University of Zurich[UZH]

Distributed scheduling using DCOPs in Signal/Collect

Thesis January 17, 2015

Daniel Hegglin
of Oerlikon ZH, Switzerland

Student-ID: 08-721-102
dani.hegglin@gmail.com

Advisor: **Mihaela Verman**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

First and foremost, I'd like to thank my advisor Mihaela Verman, Research Assistant and PhD Candidate at the DDIS group for her excellent support during the course of this thesis. She was always reachable and helped me with many technical and formal questions. Without her continous efforts and valuable inputs, this work would not have been possible.
Special thanks go to Dr. Thomas Scharrenbach for arranging the thesis proposal and his guidance at the beginning of this work. I'd also like to thank Philip Stutz for his work on the Signal/Collect framework that made my research possible and his support in technical questions regarding the usage of the framework.
Finally, I'd like to thank the Dynamic and Distributed Systems Group (DDIS) at the University of Zurich and Prof. Abraham Bernstein for the opportunity to write my master thesis in their department.

Zusammenfassung

Distributed Constraint Optimization (DCOP) ermöglicht Problemlösungen in beispielweise Terminplanung, Verkehrsflusskontrolle oder dem Management von Sensor Netzwerken. Es ist ein gut erforschtes Feld und es wurden viele verschieden Algorithmen zur Berechnung vorgestellt. Allerdings wird häufig von einer statischen Problemdefinition ausgegangen und der Aspekt von in der Realität häufig auftretenden Änderungen an der Problemstellung findet oft wenig Beachtung. Ausserdem fehlt es an einem soliden theoretischen Fundament und standardisierten Verfahren um die Performanz von DCOP Algorithmen hinsichtlich sich ändernder Probleme zu erfassen. Diese Arbeit hatte das Ziel das Verhalten und die Leistung von verschieden Arten von DCOP Algorithmen in dynamischen Umgebungen mit einem Fokus auf lokale, iterative Algorithmen und Hauptaugenmerk auf den MaxSum Algorithmus zu untersuchen. Zum Vergleich wurde eine komplette und eine lokal, iterative "message-passing" sowie eine "best-response" Variante implementiert. Während der Implementation des MaxSum Algorithmus wurde eine Variation von der Ã¼blichen Graphenstruktur ausprobiert. Zum Test eines realen Problems wurde Terminplanung ausgewählt und als DCOP formuliert. Es wurde ausserdem ein Framework entwickelt, welches die dynamische Änderungen von Constraints, Variablen und der Problemdomäne ermöglicht. Die Algorithmen wurden mit Fokus auf Qualität über Zeit, sowohl in einer statischen wie auch in einer dynamischen Umgebung getestet. Diese Arbeit schlägt ausserdem eine Lösung zur Speicherung, Weiterverarbeitung und Überwachung der Resultate der Berechnungen in Echtzeit vor, welche die Performanz der Algorithmen nicht beeinflusst.

Abstract

Distributed constraint optimization allows to solve problems in domains like scheduling, traffic flow management or sensor network management. It is a well-researched field and various algorithms have been proposed. However, the dynamic nature of some of these problems in the real world have been overlooked by researchers and problems are often assumed to be static during the course of the computation. The benchmarking of distributed constraint optimization algorithms (DCOP) with changing problem definitions currently lacks a solid theoretical foundation and standardized protocols. This thesis aimed to measure the performance of different types of DCOP algorithms on dynamic problems with a focus on local-iterative algorithms and especially on the MaxSum algorithm and possibly contribute to the field. A complete, a local-iterative message-passing and a local-iterative approximate best-response algorithm for distributed constraint optimization have been implemented for comparison. In the implementation of the MaxSum algorithm, a variation of the usual graph structure has been attempted. As a real-world use case for benchmarking, the meeting scheduling problem has been mapped as distributed constraint optimization problem. A framework has been designed that allows dynamic changes to constraints, variables and the problem domain during run-time. The algorithms have been benchmarked in a static, as well as in a dynamic environment with various parameters and with a focus on solution quality over time. This thesis further proposes a solution to store, further process and monitor the results of the computation in real-time without affecting the performance of the algorithms.

Table of Contents

1

Introduction

1.1 Motivation & Goal

Distributed constraint optimization allows to solve a broad category of problems where multiple agents are involved and a global utility function needs to be optimized. Problems range from graph coloring [Modi et al., 2005] to task allocation and scheduling [Zhang et al., 2002], traffic congestion management [Leeuwen et al., 2002] or to disaster recovery [Hiroaki et al., 1999]. The distributed variant of constraint optimization has been extensively adressed by researchers and numerous algorithms with varying design approaches have been proposed. However, these algorithms were often designed based on the premise that problems are static in their predefined state and do not change over the course of the problem solving process. A form of dynamic changes can be achieved with a step by step procedure, where the problem definition is updated after each step, but such a protocol does not work in a distributed manner with multiple involved agents [Petcu and Faltings, 2007]. Many distributed problems have dynamic properties and in a world with ever-increasing complexity and speed, those become continually more relevant for real-world applications. Constraints can change, but also the involved variables as well as the problem domain itself. Imagine for example an optimization software of a global logistics company where vehicles can get damaged and orders could be changed to other adresses or even canceled. A recalculation of the complete problem set with a new static definition seems inefficient. Research in dynamic distributed constraint optimization has started to gain momentum. The benchmarking of distributed constraint optimization algorithms with changing problem definitions currently lacks a solid theoretical foundation, but researchers have started to develop benchmarking protocols that aim to standardize the process [Mailler and Zheng, 2014].

This thesis tries to explore dynamic distributed constraint optimization by implementing a complete and two local-iterative variations of algorithm approaches and compare their performance in a dynamic environment. The complete algorithm acts as a baseline and the main focus is on the performance of the local-iterative types of algorithms. They do not guarantee complete solutions but are able to provide a good solution in a faster way as they calculate utilities on a local level with a lower communication overhead. They also have been proven to be more scalable because of this lack of organizational overhead

[Chapman et al., 2011]. These attributes indicate their potential ability to adapt faster to problem changes and keep a better stability. The main focus lays on the abilities of the MaxSum algorithm, which is a local-iterative message-passing algorithm. It is further a goal to show ways of benchmarking these type of problems from various aspects. The real-world use case for the thesis will be meeting scheduling and the software will be implemented with the Signal/Collect framework, a graph processing engine developed at the Dynamic and Distributed System Group at the Departement of Informatics of the University of Zurich.

The implemented software is made available online under the Apache license version 2.0.[1][2]

1.2 Structure

First, an overview will be given about various definitions and aspects of constraint optimization in general, as well as the aspects of the distributed and dynamic variations. Further, an examination will be provided about different approaches of algorithms to solve constraint optimization problems and their advantages and disadvantages in context of solution quality over time, scalability and adaptability to changes.

Secondly, the meeting scheduling problem definition and the mapping to a distributed constraint optimization problem, as well as the algorithm mapping to the Signal/Collect programming model will be detailed in the design chapter. Further, the design considerations for a framework for dynamic changes will be explained and the solution for data collection will also be briefly introduced.

Finally, the performed benchmarks will be evaluated and discussed. In a first series, the algorithms will be tested in static environments to evaluate the implementation. In a second series, various tests on changing constraints, variables and the domain with different rates and different problem densities will be run to determine the performance of the algorithms in dynamic environments. To wrap up, further work possibilities and limitations of the thesis will be pointed out and a conclusion about the achieved contributions and results will be given.

[1] https://github.com/danihegglin/DynDCO
[2] http://www.apache.org/licenses/LICENSE-2.0

2

Background & Related Work

In this section, constraint optimization and the distributed, as well as dynamic variants are briefly explained and brought into context of the related work. Also, the meeting scheduling problem will be described and different algorithm designs and their advantages and disadvantages are going to be briefly discussed.

2.1 Dynamic Distributed Constraint Optimization

A constraint optimization problem (COP) contains a set of variables $V = \{V_1, V_2, ..., V_n\}$. These variables are assigned to a value or state $s_j \in S_j$, which is contained in a set of possible values defined by a finite problem domain $D = \{D_1, D_2, ..., D_n\}$. A constraint $C =< V_c, R_c >$ contains one (unary), two (binary) or multiple (k-ary) variables and their relationship. The constraint defines a rule for the variables that needs to be fulfilled. One of those rules could be that none of the variables should take the same value. This would for example be the case for a meeting scheduling problem where none of the meetings should take place at the same time.
A utility function for the constraint c_k on variable state s in the form of $u_{c_k}(s_{c_k})$ needs to be formulated that defines a certain cost respectively reward for a given configuration of the involved states. The global utility function u_g would then be the summation of all utility functions of all constraints.

$$u_g(s) = u_{c_1}(s_{c_1}) \oplus \cdots \oplus u_{c_k}(s_{c_k}) \oplus \cdots \oplus u_{c_l}(s_{c_l})$$

Constraints can be attributed with varying levels of importance through weighting. One can, instead of so-called soft constraints, define hard constraints by multiplying their utility instead of using addition in the global utility function. By defining the utility of a violated hard constraint as 0, the global utility would also go to 0 if this hard constraint is not satisfied [Chapman et al., 2011, Petcu and Faltings, 2003]. A problem only containing hard constraints would represent a constraint satisfaction problem (CSP). A formal definition of such a combined utility function including soft constraints (SC) and hard constraints (HC) would look like the following formula, where the product of all hard constraints is multiplied by the sum of all utilities of a state s in the soft constraint utility functions:

$$u_g(s) = \prod_{hc_k \in HC} u_{SC_g}(s) \Big(\sum_{sc_k \in SC} u_{SC_g}(s) \Big)$$

The definition of a distributed constraint optimization problem (DCOP) extends the basic constraint optimization by distributing sets of variables to autonomous agents. These agents all have the goal to maximize the utility of their variables in a private utility function and thereby also contribute to a global utility function. Agent's whose variables are linked to a common constraint are called neighbours [Chapman et al., 2011, Farinelli et al., 2012, Petcu and Faltings, 2003].

The problem definition is, as a further extension to the optimization, moved from a static to a dynamic attribute. Constraints can change and therefore change neighbourhoods and the outcome of private and global utility functions. A change of constraints inherently changes the area of satisfying solutions if hard constraints have been included in a problem definition. [Nguyen and Yao, 2012] state that changing the constraints might lead to the discovery of a better global optima. [Mailler and Zheng, 2014] define a dynamic DCSP as a sequence of DCSPs $\{P_0, P_1, ..., P_n\}$ where every DCSP is a static problem definition. P_i is therefore a result of the previous DSCP in the sequence and of the added and removed constraints: $P_i = P_{i-1} + c_{i^a} - c_{i^r}$. This definition should also hold for DCOPs. Utility functions could also be dynamically changed. Modifying this property could especially have an impact on real-world problems like meeting scheduling, where it could move the global optima from one disconnected solution space to another [Nguyen and Yao, 2012]. Furthermore, variables could be added or removed in a dynamic setting and the problem domain D also could be changed during the course of the problem solving process.

2.2 Meeting Scheduling Problem

Scheduling is the problem of allocating tasks to a given set of ressources in an optimal order. The meeting scheduling problem is an exemplary type of this family of problems and is supposedly well-know to all of us. Participants of a meeting have private schedules with preferences when a meeting should be held according to their calendar. The challenge is to identify a time for a meeting that maximizes the preferences of all participants while being valid in then sense that every person is able to attend [Farinelli et al., 2012]. [Angulo and Godo, 2007] have formally defined a meeting scheduling problem as:

- $P = p1, p, ...pn$ is the set of people where every person has a calendar that hilds r slots, $S = s1, s2, s3$
- $M = m1, m2, ..., m3$ is a set of k meetings
- $At = at1, at2, ..., atk$ defines all attendee's of a meeting

The c parameter has been neglected as it is not relevant to this thesis. From the definition of a valid solution, one can derive two important criteria to the problem solving process:

Validity Criterium 1. All participants need to agree on the same time for the meeting.

Validity Criterium 2. Meetings need to be scheduled in a way that there are no overlaps of meeting times in the schedules of the participants.

There is further an inherent privacy aspect to the problem. Meeting participants are often not willing to share their schedules with others except to find a time for the specific meeting. It will later be shown that some of the algorithms can guarantee this privacy to a certain degree. The meeting scheduling problem will be mapped as a distributed constraint optimization problem in the design chapter [Farinelli et al., 2012] [Angulo and Godo, 2007].

2.3 Algorithm Design Approaches

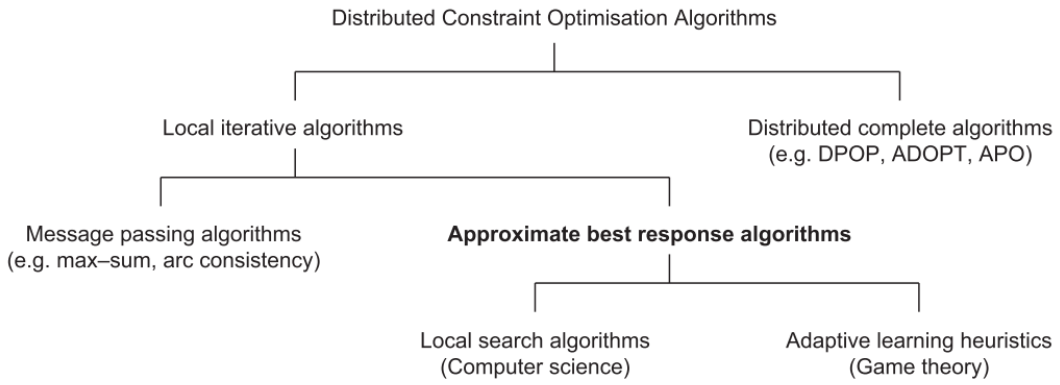


Figure 2.1: Categorization of DCO algorithms [Chapman et al., 2011]

[Chapman et al., 2011] categorize distributed constraint optimization algorithms into local-iterative and distributed complete algorithms. They further divide local-iterative into message-passing algorithms and approximate best-response algorithms (Figure 2.1). The following subsections are going to explain the differences between the three categories and introduce the specific algorithms, which have been chosen from these three different approaches for benchmarking. Advantages, as well as disadvantages will be described and which behaviour one can expect of these algorithms under certain parameter configurations.

2.3.1 Distributed Complete

Distributed complete algorithms always discover a configuration of value assignments for a set of variables that maximizes the global utility function. This completeness guarantee increases the complexity of computation and leads to exponentially growing message numbers or calculations when increasing the amount of variables in a problem. Messages between agents often contain complex structures and constraint problems usually need to be transformed to a extensive graph structure [Chapman et al., 2011]. These types of algorithms therefore are not expected to scale well and quickly find qualitative solutions, but they fit well if one wants to find the maximal utility of a problem.

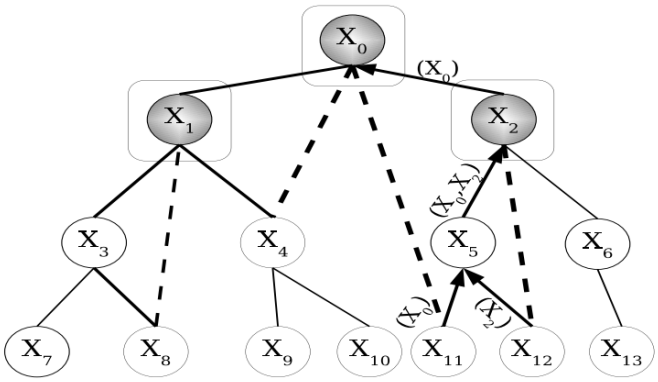


Figure 2.2: Pseudotree for DPOP [Petcu and Faltings, 2003]

For this thesis, it was decided to implement the Dynamic Programming OPptimization algorithm (DPOP) proposed by [Petcu and Faltings, 2003] as a comparison to the local-iterative approaches. In this algorithm, constraint optimization problems need to be converted to a pseudotree (Fig. 2.2), which is a modification of a DFS Tree. The original DCOP graph is transformed in a way that previous neighbours are placed in the same branches of a binary tree. They are connected trough ordinary tree edgesand additionally, so-called back-edges between unconnected previous neighbours are established. The leaf nodes propose UTIL messages containing their utility values for each value assignment upwards the tree and the root node sends a VALUE message downwards, containing the best value to choose as a variable state. Nodes in the middle of the tree propagate UTIL and VALUE messages. The message structure is fairly complex as it involves all the utilities of the pseudoparents connected by the back-edges and their context in the graph, which increases the message size exponentially. The number of messages on the other hand is linear [Petcu and Faltings, 2003].

2.3.2 Local-Iterative - Best Response

In a local-iterative best-response algorithm, agents only communicate their current state, e.g. their value assignment and react to these value messages in the best possible way from their perspective. The agents are only connected to their neighbours with

whom they share constraints and there exists no complex graph structure controlling the message flow [Chapman et al., 2011]. Through this local property, the types of algorithms should be inherently scalable as the messages and computations do not increase exponentially. Further, this approach is optimal from a privacy perspective as the neighbours only share their current preference and no other details of their schedule [Chapman et al., 2010].

For this thesis, it was decided to implement the Maximum-Gain Messaging algorithm (MGM). In this algorithm, agents calculate the maximal gain in utility they can achieve when assigning to another value and send this value as a message. If of all messages they received, they have the highest gain, the value is changed. Otherwise the local value stays the same. This algorithm fullfills the anytime property, i.e it can provide a solution at every timepoint during calculation and also reaches good solutions quickly [Chapman et al., 2010]. As the decision of an agent depends on a complete set of message of all it's neighbours, this algorithm will supposedly not perform well in asynchronous running mode. This type of algorithm does further not always converge and does not always provide the optimal solution to a problem.

2.3.3 Local-Iterative - Message Passing

The difference of message-passing to best-response algorithms lays in the fact that the agents send and receive messages containing a specific data structure, which contains the utilities respectively costs that various assignments hold for a local variable. Received messages are used to calculate the next message, which is sent to the connected neighbours. These types of algorithms are, like best-response algorithms, able to provide an acceptable solution in a short period of time, but also share the charasteric to sometimes not converge or not providing an optimal solution [Chapman et al., 2011].

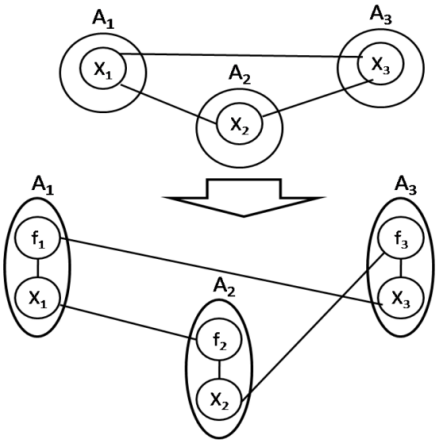


Figure 2.3: Conversion of a general DCOP to a factor graph [Zivan and Peled, 2012]

For this thesis, it was decided to implement the MaxSum algorithm introduced by

[Farinelli et al., 2008]. The algorithm has currently gotten a lot of attention from researchers. For this work, the algorithm is especially interesting because of its proposed abilities in dynamic environments. [Farinelli et al., 2008] wrote in their paper:

> [...] we note that if messages are continuously propagated, and the states of the agents are continuously updated, then the algorithm may be applied to dynamic problems where the interactions between agents, or the utilities resulting from these interactions, may change at any time.

In MaxSum, the original DCOP graph is transformed to a factor graph, which is a form of a bipartite graph and of cyclic nature (Fig. 2.3). An agent is after the transformation made up of a variable and a function node, whereas variables are connected to all corresponding function nodes of their previous neighbours. The function nodes are vice versa connected to all previous neighbours of its variable node. The messages sent from variable nodes differ from the function nodes. A message from variable to function contains for every value $d \in D_x$ the sum of utilities regarding this value, which the node has received from all connected function nodes. It is important to note that this sum does not include the values provided by the message target. The values are normalized at this point to avoid an infinite increase of the sum of the utilities. A message from a function node to a variable node holds for every value $d \in D_x$ the summation of all costs received from all connected variable nodes except the message receiver and the original cost of the constraint represented by the function node [Zivan and Peled, 2012].

3

Design

In this section, the benchmark problem will be defined and mapped as DCOP, the framework design will be explained and the mapping of the algorithms on to the Signal/Collect framework will be described. Additionally, the design and considerations regarding the monitoring platform will be presented.

3.1 Meeting Scheduling Problem

3.1.1 Formal Definition as DCOP

The formulation of the meeting scheduling problem follows the basic definition of a distributed constraint optimization problem. Agents, variables and their relationships, as well as constraints shall be formulated. The components of a meeting scheduling problem are participants, their schedule, meetings and a given timeframe. For the sake of simplicity, it was decided to not take travel time between meetings or other parameters into consideration as certain researchers have done. It was also decided to use utilities instead of costs.

Definition 1. *Participant - has preferences and meeting he/she need to attend*

Definition 2. *Meeting - has participants and needs to be held at an agreable time*

[Maheswaran and Tambe, 2004] propose three different ways of mapping a meeting scheduling problem to variables (Figure 3.1). TSAV (Time Slots As Variables), EAV (Event As Variables) and PEAV (Private Events As Variables). In EAV, every participant holds a private variable containing the preference value for a specific event. PEAV is a modification of the EAV paradigm where agents do not share their local valuations. It was decided to follow the PEAV principle and model every meeting participation of an agent as one variable instead of using timeslots as variables. An agent therefore can hold multiple variables. This paradigm has also been tried by other researchers, which further established confidence in the decision [Petcu and Faltings, 2003].

Definition 3. *Agent - holds one variable per meeting participation*

Definition 4. *Variable - represents one meeting participation*

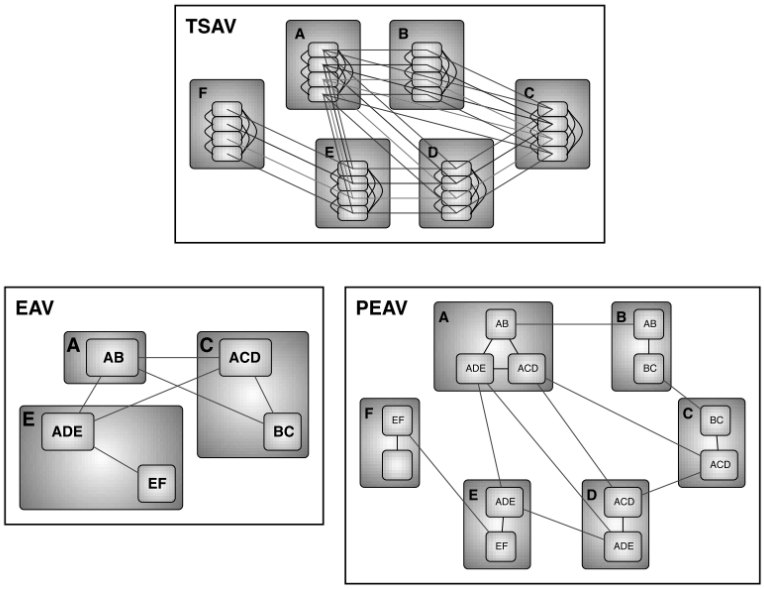


Figure 3.1: Different paradigms of mapping the meeting scheduling Problem [Maheswaran and Tambe, 2004].

A variable takes on a value $s_i \in S_i$ in a defined problem domain D_i. In the formulation of the meeting scheduling problem, the domain represents a finite set of timeslots and the variable assigns to one of these timeslots. This value represents the currently locally chosen timeslot for a specific meeting.

Definition 5. *Domain - holds a finite set of possible timeslots to schedule a meeting*

Definition 6. *Value - assignment to a timeslot of the available timeslots in the Domain*

From the problem definition in chapter 2, one can derive soft and hard constraint for the meeting scheduling problem. Soft constraints can possibly be constructed from the preferences of the participants and utilized to maximize the utility [Franzin et al., 2002]. Three differently weighted soft constraints have been defined to model preferences of agents. Preferred timeslots gain the highest utility, followed by free timeslots and blocked timeslots, which gain no value at all. Further, a timeliness soft constraint was defined that adds a higher utility to earlier timeslots. All of the soft constraints have unary relationships, i.e. are local. Additionally, two hard constraints with k-ary relationships to variable neighbours need to be formulated. The first would be an equality constraint on the assigned timeslot value for a specific meeting between all variables related to this meeting. The second is a difference constraint of assigned values between all variables of an agent [Farinelli et al., 2012] [Angulo and Godo, 2007]. A local utility function $u_l(s)$ would therefore include the sum of all soft constraints multiplied by the product of the

hard constraints analogous to the global function defined in chapter 2.

$$u_l(s) = \prod_{hc_k \in HC} u_{SC_g}(s) \Big(\sum_{sc_k \in SC} u_{SC_g}(s) \Big)$$

The conclusions from this formal definition in regards to the general structure of the algorithms is to have one variable represent each meeting participation of an agent. The agent is therefore an abstract definition of a set of meeting participation nodes. All variables of an agent should share a reference to an integrating agent vector, where meeting times are registered. This agent vector acts as a difference hard constraint between the different meeting participations of an agent. Further, all variables attending a meeting also share a reference to the meeting vector where every agent shares his current preference. This represents the aforementioned equality hard constraint. It was further decided to implement the given local utility function in the framework as a generalized method, as the structure repeats itself in all thre algorithms and because it is further helpful for comparison to have the exact same utility function implemented.

3.1.2 Problem Dataset Generation

During the course of the thesis, it was necessary to find a dataset for the benchmarking. The Frodo2[1] framework or for example the dataset from AAMAS 2004 [2] do provide a couple of datasets for meeting scheduling. But because it was considered that in the benchmarks one would need to be able to produce problems with different densities and scale to high numbers of participants, as well as change constraints dynamically it was decided to generate meeting participations and agent schedules randomly. It was also chosen to limit the number of meeting participations per agent to keep the benchmarks more realistic analogously to [Chun et al., 2003].

- The blocked timeslots in a schedule are based on the percentage given trough the density parameter
- Preferences for meetings are chosen randomly from free timeslots in the schedule
- The number of meeting participations is chosen randomly from 1-5
- The meeting participations are chosen at random

[1] http://frodo2.sourceforge.net
[2] http://teamcore.usc.edu/dcop/

3.2 Framework

3.2.1 Signal / Collect

The foundation of the implementations in this thesis is the Signal/Collect framework[3] [Stutz et al., 2010], which is built on top of Akka[4] and written in Scala[5]. It is a graph processing engine with a programming paradigm comparable to Map/Reduce [Dean and Ghemawat, 2008]. The main components are vertices and edges, as well as a graph structure where those components are added. A vertex has a state and sends signals along it's edges to connected vertices, which can contain any datatype. The signal usually is the state of the vertex or calculated in context of it. Vertices gather the signals of connected nodes and run the defined collect function on the received information. The state of a vertex is usually adjusted according to the results of these calculations and the next signal that is sent will include this state. This model allows to reduce complex algorithms to a few lines of code and is applicable for many problems. The framework further has the capability of running graph processings asynchronously or with synchronous signal steps and it is possible to distribute the system on multiple machines. Reasons for choosing this framework are the excellent structural fit for distributed constraint optimization problems, synchronous and asynchronous run modes and the possiblity to add and remove vertices during run-time as it allows for dynamically changing problems.

3.2.2 Structure & Functionality

A specific framework for benchmarking dynamic problems has been implemented. It was considered that the basic structure of the framework should help with the control of the benchmark runs and add the general ability to change a problem during runtime. This functionality was designed as problem agnostic and abstracted. The main hierarchy in the framework is the vertex stack. A `BasicVertex` has been implement containing a basic convergence function and control parameters related to the number of signal steps. Further, a `MeetingSchedulingVertex` has been implemented. This vertex implements all generalizable functions of the three benchmark algorithms described in chapter 3.1. A main component is the handling of the agent vector and the meeting vectors, a convergence function for meeting scheduling, the local utility function and data storage functions. To generalize dynamic change functions, a `DynamicVertex` has been created, which implements methods for changing constraints and the value domain of the vertex. The specific vertex implementations of the three implement DCOP algorithms extend the `DynamicVertex`.

The initial design consideration to introduce change to the constraints of the agents

[3] http://uzh.github.io/signal-collect
[4] http://akka.io
[5] http://www.scala-lang.org

has been to create a special vertex as part of the graph in Signal/Collect as the framework supports multiple types of vertices and messages in one graph execution. However, the vertex needed to be paused in the case of interval changes for a certain time and this caused errors during execution. Akka distributes multiple actors to the same thread and through pausing the `DynamicVertex`, other vertices were blocked. It therefore was decided to run the change controller in a separate thread alongside the graph execution. The main ability of the controller is to change constraints at a given interval and percentage (of all constraints in the problem). This allows testing of the stability of the algorithms. It is further possible to run a single change after a given interval. In the case of the meeting scheduling problem implementation this is only related to soft constraints as hard constraints can be handled by adding or removing variables with the second function. The controller also implements a variable change function, which similarily can be run at a certain interval or at one timepoint. One can add parameters to create a new neighbourhood or use existing relationships and add new variables or remove existing vertices. Instead of being defined by percentage, this change is given by number. As a third function, the controller does change the domain in the whole problem for all agents. In the use case of meeting scheduling, this increases or reduces the available timeslots.

The parameters and run modes of the framework have been designed with the benchmarks of this thesis in mind. It is possible to pass general run parameters for algorithm type, for Signal/Collect run mode (synchronous/asynchronous) and one can specify to the software to run in normal mode or in one of the dynamic change variations (`changeConstraints`, `changeVariable`, `changeDomain`) with specific parameters. It is further possible to add meeting scheduling specific parameters. For this thesis, the parameters for problem density (blocked timeslots percentage), timeslots, number of meetings and number of meeting participants were defined. For testing purposes, it is possible to start the framework via SingleTest or MultiTest. SingleTest runs one setting once and MultiTest allows to specifiy a range and scale of agents and meetings that should be tested.

3.2.3 Monitoring Platform

The storage and monitoring of the utilities, quality levels, conflicts and run statistics of the calculations is usually done by writing the results to a log file. It was decided to use an alternative method during the course of this thesis. Mainly because it was a desirable function to automatically post-process the results of the bechmarks on a detached machine and because real-time visibility was considered to be useful during the implementation of the algorithms.
Sending the results with non-blocking asynchronous HTTP requests[6] to a restful API on a dedicated server was considered to be a viable option and worth trying out. The Play

[6] http://dispatch.databinder.net

Framework[7] has been chosen for implementation because it is highly scalable and able to handle thousands of simultanous connections, is lightweight as well as non-blocking and allows to process results on-the-fly with code written in Java or Scala. It was also chosen because the Akka framework, which is also the foundation of Signal/Collect is tightly integrated and the actors concept is an integral part of the platform. For every benchmarking run, an actor is created that handles all the relevant incoming messages. It is therefore possible to run multiple benchmarks in parallel. The framework further allows to visualize the global utility of the graph in real-time via websockets.

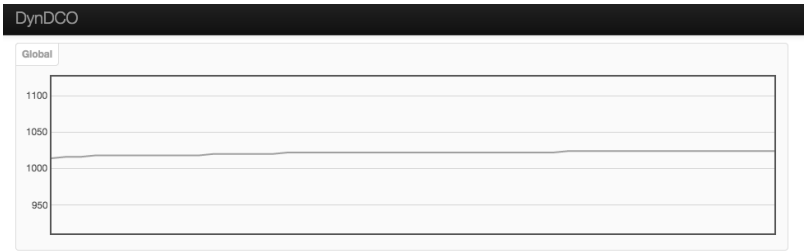


Figure 3.2: The real-time view of the Monitoring Platform

3.3 Mapping of DPOP

The following sections describes the mapping of the graph structure of the DPOP algorithm and the behaviour of it's nodes described in chapter 2.3.1 to the Signal/Collect Framework in regards to the meeting scheduling problem. The implementation is based on [Petcu and Faltings, 2003].

3.3.1 Graph Structure

Following the definition of [Petcu and Faltings, 2003] and the descriptions in chapter 2.3.1, the graph was arranged as a pseudotree. As mentioned in chapter 3.1.1, every agent is represented through meeting participation nodes as a general rule for the algorithm implementation. To create a pseudotree, one needs to transform an existing graph. Therefore, a graph has been constructed where all meeting participation vertices that share the same meeting are connected with edges. The chosen implementations of Signal/Collect components for the graph are the `DataGraphVertex` and the `StateForwarderEdge` classes. After the inital graph is created it is transformed to a pseudotree, which is a root tree with binary parent-child relationships that contains the same vertices as the original graph. An important attribute is that previously connected vertices are put into the same branch [Petcu and Faltings, 2003]. As one can see in Figure 3.3, the original participants have been put into the tree according to this rule. The visible participant of meeting three is inserted into the tree at a lower level than

[7] https://www.playframework.com

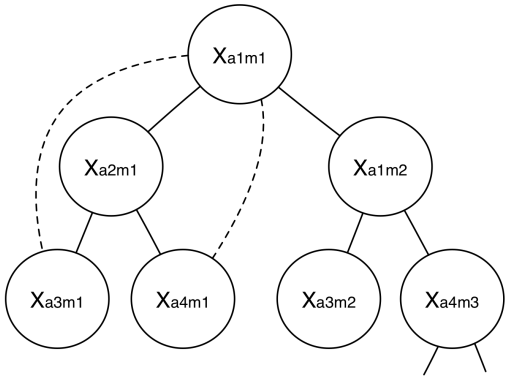


Figure 3.3: An example DPOP pseudotree

the vertices of meeting one and two, but does also hold this property. The original edge connections remain as so-called back-edges (dashed line) and the new connections are also implemented as Signal/Collect edges (solid line). This tree arrangement leads to cycles. The vertices of the same agent share an agent vector, which is not visualized in Figure 3.3.

3.3.2 Vertex Functions

In DPOP, a node does receive util messages from it's children and value messages from the parent vertex in the graph. These messages have been implemented as a `DPOPMessage` signal object that either is a util or a value message. The collect function of a vertex does decide from whom the message has been received and stores them in their respective vector. The following util propagation and value propagation phases and the calculations of their respective handler functions are defined by the position of the vertex in the tree.

```
1  Algorithm: Util Message Handler (X_k, UTIL_{X_k}(X_i))
2  store UTIL_{X_k}(X_i);
3  if UTIL messages from all children arrived then
4  |  if Parent(X_i) == null (that means X_i is the root) then
5  |  |  v_i^* ← ChooseOptimal(null) ;
6  |  |  send VALUE(X_i, v_i^*) to all C(X_i) (sends value to all children);
7  |  else
8  |  |  UTIL_{x_i}(P(X_i)) ← ComputeUtils(P(X_i), PP(X_i)) (utilities from the
   |  |  parent and pseudoparents);
9  |  |  sendMessage(P(x_i), UTIL_{x_i}(P(X_i));
10 |  end
11 end
12 return;
```

Algorithm 1: DPOP Util Message Handler [Petcu and Faltings, 2003]

A leaf node always does compute its utilities in the util propagation phase as it does not have any children nodes and receives no util messages. In the case of the meeting scheduling problem, utilities for every timeslot are being calculated. If the vertex is the root of the tree it does only receive util messages. Those include the combined utilities of all nodes along the path to the top. The vertex then chooses the optimal timeslot value for every meeting. If the node is in the middle, it does receive both types of messages. Such a vertex computes the combined utilities from the received utility messages and its own utilities during the util propagation phase (Algorithm 1).

```
1  Algorithm: Value Message Handler VALUE^{X_i}_{P(x_i)}
2  add all X_k ← v_k^* ∈ VALUE^{X_i}_{P(x_i)} to agentView;
3  X_i ← v_i^* = ChooseOptimal(agentView);
4  Send VALUE^{X_l}_{X_i} to all X_l ∈ C(X_i) ;
```

Algorithm 2: DPOP Value Handler [Petcu and Faltings, 2003]

In the value propagation phase, all received values are combined by the vertices and an optimal value for the vertex specific meeting is chosen locally (Algorithm 2). In the implementation, the util propagation phase and the value propagation phase are calculated successively and the resulting utilities and values are stored in a `DPOPMessage`. The message is then sent to all connected vertices.

3.4 Mapping of MGM

The following section describes the mapping of the graph structure of the Maximum-Gain Messaging algorithm and the functions of the nodes described in chapter 2.3.2

to the Signal/Collect Framework in regards to the meeting scheduling problem. The implementation is based on [Chapman et al., 2010].

3.4.1 Graph Structure

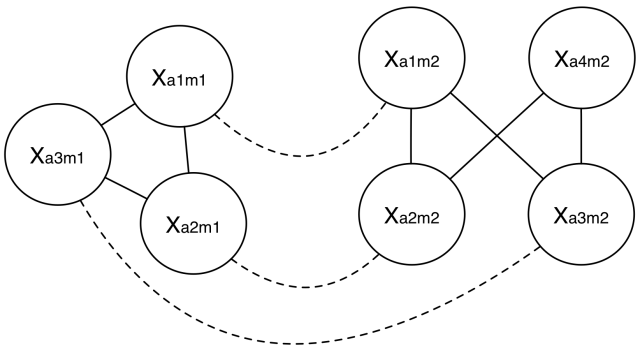


Figure 3.4: An example MGM graph

Mapping the meeting scheduling problem to the structure of the MGM algorithm was quite straightforward. The participant vertices of one meeting are all connected to each other through a Signal / Collect edge comparable to the initial graph setup in the DPOP implementation. The vertices have been implemented with the `DataGraphVertex` as base class and the edges are of the class `StateForwarderEdge`. As one can see in Figure 3.4, the vertices of the same agent are created with a reference to the same agent vector (dashed line) and therefore to each other. This connection is virtual and not implemented as Signal/Collect edge (solid line) in the graph.

3.4.2 Vertex Functions

In MGM there are also two types of messages. A gain message and a value message. Similar to the implementation of DPOP, a `MGMMessage` has been implemented, which can be both and is interpreted by the collect function as the respective type. Initially, a local best-gain is generated by comparing the utility increase from the initialy random meeting timeslot preference to the timeslot with the highest utility and therefore highest gain when changing to this value. This best-gain value is sent to the neighbours. If a vertex has received gain messages, it does compare them to the last local best-gain value and determines if one message contains a higher gain value. If the local gain is still the highest, the vertex converges to the value on which the last best-gain has been calculated. A message with the local state, i.e. the meeting preference is sent to the other vertices. If the vertex does not have the highest gain, it does continue by sending its gain value again.

Algorithm: Maximum-Gain Messaging
currentReward = $u_i(si = currentState, s_{-i})$;
for $j = 1 : J$ **do**
  $stateGain(j) = u_i(s_i = j, s_{-i}) - currentReward$;
end
bestGainState = $argmax_j(StateGain)$;
bestGainValue = stateGain(bestStateGain);
sendBestGainMessage(allNeighbours, bestGainValue);
neighbourGainValues = getNeighbourGainValues(allNeighbours);
if $bestGainValue > max(neighbourGain)$ **then**
  newState = bestGainState;
  sendStateMessage(allNeighbours, newState);
end

Algorithm 3: MGM Pseudocode [Chapman et al., 2010]

3.5 Mapping of MaxSum

The following sections describe the mapping of the graph structure of and the behaviour of it's nodes described in chapter 2.3.3 to the Signal/Collect Framework in regards to the meeting scheduling problem.

3.5.1 Graph Structure

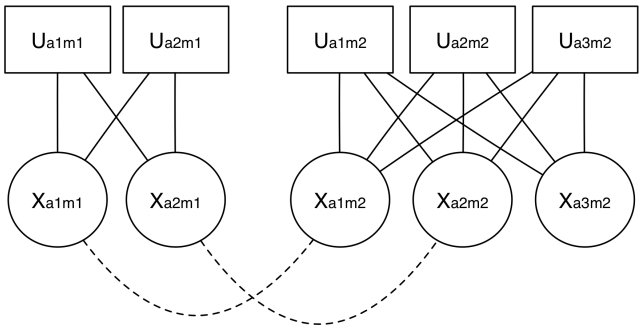


Figure 3.5: An example MaxSum graph

A MaxSum graph is a tranformation of a common DCOP graph to a factor graph, where each variable has an additional function node attached. This function node represents a constraint and is connected to the previous neighbours. The variable node is connected to the function node of it's previous neighbours. Mapping the MaxSum algorithm to Signal/Collect and the meeting schedule problem has initally been a challenge as it was the starting point for the implementations. All the factor graphs in the MaxSum

related papers where described as binary constraints between two nodes. An early consideration was to have multiple factor nodes per variable for each relationship to other meeting participants, where the variable represents the complete schedule of an agent. This approach does not work in regards to the definition of a factor graph and further does not fit the message-passing scheme of MaxSum. It was concluded that a factor graph is derived from a bipartite graph and it should therefore be possible to have k-ary connections from a function node to multiple variable nodes. In the finally implement structure every meeting participation is modelled as Signal/Collect vertex and is connected to a function node through an edge. Every other participant variable of a specific meeting is linked to this function node as well. Through this, all messages are passed to every participant function node and evaluated on the constraints of the agent.

3.5.2 Vertex Functions

In MaxSum, every neighbour of a node receives a customized message. In both vertex types in the graph, the recipient-specific message is created based on the received utilities from all its neighbours except the utilities of the recipient in question (Algorithm 4). In the meeting scheduling problem, these utilities are a vector of all timeslots and the utility of a specific timeslot. One can see the basic process in figure 4. In the variable vertex x_n, the combined utilities are additionaly normalized with a normalization function a_{nm} to prevent the utility values from increasing towards infinity. The message creation function $Q_{n\to m}(x_n)$ for every neighbour of the variable node therefore can be defined like this [Farinelli et al., 2008]:

$$Q_{n\to m}(x_n) = a_{nm} + \sum_{m' \in M(n)\setminus m} R_{m'\to n}(x_n)$$

Whereas n stands for the value vertex and m for the function vertex. For every value $d \in D_x$ the combined utility is calculated as (adjusted for utilities instead of costs analogous to [Zivan and Peled, 2012]):

$$\sum_{f' \in F_x, f' \neq f} utility(f'.d) - a$$

Where F_x is the set of neighbours of a variable vertex. $f'.d$ is the utility value received from a neighbouring function node and a is the normalization value . In the implementation, it was decided to normalize the utilities by adjusting them to a value between 0 and 1 based on the maximal possible utility of a timeslot instead of substracting at a certain scale as only the differences between the utilities matter [Zivan and Peled, 2012]. Otherwise, these functions have been implemented as defined in the literature. The message creation function of a function vertex f_m additionally used the defined constraints of the agent to the utilities and therefore adds the information to find the optimal solution for all neighbours. It then chooses the assignment with the maximal utility for a value

$d \in D_x$ and adds this utility to the timeslot vector. In a formal definition, the function is decribed like this [Farinelli et al., 2008]:

$$R_{m \to n}(x_n) = max_{x_m \setminus n} \left(U_m(x_m) + \sum_{n' \in N(m) \setminus n} Q_{n' \to m}(x_{n'} \right)$$

Whereas, U_m is the local utility function based on the soft and hard constraints of an agent. A variable vertex and a function vertex both send a message with essentially the same data structure implemented as `MaxSumMessage`. The message contains a hash, which stores the specific message for every node that is connected to this vertex. This message contains the timeslot vector with the utilities for every possible meeting time. The receiver node only takes the message that was created for it from every `MaxSumMessage` object and calculates the aforementioned functions. The implementation of the collect function of the vertex has been created in a way that it processes all available signals on each step in synchronous mode and reacts immediately to every signal in asynchronous mode.

```
1  Algorithm: Max-sum (node n)
2  N_n ← all of n's neighboring nodes;
3  while no termination condition is met do
4  |  collect messages from N_n;
5  |  foreach n' ∈ N_n do
6  |  |  if n is a variable-node then
7  |  |  |  produce message m_{n'} using messages from N_n \ {n'};
8  |  |  end
9  |  |  if n is a function-node then
10 |  |  |  produce message m_{n'} using constraint and messages from N_n \ {n'};
11 |  |  end
12 |  |  send m_{n'} to n';
13 |  end
14 end
```

Algorithm 4: Standard MaxSum Pseudocode [Zivan and Peled, 2012]

4

Benchmark & Discussion

In this chapter the three algorithms are benchmarked with different parameters and scenarios. The first series of tests is conducted in a static environment for a basic understanding of the algorithm performance and the second series is held in various dynamic settings to explore the dynamic capabilities of the algorithms. The testing environment was the minion cluster of the departement of Informatics at the University of Zurich. The cluster consists of 16 machines and each machine has 128 GB RAM and two E5-2680 v2 at 2.80GHz processors. Every processor has 10 cores and the interlink between the machines is a 40Gbps Infiniband setup. The cluster has different partition speeds (slow, fast, superfast). All tests were conducted on superfast partitions.

4.1 Results I: Algorithms Performance in Static Environments

4.1.1 Solution Quality over Time

The main focus of the work has been on the performance of the algorithms in terms of solution quality over time. In this section, the three algorithms are going to be compared on their behaviour and the influence of the parameters density, number of agents and run mode (synchronous/asynchronous) in terms of this attribute.

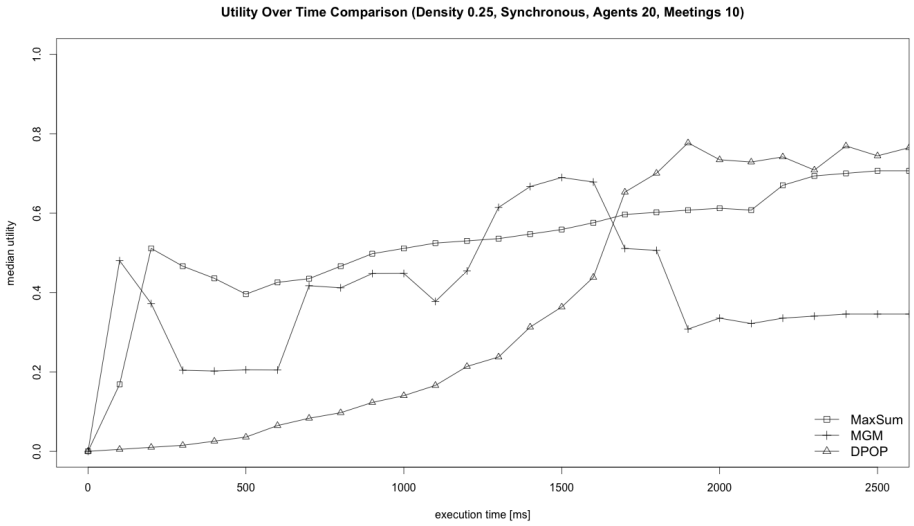


Figure 4.1: Utility over Time Comparison (Density 0.25, Synchronous, Agents 20, Meetings 10)

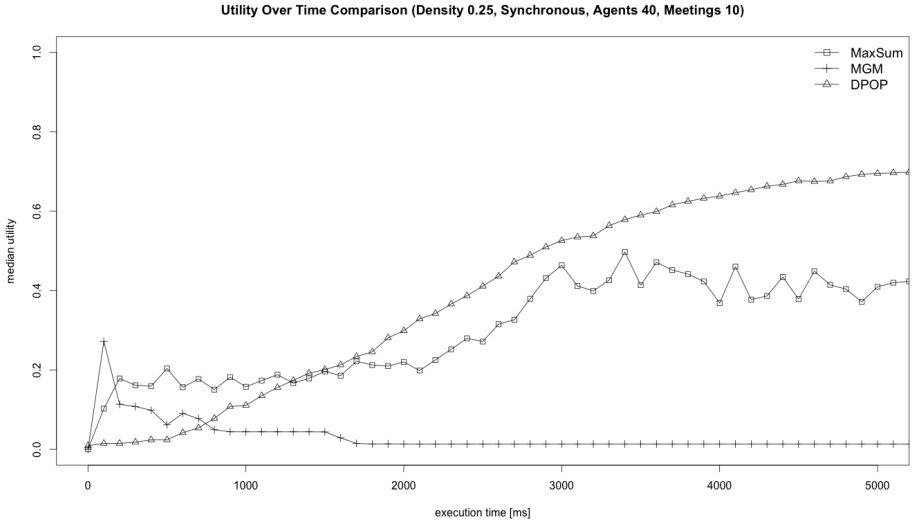


Figure 4.2: Utility over Time Comparison (Density 0.25, Synchronous, Agents 40, Meetings 10)

One can see in the figure 4.1 and 4.2 that the algorithms do show the expected performance at the start of the benchmark. MGM and MaxSum both increase quite fast at the beginning of the run, whereas MGM increases a bit faster. Because the algorithms do not always converge, the mean utility in the end is lower compared to DPOP. The MaxSum Algorithm shows the stronger performance in regards of convergence and qual-

ity of solution than MGM. In figure 4.2 one can see the sometimes erratic behaviour of the MGM algorithm. This could well be a problem with the implementation instead of the algorithm itself. The DPOP algorithm shows a steady increase with a slow start as it was expected. By increasing the number of agents, the time to reach a certain level of quality increases for all three algorithms. A further measurement for quality has been created as a combination of the percentage of accordance on a meeting time and the percentage of overlaps in an agents schedule. The figures can be found in the appendix (Figure A.1, Figure A.2) as they are quite similar to the utility benchmarks.

TABLE HERE

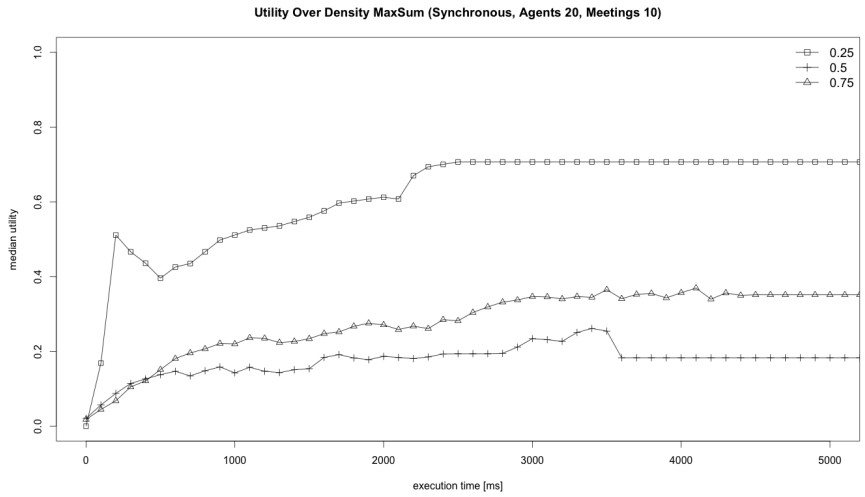


Figure 4.3: Utility over Density MaxSum (Synchronous, Agents 20, Meetings 10)

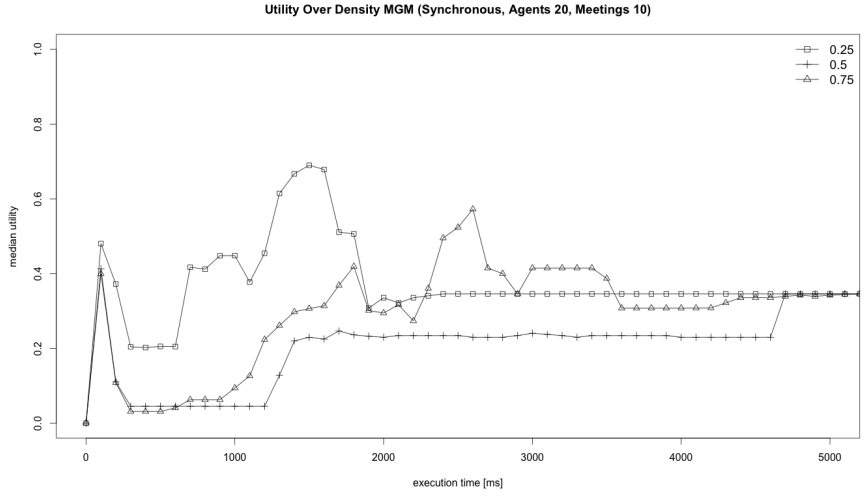


Figure 4.4: Utility over Density MGM (Synchronous, Agents 20, Meetings 10)

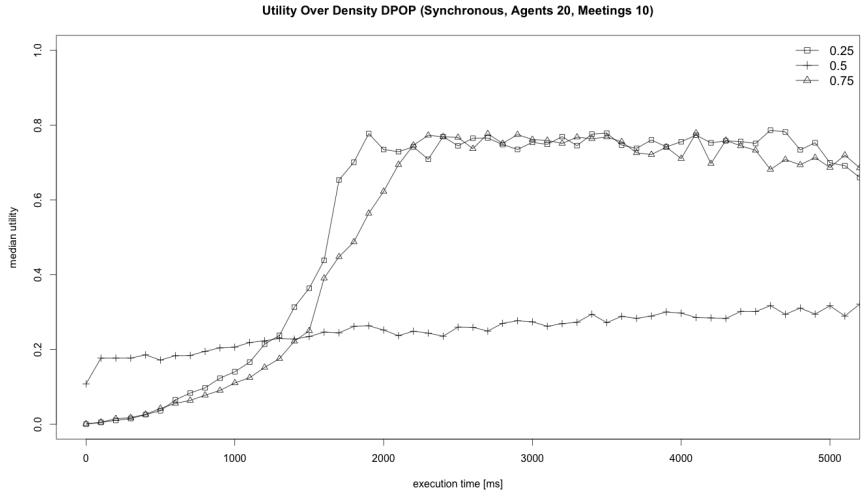


Figure 4.5: Utility over Density DPOP (Synchronous, Agents 20, Meetings 10)

Density has been defined as the number of blocked timeslots in the participations schedules. One would expect this density parameter to have a linear effect and decrease the median utility as higher as the value is set. The figures 4.3, 4.4, 4.5 present a rather different image. The value with the most impact has been 0.5, 0.25 and 0.5 have performed quite similar. This is especially the case with the DPOP algorithm. It is suspected that this is a specific property of the meeting scheduling problem as the density parameter on one hand opens up many timeslots by having a low number of blocked timeslots and on the other hand favors a few not blocked slots on high percentages.

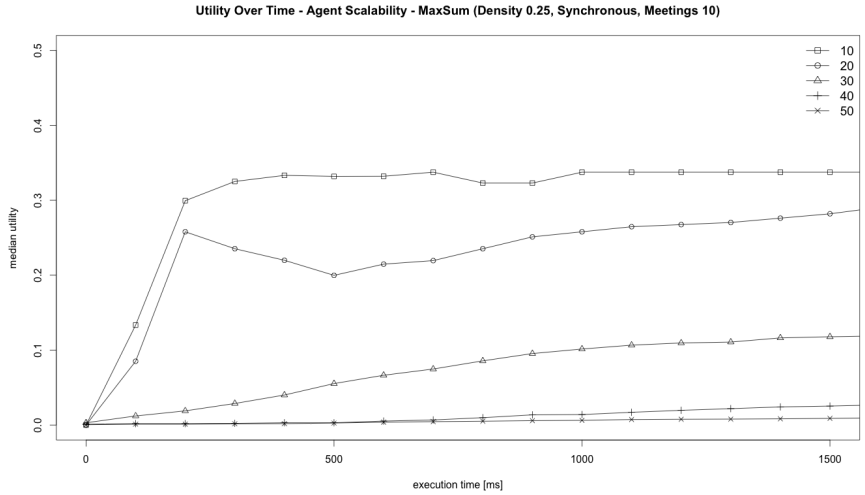


Figure 4.6: Utility over Time - Agent Scalability - MaxSum (Density 0.25, Synchronous, Meetings 10)

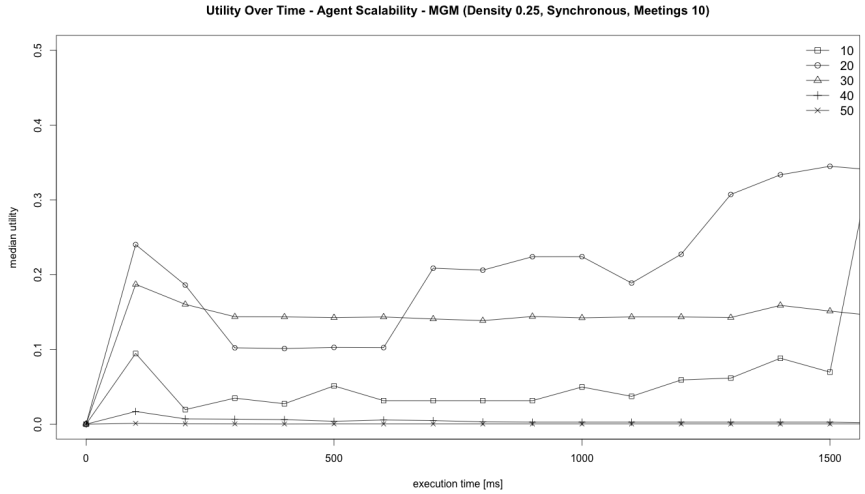


Figure 4.7: Utility over Time - Agent Scalability - MGM (Density 0.25, Synchronous, Meetings 10)

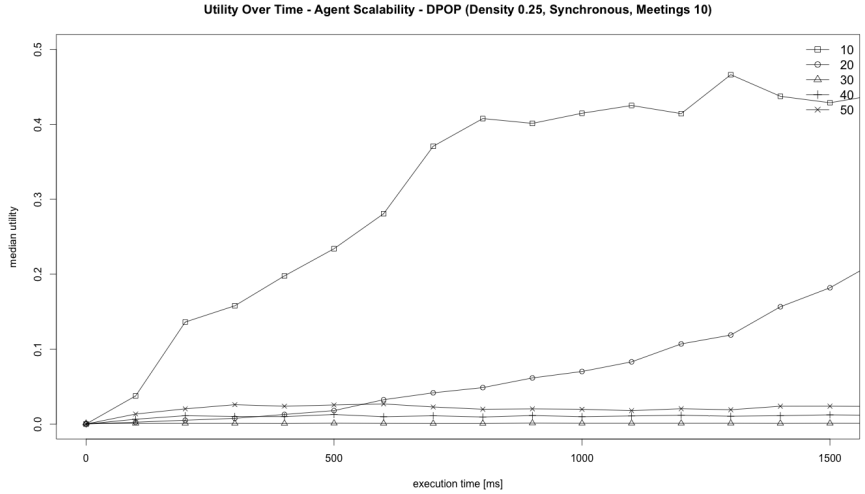


Figure 4.8: Utility over Time - Agent Scalability - DPOP (Density 0.25, Synchronous, Meetings 10)

By testing the scalability of the algorithms on the number of agents with a fixed density and a fixed amount of meetings, it could be discovered that MaxSum scales fairly well and steadily (Figure 4.6). MGM on the other hand again shows inconsistent behaviour, whereas the best performance can been seen with 20 agents (Figure 4.7). For DPOP one can see that the time to reach a certain quality increases drastically with added agents. This property has been expected due to the complexity increase in the messages with an increase of the graph size (Figure 4.8).

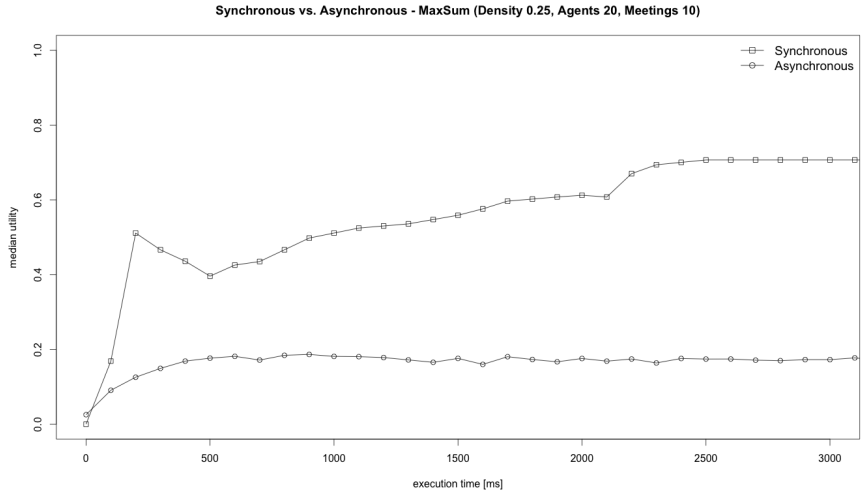


Figure 4.9: Synchronous vs. Asynchronous - MaxSum (Density 0.25, Agents 20, Meetings 10)

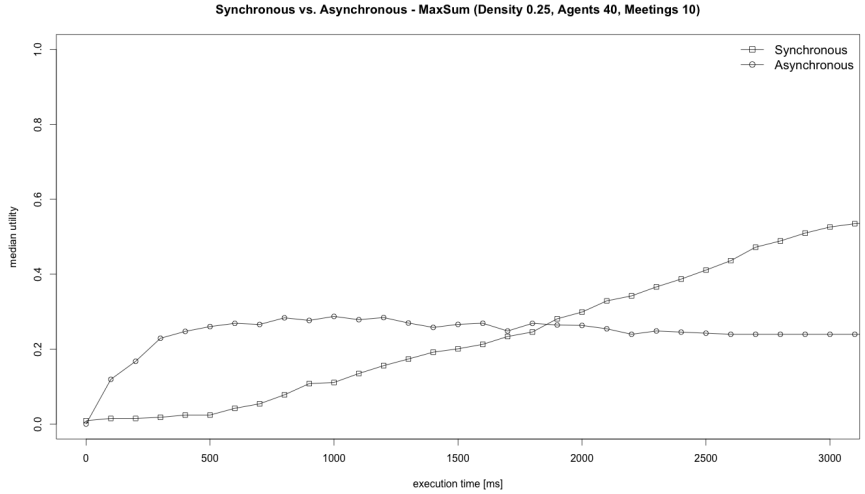


Figure 4.10: Synchronous vs. Asynchronous - MaxSum (Density 0.25, Agents 40, Meetings 10)

An interesting property of MaxSum has been discovered by comparing the synchronous and asynchronous run mode. It seems that with low numbers of agents the performance of the synchronous mode is better at the start of a run, but the asynchronous variation is faster with increased amounts of agents (Figures 4.9, 4.10). The MaxSum algorithm shows some interesting scalability properties in asynchronous mode, which can also be seen in Figure 4.13. MGM seems to converge faster in asynchronous mode on a low number of agents (Figure 4.11). DPOP does not seem to profit from the asynchronous

mode and rather slows down (Figure 4.12).

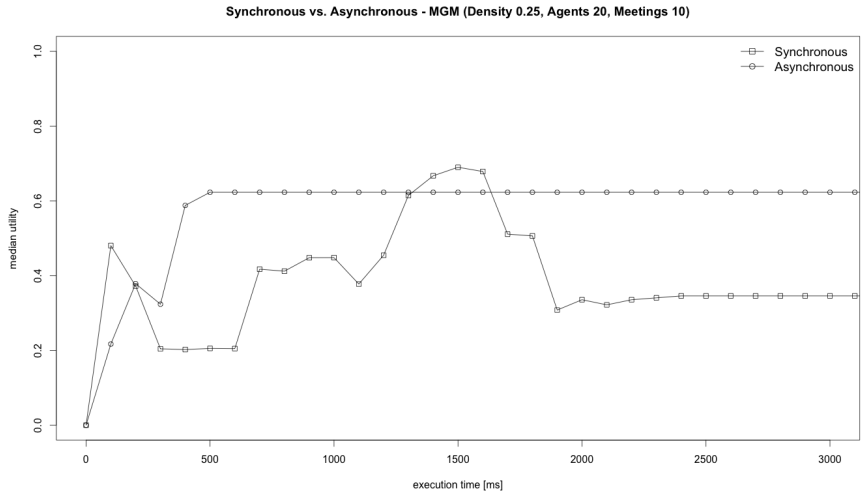


Figure 4.11: Synchronous vs. Asynchronous - MGM (Density 0.25, Agents 20, Meetings 10)

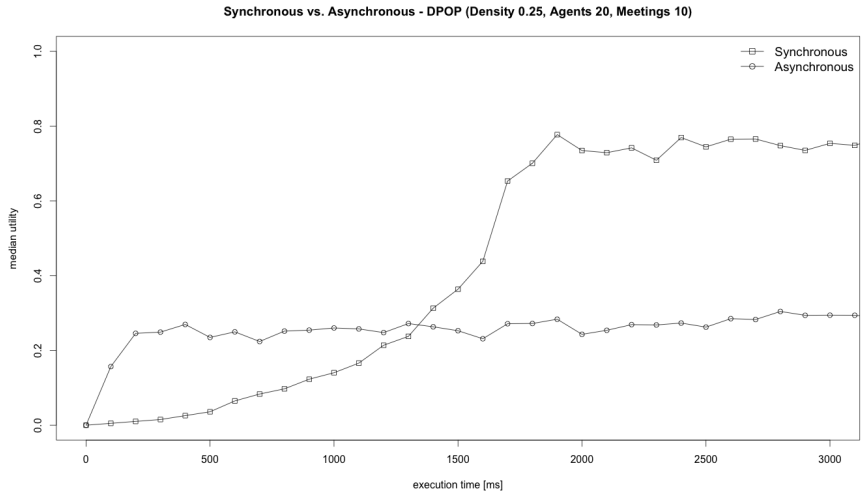


Figure 4.12: Synchronous vs. Asynchronous - DPOP (Density 0.25, Agents 20, Meetings 10)

4.1.2 Time to Convergence

In this section, the time to convergence is going to be analyzed. The focus lays on the scalability properties of the algorithms in different densities and run modes (synchronous/asynchronous) in regards to agents. Meetings have, because of the participa-

tion limitation a limited influence on the performance of the algorithms when meeting numbers are increased.

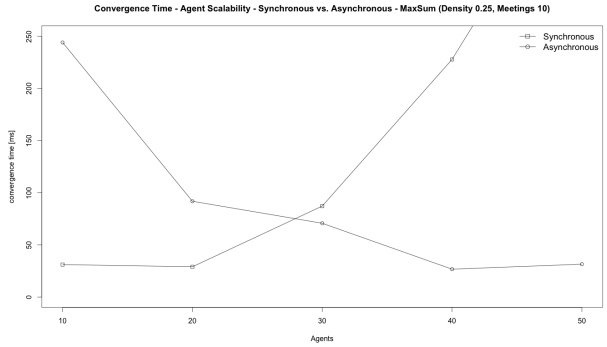


Figure 4.13: Convergence Time - Agent Scalability - MaxSum (Density 0.25, Meetings 10)

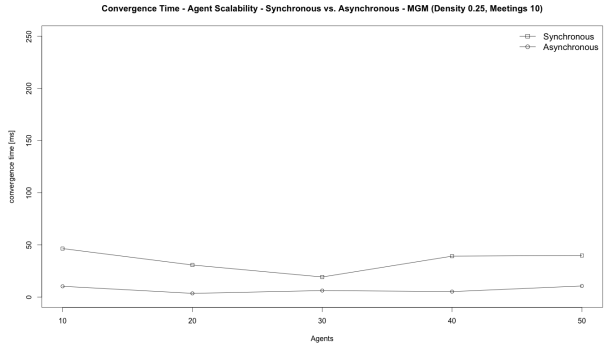


Figure 4.14: Convergence Time - Agent Scalability - MGM (Density 0.25, Meetings 10)

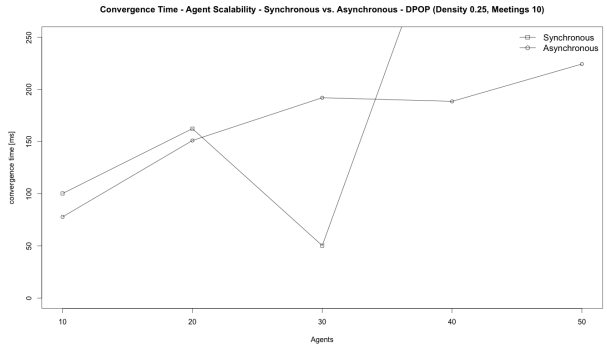


Figure 4.15: Convergence Time - Agent Scalability - DPOP (Density 0.25, Meetings 10)

MaxSum again presents an obscure behaviour by actually converging faster in asyn-

chronous mode with an increased number of agents. When run in synchronous mode, the algorithm shows slower convergence times with added agents (Figure 4.13). The MGM algorithm does seem to profit from the asynchronous mode on increased amounts of vertices in the graph, as it stays very stable on a lower level of converges time than the synchronous variant (Figure 4.14). DPOP, also seems to converge faster in asynchronous mode (Figure 4.15).

TABLE HERE

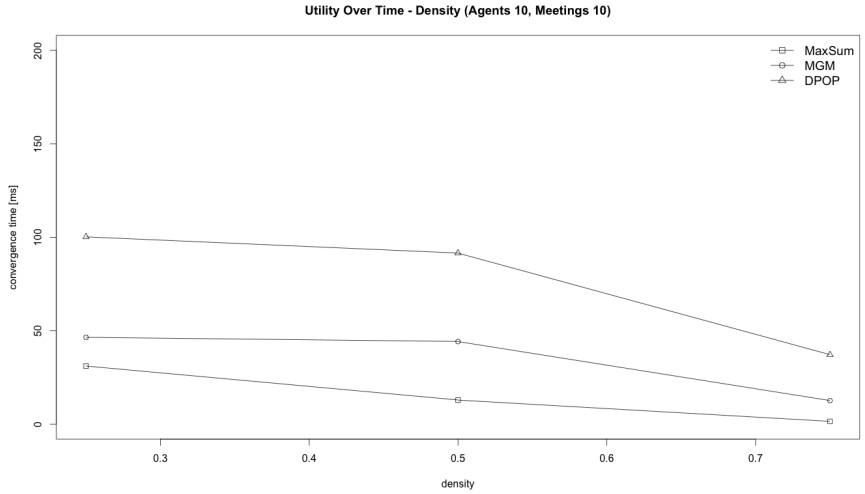


Figure 4.16: Utility over Time - Agents 10, Meetings 10

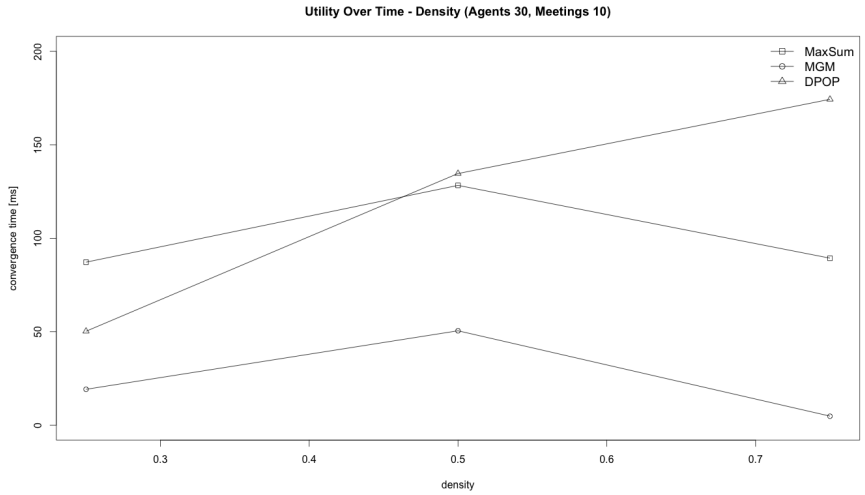


Figure 4.17: Utility over Time - Agents 30, Meetings 10

A run with 10 Agents shows the expected effect of a problem density increase (Figure

4.16). The benchmark with 30 agents again shows the case that MaxSum and MGM run comparably fast on 0.25 and 0.75, but have an increased convergence time on density value 0.5 (Figure 4.16). DPOP does not scale the same way and increases significantly on density 0.75. This observation seems to be related to the local-iterative nature of MaxSum and MGM.

4.2 Results II: Algorithms Performance in Dynamic Environments

In this section, the benchmarks on dynamic abilities of the algorithms will be shown. Some parameters needed to be fixed during the benchmarks, as otherwise there would have been too many results to process. The test case has been 30 agents and 10 meetings.

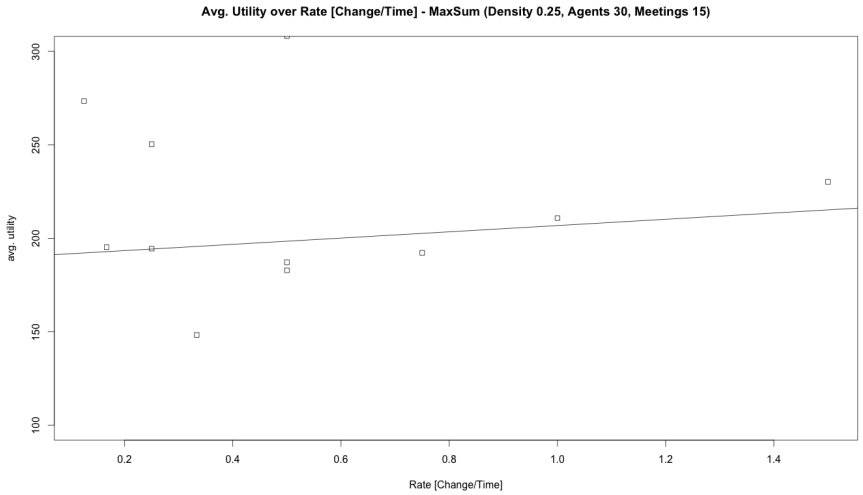
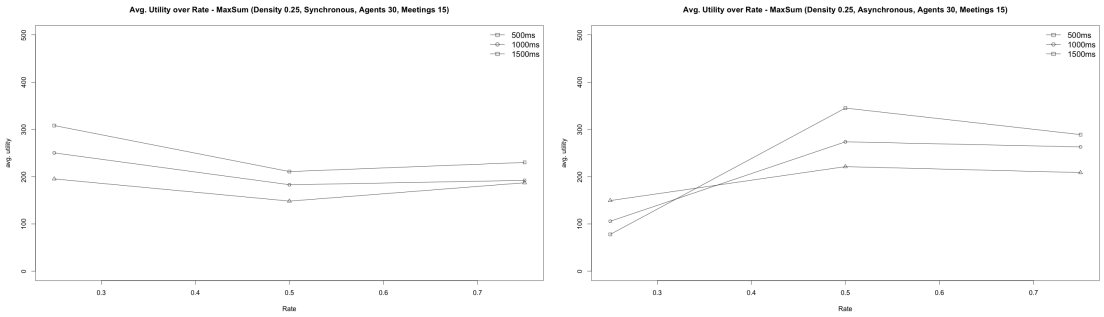


Figure 4.18: Avg. Utility over Rate [Change/Time] - MaxSum

One value for stability has been chosen to be change rate over average utility comparable to [Mailler and Zheng, 2014]. Instead of using the conflicts value, it was decided to use the utility value. The rate is defined as dP/dt, whereas dP is the amount of change to constraints and dt is the difference in time. It was modelled to use different amounts of change. On can see in Figure 4.18 that the utility does increase over the

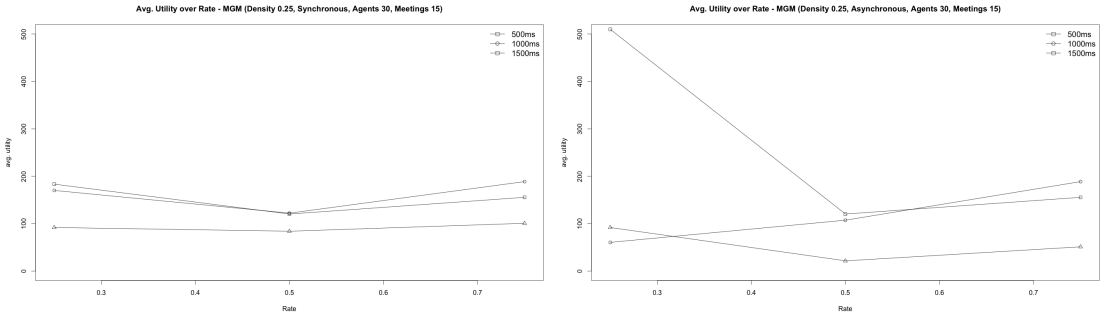
EXTENSION HERE



(a) Avg. Utility over Rate - MaxSum - Synchronous

(b) Avg. Utility over Rate - MaxSum - Asynchronous

Figure 4.19: Comparison of Avg. Utility over Rate - MaxSum - Asynchronous vs. Synchronous



(a) Avg. Utility over Rate - MGM - Synchronous

(b) Avg. Utility over Rate - DPOP - Synchronous

Figure 4.20: Comparison of Avg. Utility over Rate - MGM - Synchronous vs. Asynchronous

A further approach with Rate on the x-axis has been attempted, and it could be shown that MaxSum has the ability to handle various amounts of change well (Figures 4.19a, 4.19b. The synchronous mode does increase the performance significantly on change rate 0.5. MGM also does perform in a stable manner in the tested synchronous mode (Figure 4.20a). DPOP does not seem to be able to handle short intervals of change well as seen on value 500ms, but the algorithm shows stability on 1000ms and 1500ms change rates 4.20b.

TABLE HERE

DYNAMIC VARIABLES HERE

5

Limitations & Future Work

Limitations inherent in this thesis are the number and types of algorithms and approaches, as well as the focus on the specific problem of meeting scheduling. To generalize the results of this thesis on the performance of the algorithms in respect to solution quality over time and in a dynamic constraint environments, one would need to benchmark different problems with other constraint settings and compare additional algorithms on the framework. Future work could include the benchmarking of other problems than meeting scheduling with the given algorithm implementations and structure. Especially, the MaxSum algorithm seems promising for problems that require a quick solution to a problem like network traffic routing or high-frequency sensor networks. The meeting scheduling problem could also be further explored by increasing the amount of maximum meeting participations of an agent. The amount of participations was limited during the course of this thesis to not further expand the number of possible cases for benchmarking, but it could be interesting to see how this affects the overall performance of the algorithms as the complexity to find a converging solution increases with more participants in a meeting.

Certain aspects of dynamic problem changes have not been investigated in the benchmarking chapter of this thesis. The first aspect would be changing of domain spaces during run-time. It would be a possibility to study the effects of increases and decreases at given intervals and percentages. Decreasing domain value sets supposedly render a complete solution impossible by for example in meeting scheduling reducing the timeslots to a minimum. Increasing the domain space as the only dynamic property would be expected to have a low impact as the number of meetings does not increase and therefore no need for more timeslots would emerge. It would therefore be interesting to see how the combination of multiple change types affects the problem solution process as a second aspect. A dynamic environment like this could for example be a possibility for a real-time scheduling system, which continously integrates new information into a overall problem. One could investigate the effects of dynamically adding variables and constraints at the same time and simultanously increasing the domain space. In this case, the stability properties of the algorithms would be of great importance too and could also be tested and further studied.

6

Conclusions

In this thesis, the meeting scheduling has been mapped to a distributed constraint optimization problem. The formal definition has been derived from the literature and a local utility function has been formulated. An general description was given on complete distributive, local-iterative best-response and local-iterative message passing algorithms in the research area of distributed constraint optimization based on the categorizations of [Chapman et al., 2011]. Further, the specific algorithms MaxSum, Maximum-Gain-Message (MGM) and Distributed Pseudotree Optimization Procedure (DPOP) have been described in terms of their graph structure and communication behaviour and advantages as well as disadvantages have been pointed out. The algorithms have been mapped to the programming paradigm of the Signal/Collect framework on top of an implemented framework for benchmarking and dynamic changes based on these descriptions and specified to the meeting scheduling problem. Solutions had to be found to map soft constraints as well as hard constraints into the graph structure and vertices for all three algorithms in a manner that the performance values could be compared. Additionally, a monitoring and storage solution has been proposed that allows for immediate processing of values from the graph and real-time monitoring of the performance.

In the mapping of MaxSum, an approach for the graph structure has been taken that varies from the commonly described factor graphs in the papers. During the mapping process, problems arose when only on factor node was present in the graph because of the inherent message structure defined by the algorithm formulation. Instead of binary connections to a factor node from to variable nodes, it was chosen to allow multiple respectively k-ary connections. This choice was based on the fact that factor graphs are based on bipartite graphs, which allow such connection setups.

The framework has proven to be a good starting point to benchmark dynamically changing problems during runtime and could be further extended for future research. The monitoring platform has proven to be very helpful in the process of implementation as well as during the evaluation. The benchmarking has delivered some interesting data on the performance of the algorithms. The comparison between the three approaches in terms of Time to Solution has shown the abilities of the local-iterative implementations to deliver a certain level of quality quicker than the complete variation, but also reach a lower median utility respectively do not converge every time. Surprisingly, the MGM

algorithm did fairly well in asynchronous mode even if the implementation does not wait for a complete set of neighbour messages. The reason could be the limited amount of participants per meeting and the low amount of delay in the system. The MaxSum algorithm has shown an interesting property of scaling very well and even improving the convergence rate over the amount of agents in asynchronous mode, whereas it did not scale well in synchronous mode. The influence of problem density has shown to be comparable along the local-iterative algorithms. To benchmark the dynamic properties, a fairly new proposal by [Mailler and Zheng, 2014] has been adjusted to utilities instead of conflicts and tested. This evaluation method showed to deliver useful results. Further, another benchmark method has been attempted, which also could help to gain more insight into dynamic constraint optimization problems.

References

[Angulo and Godo, 2007] Angulo, C. and Godo, L. (2007). Distributed meeting scheduling. *Artificial Intelligence Research and ...*, pages 125–136.

[Chapman et al., 2010] Chapman, A. C., Rogers, A., and Jennings, N. R. (2010). Benchmarking hybrid algorithms for distributed constraint optimisation games. *Autonomous Agents and Multi-Agent Systems*, 22(3):385–414.

[Chapman et al., 2011] Chapman, A. C., Rogers, A., Jennings, N. R., and Leslie, D. S. (2011). *A unifying framework for iterative approximate best-response algorithms for distributed constraint optimization problems*, volume 26.

[Chun et al., 2003] Chun, A., Wai, H., and Wong, R. Y. (2003). Optimizing agent-based meeting scheduling through preference estimation. *Engineering Applications of Artificial Intelligence*, 16(7-8):727–743.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). MapReduce : Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107.

[Farinelli et al., 2008] Farinelli, A., Rogers, A., Petcu, A., and Jennings, N. R. (2008). Decentralised Coordination of Low-Power Embedded Devices Using the Max-Sum Algorithm. (Aamas):12–16.

[Farinelli et al., 2012] Farinelli, A., Vinyals, M., Rogers, A., and Jennings, N. R. (2012). Chapter 12 Distributed Constraint Handling and Optimization. pages 1–43.

[Franzin et al., 2002] Franzin, M., Freuder, E., Rossi, F., and Wallace, R. (2002). Multi-agent meeting scheduling with preferences: efficiency, privacy loss, and solution quality. *... on Preference in AI and CP*.

[Hiroaki et al., 1999] Hiroaki, K., Tadokoro, S., Noda, I., Matsubara, H., Takahashi, T., Shinjou, A., and Shimada, S. (1999). RoboCup Rescue : Search and Rescue in Large-scale Disasters as a Domain for Autonomous Agents Research and Robotics Can Make. pages 739–743.

[Leeuwen et al., 2002] Leeuwen, P. V., Hesselink, H., and Rohling, J. (2002). Scheduling Aircraft Using Constraint Satisfaction. 76.

[Maheswaran and Tambe, 2004] Maheswaran, R. and Tambe, M. (2004). Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. *Proceedings of the ....*

[Mailler and Zheng, 2014] Mailler, R. and Zheng, H. (2014). A new analysis method for dynamic distributed constraint satisfaction. *Proceedings of the 2014 international conference ...*, pages 901–908.

[Modi et al., 2005] Modi, P., Shen, W., Tambe, M., and Yokoo, M. (2005). Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180.

[Nguyen and Yao, 2012] Nguyen, T. T. and Yao, X. (2012). Optimization - The Challenges. 16(6):769–786.

[Petcu and Faltings, 2003] Petcu, A. and Faltings, B. (2003). A Scalable Method for Multiagent Constraint Optimization.

[Petcu and Faltings, 2007] Petcu, A. and Faltings, B. (2007). Optimal solution stability in dynamic, distributed constraint optimization. *Proceedings of the 2007 IEEE/WIC/ACM ....*

[Stutz et al., 2010] Stutz, P., Bernstein, A., and Cohen, W. (2010). Signal/collect: graph algorithms for the (semantic) web. *The Semantic Web-ISWC 2010*.

[Zhang et al., 2002] Zhang, W., Xing, Z., and Louis, S. (2002). Distributed Breakout vs . Distributed Stochastic : A Comparative Evaluation on Scan Scheduling. *Proceedings of the AAMAS-02 workshop on Distributed Constraint Reasoning*, pages 192–201.

[Zivan and Peled, 2012] Zivan, R. and Peled, H. (2012). Max/min-sum distributed constraint optimization through value propagation on an alternating dag. *Proceedings of the 11th International Conference on ...*, (June):4–8.

A

Appendix 15

A.1 Results I: Additional Data

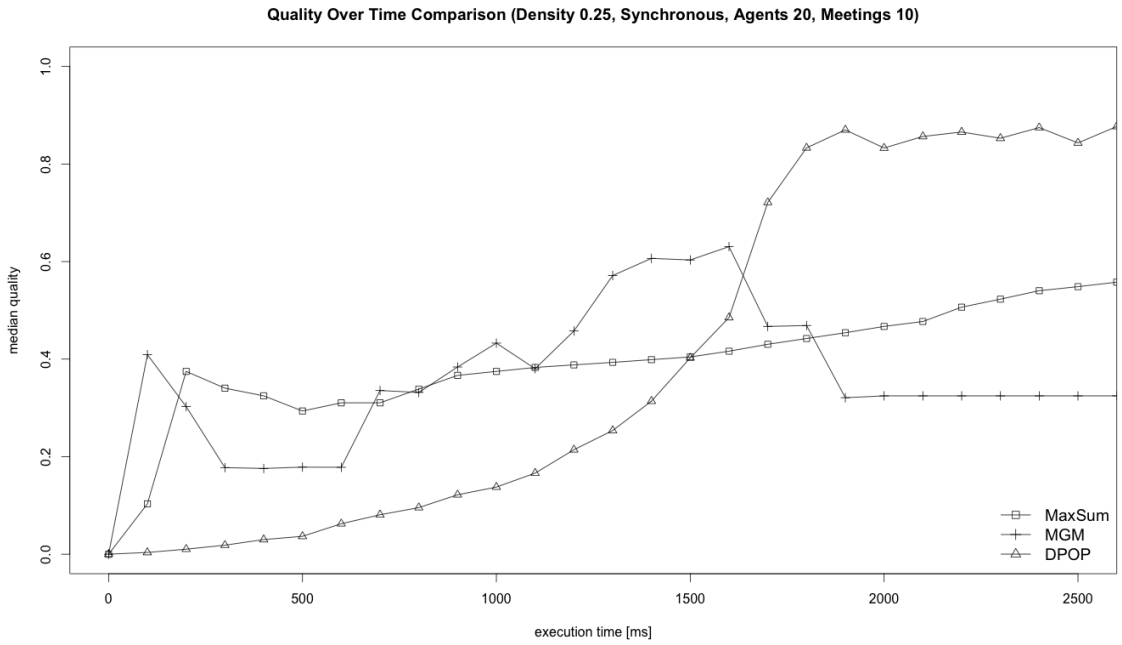


Figure A.1: 20/10, 0.25 density, all three, utility

A.2 Results II: Additional Data

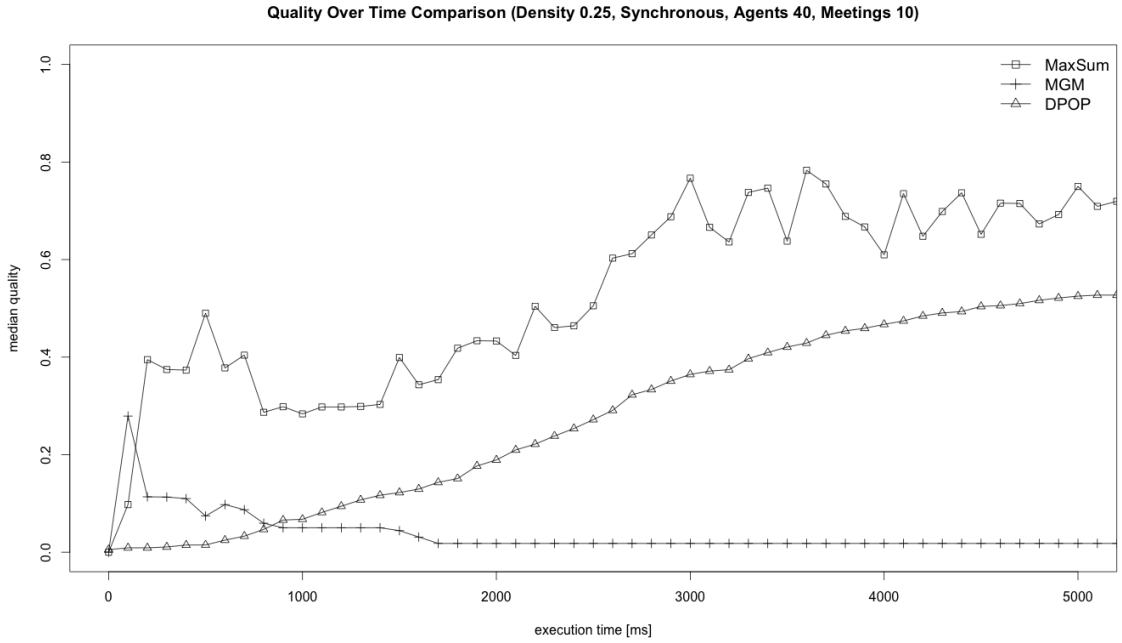


Figure A.2: 20/10, 0.25 density, all three, utility

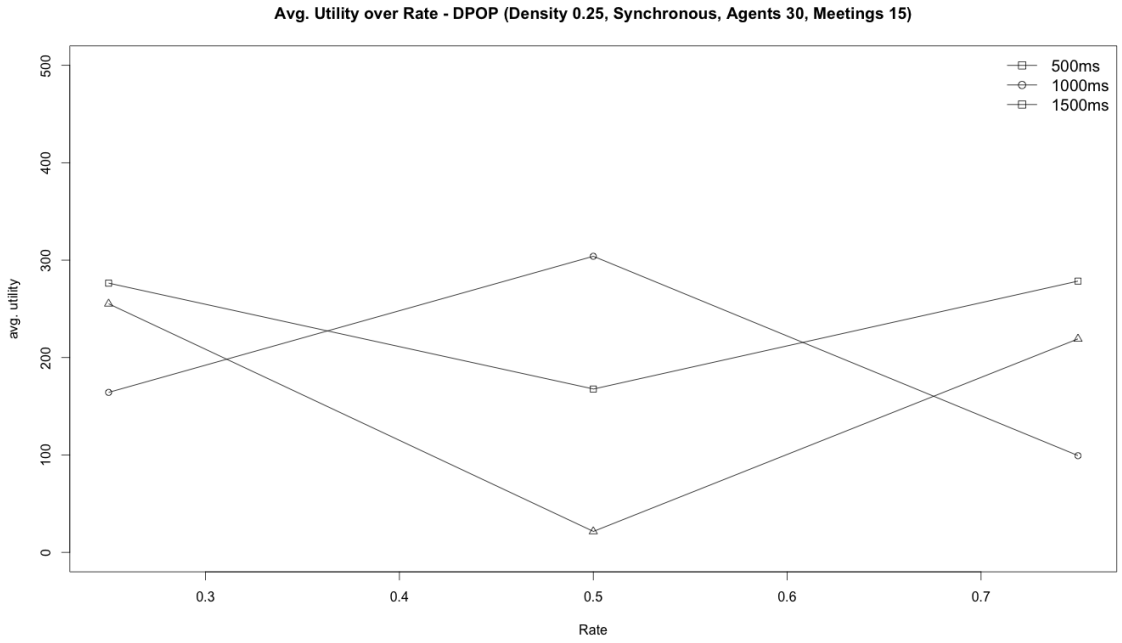


Figure A.3: 20/10, 0.25 density, all three, utility

List of Figures

List of Tables