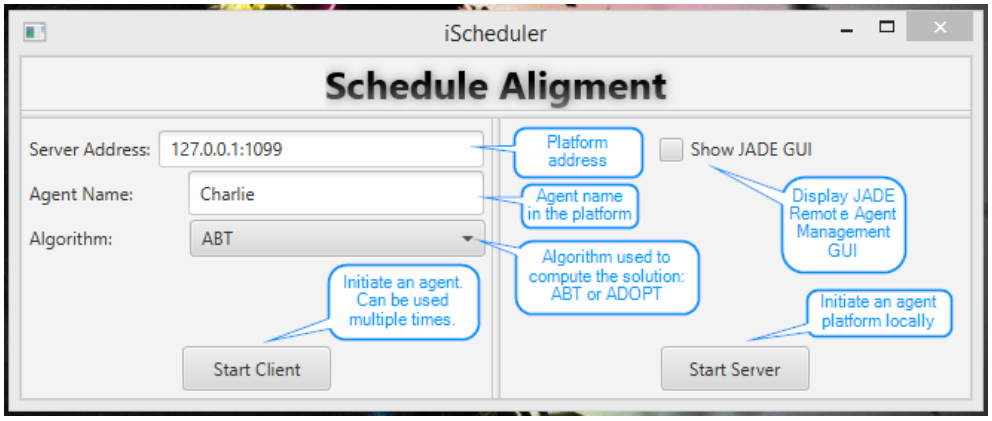

Figure 9: Main interface - start a JADE platform or create multiple agents

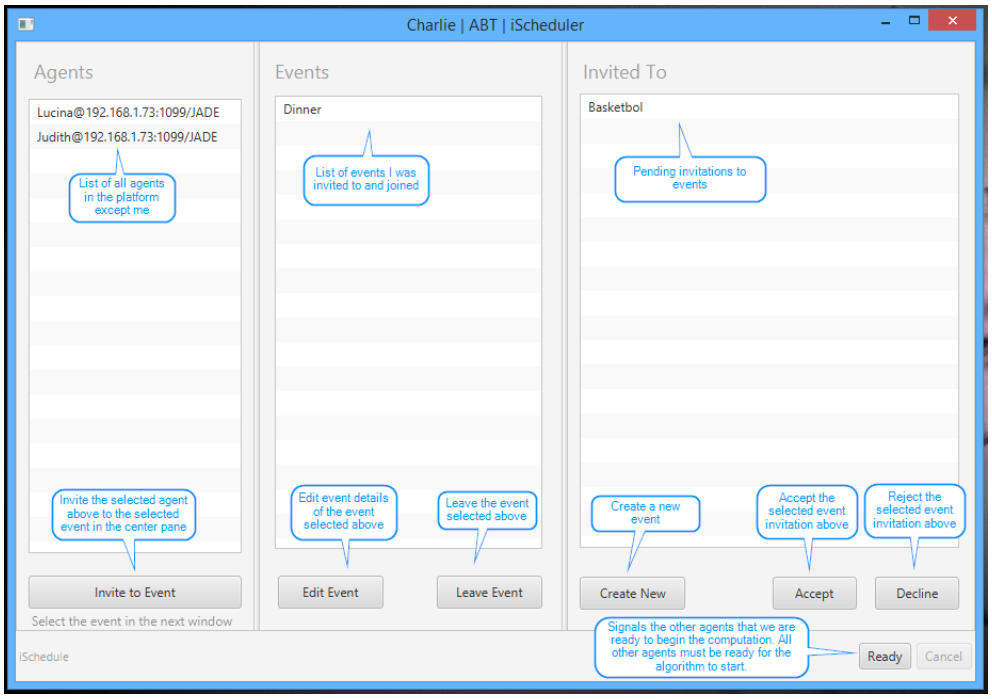

Figure 10: Main agent interface - display events, events invitations and other agents

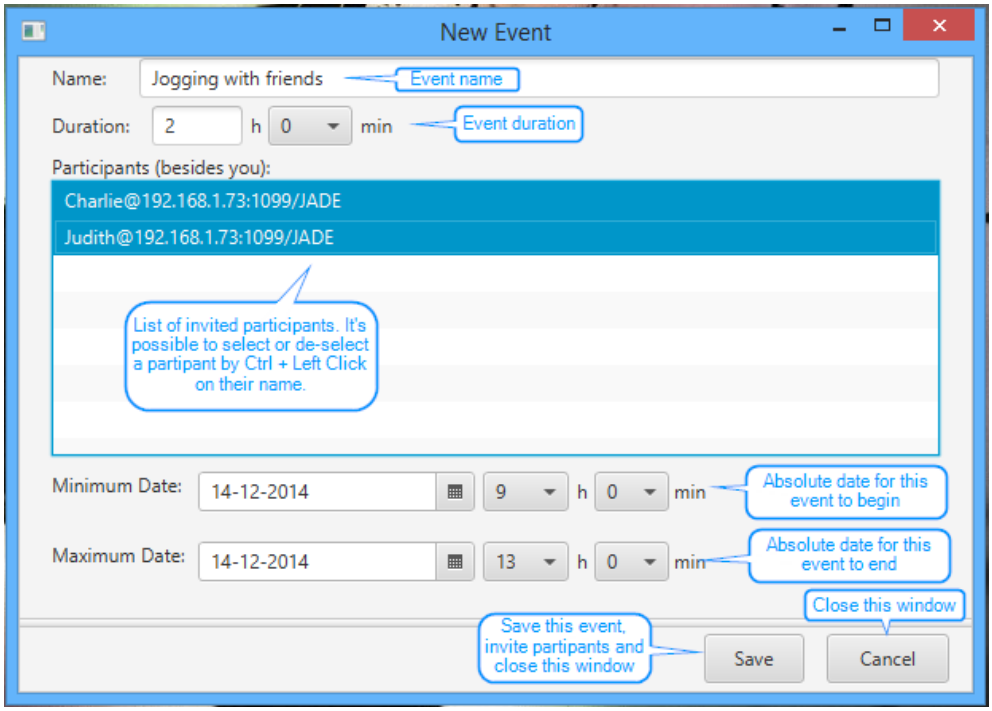

Figure 11: Create event interface - create a new event, define event details and invite other agents

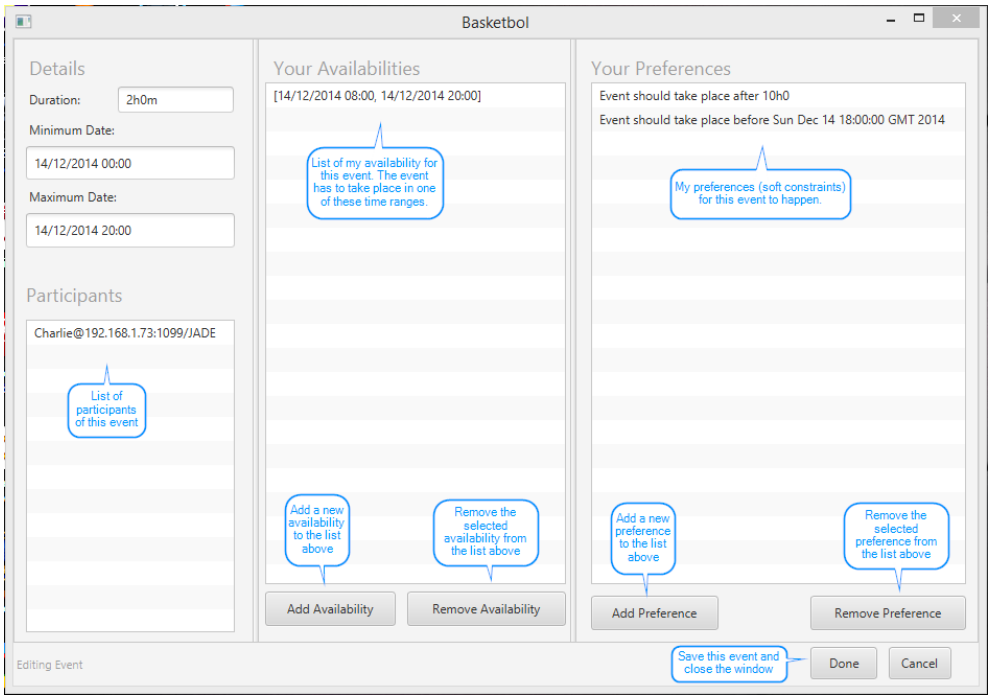

Figure 12: Edit event interface - accept an event invitation or edit an existing event, define your availabilities and preferences

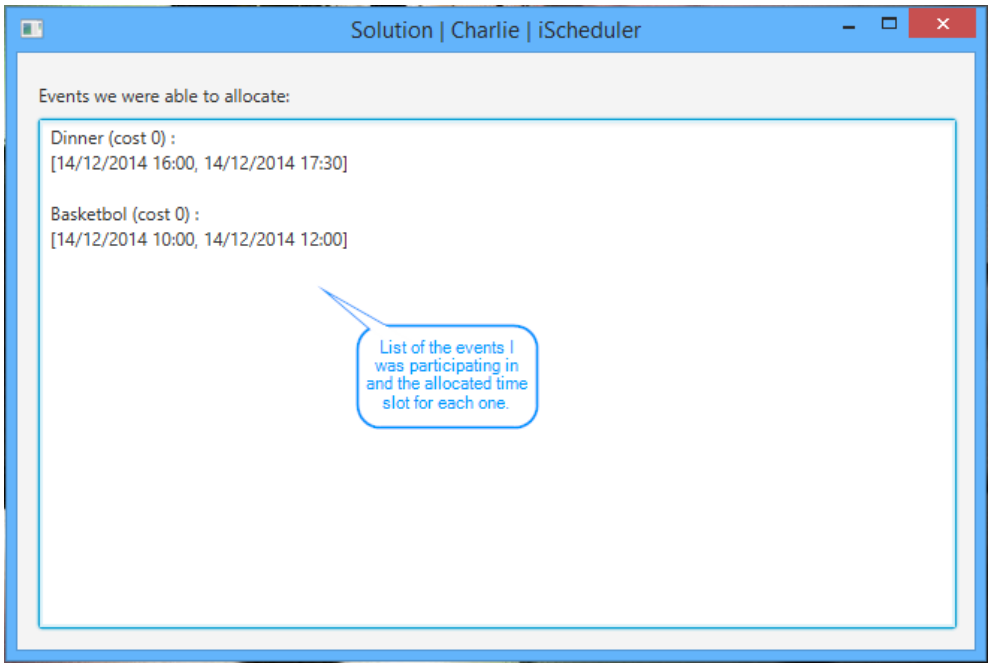

Figure 13: Solution interface - displays the allocated time slot for each event