University of Zurich[UZH]

Scaling Message Passing Algorithms for Distributed Constraint Optimization Problems in Signal/Collect

Dynamic and Distributed Information Systems

Bachelor Thesis October 10, 2013

Genc Mazlami
of Bilten GL, Switzerland

Student-ID: 09-923-061
genc.mazlami@uzh.ch

Advisor: **Mihaela Verman**

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

Acknowledgements

This thesis constitutes the final step towards the completion of my bachelor studies in computer science. The subject of the thesis, as well as the used frameworks, languages and techniques were new for me and were quite of a challenge. Hence, the help of some persons was very essential and should be credited.
Here, I would like to take the chance to thank all persons that contributed to the completion of this thesis in some way.
My biggest gratefulness goes to Mihaela Verman, PhD candidate at the Dynamic and Distributed Information Systems Group (DDIS) at the Department of Computer Science of the University of Zurich. She coached me during the months of the work on this thesis and never hesitated to help with useful tips and comments as well as help in technical or algorithmical questions. Without her effort and continuous support, the completion of the thesis would not have been possible.
A special appreciation goes to Daniel Strebel for his useful tips in programming questions and especially to Philip Stutz for his continuous support and help related to Signal/Collect specific questions and questions with respect to the distribution of the algorithm on a multiple machine cluster. His help has contributed a lot to the successful completion of the thesis.
I would like to thank the Dynamic and Distributed Information Systems Group at UZH, and especially its head, Prof. Dr. Abraham Bernstein, for giving me the opportunity to work in a professional environment on an interesting and challenging subject.
Last but not least, I'd like to thank my friends and family, who helped in the completion of the thesis through their ongoing motivational speeches and their critical eyes while proofreading the chapters of the thesis.

Genc Mazlami, Bilten GL, Switzerland.
October 10, 2013

Abstract

The concept of Distributed Constraint Optimization Problems (DCOPs) is becoming more and more relevant to research in fields such as computer science, engineering, game theory, and others. Many real world problems, such as congestion management in data communication or traffic, and applications on sensor networks, are potential application fields for DCOPs. Hence, there is a need for research on different algorithms and approaches to this class of problems.

This thesis considers the evaluation and distribution of the Max-Sum algorithm. Specifically, the thesis first illustrates a detailed example computation of the algorithm in order to contribute in the understanding of the algorithm. The main contribution of the thesis is the implementation of the Max-Sum algorithm in the novel graph processing framework Signal/Collect. Also, a theoretical complexity analysis of said implementation is performed. Based on the implementation, a second contribution of the thesis follows: The benchmarking of the Max-Sum algorithm and its comparison to the DSA-A, DSA-B and Best-Response algorithms. The benchmark first tries to reproduce the results found in [Farinelli et al., 2008] by analyzing the conflicts over the execution cycles and the cycles until convergence. Then, the thesis contributes new empirical results by evaluating and comparing synchronous and asynchronous Max-Sum with respect to conflicts over time and time to convergence. Also, the analysis of the relation between the execution cycles and the execution time will be part of the novel contribution.

Another main contribution of the thesis is the distributed evaluation of the algorithm on a multiple machine cluster. The benchmarks on multiple machines first compare the solution quality of asynchronous and synchronous Max-Sum on multiple machines. This is followed by an analysis of how the number of machines used in the execution impacts the results for the conflicts over time. The thesis also adresses performance questions raised by the theoretical complexity analysis by analyzing the influence of the average vertex degree on the solution quality.

Zusammenfassung

Das Konzept der Distributed Constraint Optimization Problems (DCOPs) gewinnt immer mehr an Relevanz in Forschungsgebieten wie der Informatik, Ingenieurwissenschaften, Spieltheorie oder anderen. Viele Probleme aus der realen Welt, wie z. B. Überlastkontrolle in Kommunikationstechnologien oder Verkehrssystemen, und Anwendungen in Sensornetzwerken, sind potenzielle Anwendungsgebiete für DCOPs. Daher besteht ein Bedarf and Forschung an verschiedenen Ansätzen und Algorithmen in dieser Problemklasse.
Diese Arbeit beschäftigt sich mit der Evaluation und Verteilung des Max-Sum Algorithmus (auf mehreren Maschinen). Die Arbeit leistet einen Beitrag zum Verständnis des Max-Sum Algorithmus indem sie eine detaillierte Beispielberechnung darlegt. Der wissenschaftliche Hauptbeitrag dieser Arbeit ist jedoch der Entwurf und die Implementierung des Max-Sum Algorithmus in einer modernen Graph-Processing Umgebung namens Signal/Collect. Des Weiteren leitet die Arbeit eine theoretische Komplexitätsanalyse der besagten Implementierung her. Mit Hilfe der Implementation wird der zweite Hauptbeitrag dieser Arbeit realisiert: Die Evaluation des Max-Sum Algorithmus im Vergleich zu den DSA-A, DSA-B und Best-Response Algorithmen. Die Experimente versuchen zum Einen die Resultate aus [Farinelli et al., 2008] zu reproduzieren, indem die Konflikte pro Zyklus und die Anzahl Zyklen bis zur Konvergenz gemessen wird. Zum Anderen wird ein neuer Beitrag geleistet in Form von empirischen Resultaten bezüglich der Konflikte pro Zeiteinheit und bezüglich der Zeit bis zur Konvergenz. Verglichen werden dabei die synchrone und asynchrone Version des Max-Sum Algorithmus. Ausserdem wird die Beziehung zwischen einem Zyklus und der Ausführungszeit analysiert.
Ein weiterer Hauptbeitrag dieser Arbeit ist die verteilte Ausführung und Evaluation des Max-Sum Algorithmus auf mehreren Maschinen. Die verteilte Evaluation vergleicht zuerst die Lösungsqualität vom synchronen und asynchronen Max-Sum Algorithmus auf mehreren Maschinen. Danach wird der Einfluss der Anzahl benutzter Maschinen auf die Lösungsqualität analysiert. Zu guter Letzt werden Fragen welche durch die theoretische Komplexitätsanalyse aufgeworfen werden durch empirische Versuche bearbeitet. Diese Versuche analysieren den Einfluss des mittleren Grades der Knoten im Graph auf die Lösungsqualität.

Table of Contents

1 Introduction

In modern information technology and computing, distributed concepts and systems have become widespread. Among the great variety of tasks and problems that this family of systems tries to solve, there are problems or categories of problems that attract high attention because they are finical and of particular interest in nowadays distributed computing. For example, the tasks of optimally configuring a large scale system in a distributed fashion [Chapman et al., 2011] or a cooperative situation where multiple participating entities aim to achieve a common goal through cooperation [Shoham and Leyton-Brown, 2009] are such problems. This family of problems can be formalized under the term of *distributed constraint optimization problems* (DCOPs), which is a generalization of the *distributed constraint satisfaction problems.*

Distributed constraint optimization problems are finical because the goal in those situations is shared among various agents [Shoham and Leyton-Brown, 2009]. Hence, since a central coordinator is not available and the system designers want to make good use of distributed resources [Chapman et al., 2011], DCOPs pose additional difficulties compared to centralized constraint optimization problems. Also, communication restrictions and / or complicated system topologies make it difficult or costly to collect all the necessary information at a location where a solution can be computed [Chapman et al., 2011]. This adds more complexity to these distributed problems and encourages system designers to take the fully-distributed approach which is provided by some DCOP solutions. The interest in DCOPs is also motivated through the wide range of real-world problems they can support and solve. Examples of such real-world applications for DCOPs are congestion management and scheduling in traffic [van Leeuwen et al., 2002] or applications in sensor networks [Zhang and Xing, 2002].

While there are multiple approaches to solve distributed constraint optimization problems - these approaches will be covered briefly in the following chapter, as well as a formal definition of DCOPs - this thesis is mainly intended to cover one of these approaches, namely the *local iterative message passing algorithms.* More precisely, the Max-Sum algorithm [Farinelli et al., 2008], which is one of the main representatives of the mentioned group, shall be investigated in detail.

1.1 Motivation

Unlike classical approaches in distributed constraint optimization, which mostly try to achieve provable optimal and complete solutions to DCOPs, the family of local iterative message passing algorithms delivers only approximate solutions for problems in the distributed constraint optimization field (cf. chapter 2). While the complete algorithms deliver optimal solutions to the problems, they have essential drawbacks in terms of time, computational complexity and memory overhead (cf. chapter 2), which makes these unsuitable for a wide range of applications.

With the advent of the ubiquitous computing paradigm [Weiser, 1991] and the growing importance of embedded systems, multiagent systems or sensor networks, the requirements posed to distributed algorithms have changed. Since the uprising computing paradigms mentioned in Weiser's work have lead to highly embedded and

low-price computing entities, microprocessors and memory modules are integrated into various devices, sensors and systems, thus forming *the internet of things.* As a consequence, the computing entities (devices, processors etc.) in such systems provide only limited computational power and are also limited in terms of memory size and power consumption. Since most of the aforementioned systems have a highly distributed nature, there is clearly a need for distributed coordination between the participating entities. In other words, these new styles of computing are a primary application field for distributed constraint optimizations. Hence, distributed constraint optimization algorithms that are intended to operate in such an environment with weak hardware resources have to take into account the limits set by the underlying technology.
This fact shows that the complete distributed constraint algorithms are not suitable for the above-mentioned application fields. These new needs and requirements demand a different approach to DCOPs and motivate the research on other algorithms, such as the local iterative message passing algorithms.

1.2 Goals

In the following, the concrete goals of this thesis will be outlined. The goals include the implementation and evaluation of a chosen algorithm of the local iterative message passing category.

Implementation

For local iterative message passing algorithms, the solution quality is an important measure, but since many of the real-world applications have limited decision-times, time is also an important aspect. Therefore, it would be interesting to evaluate the performance of such algorithm in terms of time versus solution quality. To be able to do such evaluations, an iterative message passing algorithm, such as the Max-Sum algorithm, will be implemented. An implementation would benefit from being done in an appropriate large-scale graph processing frame work, such as the Signal/Collect framework (cf. chapter 4).

Evaluation

The implemented algorithm should be evaluated in practice. It makes sense to evaluate the Max-Sum algorithm against other local algorithms while leaving out the complete solutions. Therefore, local-iterative best-response algorithms, such as best-response [Chapman et al., 2011] and distributed stochastic algorithms [Zhang et al., 2005] will be used as candidates. The evaluation should concentrate on appropriate performance indicators and measures. A first evaluation will be done on a single machine. A stretch goal for the evaluation would be to execute and benchmark the implementation distribute on multiple machines and analyze scaling properties.

2

Background and Related Work

In order to implement and evaluate an algorithm that solves distributed constraint optimization problems, an understanding about what a DCOP is has to be established. This chapter aims to clearly define DCOPs and deliver an overview of the current state-of-the-art research in the discipline of DCOPs. Besides that, the focus will be set on a particular group of DCOP algorithms, namely the local iterative messsage passing algorithms.

2.1 Distributed Constraint Optimization Problems

A term often used in the field of distributed systems, multiagent systems or distributed coordination is the term of *distributed constraint satisfaction*. It is important not to confuse *distributed constraint satisfaction* with *distributed constraint optimization* although these two terms sometimes are used interchangeably. Hence, a definition and distinction of both is necessary.

The distributed constraint satisfaction problem (DisCSP) is a concept that models interactions and decisions in a distributed or multiagent setting [Modi, 2003]. In a DisCSP model, there is a group of agents interacting to achieve a global goal. Each agent has control over a subset of the global set of variables for the corresponding problem. [Modi, 2003] modeles the global goal of the problem as a set of constraints

where each of the agents only has knowledge of the constraints which have influence on the agents controlled variables. The solution to the problem - e.g. achieving the global goal defined in this global function - is then found by having the agents choose values for their variables which lead to a global satisfaction of the objective function [Modi, 2003]. [Modi, 2003] notes that the constraints used in such a problem setting have to be strictly propositional (as in propositional logic). This means, the constraint can only have *true* or *false* results. A solution of a DisCSP is considered valid if *all* constraints participating in the global objective function are satisfied by the chosen values [Modi, 2003].
This model is inherently discrete and thus, may be too limited for many real-world problems. In most real-world problems, there is no binary decision whether a solution is *true* or *false* but solutions in real-world settings tend to have gradual degrees of quality [Modi, 2003]. This aspect can not be mapped onto the classical DisCSP model. A possible solution to this is to generalize real-world problems where it is impossible to satisfy all constraints. An optimal solution for such a problem can then be found by minimizing the number of unsatisfied constraints, or stated in a different way, by maximizing the reward for satisfying as many constraints as possible. An often used model in DCOPs follows the idea of a utility function (sometimes also referred to as cost function or valued constraint). A utility function is a function that computes a number which expresses how good the constraints in the problem setting of a DCOP are satisfied. Hence, a utility function is - in contrast to the presented binary and constant nature of constraints - a reward or a penalty that says how good a certain assignment or solution is [Modi, 2003].

Following the idea of a utility function, a DCOP paradigm has been proposed in [Modi, 2003] and is usually used as a standard definition for *distributed constraint* *optimization problems* (DCOPs) in later literature and research.

2.1.1 Definition

The presented definition of DCOPs in this section mainly follows [Chapman et al., 2011] which gives a clear definition and introduction of DCOPs.

A **DCOP** is a tuple

$$< V, A, \delta, C, u_g >$$

where

$$V := \{v_1, v_2, ..., v_k\}$$

is a finite set of variables, all of them having the same importance to the general problem, and

$$A := \{a_1, a_2, ..., a_n\}$$

is a finite set of agents, each independently controlling the value of a subset of the variables in V, and

$$\delta := \{D_1, D_2, ..., D_k\}$$

is a finite set of discrete and finite domains from which the variables in V can take values, and

$$C := \{c_1, c_2, ..., c_l\}$$

is a finite set of constraints.

Each constraint has an associated function $u_{c_k}(s_{c_k})$ where s_{c_k} is the configuration of states of the variables v_{c_k} involved in the constraint c_k. The function $u_{c_k}(s_{c_k})$ computes a penalty for violating or a reward for satisfying the constraint c_k. By combining these functions, the global utility function u_g can be formed as in:

$$u_g(s) = u_{c_1}(s_{c_1}) \oplus ... \oplus u_{c_k}(s_{c_k}) \oplus ... \oplus u_{c_l}(s_{c_l}) \tag{2.1}$$

The operator $\oplus$ in equation (2.1) can be an arbitrary, commutative and associative binary operator. Furthermore, the aggregation operator $\oplus$ has to also ensure that an increase in the number of satisfied constraints results in an increase in the global utility function u_g [Chapman et al., 2011]. Hence, the operator has to be strictly monotonic:

$$\forall a, b, c \in D : a < b \Rightarrow c \oplus a < c \oplus b$$

For the sake of simplicity, in this thesis, the arithmetic addition will be used as an aggregation operator, yielding a global utility function of the form:

$$u_g(s) = \sum_{c_k \in C} u_{c_k}(s) \tag{2.2}$$

The goal of a DCOP is then to maximize the reward for satisfying constraints by letting the agents find a variable configuration s^* such that:

$$s^* \in \underset{s \in D \in \delta}{argmax}\, u_g(s)$$

Expressed in words, the DCOP can be solved by letting the agents $a_i \in A$ jointly maximize the global utility function $u_g(s)$ [Chapman et al., 2011].

2.1.2 Extensions

In the presented definition, all constraints have the same importance. While this concept is sufficient for most research and theoretical work, real-world use cases may demand different levels of importance for the different constraints. This can be mapped into the DCOP definition in equation (2.1) and (2.2) by simply having multiplicative coefficients $c_i \in \mathbb{R}$ used as weights for the corresponding constraints according to their importance for the global problem [Chapman et al., 2011].

Another property of the definition above which might not be optimal for real-world applications of DCOPs is the fact that, in this definition, there are no *hard* constraints, which by all means would have to be fulfilled. If an application would have such hard constraints as a part of the global problem, they could be introduced into the DCOP description by inserting a multiplicative element in the equation (2.2) as follows [Chapman et al., 2011]:

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

HC denotes the set of hard constraints, whereas SC denotes the set of soft constraints.

Please note that these extensions will not be used in the remainder of the thesis, as for the rest of the work, the thesis will use the definition in (2.1) or respectively (2.2) .

2.2 Related Work

This section aims to provide an overview over the state of the research in the area of DCOPs. Different views, approaches to DCOPs and algorithms will be briefly discussed. Since this is only an overview, interested readers who want to take deeper looks into the mentioned subjects may refer to the cited papers.

2.2.1 Game-theoretic View

In other sections, the topic was presented as it is described mainly in computer science literature. On a second thought, it is clear that DCOPs are not only a concept known to computer science or engineering, but are open for a broader range of disciplines and applications.
For example, [Maheswaran et al., 2005] presents a definition which categorizes DCOPs in terms of game-theoretic concepts. This insight is not a novel thing; since the beginning of research in the field of DCOPs, researchers have always reasoned about DCOPs in concepts of game theory. From a game theoretic perspective, the agents participating in a DCOP can be seen as players in a strategic game, trying to maximize their own utility function. [Maheswaran et al., 2005] introduces a way to decompose a DCOP into an equivalent graphical game. The concept of a graphical game was introduced in [Kearns et al., 2001] and constitutes a graph-based representation of a game. A graph G represents a game, where the nodes represent the players. Each player is assumed to have a utility function that depends on the player itself, and on some of the neighbouring players. This relation is represented through an edge connecting two nodes, if and only if their respective utility functions are dependent on the strategy the connected node or player chooses. By using techniques and algorithms exchanging messages locally, it is possible to compute locally optimal solution to DCOPs. [Maheswaran et al., 2005] proves that, when mapped onto game-theory concepts, the optimal solution of a DCOP is a Nash-Equilibrium. This perspective of DCOPs opens a variety of applications and questions for DCOPs. Distributed constraint optimizations are not limited to computer science, but are

available for a broader range of research areas such as economics, social sciences and others. This can be a chance for the research field to profit from high-quality solutions derived in other domains, e.g. solutions of DCOPs adapted from game-theory, economics or social sciences or vice versa.

2.2.2 Approaches and Algorithms

As mentioned in the introductory chapter of this thesis, DCOP algorithms can be classified into groups depending on the technique they are using to find a solution. [Chapman et al., 2011] provides a classification which takes into account the most recent algorithms and developments in the field of DCOPs. It is depicted in Figure 2.1. It is important to note that the categories only cover the fully distributed algorithms and leaves aside centralized approaches to DCOPs. The taxonomy in [Chapman et al., 2011] separates DCOP algorithms in two main groups, the *distributed complete algorithms* algorithms and the *local iterative algorithms*.
The distributed complete algorithms are characterized by the fact that all of its representants provide globally optimal solutions for DCOPs. According to
[Chapman et al., 2011], the algorithms in this category are said to always find the optimal solution. This class is also characterized through the inherent complexity of their algorithms in terms of computation, messages or memory. Most of the algorithms in this class have exponential complexity in either computation or message size. Some examples of such algorithms are the ADOPT algorithm (Asynchronous Distributed Optimization) [Modi, 2003] or the DPOP algorithm (Dynamic Programming Optimization) [Petcu and Faltings, 2005]. The ADOPT and DPOP algorithms both solve DCOPs by creating a constraint graph as a first step. In other words, they represent the dependencies between agents, constraints and variables in form of a mathematical graph where agents, constraints and variables are mapped onto the

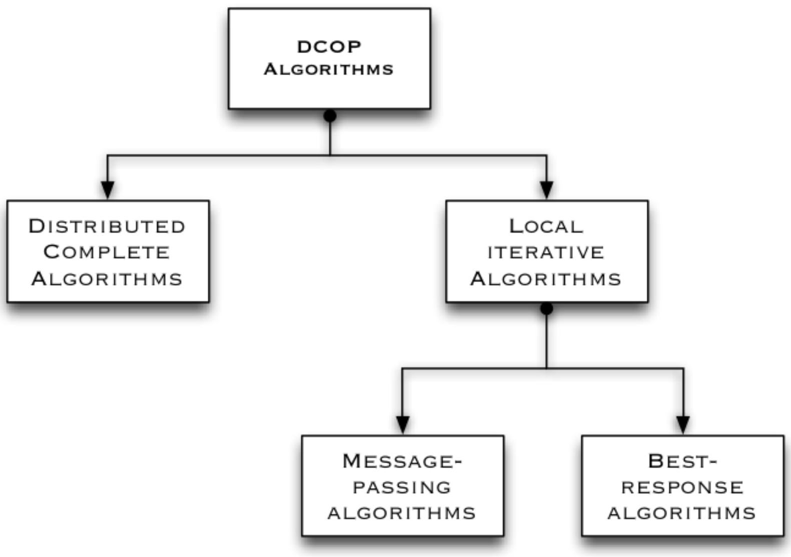

DCOP Algorithms
Distributed Complete Algorithms
Local iterative Algorithms
Message-passing algorithms
Best-response algorithms

Figure 2.1: Categorization of DCOP algorithms according to [Chapman et al., 2011]

nodes of the graphs, and the dependencies between them are modeled as edges between the nodes. Then the procedure continues by restructuring and arranging the constructed graph into a depth first search (DFS) tree, over which the actual computation according to the respective algorithms is executed [Farinelli et al., 2008]. The second group, the local iterative algorithms, are divided in Chapman's work further into two smaller groups, the *local iterative best-response algorithms* and the *local iterative message-passing algorithms*. The algorithms in this group are not capable of finding globally optimal solutions to DCOPs. Nevertheless, the algorithms of this category are deployed in real-world applications and systems since they are applicable in situations where it is reasonable to trade timeliness against optimality of the solution [Chapman et al., 2011].

The *local iterative best-response algorithms* all work by having the participating agents exchange messages containing the state of the agent itself. In addition to the messages containing the states of the agents, agents can observe the strategies

of their neighbors in the problem graph. This makes this category of algorithms applicable for game-theoretic views and applications of DCOPs, such as maximizing social welfare among a group of independent agents (cf. section 2.2.1). Hence, it is not surprising that some of the most known algorithms of this group have their origin in game-theory. Examples of such algorithms are the fictious play algorithm [Brown, 1951] and the adaptive play algorithm [Young, 1993]. A common approach in this category of algorithms is based on randomness or stochastic
[Chapman et al., 2011]. A prime example of such an approach is the *distributed stochastic algorithm (DSA)* [Zhang et al., 2005] , [Zhang and Xing, 2002].
The second subgroup of the local iterative algorithms, the *local iterative message-passing algorithms*, are the most interesting with respect to this thesis, since one of its main representatives, the Max-Sum algorithm [Farinelli et al., 2008], shall be implemented and investigated. As the name implies, in local iterative message-passing algorithms agents exchange messages to find solutions to DCOPs and DisCSPs. The messages usually contain data structures representing the local variable configurations of the sender agents. Receiving agents use these messages to construct new messages to pass on to other agents [Chapman et al., 2011]. Another example of an algorithm of this category (besides the Max-Sum algorithm) is the distributed arc consistency algorithm [Cooper et al., 2007].

2.2.3 Evaluations and Performance

In the introductory sections of this thesis, the importance of performance and scalability for DCOP algorithms was motivated. This thesis aims to evaluate the performance of certain algorithms, hence it is interesting to investigate briefly what kinds of results related research has found when evaluating DCOP solutions. A standard use case for evaluating algorithms in distributed systems, multiagent systems or sensor networks is the *distributed graph coloring problem.*

In distributed graph coloring problems, typically an undirected graph $G =< V, E >$ with $V := \{v_1...v_n\}$ the set of vertices and $E := \{e_1...e_m\}$ the set of edges and a set of globally possible colors $C = \{c_1...c_k\}$ are given as inputs to the problem. It is then a goal of the distributed graph coloring problem to assign to each vertex v_i a color c_j in such a way that the number of conflicts with other vertices is minimized. A conflict occurs when two vertices v_i and v_j connected through an edge are colored with the same color. The described problem can be mapped to a DCOP where the vertices to be colored are agents that have to select their respective color. The DCOP's goal is then to find a global coloring of the vertices / agents such that the overall number of conflicts is minimized.

In [Farinelli et al., 2008] the Max-Sum algorithm is derived from the Sum-Product [Kschischang et al., 2001] and the Max-Product algorithm [Farinelli et al., 2008], and its usage to solve DCOPs is illustrated. The work is concluded by an evaluation of the Max-Sum algorithm against the distributed stochastic algorithm (DSA), a best-response (BR) algorithm and the DPOP algorithm on a distributed graph coloring problem. A first evaluation measured the number of conflicts after 50 execution cycles of the algorithms. The results of [Farinelli et al., 2008] show that the Max-Sum algorithm outperforms DSA, BR and DPOP when evaluated on a set of random colorable graphs. The number of conflicts that Max-Sum produces in this test case is around 1/5 of the number of conflicts for the other candidate algorithms. Despite being superior in tests on random graphs, the Max-Sum algorithm has shown to be performing less well when executed on the ADOPT graph dataset[1]. Especially for a small number of agents, [Farinelli et al., 2008] have found out that Max-Sum performs worse than the other candidates. The authors base the

[1] *http://teamcore.usc.edu/dcop/*

decrease in performance on this dataset on the fact that the graph datasets from the ADOPT repository have a more complex structure than the other datasets, especially in terms of the number and size of the loops occurring in the graphs. This may have lead to cyclic behavior of the messages being passed by the Max-Sum algorithm, resulting in non-convergence. This issue has been addressed and fixed in [Farinelli et al., 2008] through a modification of the algorithm's utility function. Besides the solution quality (number of conflicts), [Farinelli et al., 2008] have evaluated the performance in terms of execution cycles until convergence occurs. In their results, the Max-Sum algorithm delivers the weakest performance with the highest number of steps needed until convergence. It will be interesting to see if the evaluations in this thesis will reproduce these results.
In the next section, a formal definition of the Max-Sum algorithm follows.

3

The Max-Sum Algorithm

The concept on which the Max-Sum algorithm and its relatives are based on is widely used in fields like information theory, artificial intelligence or signal processing. The concept is sometimes referred to as the *generalized distributive law* [Aji and McEliece, 2000]. It uses the fact that the functions involved in the computations can be factorized to be expressed as products of simpler functions in order to simplify the computation by decomposing it [Farinelli et al., 2008].
The Max-Sum algorithm is in fact only a modification of the Sum-Product algorithm [Kschischang et al., 2001] or respectively, the Max-Product algorithm [Farinelli et al., 2008].

3.1 Derivation and Definition

The following derivation mainly follows [Farinelli et al., 2008] and [Waldock et al., 2008] and will begin by introducing the Sum-Product algorithm, from which the Max-Sum algorithm will be derived. All formulas have their origin in [Farinelli et al., 2008] and for the sake of readability, there will be no citations on the formulas or the corresponding description text.
Let F be a function that is dependent on N variables $\mathbf{x} = \{x_1...x_N\}$. The function

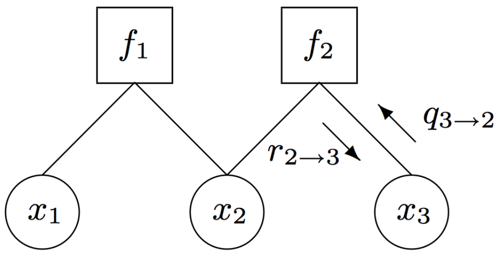


Figure 3.1: Example of a factor graph from [Farinelli et al., 2008]

F is analog to the global utility function $u_g(s)$ mentioned in the previous chapter (c.f. equation 2.2). F is defined as a product of M factors such that:

$$F(\mathbf{x}) = \prod_{m=1}^{M} f_m(\mathbf{x}_m) \tag{3.1}$$

Each of the factors $f_m(\mathbf{x}_m)$ is a function defined over a subset of the variables in $\mathbf{x}$, called $\mathbf{x}_m$. Functions of this form can be visualized through a *factor graph.* A factor graph is a bipartite graph that has two kinds of nodes: *function nodes* and *variable nodes.*

Example: Figure 3.1 shows a factor graph representation of a function $F(\mathbf{x}) = f_1(x_1, x_2)f_2(x_2, x_3)$ with $\mathbf{x} = \{x_1, x_2, x_3\}$. The rectangular vertices always represent a function node, while the circled vertices represent variable nodes [Farinelli et al., 2008].

The Sum-Product algorithm allows to compute the total dependency of the global function $F(\mathbf{x})$ on a single variable $x_n \in \mathbf{x}$. This dependency is called the *marginal function.*

The Sum-Product algorithms computes the marginal functions of all variables simultaneously. It does so by iteratively exchanging messages between the nodes of the factor graph. The algorithm has two types of messages: messages from function nodes to variable nodes and messages from variable nodes to function nodes.

In order to define the format of the messages being passed, let $\mathcal{M}(n)$ be the set of indexes of functions connected to the variable x_n in the factor graph. Let then $\mathcal{N}(m)$ be the set of indexes of variables connected to the function f_m in the factor graph. For example, in figure 3.1, $\mathcal{M}(2) = \{1, 2\}$ and $\mathcal{N}(1) = \{1, 2\}$. Also, let $x_m \backslash n = \{x_{n'} : n' \in \mathcal{N}(m) \backslash n\}$ be the set of variables attached to function m, except for variable x_n. The messages are then defined as follows:

Variable → function:

$$q_{n \to m}(x_n) = \prod_{m' \in \mathcal{M}(n) \backslash m} r_{m' \to n}(x_n) \tag{3.2}$$

The message from variable to function is defined as the product of all messages the current variable has received from its neighboring functions except for the message from the function to which the currently computed message is intended to be sent.

Function → variable:

$$r_{m \to n}(x_n) = \sum_{x_m \backslash n} \left(f_m(\mathbf{x}_m) + \prod_{n' \in (N)(m) \backslash n} q_{n' \to m}(x_{n'}) \right) \tag{3.3}$$

The message from function to variable is defined as a sum over all variables connected to the current function. In each step of the summation, the value of the current function $f_m(\mathbf{x}_m)$ plus the product of the messages received from all neighboring variables except for the variable to which the current message is being sent, are aggregated to a sum.

The leaf nodes initiate the algorithm with the following initial messages:

$$q_{n \to m}(x_n) = 1 \qquad r_{m \to n}(x_n) = f_m(x_n)$$

After the exchange and receipt of all messages from neighbors, a variable node can compute its marginal value by taking the product of all messages it has received:

$$z_n(x_n) = \prod_{m \in \mathcal{M}(n)} r_{m \to n}(x_n)$$

[Farinelli et al., 2008] note that the convergence of sum-product algorithm has been proved only on acyclic graphs [Pearl, 1988]. There is a theoretical risk for the algorithm to oscillate or fail to converge when applied on cyclic graphs. This applies also to the derivations of the Sum-Product algorithm, the Max-Product and Max-Sum algorithms. Despite this fact, it is common to apply the algorithms also on cyclic graphs in practice, since empirically, the algorithms often converge well on all types of graphs; a technique which is known as *loopy belief propagation* [Pearl, 1988]. When translated to DCOPs, the goal in such a setting would be to maximize the global function $F(\mathbf{x})$ by finding an optimal configuration of the variable state of the functions $f_m(\mathbf{x}_m)$. In other words, the goal is to find $arg\ \max_x F(\mathbf{x})$
[Farinelli et al., 2008]. Since the Sum-Product algorithm computes the marginal functions only, it is not suitable for this kind of problem. [Farinelli et al., 2008] propose the use of a derivation of the Sum-Product algorithm: The Max-Product algorithm [Pearl, 1988].
For the Max-Product algorithm, the format of the function-to-variable message in equation 3.3 has to be modified by replacing the summation with a maximization function:

$$r_{m \to n}(x_n) = \max_{x_m \backslash n} \left(f_m(\mathbf{x}_m) + \prod_{n' \in (N)(m) \backslash n} q_{n' \to m}(x_{n'}) \right)$$

Through this modification, $z_n(x_n)$ represents the maximization of the function $F(\mathbf{x})$:

$$z_n(x_n) = \max_{x_m \backslash n} \prod_{m=1}^{M} f_m(\mathbf{x}_m)$$

Thus, it is possible to find $x^* = arg\max_x \prod_{m=1}^{M} f_m(\mathbf{x}_m)$ since each component of the vector of variable configurations x^* is given by:

$$x_n^* = arg\max_{x_n} z_n(x_n)$$

This way, a global maximization task is solved through local message passing [Farinelli et al., 2008].

Until now, it was assumed that the utility function $F(x)$ has the form of equation 3.1. If the utility function has a form as in:

$$\sum_{m=1}^{M} U_m(\mathbf{x}_m)$$

the Max-Product algorithm is not suitable anymore, since it is a summation that is being maximized and not a product as before. As mentioned in equation 2.2, this thesis uses the summation as an aggregation operation. This makes changes to the algorithm necessary and leads to the Max-Sum algorithm. The Max-Sum algorithm is the result of taking the Max-Product algorithm to the logarithm space. This is done through the usage of the following identities [Farinelli et al., 2008]:

$$R_{m\to n} = log\, r_{m\to n}$$

$$Q_{n\to m} = log\, q_{n\to m}$$

$$Z_n(x_n) = log\, z_n(x_n)$$

$$U_m(\mathbf{x}_m) = log\, f_m(\mathbf{x}_m)$$

[Farinelli et al., 2008] notice that, since the algorithms presented here are applied also on cyclic graphs, it is possible for messages to continue propagating in loops and thus growing indefinitely. Thus, [Farinelli et al., 2008] to introduce a *normalization* for the message formats. Hence, the messages for the Max-Sum algorithm are defined as follows:

Variable → function:

$$Q_{n\to m}(x_n) = \alpha_{nm} + \sum_{m' \in \mathcal{M}(n)\setminus m} R_{m'\to n}(x_n) \tag{3.4}$$

where α_{nm} is a normalization scalar chosen such that:

$$\sum_{x_n} Q_{n\to m}(x_n) = 0$$

Function → variable:

$$R_{m\to n}(x_n) = \max_{x_m \setminus n}\left(U_m(\mathbf{x}_m) + \sum_{n' \in (N)(m)\setminus n} Q_{n'\to m}(x_{n'}) \right) \tag{3.5}$$

The marginal functions represent solutions to the maximization problem:

$$Z_n(x_n) = \max_{x_m \setminus n} \sum_{m=1}^{M} U_m(\mathbf{x}_m)$$

and can be computed through:

$$Z_n(x_n) = \sum_{m \in \mathcal{M}(n)} R_{m\to n}(x_n)$$

Thus, by finding $arg\max_{x_n} Z_n(x_n)$, each agent is able to determine which state it should adopt such that the global utility function is maximized [Farinelli et al., 2008].

3.2 Example Computation

In order to illustrate the Max-Sum algorithm on a problem, in this section, an example execution of the Max-Sum algorithm on a three-agent two-color graph coloring problem will be illustrated. The example is according to [Farinelli et al., 2008], but the level of detail is much greater because this thesis aims to contribute in the understanding of the Max-Sum algorithm in practice. Hence, some of the computations are shown in an extended presentation in contrast to [Farinelli et al., 2008].
Let's assume we have three agents connected to each other as illustrated in figure 3.2.

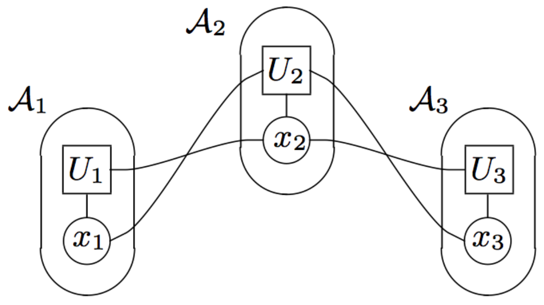


Figure 3.2: Formation of the agents for the example [Farinelli et al., 2008]

In figure 3.2, $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ represent the agents involved in the problem, U_1, U_2, U_3 represent the agent's utility functions, and x_1, x_2, x_3 represent the variables on which the connected utility functions depend.
As explained in section 2.2.3, in distributed graph coloring problems it is our goal to assign to each agent a color in such a way that the global number of conflicts (e.g. two neighbors having the same color) is minimized. The utility function in

this setting is defined as [Farinelli et al., 2008]:

$$U_m(x_m) = \gamma_m(x_m) - \sum_{i \in \mathcal{N}(m) \backslash m} x_m \oplus x_i \tag{3.6}$$

where:

$$x_i \oplus x_j = \begin{cases} 1, & \text{if } x_i = x_j. \\ 0, & \text{otherwise.} \end{cases}$$

and $\gamma_m(x_m) << 1$ is the agent's preference for a certain color when there is no conflict. In short, $U_m(x_m)$ equals the agents preference depending on the value of x_m minus the sum of the conflicts with the neighbors, where the neighbors color takes a value coming from the maximization variables in 3.5. $\gamma_m(x_m)$ is a vector with the length equal to the number of possible colors in the graph coloring problem. The first element of the vector indicates the preference for the first color, the second element indicates the preference for the second color and so on.
The initial preferences for a color of the three agents are as follows:

$$\mathcal{A}_1 : \gamma_1(x_1) = [0.1, -0.1]$$

$$\mathcal{A}_2 : \gamma_2(x_2) = [-0.1, 0.1]$$

$$\mathcal{A}_3 : \gamma_3(x_3) = [-0.1, 0.1]$$

We compute the messages from variables to functions, and from functions to variables, and the marginal function for each of the three agents, step-by-step.
Agent $\mathcal{A}_1$:

The initial messages from variables to functions are all set to zeros:

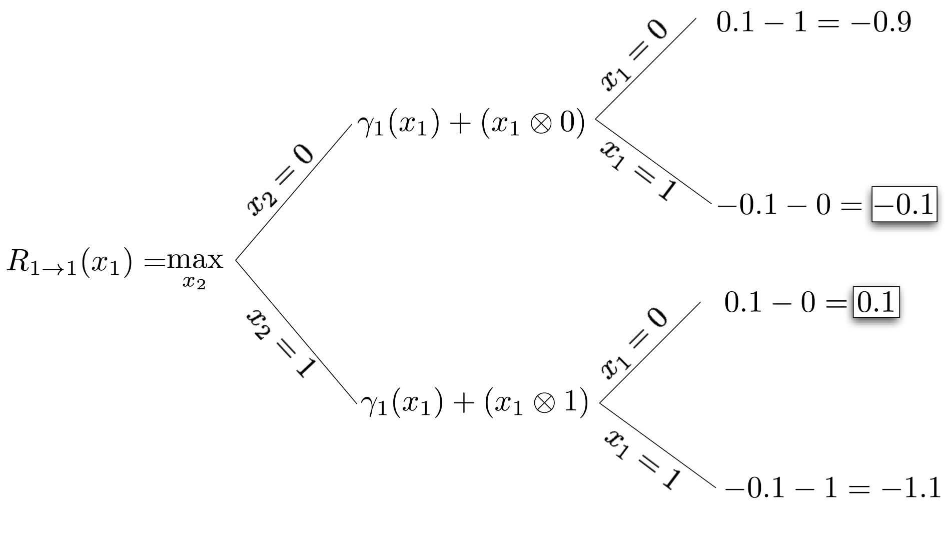
$$Q_{1\to 1}(x_1) = [0,0]$$

$$Q_{2\to 1}(x_1) = [0,0]$$

The messages from functions to variables can then be computed through equation 3.5:

$$\begin{aligned} R_{1\to 1}(x_1) &= \max_{x_2} \left(U_1(x_1,x_2) + Q_{2\to 1}(x_1) \right) \\ &= \max_{x_2} \left(\gamma_1(x_1) - (x_1 \otimes x_2) + Q_{2\to 1}(x_1) \right) \end{aligned}$$

The easiest way to see how the values for the elements of the message are computed, is by inspecting the maximization tree being followed:


$R_{1\to 1}(x_1) = \max_{x_2}$
$x_2 = 0$
$x_2 = 1$
$\gamma_1(x_1) + (x_1 \otimes 0)$
$\gamma_1(x_1) + (x_1 \otimes 1)$
$x_1 = 0$
$x_1 = 1$
$x_1 = 0$
$x_1 = 1$
$0.1 - 1 = -0.9$
$-0.1 - 0 = \boxed{-0.1}$
$0.1 - 0 = \boxed{0.1}$
$-0.1 - 1 = -1.1$

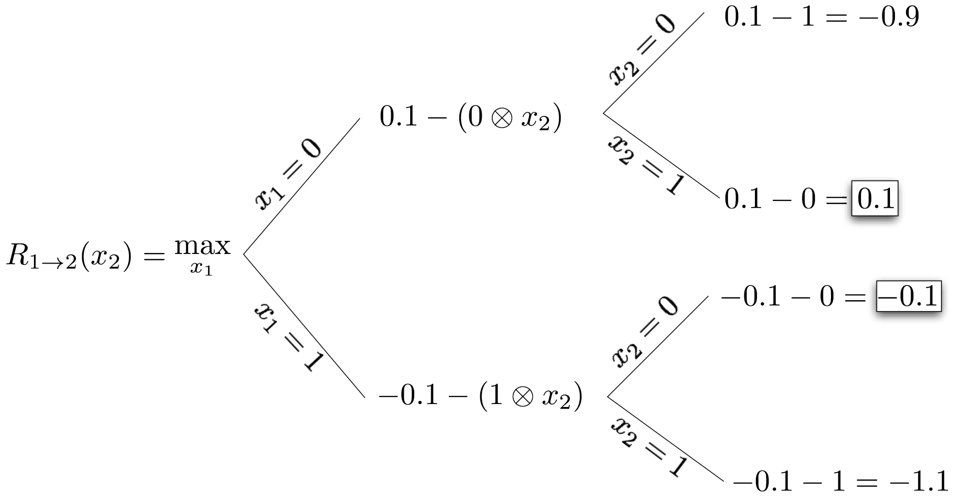
The vector $R_{1\to 1}(x_1)$ represents the resulting message containing the preference for each color configuration for variable x_1, and hence the result is composed of the max value of $R_{1\to 1}(x_1)$ for $x_1 = 0$ and for $x_1 = 1$. The corresponding values are shown in boxes at the end of the decision tree above and hence the result for $R_{1\to1}$ is:

$$R_{1\to1}(x_1) = [0.1, -0.1]$$

The computation of $R_{1\to2}(x_2)$ follows the same steps:

$$\begin{aligned} R_{1\to2}(x_2) &= \max_{x1}\left(U_1(x_1, x_2) + Q_{1\to1}(x_1)\right) \\ &= \max_{x1}\left(\gamma_1(x_1) - (x_1 \otimes x_2) + Q_{1\to1}(x_1)\right) \end{aligned}$$

Also, for $R_{1\to2}(x_2)$ a decision tree helps in the computation:



Since $R_{1\to2}(x_2)$ is dependent on x_2 the result can be computed by finding the maximum values for $x_2 = 0$ and $x_2 = 1$ in the decision tree. Hence the result is:

$$R_{1\to2}(x_2) = [-0.1, 0.1]$$

The marginal function Z_n for the first agent is defined as:

$$Z_1(x_1) = R_{1\to1}(x_1) + R_{2\to1}(x_1)$$

And since the message $R_{2\to1}(x_1)$ has not been computed yet, it is assumed to be equal to $[0, 0]$. Hence the marginal function for agent $\mathcal{A}_1$ can be computed according to equation 3.1:

$$Z_1(x_1) = R_{1\to1}(x_1) + R_{2\to1}(x_1) = [\mathbf{0.1}, -0.1]$$

The vector resulting is $[\mathbf{0.1}, -0.1]$ and the color of the variable x_1 owned by agent $\mathcal{A}_1$ is set to 0, since the element at index 0 in the resulting vector has the greatest value.

Agent $\mathcal{A}_2$:

From variable to function:
Again, the messages $R_{2\to2}$ and $R_{3\to2}$ have not yet been computed but they are involved in the computation of the messages from variable to function for the second agent. Hence, $R_{2\to2}$ and $R_{3\to2}$ are also assumed to be equal to $[0, 0]$. So the messages result in:

$$Q_{2\to3}(x_2) = R_{1\to2}(x_2) + R_{2\to2}(x_2) = [-0.1, 0.1]$$

$$Q_{2\to1}(x_2) = R_{2\to2}(x_2) + R_{3\to2}(x_2) = [0, 0]$$

$$Q_{2\to2}(x_2) = R_{1\to2}(x_2) + R_{3\to2}(x_2) = [-0.1, 0.1]$$

From function to variable:

In the computation of $R_{2\to3}$, $R_{2\to1}$ and $R_{2\to2}$ a decision tree as for $R_{1\to2}$ and $R_{1\to1}$ is helpful for the understanding. The computation for $R_{2\to3}$ would be:

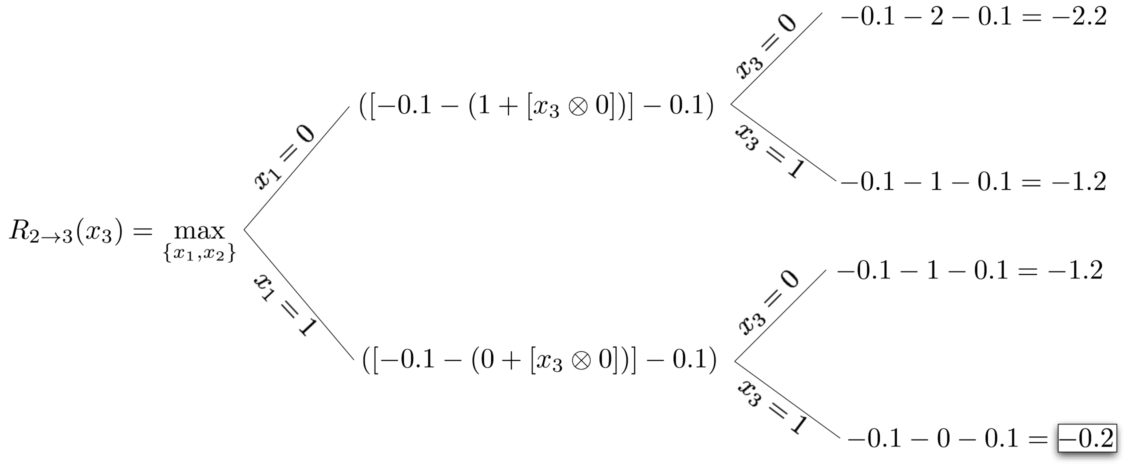
$$R_{2\to3}(x_3) = \max_{\{x_1,x_2\}} \left([\gamma_2(x_2) - \sum_{i\in\{1,3\}} x_i \otimes x_2] + [Q_{1\to2}(x_1) + Q_{2\to2}(x_2)] \right)$$

Since for $R_{2\to3}$ the *max* is taken over both x_1 and x_2, the computation follows two trees.

For $x_2 = 0$:

$$R_{2\to3}(x_3) = \max_{\{x_1,x_2\}} ([-0.1 - ([x_1 \otimes 0] + [x_3 \otimes 0])] + [Q_{1\to2}(x_1) - 0.1])$$

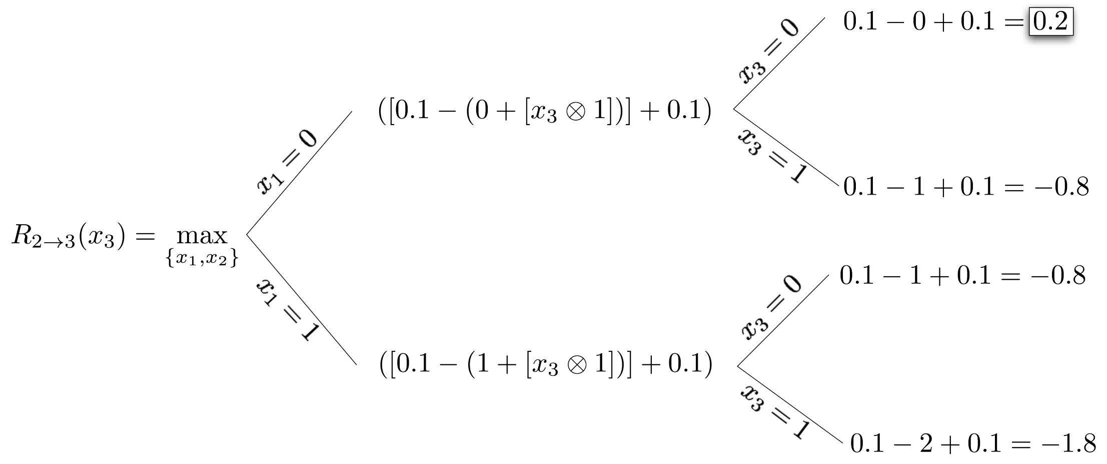
And hence, the decision tree when $x_2 = 0$ is:


$R_{2\to3}(x_3) = \max_{\{x_1,x_2\}}$
$x_1 = 0$
$x_1 = 1$
$([-0.1 - (1 + [x_3 \otimes 0])] - 0.1)$
$([-0.1 - (0 + [x_3 \otimes 0])] - 0.1)$
$x_3 = 0$
$x_3 = 1$
$x_3 = 0$
$x_3 = 1$
$-0.1 - 2 - 0.1 = -2.2$
$-0.1 - 1 - 0.1 = -1.2$
$-0.1 - 1 - 0.1 = -1.2$
$-0.1 - 0 - 0.1 = \boxed{-0.2}$

For $x_2 = 1$:

$$R_{2\to3}(x_3) = \max_{\{x_1,x_2\}} ([0.1 - ([x_1 \otimes 1] + [x_3 \otimes 1])] + [Q_{1\to2}(x_1) + 0.1])$$

and the decision tree when $x_2 = 1$ is:


$R_{2\to3}(x_3) = \max_{\{x_1,x_2\}}$
$x_1 = 0$
$x_1 = 1$
$([0.1 - (0 + [x_3 \otimes 1])] + 0.1)$
$([0.1 - (1 + [x_3 \otimes 1])] + 0.1)$
$x_3 = 0$
$x_3 = 1$
$x_3 = 0$
$x_3 = 1$
$0.1 - 0 + 0.1 = \boxed{0.2}$
$0.1 - 1 + 0.1 = -0.8$
$0.1 - 1 + 0.1 = -0.8$
$0.1 - 2 + 0.1 = -1.8$

Since $R_{2\to3}(x_3)$ represents a preference vector for the configurations of x_3, the result of $R_{2\to3}(x_3)$ is found by taking the maximum value of the decision tree for $x_3 = 0$ and the maximum value for $x_3 = 1$. The respective values are surrounded by boxes in the two decision trees above. Hence:

$$R_{2\to3}(x_3) = [0.2, -0.2]$$

Since it would be an unnecessary repetition that wouldn't contribute in the understanding of the algorithm, the decision trees for the messages $R_{2\to1}(x_1)$ and $R_{2\to2}(x_2)$ are left out and just the results are shown:

$$R_{2\to1}(x_1) = [0.2, -0.2]$$

$$R_{2\to2}(x_2) = [-0.1, 0.1]$$

The marginal function results in:

$$Z_2(x_2) = R_{1\to2}(x_1)+R_{2\to2}(x_2)+R_{3\to2}(x_3) = [-0.1, 0.1]+[-0.1, 0.1]+[0, 0] = [-0.2, \mathbf{0.2}]$$

Hence the color of the variable x_2 in control of agent $\mathcal{A}_2$ is 1.

Agent $\mathcal{A}_3$:

From variable to function:

$$Q_{3\to2}(x_2) = \sum_{m'\in\mathcal{M}(3)\setminus 2} R_{m'\to3}(x_3) = R_{3\to3}(x_3)$$

and since $R_{3\to3}(x_3) = [0, 0]$ (because it has not yet been computed and is assumed to be all zeros):

$$Q_{3\to2}(x_2) = [0, 0]$$

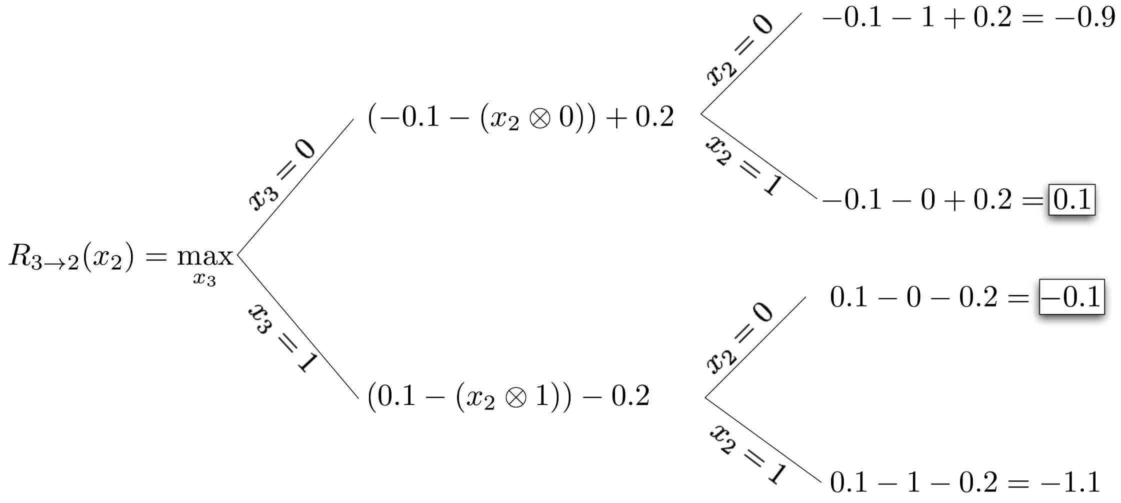
and the second message from variable to function for agent $\mathcal{A}_3$ equals:

$$Q_{3\to3}(x_3) = \sum_{m'\in\mathcal{M}(3)\setminus 3} R_{m'\to3}(x_3) = R_{2\to3}(x_3) = [0.2, -0.2]$$

From function to variable:

$$R_{3\to2}(x_2) = \max_{x_3}(\gamma_3(x_3) - (x_2 \otimes x_3) + Q_{3\to3}(x_3))$$

again, the computation follows a decision tree:


$R_{3\to 2}(x_2) = \max_{x_3}$
$x_3 = 0$
$x_3 = 1$
$(-0.1 - (x_2 \otimes 0)) + 0.2$
$(0.1 - (x_2 \otimes 1)) - 0.2$
$x_2 = 0$
$x_2 = 1$
$-0.1 - 1 + 0.2 = -0.9$
$-0.1 - 0 + 0.2 = \boxed{0.1}$
$x_2 = 0$
$x_2 = 1$
$0.1 - 0 - 0.2 = \boxed{-0.1}$
$0.1 - 1 - 0.2 = -1.1$

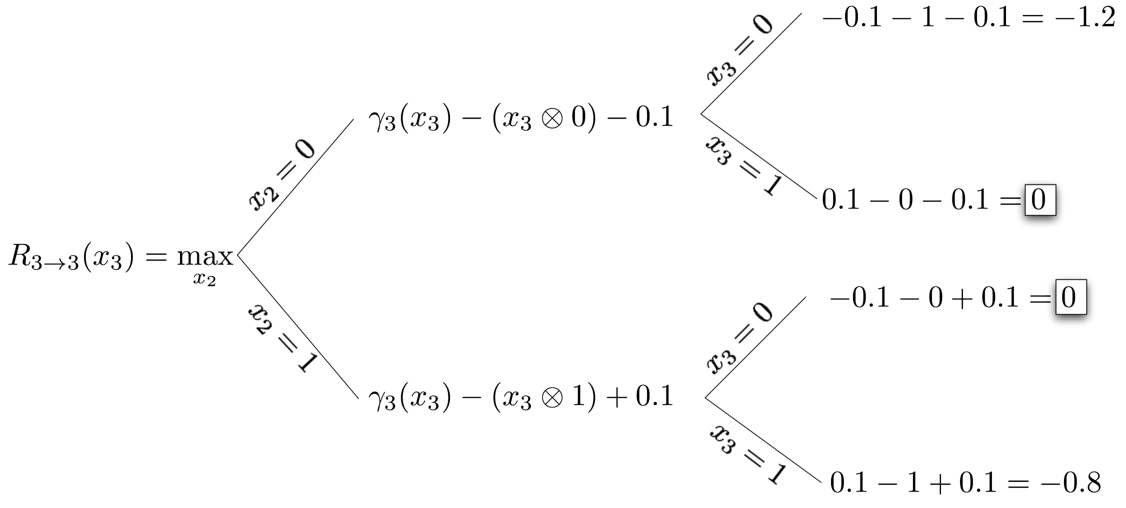
The values in the boxes are the maximum values of the tree for $x_2 = 0$ and $x_2 = 1$, and hence:

$$R_{3\to 2}(x_2) = [-0.1, 0.1]$$

The equation for $R_{3\to 3}$ is:

$$R_{3\to 3}(x_3) = \max_{x_2}(\gamma_3(x_3) - (x_2 \otimes x_3) + Q_{2\to 3}(x_2))$$

and the decision tree is:


$R_{3\to3}(x_3) = \max_{x_2}$
$x_2 = 0$
$x_2 = 1$
$\gamma_3(x_3) - (x_3 \otimes 0) - 0.1$
$\gamma_3(x_3) - (x_3 \otimes 1) + 0.1$
$x_3 = 0$
$x_3 = 1$
$x_3 = 0$
$x_3 = 1$
$-0.1 - 1 - 0.1 = -1.2$
$0.1 - 0 - 0.1 = \boxed{0}$
$-0.1 - 0 + 0.1 = \boxed{0}$
$0.1 - 1 + 0.1 = -0.8$

and hence the result is:

$$R_{3\to3}(x_3) = [0, 0]$$

Having computed the messages, the last marginal function Z_3 can be computed:

$$Z_3(x_3) = R_{2\to3}(x_3) + R_{3\to3}(x_3) = [\mathbf{0.2}, -0.2]$$

The resulting color configuration can then be derived from the marginal of each variable:

$$Z_1(x_1) = [\mathbf{0.1}, -0.1] \to \text{variable } x_1 \text{ has color } 0$$

$$Z_2(x_2) = [-0.2, \mathbf{0.2}] \to \text{variable } x_2 \text{ has color } 1$$

$$Z_3(x_3) = [\mathbf{0.2}, -0.2] \to \text{variable } x_3 \text{ has color } 0$$

And as can be easily seen, there are no conflicts between the color configurations of the variables, hence the Max-Sum algorithm has found the optimal solution to the graph coloring problem.

4

Implementation

In the previous chapters, the aims of the thesis were outlined and motivated, an overview of the current related research in the area of distributed constraint optimizations was given, and the main aspect of the thesis, the Max-Sum algorithm, was introduced and defined. The following chapter covers one of the main contributions of this thesis: The design and implementation of the Max-Sum algorithm in an appropriate distributed graph computing framework.
First, the choice of the framework is motivated. In the following sections, the software design and mapping aspects are discussed, while in the later sections implementation details are described.

4.1 Frameworks

Distributed problems pose sensitive requirements when it comes to the implementation. Scalability is a main concern [Tanenbaum and Van Steen, 2007] and thus, engineers and researchers have to find methods to speed up distributed computations even when they are performed across thousands of nodes. While the computational power of hardware devices and processors increases at high rates, and is expected to continue in this fashion [Schaller, 1997], it is not a valuable option to rely on a progress in hardware performance in order to build and implement large-scale distributed systems to solve scientific problems [Stutz et al., 2010]. Hence, the speed-up of such systems has to be achieved by building frameworks that exploit properties such as parallelism, asynchronicity of computation, and others. In the context of large-scale distributed graph problems, there are some existing frameworks that provide useful approaches to large-scale problems.

A well-known and broadly used framework in this context is the MapReduce framework [Dean and Ghemawat, 2008]. MapReduce consists mainly of a Map-function and a Reduce-function which both take key-value pairs as input and produce key-value pairs as output. The distribution of the computational task is achieved by parallelizing the Map-invocation across multiple machines (real or virtual) by splitting the input data into M split parts, where M is the number of distributed machines [Dean and Ghemawat, 2008]. The Reduce-function is distributed in a similar fashion, and a more detailed presentation of it is offered in
[Dean and Ghemawat, 2008]. Because of its strict key-value orientation, computations may have to be mapped to this key-value model in order to be executable on MapReduce [Stutz et al., 2010], and hence, it may not be suitable for problems such as DCOPs.

A framework that tries to overcome the mentioned drawback of MapReduce is the GraphLab framework [Low et al., 2010]. GraphLab allows its users to issue complex computational problems by providing a data model that makes it possible to map complex graph structures in contrast to MapReduce's key-value structure. The data model consists of a data graph and a shared data table [Low et al., 2010]. Although GraphLab provides better means of mapping graph-based problems, it has non-obvious efficient distribution [Bernstein, 2012] and hence may not be optimal for the goal of this thesis.

Another interesting candidate is the Pregel-framework [Malewicz et al., 2010]. Pregel is a Google development and executes in discrete, synchronous computation steps, which are called supersteps in Pregel-terminology. The user specifies a function that will be executed concurrently on every single vertex at a superstep S [Malewicz et al., 2010]. Functions on vertices have access to messages received by the vertex in the superstep $S - 1$ and send messages to other vertices that will be processed at superstep $S + 1$ [Malewicz et al., 2010]. Pregel achieves great results in terms of performance and has proven scalability and error recovery [Bernstein, 2012]. Despite its nice properties, Pregel has the essential drawback of being inherently synchronous as well as the fact that the framework is not open-source [Bernstein, 2012]. A very promising candidate is the Signal/Collect framework [Stutz et al., 2010]. Signal/Collect is a programming model for synchronous and asynchronous graph problems. It provides the possibility to define multiple vertex types and can be used in synchronous, as well as in asynchronous modes. Furthermore, it provides useful mechanisms to aggregate statistics over the processed graphs and detect algorithm convergence by built-in mechanisms. These properties make Signal/Collect the most suitable one among the here-mentioned frameworks, especially with respect to distributed constraint optimization problems. A DCOP algorithm such as the Max-Sum algorithm, may profit from being implemented in the Signal/Collect framework since the asynchronous possibilites of Signal/Collect allow a further speedup of the computation [Stutz et al., 2010]. The possibility to introduce multiple user-defined vertex types makes it suitable for the Max-Sum algorithm, where one has to deal with multiple vertex types such as function and variables nodes. Also the mechanisms to measure graph-wide statistics and the possibility to implement convergence detection through built-lin structures come in handy in a task as the one this work deals with. Due to these considerations, the implementations in this thesis will be done in the Signal/Collect framework, which will be described in more detail in the next section.

4.2 Signal/Collect

In the Signal/Collect programming model, all computations are executed on a graph where the vertices are treated as the main computational units. The vertices may have internal state and can interact through messages - *signals* in Signal/Collect terminology - which are sent along the edges of the graph to other vertices [Stutz et al., 2010]. The vertices can *collect* incoming signals from their neighbors, perform any computation involving received information and / or internal state, and then continue signaling to their neighbors [Stutz et al., 2010]. Formally, a compute graph in Signal/Collect is defined as a directed graph $G =< V, E >$, where V is the set of vertices and E is the set of edges.

Every edge $e \in E$ has the following attributes:

- **e.source**: The source vertex (or a pointer to it)
- **e.sourceId**: Id of the source vertex
- **e.targetId**: Id of the target vertex

Every vertex $v \in V$ has attributes:

- **v.id**: A unique Id for every vertex
- **v.state**: The internal state of the vertex
- **v.outgoingEdges**: A list of all edges $e \in E$ where $e.source = v$

- **v.signalMap**: A map data structure that stores key-value pairs with the Id's of the neighbor vertices as keys and the most recently received signals from the corresponding vertices as values

- **v.uncollectedSignals**: A list containing the signals that arrived since the last execution of the collect operation on v

In addition to the attributes, vertices $v \in V$ also have an abstract procedure named *v.collect* while edges $e \in E$ have an abstract signal procedure $e.signal$. A user can express an algorithm in the Signal/Collect model by specifying the signal and collect procedures.
Signal/Collect allows to execute algorithms in a variety of execution modes. The most important with respect to this thesis are the *Synchronous* execution mode, to which the text will simply refer to as synchronous or SYNC and the *PureAsynchronous* execution mode to which the text will refer to as ASYNC or asynchronous. Signal/Collect[1] provides an execution platform developed in Scala, and is released under Apache License 2.0. It achieves parallelization of its computations through exploitation of multi-core processor architectures [Stutz et al., 2010]. Worker threads are each responsible for a certain part of the graph by assigning them the vertices and their respective edges through a hash function [Stutz et al., 2010].

4.3 Implementation Design

Since it is unsuitable to cover every detail of the implementation, the following sections of this chapter will cover the most important design decisions in the implementation of the Max-Sum algorithm.

[1] *http://www.ifi.uzh.ch/ddis/research/sc*, Revision: 1374748765

4.3.1 Vertex and Edge Mapping

The algorithm takes an arbitrary undirected graph as an input. The graph represents the agents involved in the DCOP and the structure of their communication as can be seen in figure 4.1.

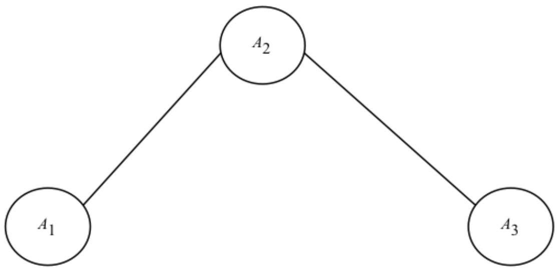


Figure 4.1: Example of an input graph for the algorithm

The input graph needs to be transformed further in order to be processable by the Max-Sum algorithm; it needs to be in the form of a factor graph suitable for graph coloring problems as introduced in section 2.2.3. This is done by expanding every agent node A_n into two different vertices, a function vertex, denoted U_n, and a variable vertex, denoted x_n [Farinelli et al., 2008]. Each function vertex U_n is then connected to its own variable vertex x_n and to the variable vertex of its neighboring agents. An example is shown in figure 4.2.

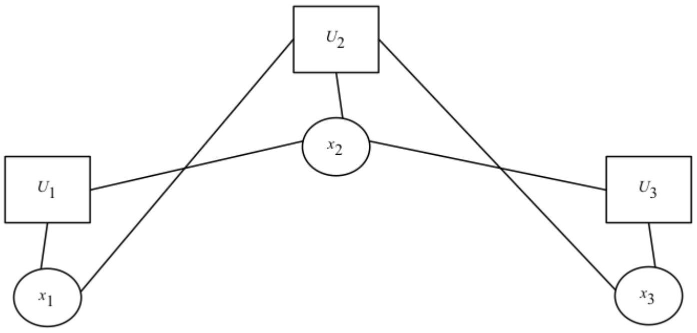


Figure 4.2: Transformed input graph

However, the factor graph needs to be mapped on Signal/Collect vertex and edge types, in order to run the algorithm in the Signal/Collect framework. Since there are two types of vertices in factor graphs and the Max-Sum algorithm, it is necessary to define specific vertex types in Signal/Collect in order to be able to map the theoretical concepts of the Max-Sum algorithm on a software design. In Signal/Collect this can be easily done by extending one of the basic vertex classes provided by the Signal/Collect library.
The design is as follows: Two vertex types (or classes) are defined. The class `FunctionVertex` represents the utility function vertices U_n in the factor graph, and the class `VariableVertex` represents the variable vertices x_n, as the name implies. Both have a common superclass, `MaxSumVertex`, that provides generic and common functionality needed in both vertex types. Furthermore, two types of edges will be defined: `FunctionToVariable` computing the messages $R_{m \to n}$ and `VariableToFunction` computing the messages $Q_{n \to m}$. The reason to have two different edge types to represent undirected graph connections lies in the nature of the Max-Sum algorithm. The Max-Sum algorithm propagates two types of messages where the type of the message depends on the direction in which the message is sent. Hence, in order to compute both types, the implementation needs a separated view of the bidirectional edges of the factor graph. The message from variable to function $Q_{n \to m}$ (c.f. equation 3.4) will be computed by a `VariableToFunction` edge instance, and the message from function to variable $R_{m \to n}$ (c.f. equation 3.5) will be computed by a `FunctionToVariable` instance.

Thus, the factor graph that resulted from the transformation of the input graph needs to be further transformed in order to result in the presented Signal/Collect vertex and edge structure. Figure 4.3 shows the mapped version of the factor graph from figure 4.2. The rectangular nodes with the captions FV_n denote the instances

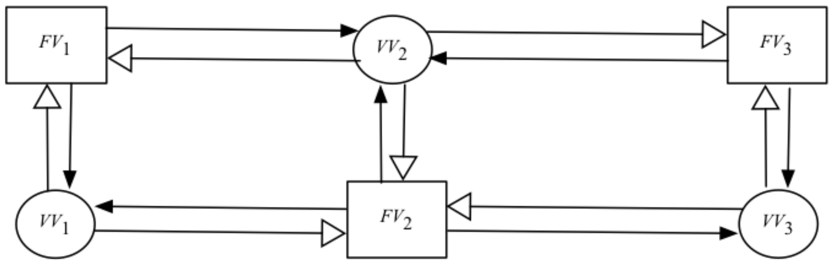


Figure 4.3: Resulting mapping of the factor graph on to signal/collect vertex and edge types

of the class `FunctionVertex`, while the circle nodes with the captions VV_n represent instances of the class `VariableVertex`. The arrows with empty white arrow heads show the instances of `VariableToFunction` (V2F) edges and the arrows with bold black arrow heads represent instances of `FunctionToVariable` (F2V) edges. The whole process that transforms the input graph to a factor graph and then to a Signal/Collect mapping will be done automatically on the startup of the system. The user only passes a file name and a path to the file containing the unmapped graph either in edge list format or in ADOPT format.

4.3.2 Unique Ids and Messages

In order to have a concise and practical identification scheme for the vertices in the implementation, a special id type named `MaxSumId` had to be defined. A `MaxSumId` instance basically has three fields. The most important field is `id : String`, a character string that represents the unique identification of a vertex in the implemented system. A vertex can be either an instance of `VariableVertex` or an instance of `FunctionVertex`, hence `id : String` will have the form *v1234* for `VariableVertex` instances, or *f1234* for `FunctionVertex` instances. The field

`idNumber : Int` stores only the id of the vertex without the corresponding prefix (v or f). This is useful in situations where one wants to find out the id of a variable that belongs to the same agents as a given function or vice versa. The third field is `isVariable : Boolean` determines if the vertex having this id is a variable or a function vertex. This will also be needed during the execution when the program has to perform some computations only on one of the two vertex groups. Besides the presented fields, an instance of `MaxSumId` also provides two methods `equals(other : MaxSumId)` and `hashCode()`. The equals method overrides the standard Scala equality check to adapt equality comparison for `MaxSumId`. The hashCode method is necessary since instances of `MaxSumId` are intended to be used as keys in map-like data structures. As in Java, Scala maps keys in those data structures by computing their hash code. Hence, an appropriate hash computation method was defined.
The type of the signal being passed along the edges of the graph is a custom type named `MaxSumMessage`. A `MaxSumMessage` is a container class for three things: the value of the message being passed, the source vertex id where it came from and the target vertex id where it is intended to arrive. The values in instances of `MaxSumMessage` are of the type `ArrayBuffer[Double]` where the length of the `ArrayBuffer[Double]` is equal to the number of colors in the graph coloring problem. The source and target ids are instances of `MaxSumMessage`.

4.3.3 Message Computation

The implementation presented in this thesis differs from the theoretical concept in some points and follows own approaches. One of the issues where the implementation differs from the concept proposed in [Farinelli et al., 2008] is the computation of the messages. While in [Farinelli et al., 2008] the vertices act as the main computational units computing the messages, the implementation in this thesis uses the edges as main computational units. The messages will be computed through the edge instances by accessing necessary data from the respective edges source vertex. This is especially convenient in the Signal/Collect environment, since by definition, edges compute the signals. Furthermore, each function-variable neighbor pair involves different variables in the computation, and hence it would be inefficient to perform the computation on the vertices where one would have to choose the correct variables first. By letting the edges perform the computation, the implementation ensures that only the needed neighbor structure is available at the computation location of the message.

The computation of the messages $Q_{n\to m}(x_n)$ from variables to functions is executed on instances of `VariableToFunction` and is straightforward. The source vertex aggregates all received messages in its collect method, which then are used by the edge instance of the V2F class to sum up and compute $Q_{n\to m}$ in its signal method, according to equation 3.4. In order to avoid having the messages grow indefinitely over execution time, a (negative) normalization factor α_{nm} is added to the sum of received messages. As mentioned in [Farinelli et al., 2008] and in section 3.1 of this thesis, the normalization factor α_{nm} should be chosen such that:

$$\sum_{x_n} Q_{n\to m}(x_n) = 0$$

Following equation 3.4, the normalization has to be included in the computation of the message as in:

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

To compute the normalization factor α_{nm}, one has to restructure the equation from above to:

$$\sum_{x_n=0}^{|C|} \left(\alpha_{nm} + \sum_{m' \in \mathcal{M}(n) \setminus m} R_{m' \to n}(x_n) \right) = 0$$

$$\sum_{x_n=0}^{|C|} \alpha_{nm} + \sum_{x_n=0}^{|C|} \left(\sum_{m' \in \mathcal{M}(n) \setminus m} R_{m' \to n}(x_n) \right) = 0$$

$$|C| * \alpha_{nm} + \sum_{x_n=0}^{|C|} \left(\sum_{m' \in \mathcal{M}(n) \setminus m} R_{m' \to n}(x_n) \right) = 0$$

And hence, the value of the normalization factor results in:

$$\alpha_{nm} = -\frac{\sum_{x_n=0}^{|C|} \left(\sum_{m' \in \mathcal{M}(n) \setminus m} R_{m' \to n}(x_n) \right)}{|C|}$$

The factor α_{nm} has to be computed separately for each new message $Q_{n \to m}(x_n)$. Note that $|C|$ stands for the number of colors involved in the graph coloring problem being processed.

For the messages $R_{m \to n}(x_n)$ from functions to variables, the computation is more involved in contrast to $Q_{n \to m}$. As can be seen in equation 3.5, the computation of $R_{m \to n}$ demands a maximization over several variables. In the hands-on example of the previous chapter, the maximization task was solved by following a tree exploring all possible variable configurations and solutions of equation 3.5. Hence, the implementation would have to compute the value of formula 3.5 for every possible variable configuration of the variables involved in the computation and then store the different solutions in a simple data structure, such as a table. One then could easily find the maximum by inspecting the result values in the table and comparing them. In the implementation for this thesis, this described task of constructing all possible configurations and finding the max value among them is achieved through a procedure similar to the backtracking method described in [Sedgewick and Wayne, 2011].

Algorithm 1 The computation of the message $R_{m \to n}(x_n)$

```
function R_M_N(neighborVariables)
    for i ← 0 to neighborVariables.size - 1 do
        varValues[i] = 0
    end for
    for outerColor ← 0 to numOfColors - 1 do
        varValues[0] = outerColor
        $R_{m \to n}$[outerColor] = backtrack(neighborVariables , varValues , 1)
    end for
end function
return $R_{m \to n}$
```

Algorithm 1 shows the encapsulating function that computes the message $R_{m \to n}(x_n)$ on a `FunctionToVariable` edge. The function takes an array-type paramater called `neighborVariables` holding the ids of all neighboring `VariableVertex` instances. An array with the same length as `neighborVariables` is initialized with zeros and named `varValues`. This array represents the current color configuration of the variables in `neighborVariables`. The second loop goes over the possible color values for `outerColor`. `outerColor` is the color of the variable on which the message $R_{m \to n}(x_n)$ being computed depends on. For example, in the computation of the message $R_{1 \to 2}(x_2)$, `outerColor` would iterate over the possible color values for x_2 and then compute the result of equation 3.5 separately for each value of `outerColor`. The separate results are then stored in the final message array $R_{m \to n}$ with `outerColor` as an index.

Algorithm 2 The recursive backtracking function

```
function BACKTRACK(variableNames , variableValues , index)
    max = 0.0
    if index == variableNames.size - 1 then                ▷ Exit condition of the recursion
        for color ← 0 to numOfColors - 1 do
            max = 0.0
            variableValues[index] = color
            max = max(max, equation(variableNames , variableValues))
        end for
    else                                                   ▷ Recursive branch
        for color ← 0 to numOfColors - 1 do
            max = 0.0
            variableValues[index] = color
            max = max(max, backtrack(variableNames , variableValues , index + 1))
        end for
    end ifreturn max
end function
```

The actual computation of the elements of the resulting message vector $R_{m \to n}$ is performed through the function `backtrack`, which is shown in Algorithm 2. The function takes two arrays and one integer as parameters. The arrays `variableNames` and `variableValues` store the ids and the color configuration of the variables involved in the computation. The integer `index` is used to iterate through the arrays and determine when the recursion has to stop.

The function will call itself recursively as long as `index` has not reached the end of `variableNames` or `variableValues` respectively. At each recursion step, max is computed for every possible color of the variable at position `index`. Hence, before calling the `backtrack` function recursively, the value of the current variable (e.g. `variableNames[index]`) is set to the current color. Then, the recursion is called, which eventually returns a floating value for max. The recursion tree representing the recursion steps of the backtracking method would be similar to the decision trees shown in section 3.2.

4.3.4 Complexity Analysis

Although this thesis puts the emphasis on the empirical performance and results of the Max-Sum algorithm, it is worth analyzing the theoretical complexity of the implementation. This may help in future implementations and experiments with the algorithm, by discovering potential room for improvement.

Having the set of possible colors $C = \{c_1, c_2, ..., c_n\}$ and $V = \{x_1, x_2, ..., x_k\}$ the set of involved variable vertices, the recursion tree of the backtracking function in Algorithm 2 will look as depicted in figure 4.4.

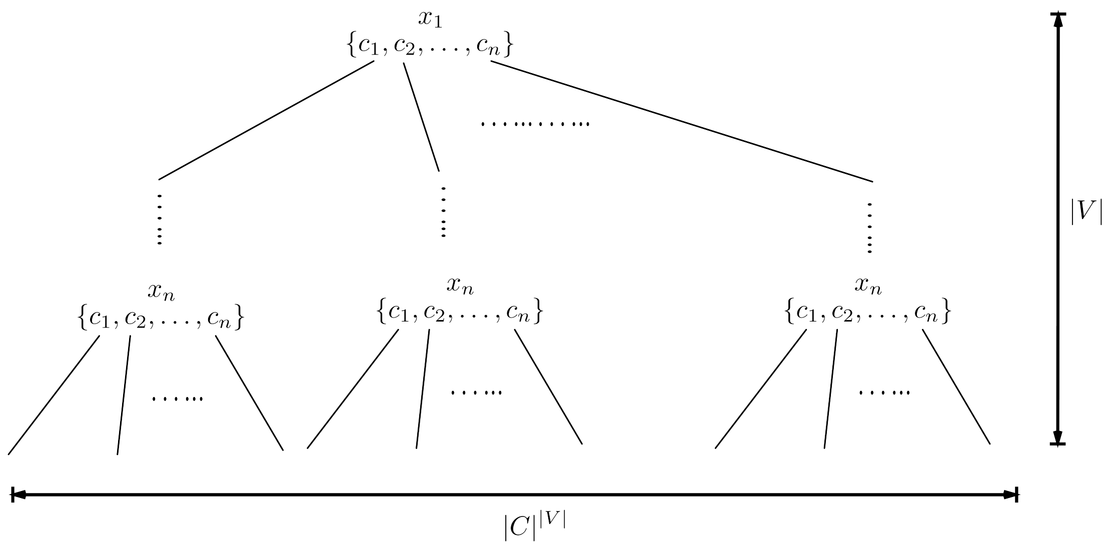

x_1
$\{c_1, c_2, \ldots, c_n\}$
x_n
$\{c_1, c_2, \ldots, c_n\}$
x_n
$\{c_1, c_2, \ldots, c_n\}$
x_n
$\{c_1, c_2, \ldots, c_n\}$
$|V|$
$|C|^{|V|}$

Figure 4.4: Recursion tree of Algorithm 2

One can easily see that the depth of the recursion tree will be equal to the cardinality of the set of variables, $|V|$. The actual computation of the possible solutions takes place at the leaves at the bottom level of the recursion tree. Figure 4.4 shows that the recursion tree has $|C|^{|V|}$ leaves. Hence, the computation of equation 3.5 will be executed $|C|^{|V|}$ times with different variable configurations. But the actual computation is not the only work that is done by the recursion. The comparisons through the *max*-function should also be taken into account. The max-comparisons take place at the "inner nodes" of the recursion tree, that is, at all nodes except for the root node and the leave nodes. The number of inner nodes in the recursion tree is:

$$|C|^1 + |C|^2 + ... + |C|^{|V|-1} = \sum_{k=1}^{|V|-1} |C|^k$$

Hence the total number of steps (computations and comparisons) of the recursion then results in:

$$\mathcal{O}(|C|^{|V|} + \sum_{k=1}^{|V|-1} |C|^k) \tag{4.1}$$

And because the following relation holds:

$$\mathcal{O}(\sum_{k=1}^{|V|-1} |C|^k) \subset \mathcal{O}(|C|^{|V|})$$

the overall worst case complexity term can be reduced to:

$$\mathcal{O}(|C|^{|V|}) \tag{4.2}$$

Equation 4.2 shows clearly that the theoretical complexity and performance of the implementation of the Max-Sum algorithm depend highly on the number of involved variable vertices $|V|$ and the number of colors $|C|$ in the graph coloring problem. The involved variables in V are the neighboring variable vertices of the source function vertex for the computed message. Because usually $|V| >> |C|$, the overall performance will depend mainly on the number of neighboring vertices. Stated in a graph-theoretical way, the complexity depends on the average degree

$deg(v)$ of the input graph.

5

Benchmarks

The previous chapter explained why the Signal-Collect framework was chosen for the implementation and how the implementation of the Max-Sum algorithm was conceptually designed. The implementation enables the empirical experiments with the Max-Sum algorithm. The first part of the experiments consists of a comparison with other candidate DCOP algorithms using a selected benchmark on a single machine. The following chapter will demonstrate how the evaluation benchmark is designed, which algorithms are used, which measurements are taken and on which input data the benchmarking is performed.

5.1 Outline

The main goal of using this benchmark is to get useful empirical evidence about the performance of the Max-Sum algorithm in comparison to other local iterative DCOP algorithms. It is of special interest to get results comparable to the empirical evaluation of the Max-Sum algorithm in [Farinelli et al., 2008]. A comparison and conclusion between the results of this thesis and the results in [Farinelli et al., 2008] may lead to interesting insights. Hence, it makes sense for the design of the benchmark to be similar to the one presented in [Farinelli et al., 2008]. On one side, it is interesting to see if the results can be reproduced, but on other hand, the evaluations presented in this thesis are implemented in a special graph processing framework and hence one might be interested in differences between the results of the benchmarks in this thesis and in [Farinelli et al., 2008].

5.2 Infrastructure

All of the following benchmarks and evaluations in this chapter were executed on a single machine with standard hardware. The machine runs on a single 64 bit processor consisting of two cores each having a maximum frequency of 2533 Mhz. The processor works on a x86 architecture instruction set and provides 3012 kB of L2-cache. The machine has a 1.07 Ghz system bus and has access to 4 GB of random access memory.
The implementation is programmed in Scala, while the machine uses Scala version 2.10.0 and Java version 1.7.0. The incorporated Signal-Collect framework has revision 1374748765.

5.3 Candidate Algorithms

Since one of the intentions of the benchmark is to be able to make comparisons to the results in [Farinelli et al., 2008], it is obvious that the candidate algorithms contesting the Max-Sum algorithm in the benchmark are the same as the ones used in Farinelli's work. A slight difference to the benchmark in [Farinelli et al., 2008] is the fact that in this evaluation the DPOP (Dynamic Programming Optimisation Protocol) algorithm will not participate in the benchmark. This is because DPOP is a complete DCOP algorithm (c.f. section 2.2.2) and this thesis concentrates mainly on the *local iterative algorithms*. Hence, the candidate algorithms to be evaluated in the benchmark are:

- Candidate 1: Max-Sum algorithm
- Candidate 2: Distributed Stochastic Algorithm [Zhang and Xing, 2002]
- Candidate 3: Best-Response algorithm [Fudenberg and Levine, 1998]

5.3.1 Distributed Stochastic Algorithm

The distributed stochastic algorithm (DSA) [Zhang and Xing, 2002] describes not only a certain algorithm, but rather a whole family of algorithms based on the same concept. To be more precise, the basic algorithm structure and steps are the same for all DSA variants, but some important decision rules vary depending of the DSA variant.

[Zhang and Xing, 2002] explain the basic structure of DSA. An adapted version for graph coloring problems is shown in Algorithm 3.

Algorithm 3 The basic structure of DSA

```
Choose color $c \in C = \{c_1, c_2, ..., c_n\}$ randomly
while termination condition not met do
    if $c$ was changed in previous iteration step then
        for $v \in neighbors$ do
            send(c,v)                                   ▷ Send new value to neighbors
        end for
    end if
    for $v \in neighbors$ do
        collect new values from v (if any)
    end for
    select and assign new value for $c$      ▷ see next table for different DSA variants
end while
```

The steps shown in Algorithm 3 are executed by every single agent participating in the DCOP. First, an agent starts by choosing a random color among the domain of possible colors in the problem. Then the agent loops until a certain termination condition is met. While looping, each agent sends its new color to all neighbors if it changed since the last iteration. It then collects potential new values for the colors of its neighbors. In the last step of each iteration, an agent decides stochastically wether to change its color value. [Zhang and Xing, 2002] motivate the idea of stochastically changing the color value by the fact that a change of the color will possibly lead to a reduction of the number of conflicts. The agent computes its new color based on its current color and on the perceived colors of its neighbor agents [Zhang and Xing, 2002] with the aim of finding a color such that the number of conflicts is reduced. If it can't find such a color, the value is not changed, if it finds such a color, the value may or may not be changed depending on the strategy of the DSA variant [Zhang and Xing, 2002]. The variants and their strategies are shown in table 5.1.

Variant	$\Delta > 0$	Conflict, $\Delta = 0$	No conflict, $\Delta = 0$
DSA-A	v with p	-	-
DSA-B	v with p	v with p	-
DSA-C	v with p	v with p	v with p
DSA-D	v	v with p	-
DSA-E	v	v with p	v with p

Table 5.1: Strategies for different DSA-Variants [Zhang and Xing, 2002]

[Zhang and Xing, 2002] have analyzed the solution quality of the different DSA variants from table 5.1. They found out that DSA-A and DSA-B, which are considered as more "conservative" variants, deliver a better solution quality on graph coloring problems. Hence, this thesis will not take into account all of the DSA variants shown in table 5.1 but will only use DSA-A and DSA-B as candidates.

Explanation of table 5.1: Δ represents the *best possible conflict reduction* between an old color value and a newly chosen one. v is the new color value that yields

Δ whereas $p \in [0, 1]$ is the probability for changing the old value to the new value v. When $\Delta > 0$, there must be a conflict, otherwise Δ could not be greater than zero. DSA-A will change its color value with probability p if the new value leads to a reduction of the number of conflicts ($\Delta > 0$). If $\Delta = 0$, there are two possible situations: Either there are conflicts, but the agent couldn't find any value reducing the conflicts, or there are no conflicts at all, and hence there is no potential new color value that could lead to a lower number of conflicts. In both cases, DSA-A will not change the value, whereas DSA-B will change the value with probability p in the first case. This is the only difference between the two variants.

5.3.2 Best-Response Algorithm

The Best-Response strategy is a concept widely used in game-theory [Fudenberg and Levine, 1998]. In a Best-Response strategy, an agent or a player chooses its own strategy such that it results in the highest possible response in terms of utility, given the current strategies or states of its neighboring agents.

Algorithm 4 Best-Response Algorithm [Farinelli et al., 2008]

```
for $v \in neighbors$ do
    neighborStates.add(v.state)
end for
newState = chooseBestState(neighborStates)
if newState != oldState then                ▷ If the state has changed inform neighbors
    for $v \in neighbors$ do
        sendState(v,newState)
    end for
end if
```

The Best-Response strategy can be applied to our DCOP graph coloring model as listed in algorithm 4. First, a node chooses its state based on the current states of its neighbors such that it has the best possible outcome (e.g. the lowest number of conflicts) [Farinelli et al., 2008]. If the new state is different from the old state, all neighbors have to be informed about the state update. This procedure is executed on all vertices to solve a graph coloring problem. [Farinelli et al., 2008] note that the Best-Response algorithm represents a lower bound on the performance of any approximate DCOP algorithm, since it uses the minimum computation and communication possible.

5.4 Metrics and Measurements

After having explained the infrastructure and candidate algorithms used in the evaluation, it is important to choose appropriate and expressive metrics or statistics to be evaluated in the benchmark.
The measures should allow conclusions about the quality of the solutions an algorithm finds and about the performance of the algorithm (e.g. time to find a solution). Several measures were chosen in order to cover both performance and quality aspects. The quality of a solution in a graph coloring setting is mainly characterized by the number of conflicts. Hence, the number of conflicts will be the main quality aspect evaluated in the benchmark. There will be conflict measurements for synchronous and asynchronous executions of the algorithms (c.f. list below). The performance of the algorithms is characterized through the execution time (or number of execution cycles in a synchronous execution) until the algorithm converges. One can clearly see that it comes in handy to be using the Signal/Collect framework since its concise way of expressing asynchronous and synchronous executions of the same algorithm makes it easy to switch between those two execution modes and allows for a more detailed evaluation.

- Conflicts:

  - Synchronous: Conflicts per execution cycle
  - Asynchronous: Conflicts over execution time
  - Relation between cycles and execution time

- Convergence:
  - Synchronous: Execution cycles until algorithm converges
  - Asynchronous: Time until algorithm converges

Note that [Farinelli et al., 2008] evaluate the Max-Sum algorithm against DSA, Best-Response and DPOP algorithms using a variety of measures, including also conflicts per execution cycle and cycles until convergence. So the choice of these measures allows for a possible comparison of results with the results in [Farinelli et al., 2008]. The conflicts over time, time to convergence and time versus cycles were not included in [Farinelli et al., 2008] and hence are a special contribution of this thesis.

5.4.1 Conflicts

The number of conflicts serves as a quality metric in the benchmark. As mentioned in section 2.2.3, a conflict in a graph coloring setting is defined as the situation when two neighboring vertices have the same color. To analyze the change of the number of conflicts over execution time, the number of conflicts after each algorithm cycle will be recorded. A cycle is defined by Farinelli as the period in which all vertices have had the opportunity to update their state and have exchanged messages [Farinelli et al., 2008]. When translating Farinelli's notion of a cycle into a Signal/-Collect compatible sense, a cycle is defined as the phase where all vertices have tried to collect and signal once. Note that a vertex in Signal/Collect may or may not execute a signal or collect execution during a cycle depending on the implementation. Since the notion of a discrete cycle only makes sense in a synchronous setting where time can be discretized into clear steps, this evaluation will be executed in Signal/Collect's synchronous mode. As in [Farinelli et al., 2008], for this measure, the algorithm will be run for a fixed number of cycles to analyze the number of conflicts at each step.

In order to make a comparison between the asynchronous and synchronous performance of the algorithm, the benchmark will analyze the number of conflicts over execution time, measured at fixed intervals.

To measure the number of conflicts at each step, Signal/Collect provides a mechanism named `AggregationOperation`, which is an abstract class defining methods that enable measuring of characteristics of the vertices iteratively at each execution cycle. It serves as a distributable way to count statistics over the whole Signal/-Collect graph on which the algorithm is being executed. For each of the candidate algorithms involved in the benchmark, a subclass of `AggregationOperation` was defined and used to take measures. The respective `AggregationOperation` calls on each vertex a method returning the number of conflicts at that vertex. For the benchmarks in this chapter, a central static object was created, to which all vertices will write their current color during computation of the algorithm. By querying this central object, all vertices can determine the color of their neighbors and hence compute their respective number of conflicts in order to return this value to the `AggregationOperation`. This approach is necessary since a vertex in Signal/Collect has only access to the IDs of its neighbors, but not to the object reference of the neighbor itself. Since the Max-Sum algorithm does not exchange the actual states of the vertices, neighbors do not know the colors of each other. The illustrated use of a central object circumvents this problem.

5.4.2 Convergence

The convergence properties help in discussing the performance of an algorithm and hence will be part of the evaluation. This indicator measures the time until the algorithm converges. There are two kinds of convergence in the context of local iterative DCOP algorithms: Message convergence and state convergence. The messages are said to converge when an updated message has the same value as the last message that was sent along the same path [Farinelli et al., 2008]. The convergence for the state of the vertices is defined analogously. The benchmark will simply count the number of algorithm cycles (c.f. section 5.4.1) until both all messages and all states in the algorithm have converged. To check message convergence, the implementation will look at the marginals of the variable vertices.
To detect convergence, Signal/Collect provides useful termination structures. Each vertex type has the methods `scoreCollect` and `scoreSignal`. Signal/Collect detects convergence of an algorithm based on these two methods and based on the `signalThreshold` and `collectThreshold` constants that are defined globally
[Stutz et al., 2010]. A Signal/Collect computation continues as long as there exists at least one vertex where `scoreCollect` will return a value higher than
`collectThreshold`, or `scoreSignal` returns a value higher than `signalThreshold` respectively
[Stutz et al., 2010]. A specific algorithm implementation in Signal/Collect, such as the Max-Sum implementation, may define its own convergence rules by overriding the two scoring methods and setting the corresponding thresholds.
In the implementations of the algorithms and benchmarks in this thesis, the
`signalThreshold` and the `collectThreshold` constants are both set to 0.0. Hence an algorithm will converge when all collect scores and signal scores are below or equal to this threshold. The scoreCollect method implemented in the vertices used

to detect convergence for the Max-Sum implementation is shown below:

Algorithm 5 scoreCollect method

```
if edgesModifiedSinceCollectOperation then
    return 1.0
else
    if stateHistory.hasConverged $\wedge$ marginalHistory.hasConverged then
        return 0.0
    else
        return 1.0
    end if
end if
```

The method shown in 5 returns 0.0 if the `stateHistory` and the `marginalHistory` of the current vertex have converged. In all other cases, the method returns 1.0 indicating that the vertex should continue to execute collect operations. The two fields named `stateHistory` and `marginalHistory` are references to instances of the type `ConvergenceHistory[T]` and `MarginalHistory[T]`, where T is any scala language type. These types were designed as a helping data structure to support the convergence detection in this thesis. The implementation of `ConvergenceHistory` is shown in listing 5.1. The need for such a data structure comes from the fact that the state and marginal changes do not propagate immediately in the factor graph. For example, if a variable vertex changes its color, this will lead to different messages arriving at one of its connected function vertices in the first execution cycle after the change. Only after a second step, the changed state of the original variable vertex will have an impact on the next variable vertex, because only now the middle function vertex will recompute a message with the new state information and send it to the next variable vertex. Following this logic, it takes multiple steps between two vertices in order to confirm convergence. This is why a data structure as the one presented here is needed.

Listing 5.1: The implementation of `ConvergenceHistory[T]`

```
class ConvergenceHistory[T](c: Int) extends Queue[T] {

  val capacity = c

  def push(element: T): Unit = {
    if (size < capacity) {
      enqueue(element)
    } else {
      dequeue
      enqueue(element)
    }
  }

  def isFull(): Boolean = {
    if (size < capacity) {
      false
    } else {
      true
    }
  }

  def hasConverged(): Boolean = {
    if (!isFull) {
      false
    } else {
      var converged = true
      val list = this.toList
      var i = 0
      while ((i < list.size - 2) && converged) {
        if (list(i) != list(i + 1)) {
          converged = false
        }
        i += 1
      }
      converged
    }
  }
}
```

`ConvergenceHistory[T]` is a subtype of the scala language type `Queue[T]`, which is a simple first-in-first-out data structure holding instances of parameter type `T`. `ConvergenceHistory[T]` extends it by adding a field named `capacity` to `Queue[T]`, thus introducing a maximum size for the history. The wrapper method `push` allows to add an element to the history. It stores the most recent state entries of a vertex. If the capacity is reached, the implementation automatically removes an element from the history according to the FIFO rule. Hence, the history will always contain the most recent entries. In addition, the type provides a `hasConverged` function that checks if the entries in the history indicate convergence. Convergence occurs when all entries in the history have the same value. It is important to note that the field `marginalHistory` in method 5.1 has the type `MarginalHistory[T]`, which is a subtype of `ConvergenceHistory[T]`. This design decision was necessary because `marginalHistory` keeps track of the convergence of the marginals at a vertex. Due to the high number of floating point operations involved in the computation of those marginals, the results may deviate because of rounding errors. Hence, detecting convergence based on equality is not an option. This is why there is a special subtype checking convergence for marginals. The implementation is shown in 5.2.

Listing 5.2: The implementation of `MarginalHistory[T]`

```
class MarginalHistory[T](c: Int, epsilon: Double)
extends ConvergenceHistory[ArrayBuffer[Double]](c) {

  val e = epsilon

  override def hasConverged() = {
    if (!isFull) {
      false
    } else {
      var converged = true
      val list = this.toList
      var i = 0
      while ((i < list.size - 2) && converged) {
        if (compare(list(i), list(i + 1))) {
          converged = false
        }
        i += 1
      }
      converged
    }
  }

  private def compare(a: ArrayBuffer[Double], b: ArrayBuffer[Double]) = {
    var equal: Boolean = true
    var i = 0
    while (i < a.length && equal) {
      if (abs(a(i) - b(i)) > e) {
        equal = false
      }
      i += 1
    }
    equal
  }

}
```

`MarginalHistory[T]` detects convergence not based on value equality, but on an epsilon-test of the pairs of marginals in the history. Instead of checking if the values are the same, the function checks wether the absolute difference of the two values in comparison is smaller or greater than a predefined value ϵ. This addresses the rounding errors in the computations of the marginals and leads to a more concise convergence detection.

5.5 Data Sets

There are 3 different graphs on which the benchmarks will be executed. The graphs were taken from the *USC Distributed Constraint Optimization Problem Repository* [1], sometimes also referred to as *ADOPT repository*. This repository is used widely throughout DCOP literature and graph coloring problems. The chosen data set is a set of undirected graphs prepared for graph coloring problems. It was used previously in [Modi et al., 2005] and [Modi et al., 2003]. All graphs in the data set are undirected and intended for graph coloring problems involving three colors.
For the benchmarks in this chapter, three graphs with different number of nodes were chosen from the mentioned data set: *adopt10*, with 10 nodes, *adopt20* with 20 nodes and *adopt40* with 40 nodes. All those graphs can be found under *http://teamcore.usc.edu/adopt/problems.tar.gz*. All of the graphs contain cycles. Since the

[1] *http://teamcore.usc.edu/dcop/*

next chapter of this thesis aims to evaluate the scaling properties of the algorithms on greater graphs and on a cluster infrastructure, there is no need for bigger graphs in this chapter.

5.6 Evaluation

In the following sections, the execution procedure and details as well as the results of the benchmarks will be presented.

5.6.1 Conflicts Over Steps

To benchmark the conflicts over steps, the algorithms have to be executed in the synchronous mode of the Signal/Collect framework. All algorithms were run for 50 execution steps, and then stopped automatically. The number of conflicts at each of the 50 steps was recorded and stored. This procedure was repeated 4 times for each algorithm. At the end, the results of the 4 repetitions were averaged. In general, the Max-Sum algorithm is expected to deliver the best performance in this kind of benchmark as implied by [Farinelli et al., 2008].

10 Nodes:
The results of the benchmark executed on graph *adopt10* are shown in figure 5.1.

As one can see in the plot, DSA-A [2] and DSA-B [3] perform in a very similar fashion, with DSA-B doing slightly better. The analyses from [Zhang and Xing, 2002] of different DSA variants have produced similar results. Both DSA variants start with a rather high number of conflicts at the beginning. With increasing number of execution steps, the number of conflicts drops in an almost exponential fashion.

[2] *https://github.com/hafenr/benchmarking-libr-algs-signal-collect*
[3] *https://github.com/hafenr/benchmarking-libr-algs-signal-collect*

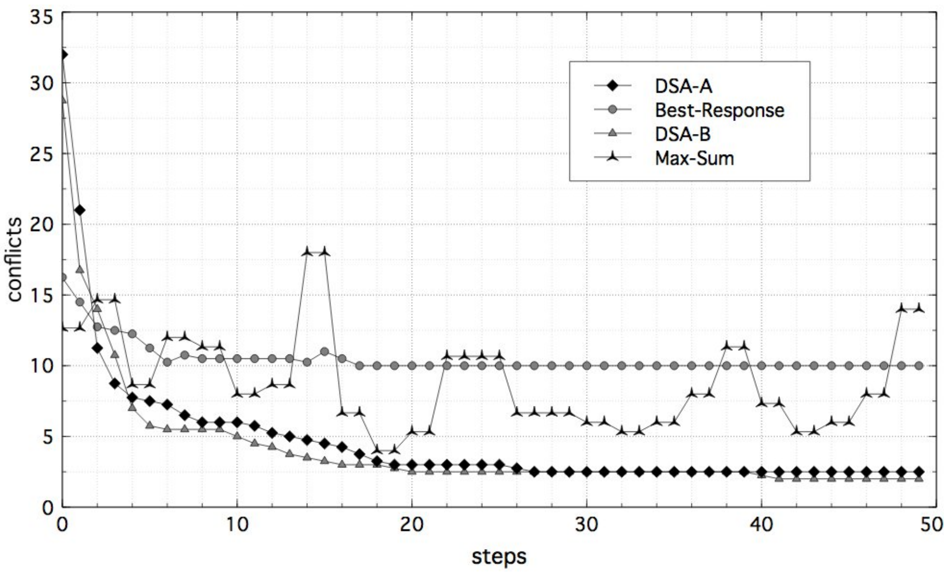


Figure 5.1: Conflicts over execution steps for Max-Sum, DSA-A, DSA-B and Best-Response algorithms, benchmarked on a 10-node ADOPT graph.

Best-Response [4] shows a nearly linear trend after a slight improvement in the first execution steps. Surprisingly, in this benchmark, the Max-Sum algorithm behaves unstable and does not really outperform the other candidates. The expectations for the Max-Sum algorithm to perform better were not yet fulfilled in the first benchmark executed on the smallest of the three graphs. The Max-Sum algorithm starts at a lower conflict level than any of the other candidates, but then instead of lowering the number of conflicts, it goes on by oscillating irregularly between three different conflict levels: 18 , 11 - 14 and 4 - 6 conflicts. It reaches several local minima at steps { 18 , 19} , { 32 , 33 } and { 42 , 43 }. At the right end of the plot, the number of conflicts starts increasing again up to a local maximum of 14. After taking a closer look, the oscillating behavior on the 10-node ADOPT graph is not as surprising. In the conflicts over time benchmark that was performed in [Farinelli et al., 2008], the authors note that the Max-Sum algorithm performs less

[4] *https://github.com/hafenr/benchmarking-libr-algs-signal-collect*

well for small graphs of the ADOPT repository with a low numbers of agents. This fact was also mentioned in section 2.2.3. In other words, [Farinelli et al., 2008] have observed the exact same behavior as in this benchmark. The poor performance observed in this benchmark may come from the fact that cyclic graphs can cause oscillating behavior in the Max-Sum algorithm [Farinelli et al., 2008]. In the case of the 10-node benchmark, cycles may have a greater impact on the behavior of the Max-Sum algorithm, since on a 10-node graph, the probability of messages being passed in a potential loop is higher than for bigger graphs. This issue has been adressed in [Farinelli et al., 2008]. The authors claim that this issue can be fixed by modifying the utility function (c.f. equation 3.6) to consider not only the constraints with the direct neighbors of a node, but also to include the constraints among these neighbors [Farinelli et al., 2008]. The utility function shown in 3.6 would have to be changed to:

$$U_m(x_m) = \gamma_m(x_m) - \sum_{i \in \mathcal{N}(m) \setminus m} \sum_{j \in \mathcal{C}(i,m)} x_m \oplus x_i$$

where:

$$\mathcal{C}(i,m) = \{k \in \mathcal{N}(m) | k > i \wedge (i \in \mathcal{N}(k) \vee k \in \mathcal{N}(i))\}$$

is the set of neighbors of the current vertices' neighbors.

20 Nodes:

The results of the benchmark executed on graph *adopt20* are shown in figure 5.2.

The benchmark on a graph of 20 nodes (c.f. figure 5.2) shows similar trends for DSA-A and DSA-B. Both show a great decrease in the number of conflicts over the first steps. Both DSA variants have almost identical trends towards the end of the steps scale. The relative difference between the conflicts for the Best-Response

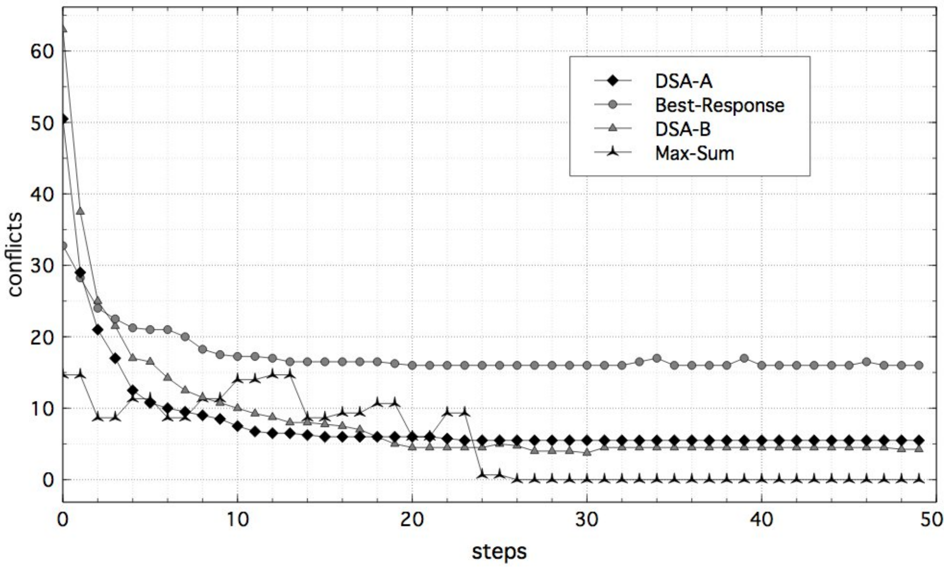


Figure 5.2: Conflicts over execution steps for Max-Sum, DSA-A, DSA-B and Best-Response algorithms, benchmarked on a 20-node ADOPT graph.

algorithm and the DSA variants are not as big as in the previous benchmark. Nevertheless, Best-Response again shows generally the highest number of conflicts. Despite the mentioned similarities to the first benchmark, the second one exposes a completely different behavior of the Max-Sum algorithm. In this case, the conflict trend of the Max-Sum algorithm confirms the results of a similar empirical analysis in [Farinelli et al., 2008] where the Max-Sum algorithm shows a better quality in terms of number of conflicts compared to DSA-A, DSA-B and Best-Response. Figure 5.2 shows that the Max-Sum algorithm starts with a very low number of conflicts in the first steps, followed by a range of ~ 20 steps where the number of conflicts oscillates only very slightly around 10 conflicts. From the 23-th step on, Max-Sum outperforms all other candidates in the benchmark. The number of conflicts drops heavily until down to one conflict, and two steps later it arrives at zero, which means the Max-Sum algorithm has solved the graph coloring problem optimally. DSA-A remains on 5 and DSA-B on 4 for the rest of the execution steps.

40 Nodes:

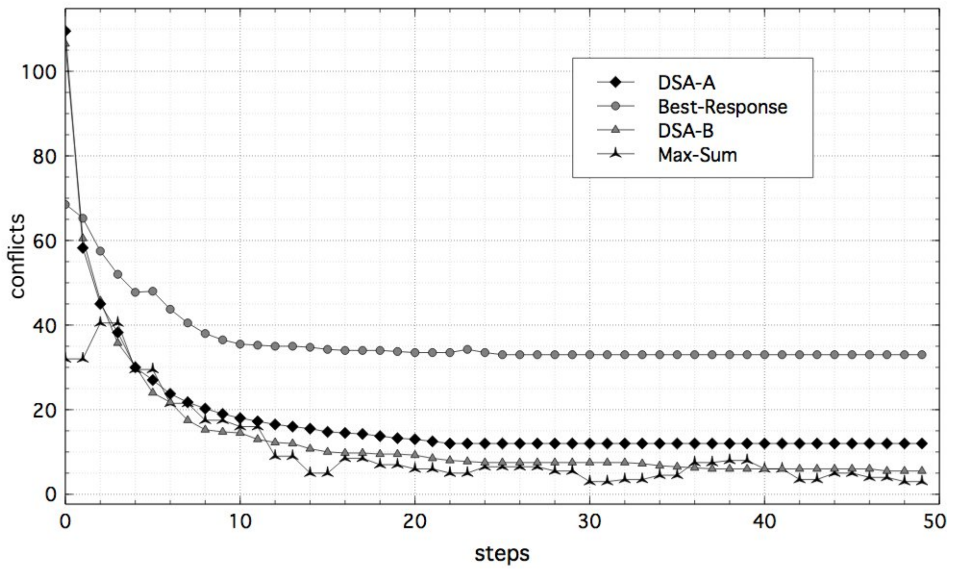


Figure 5.3: Conflicts over execution steps for Max-Sum, DSA-A, DSA-B and Best-Response algorithms, benchmarked on a 40-node ADOPT graph.

The benchmark on 40 nodes, depicted in figure 5.3, shows generally a similar figure as the benchmark over 20 nodes (c.f. 5.2). Again, DSA-A and DSA-B have a nearly exponential decrease in number of conflicts with DSA-B performing slightly better as the steps increase. Both seem to ”converge” to a stable state between 5 conflicts (DSA-B) and 12 conflicts (DSA-A). Also, the constellation of the Best-Response algorithm has not changed; it is again the weakest candidate in terms of number of conflicts. Although the difference in the conflict trends between the other three candidates and the Max-Sum algorithm is not as big as in the benchmark before, the Max-Sum algorithm again outperforms its contestants. Max-Sum shows multiple slight deviations in the lower conflicts levels towards the end of the steps scale, but still its number of conflicts is mostly lower than the conflicts of DSA-A, DSA-B and

Best-Response. Hence, once again, the Max-Sum has achieved to confirm former results of [Farinelli et al., 2008].
In general, one can say that the Max-Sum algorithm outperforms DSA-A, DSA-B and Best-Response in terms of solution quality. There is a slight outlier in the banchmark of 10-nodes, which was discussed and adressed above. Aside from the mentioned case, the acquired results in the presented conflict benchmarks confirm the expectation that the Max-Sum algorithm achieves slightly better solution quality.

5.6.2 Conflicts Over Time

In the previous benchmark, the quality of solution of the Max-Sum algorithm versus the candidate algorithms was analyzed. This thesis makes an additional contribution as it looks deeper into the quality properties (e.g. conflicts) of the Max-Sum algorithm. Instead of comparing Max-Sum to other candidates, this benchmark executes the Max-Sum algorithm synchronously and asynchronously, and then compares the resulting number of conflicts over a certain execution time t. Again, the used graph datasets were *adopt10* , *adopt20* and *adopt40* (c.f. section 5.5). The benchmarks were executed for 10'000 milliseconds and the number of conflicts were recorded at time t equal to 500, 1'000, 1'500, 2'500, 5'000, 7'500 and 10'000 milliseconds. For each graph, the benchmark was executed 4 times, and then the average of the resulting conflicts was computed. In the following, the results for each t are presented; for the sake of readability, the synchronous Max-Sum implementation will be denoted *SYNC* while the asynchronous one will be denoted *ASYNC*.

On 10 nodes:

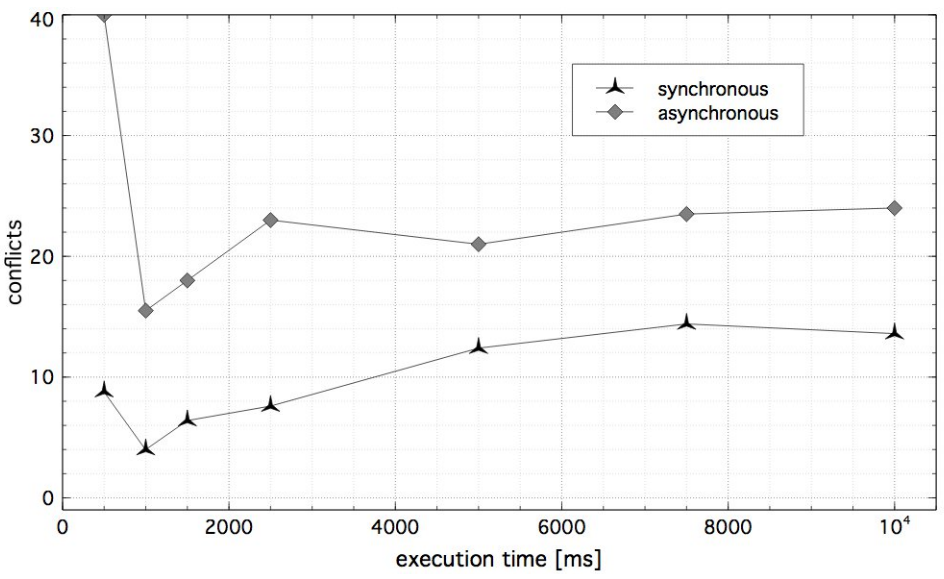


Figure 5.4: Conflicts for synchronous and asynchronous Max-Sum on a 10 node ADOPT graph.

Figure 5.4 shows the trend for the number of conflicts in the benchmark on adopt10. SYNC has a much lower initial conflict level than ASYNC. After a rather high number of conflicts in the beginning, both candidates show an abrupt drop. The trendline for SYNC is generally lower than the trendline of ASYNC. The difference between these two is very steady and remains around 10 conflicts for the whole execution time. After the initial decrease down to a local minimum, the values for both candidates show an increase with very similar slopes. This behavior is not what one would expect, since a longer execution time gives the algorithms greater chance of solving the problem and hence lowering the number of conflicts. The unnatural behavior observed in figure 5.4 is clearly based on the same cause as the poor results of Max-Sum in the previous benchmark on the 10 node adopt graph (c.f. 5.6.1). The poor results are caused by the loop structure in the smaller adopt graphs. A more detailed explanation and a modification of the algorithm that sorts out this problem was presented in section 5.6.1.

On 20 nodes:

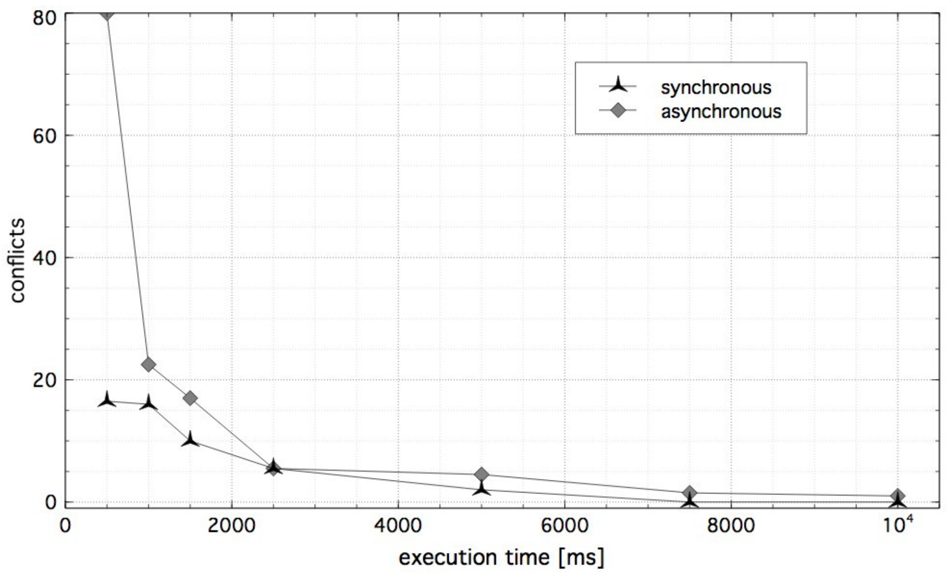


Figure 5.5: Conflicts for synchronous and asynchronous Max-Sum on a 20 node ADOPT graph.

The benchmark results for the adopt20 graph reveal a totally different situation, as depicted in figure 5.5. Again, ASYNC has a much higher number of conflicts at the first measuring interval, but then a steep decrease is observed between the intervals 500 ms and 1000 ms. While SYNC's number of conflicts was much lower at the beginning, both candidates arrive at the same number of conflicts at $t = 2500$ ms. From that point on, the trends for both candidates are very similar; they both continue to decrease in number of conflicts in a very slight fashion. SYNC has a slightly lower number of conflicts than ASYNC.

On 40 nodes:

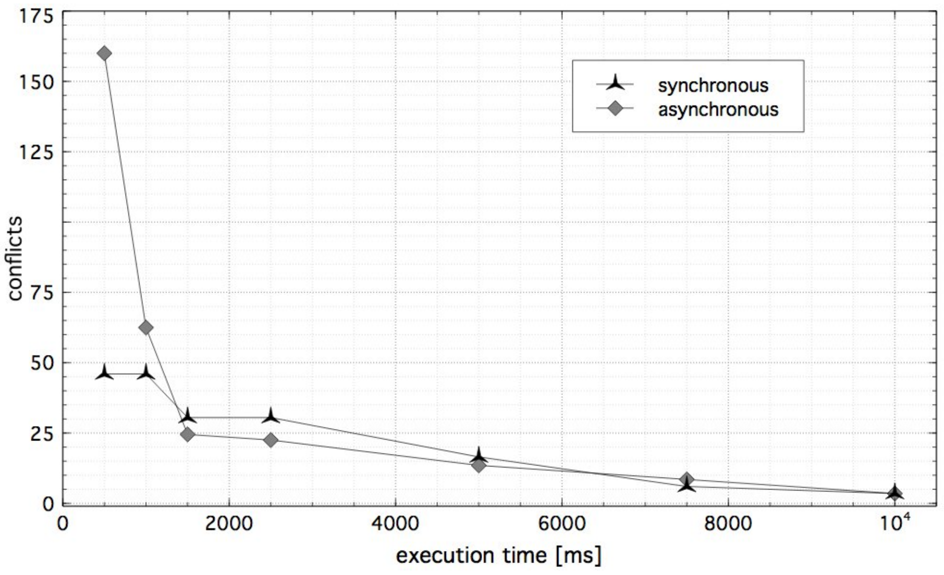


Figure 5.6: Conflicts for synchronous and asynchronous Max-Sum on a 40 node ADOPT graph.

The benchmark on 40 nodes (c.f. figure 5.6) shows a very similar result as the one benchmark on 20 nodes: After a high discrepancy in the beginning, both candidates' trend lines draw near each other and continue with a similar development towards the end of the execution interval. This time, there is no remarkable difference between the candidates towards the end. All said, this benchmark revealed three things.

First, the poor performance of the Max-Sum algorithm on small cyclic graphs, that was already mentioned in the conflicts over steps benchmark and in [Farinelli et al., 2008], was reproduced and confirmed. SYNC as well as ASYNC show similar results in the case of said graph (adopt10) and hence, the poor performance in that region is not related to the execution mode.

Second, a clear observation from the above figures is the fact that in the first time intervals after the execution starts, SYNC considerably lowers its number of conflicts and shows a much lower level of conflicts than ASYNC in those first intervals.

In the asynchronous execution of the algorithm, the vertices do not message in a specific order. There is no guarantee about the order of the nodes that are allowed to compute messages or compute new states based on received messages. Hence, in the beginning, a lot of the messages may contain no useful information yet because needed incoming messages are not available yet. In contrast, SYNC operates in clearly defined and ordered steps. Hence, all nodes will receive some useful information in the messages in the initial phase of the execution. This may be the reason why SYNC achieves better results at the beginning.
Third, there is no significant difference in the overall performance between the two candidates. It is not possible to make a clear statement wether SYNC or ASYNC delivers better results in terms of solution quality based on this benchmark. Given the results obtained here, asynchronous and synchronous executions of Max-Sum seem to perform almost equally well. Especially as the execution time increases, the number of conflicts for both variants settle down on nearly the same level of conflicts.
To get a connection between the execution cycles used in section 5.6.1 and the execution time used as a metric for the abscissa in this benchmark, the respective number of steps for the execution intervals used before has been recorded on the three different graphs. Note that this was only possible on the synchronous implementation, since for the asynchronous case there is no definition for a discrete execution cycle or step.

The relation between the number of steps and the execution time ist fairly linear for all three graphs. A linear function fitting of the form $s(t) = a + b * t$ for the data points results in:

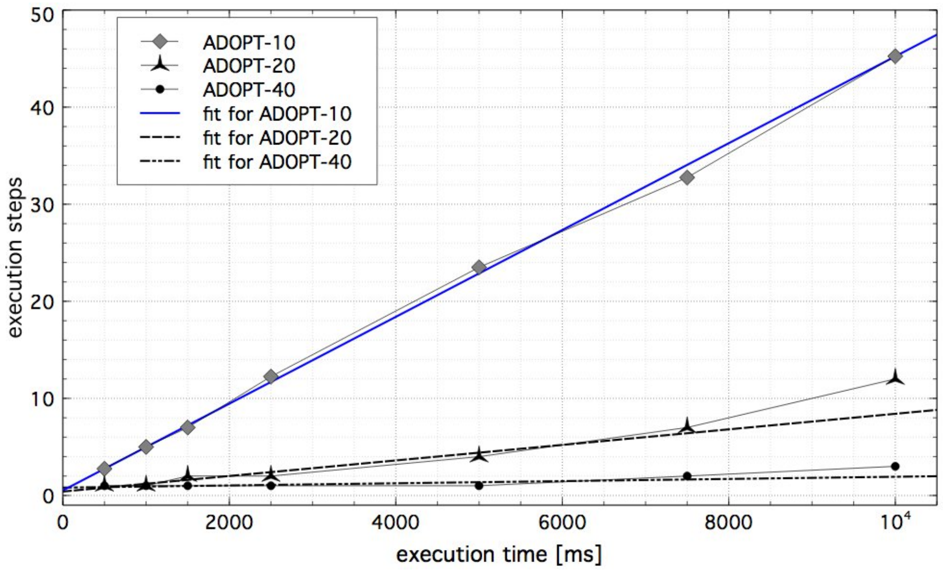


Figure 5.7: Relation between the execution time and the execution cycles for synchronous Max-Sum.

$$s_{ADOPT-10}(t) = 0.512612 + 0.00447166 * t$$

$$s_{ADOPT-20}(t) = 0.403824 + 0.000800772 * t$$

$$s_{ADOPT-40}(t) = 0.806587 + 0.000111903 * t$$

As one can see, the size of the slope of the linear functions above decreases heavily when the graph size increases. Following this, the relation function $s_{ADOPT-10}(t)$ has a slope that is greater than the slope of $s_{ADOPT-20}(t)$ by a factor of 5.58. The factor between the slopes of $s_{ADOPT-20}(t)$ and $s_{ADOPT-40}(t)$ is around 7.15.

5.6.3 Execution Steps To Convergence

In the previous sections the solution quality of the algorithms was benchmarked. This section aims to benchmark the performance of the algorithms in terms of execution steps until convergence is reached. The definition of convergence and the mechanisms to detect it were discussed in 5.4.2. The candidate algorithms were run in synchronous mode until convergence was detected. After the execution the number of execution steps was recorded. This was repeated 4 times on each of the three ADOPT graphs presented in section 5.5 and the averages were computed.

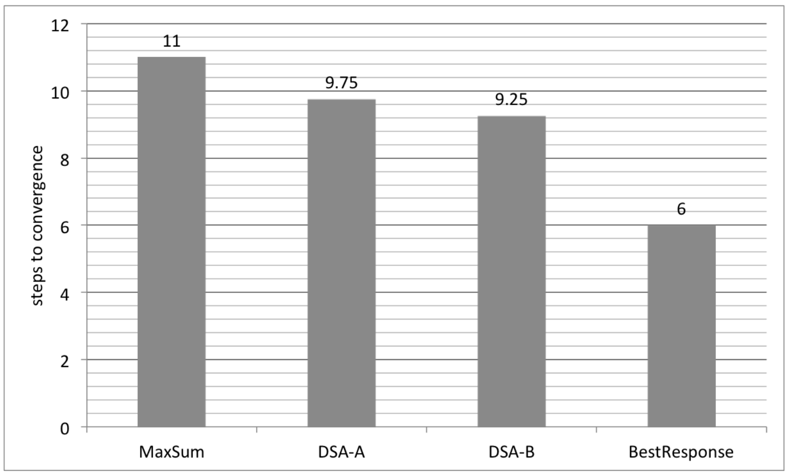


Figure 5.8: Steps to convergence for Max-Sum, DSA-A, DSA-B and Best-Response algorithms on a 10-node ADOPT graph.

The results on the graphs with 10 and 20 nodes are very similar and differ only in the number of steps for the Best-Response algorithm. Generally, the differences between the algorithms are very small. The Max-Sum algorithm has the highest number of steps until convergence for the first two graphs, while DSA-A and DSA-B achieve results that are very close to each other. Best-Response shows the best performance of all candidates.

The benchmark on the 40 node ADOPT graph shows also similar results (c.f.

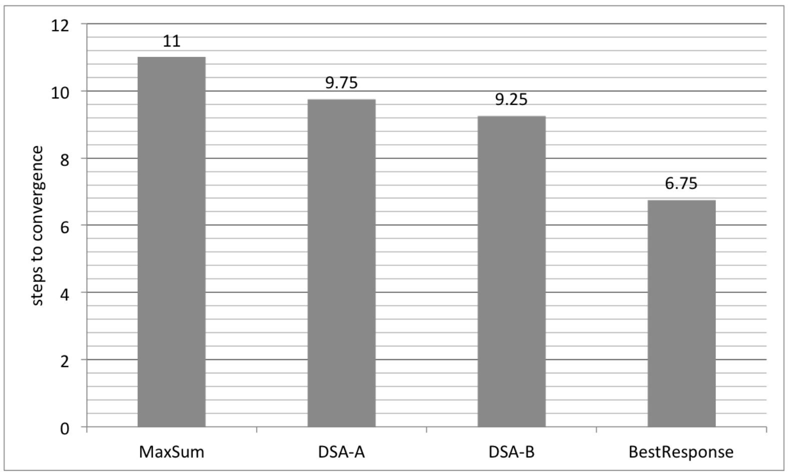


Figure 5.9: Steps to convergence for Max-Sum, DSA-A, DSA-B and Best-Response algorithms on a 20-node ADOPT graph.

figure 5.10). In this case, the differences between the algorithms are slightly larger than the ones of the benchmarks on 10 and 20 nodes. Again, Max-Sum has the weakest performance achieving 13 steps until convergence. In contrast to the previous two benchmarks, DSA-A performs slightly better than DSA-B. Best-Response again delivers the best results in this benchmark. This is mainly because Best-Response uses the lowest amount of communication and computation among the candidates. Hence, the theoretical fact that Best-Response is a lower bound on the performance of any local DCOP algorithm [Farinelli et al., 2008] was confirmed by the benchmark.

5.6.4 Execution Time To Convergence

As before, the synchronous and asynchronous versions of Max-Sum were compared. This time, the focus lies on the performance rather than on the solution quality, therefore the benchmark recorded the convergence time in milliseconds. Both versions were run on the presented datasets four times using the same convergence

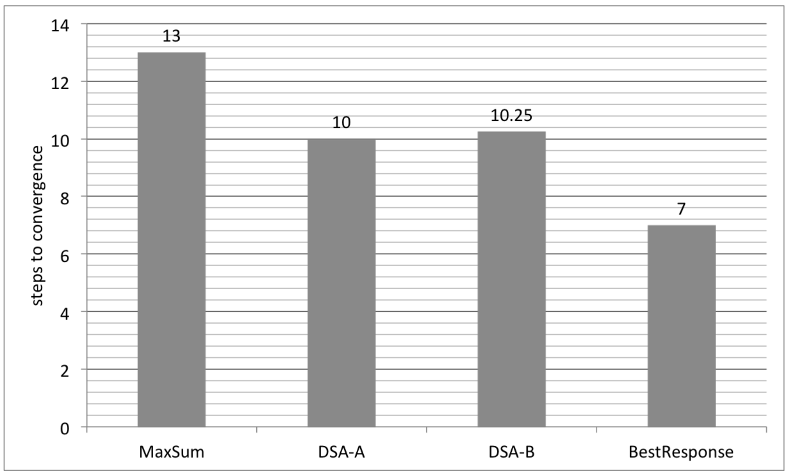


Figure 5.10: Steps to convergence for Max-Sum, DSA-A, DSA-B and Best-Response algorithms on a 40-node ADOPT graph.

detection mechanism as in section 5.6.3. The averaged results are presented below. In the following, SYNC denotes the synchronous version and ASYNC stands for the asynchronous version.

The figures 5.11, 5.12 and 5.13 illustrate that ASYNC outperforms SYNC on all data sets by a large magnitude. On adopt10, ASYNC is faster than SYNC by a factor equal to 4.13. On adopt20, the factor drops down to 2.43, while on adopt40, it is equal to 3.15. Figure 5.14 compares the trend lines of the convergence times versus the size of the graphs. The trend line of ASYNC appears to be almost linear with a steady slope, while SYNC shows a clear increase of the slope when the graph size is greater than 20 nodes.

In general, ASYNC has proved to perform much better in the computation-speed benchmarks. This kind of result was expected and the reason for it lies in the fact that SYNC loses lots of computation time in the blocking phases while waiting for all vertices to have finished their computation in order to continue the algorithm. The need for global synchronization inherently slows down algorithms such as SYNC. In

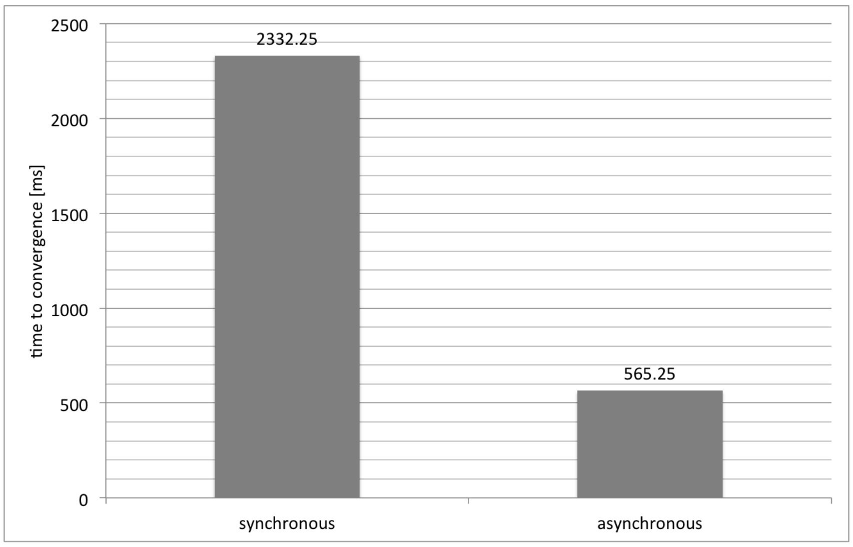


Figure 5.11: Time to convergence for asynchronous and synchronous Max-Sum on a ADOPT graph with 10 nodes.

contrast to this, ASYNC has no means of ordering between the execution phases and does not wait or block during the execution. Hence, ASYNC delivered superior results.

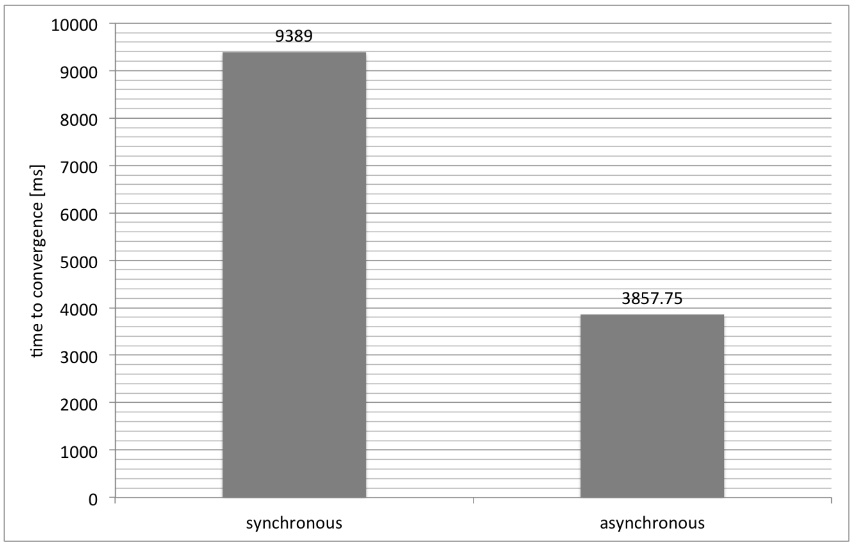


Figure 5.12: Time to convergence for asynchronous and synchronous Max-Sum on a ADOPT graph with 20 nodes.

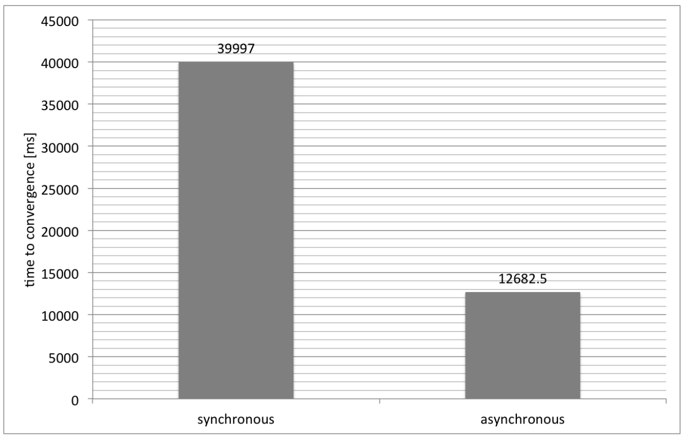


Figure 5.13: Time to convergence for asynchronous and synchronous Max-Sum on a ADOPT graph with 40 nodes.

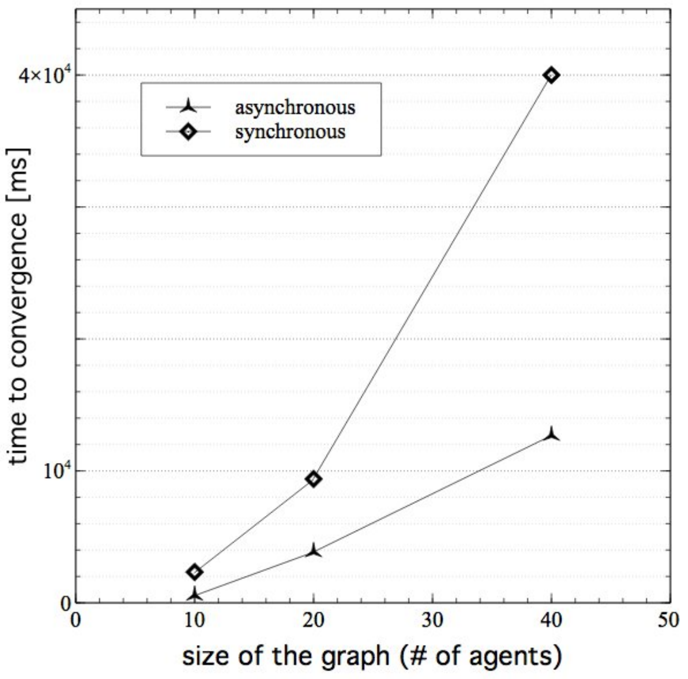


Figure 5.14: Trend lines for the convergence times of synchronous and asynchronous Max-Sum depending on the graph size

6

Scalability

The last contribution attempt of this thesis, besides the implementation and the benchmarking of the Max-Sum algorithm, is to test the scalability of the Max-Sum algorithm on proper machine infrastructure. This chapter will introduce the infrastructure on which the scalability benchmarks were run, the data sets that were used and describe the benchmark criteria and measures.

6.1 Infrastructure

The scalability benchmarks were run on *Kraken*, a cluster of 12 machines. Each machine has two twelve-core AMD Opteron™ 6174 processors and 66 GB RAM. Not all of the 12 machines on the cluster were used on all benchmarks. The number of machines used is indicated in the sections describing the benchmark itself.

6.2 Considerations on Distributability

This section will cover the aspects of the implementation that had to be modified or improved in order for the implementation to run distributed on multiple machines. As mentioned in the previous chapter, the colors of the vertices were stored in a global static object for the `AggregationOperation` to compute the number of conflicts over the entire graph. Obviously, this approach is unsuitable for a distributed use of the software. Hence, the central color and conflict management through the static object needs to be distributed and delegated to the vertices. The basic idea is to let every `VariableVertex` instance provide a function named `getNumberOfConflicts` that returns the number of conflicts at the corresponding vertex (note that `FunctionVertex` instances are not relevant to the computation of the number of conflicts since a `FunctionVertex` does not have any state or color). But how can a `VariableVertex` know the number of conflicts it is involved in? This is tricky, because in Signal/Collect, a vertex has no access to object references of its neighbors, but only access to the ids. Hence, vertices can not access the color of their neighbors in order to compute the number of conflicts they are involved in. This issue was solved by adding a method `tellNeighborsAboutColor()` to `VariableVertex` which sends special messages to all neighbors containing the vertex's current color. On the receiving side, the method `deliverSignal` has to be overriden since `VariableVertex` instances now receive two types of messages: The normal `MaxSumMessage` and the newly introduced color message. The overriden method `deliverSignal` decides wether a received signal is a `MaxSumMessage` or a color. In the latter case, the received color is put into a map data structure with the id of its sending variable as a key. Based on this map, a vertex can compute its number of conflicts when `getNumberOfConflicts` is called on it. Then the `AggregationOperation` simply extracts the number of conflicts from each vertex by calling `getNumberOfConflicts` and aggregates them to the total sum.

Just like the conflicts, the initial preferences and neighborhood structures were stored to a global static object too. This was overcome by injecting the necessary information during the loading and constructing of the Signal/Collect graph directly into the edge instances, where the information is needed to compute the messages.

6.3 Data Sets

The data sets used to evaluate the Max-Sum algorithm on the distributed infrastructure consist of synthetically generated graphs of different sizes. The graph generation works as follows: First a fully connected graph of a defined graph size is constructed and then edges are removed randomly from this graph until a predefined average degree is reached. The graphs used in the following all have an average degree of $\sim$ 3 and have sizes of 100, 200 and 300 nodes.

6.4 Evaluation

The following sections present the results to the benchmarks run on the multiple machine cluster. First, an evaluation of the conflicts over time is shown. The second benchmark analyzes the influence of the number of machines on the solution quality while the last benchmark evaluates the impact of the average vertex degree on the solution quality.

6.4.1 Conflicts Over Time

As in the previous chapter, the synchronous and asynchronous execution of the Max-Sum algorithm were compared. Again, SYNC is used as a an abbreviation for synchronous Max-Sum and ASYNC is the abbreviation for asynchronous Max-Sum. In the following, the results are presented. The intervals at which the number of conflicts was measured were 250, 300, 350, 400, 450, 500, 625, 750, 875, 1'000, 1'500, 2'500, 5'000, 7'500, and 10'000 milliseconds after execution start. The experiments were run distributed on 4 machines of the cluster.

100 agents:

In the first intervals, the difference of the number of conflicts between SYNC and ASYNC is very high. But in contrast to the analysis in section **??**, where SYNC's number of conflicts was lower at the first few measuring intervals, the results in figure 6.1 show a contrary situation. Asynchronous Max-Sum delivers significantly better results, especially at the first intervals. As the execution time increases, the difference between SYNC and ASYNC diminishes.

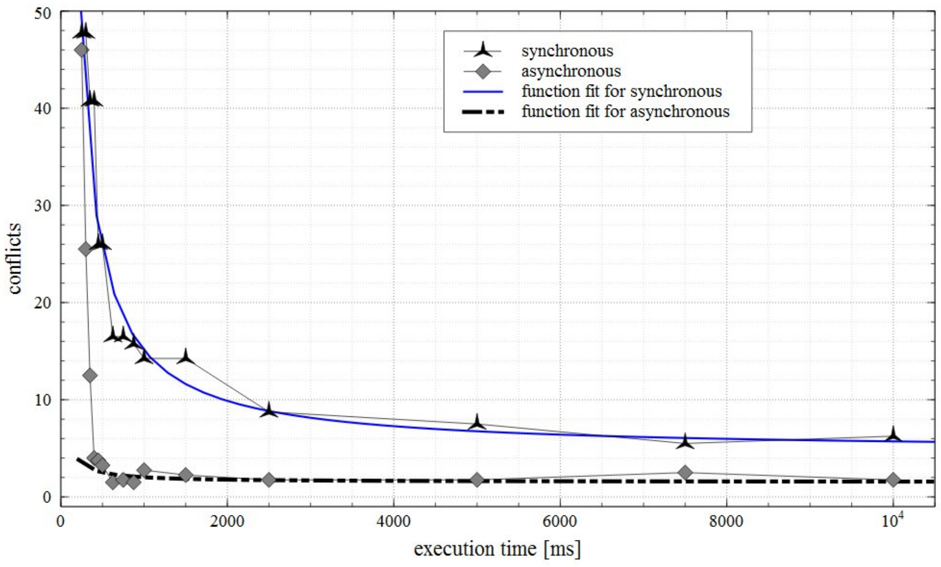


Figure 6.1: Conflicts over time for synchronous and asynchronous Max-Sum on a 100 node graph, distributed on 4 machines.

Both SYNC and ASYNC show a trend line that follows the shape of a power function of the form $f(t) = \frac{q}{t} + m$ where $q \in (0, \infty)$ and $m \in [0, \infty)$. The fitted function $f_{SYNC}(t)$ for SYNC is:

$$f_{SYNC}(t) = \frac{10407}{t} + 4.67$$

And the function fit for ASYNC equals:

$$f_{ASYNC}(t) = \frac{489.59}{t} + 1.53$$

Hence, the difference in the number of conflicts between SYNC and ASYNC when execution time goes on is: $4.67 - 1.53 = 3.14$.

200 agents:

The results of the evaluation on 200 nodes show a similar picture as for 100 nodes (c.f. figure 6.2). Again, ASYNC outperforms SYNC, but this time the difference is a little smaller than it was for 100 nodes.

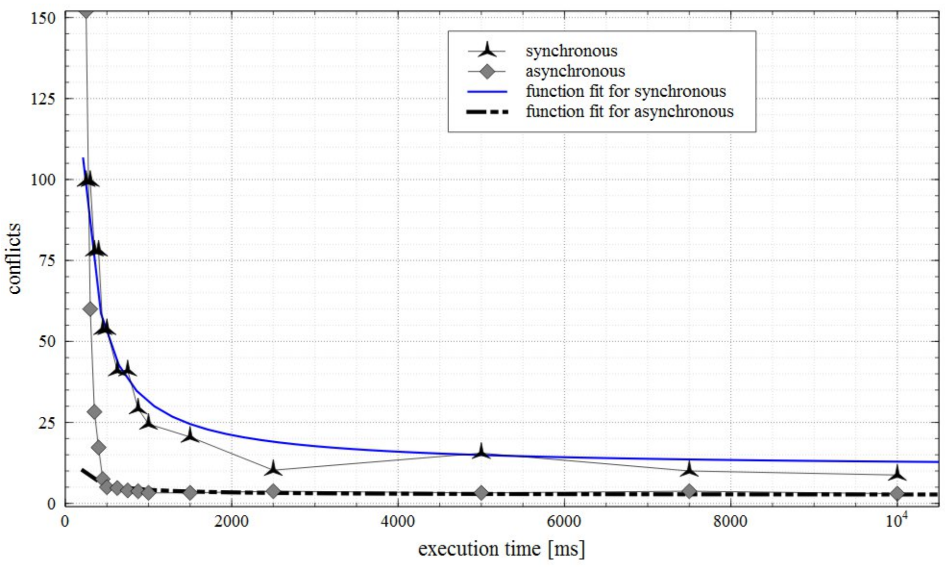


Figure 6.2: Conflicts over time for synchronous and asynchronous Max-Sum on a 200 node graph, distributed on 4 machines.

The measured data follow trend lines of the form:

$$f_{SYNC}(t) = \frac{22685.4}{t} + 5.71$$

and

$$f_{ASYNC}(t) = \frac{1608.01}{t} + 2.58$$

Hence, the final difference in the number of conflicts is $5.71 - 2.58 = 3.13$. In comparison with the evaluation on 100 nodes, the final difference has not changed significantly.

300 agents:

As shown in figure 6.3, SYNC has a much lower number of conflicts than ASYNC at the first measuring interval (t = 250 ms). But then again, ASYNC shows clearly lower conflict rates.

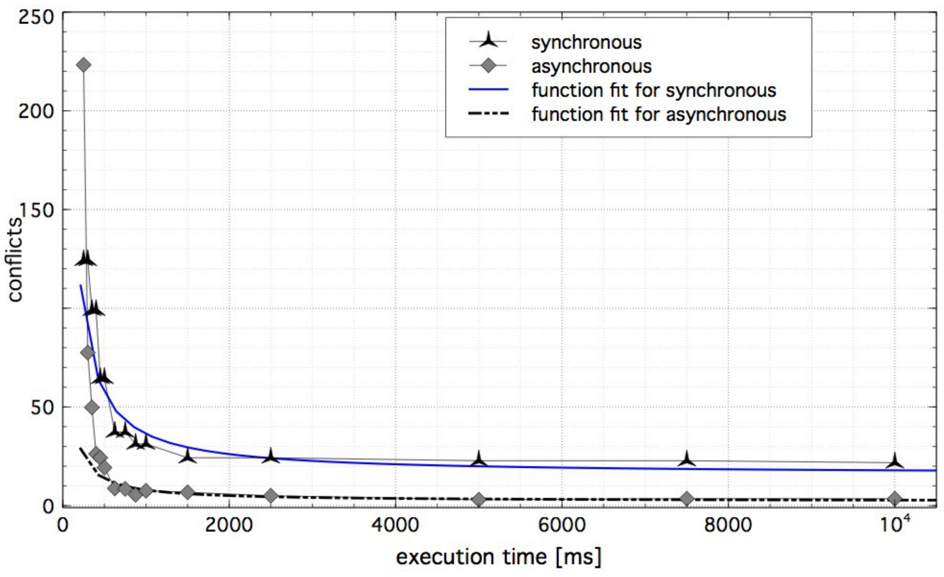


Figure 6.3: Conflicts over time for synchronous and asynchronous Max-Sum on a 300 node graph, distributed on 4 machines.

Nevertheless, the difference between ASYNC and SYNC has again decreased. The fitted function $f_{SYNC}(t)$ for SYNC is:

$$f_{SYNC}(t) = \frac{20498.75}{t} + 15.83$$

And the function fit for ASYNC equals:

$$f_{ASYNC}(t) = \frac{5678.70}{t} + 2.23$$

The prediction for the difference of the number of conflicts is $15.83 - 2.23 = 13.6$. This is a significant increase in comparison to the two values of the previous evaluations of the same kind on smaller graphs.

Generally, one can deduce that ASYNC starts with a very high number of conflicts but as the execution goes on, it delivers a significantly lower number of conflicts. This is a contrast to section 5.6.2, where ASYNC and SYNC showed very similar results in the benchmarks over number of conflicts.

6.4.2 Influence of the Number of Machines

Since the main difference between the evaluations in the last chapter and this one is the fact that the benchmarks are run distributed on multiple machines, it is interesting to analyse if the number of machines has influence on the number of conflicts over time. The following sections present the conflicts over time for SYNC and ASYNC, where the evaluations were run on 1, 2, and 4 machines of the cluster.

Asynchronous

100 agents:

As depicted in 6.4, the evaluation on a graph with 100 nodes does not show any significant differences depending on the number of machines used in the distribution of the algorithm. The run on 4 machines has the lowest number of conflicts at the first measuring interval, while the run o 2 machines lies in the middle and the run on one machine starts with the highest number of conflicts. This is as expected. But surprisingly, as execution time increases, the differences disappear and the trend lines for 4, 2, and 1 machine seem to move towards the same level.

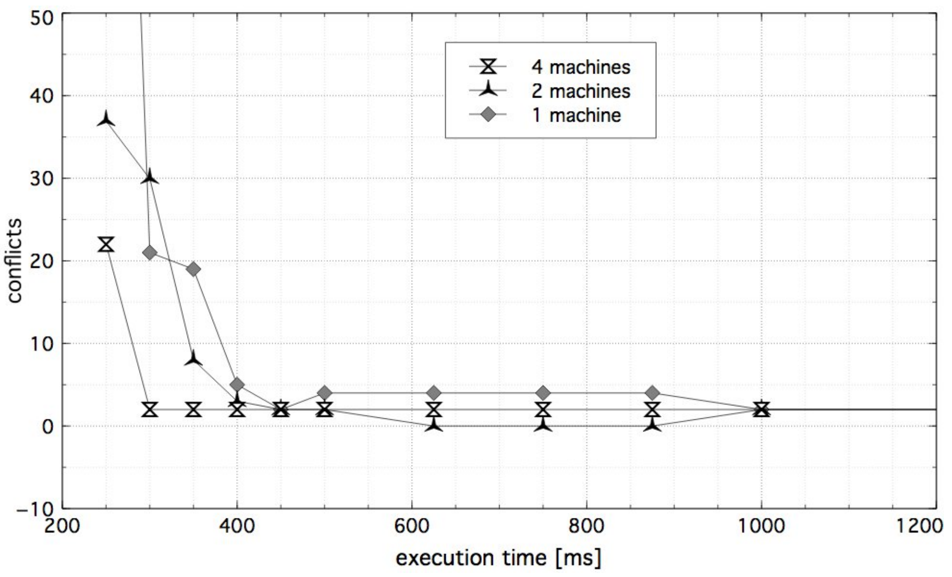


Figure 6.4: Conflicts over time for asynchronous Max-Sum on a 100 node graph, depending on the number of distributed machines.

This may be caused by the fact that, when using multiple machines on a rather small graph with 100 nodes, there is a trade-off between the increased computation speed through the distribution and the slowdown through the network communication required. In other words, the time gained through the speedup of the distribution is lost again because of the network communication between the machines which is rather slow in comparison to the computation cycles. Hence the number of the machines that are incorporated in the benchmark does not have a significant influence on the number of conflicts on small graphs.

200 agents:

Figure 6.5 representing the results for a 200 node graph shows a slightly different picture. The discrepancy between the values for one machine and the values for 2, or respectively 4 machines are a little higher than before. Again, after rather high differences for the values measured at the first intervals, the differences decrease as the execution time increases. Towards the end, all three runs arrive at almost the same level of conflicts.

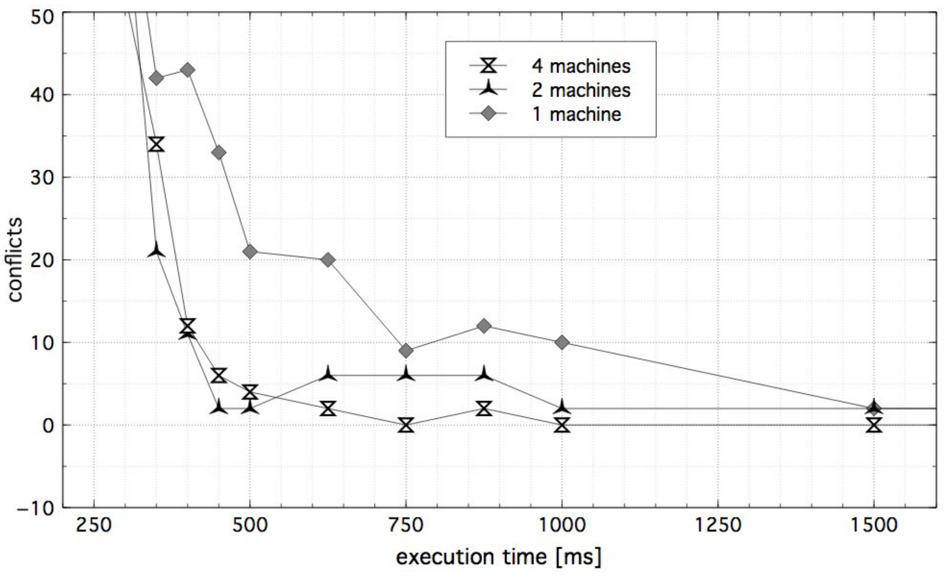


Figure 6.5: Conflicts over time for asynchronous Max-Sum on a 200 node graph, depending on the number of distributed machines.

Nevertheless, in figure 6.5, the expected differences between the runs on different numbers of machines are more clearly observable.

300 agents:

The results on the 300 node graph (c.f. figure 6.6) show the most clear trends among the three presented results for 100, 200, and 300 nodes.

Throughout the execution time, the run on 4 machines shows the lowest number of conflicts. In the first half of the execution, the run on 2 machines behaves as expected. It is waker than the run on 4 machines, but outperforms the run on 1 machine. But as the execution time increases, the 2-machine run shows a slight

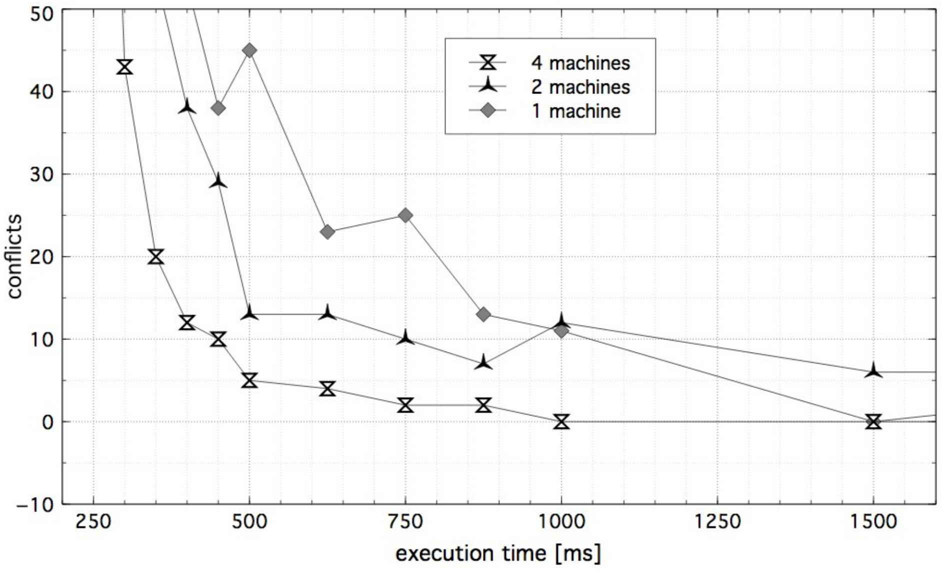


Figure 6.6: Conflicts over time for asynchronous Max-Sum on a 300 node graph, depending on the number of distributed machines.

deviation, which leads to almost the same conflict level on one and on two machines. The clearer differences in this analysis are mainly caused by the size of the graph. As the graph size increases, the advantage in computation speed that a higher number of machines provides, compensates and overtakes the disadvantage that it has (network communication cost).

Synchronous

The same procedure was also performed for the synchronous mode of the Max-Sum algorithm.

100 agents: Figure 6.7 shows a very odd behavior of the results achieved by 2 and 4 machines on 100 nodes. The number of conflicts for the execution on 4 machines oscillate slightly at the first intervals. After $t = 1000$ ms, it starts linearly decreasing its number of conflicts until it has the lowest value of the three runs at the end of the evaluation interval. The execution on 2 machines shows a similar oscillating behavior at the first intervals as the run on 4 machines. At $t = 1000$ ms, it also starts delivering lower number of conflicts, which leads to a trend-line that is almost identical to the one for 1 machine.

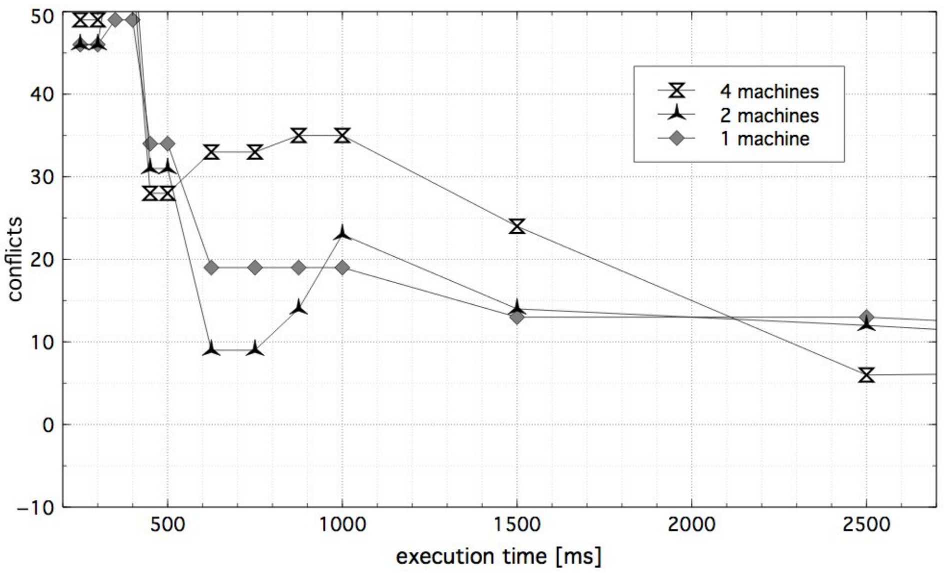


Figure 6.7: Conflicts over time for synchronous Max-Sum on a 100 node graph, depending on the number of distributed machines.

As in the asynchronous case before, on 100 nodes, there is no clear influence of the number of machines used in the number of conflicts produced.

200 agents: The results for the synchronous execution of Max-Sum on different numbers of machines are depicted in figure 6.8. There is no odd behavior like it was observed in the results for 100 nodes.

It is remarkable that despite the graph size has increased, the differences between the results on different numbers of machines have decreased. This is somewhat counterintuitive. Again, no influence of the number of machines can be deduced from these results.

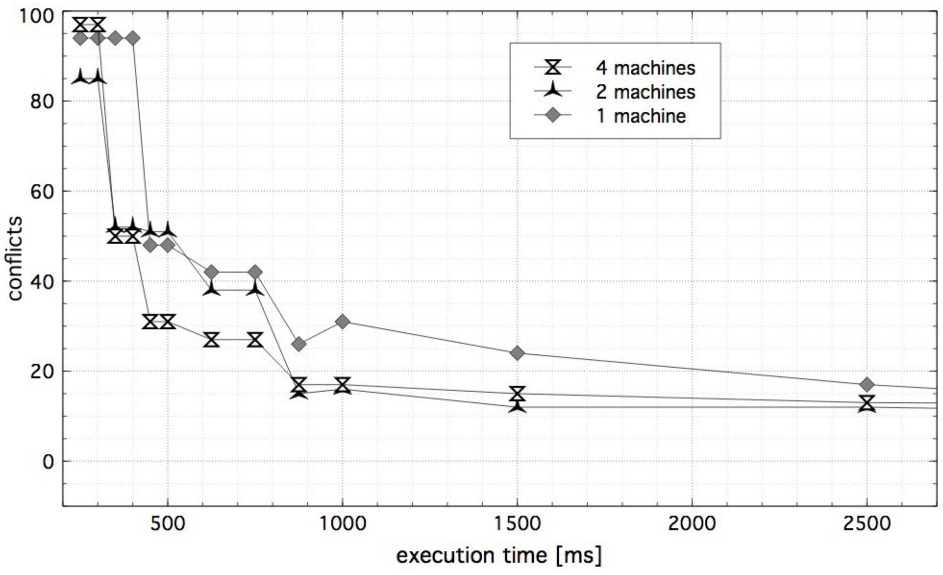


Figure 6.8: Conflicts over time for synchronous Max-Sum on a 200 node graph, depending on the number of distributed machines.

300 agents:

When executed on a 300 node graph, the results (c.f. figure 6.9) do not show significant changes in comparison to the results for 200 nodes.

The only difference one can observe, is that the execution on 4 machines achieves the lowest number of conflicts, while the runs on 2 machines and on one machine remain around the same level.

6.4.3 Influence of the Average Vertex Degree

As illustrated in the implementation chapter, the computation of the messages for the Max-Sum algorithm has a worts case complexity of $\mathcal{O}(|C|^{|V|})$. This means, the computation complexity will depend on the number of colors in the graph coloring problem and on the average degree of the vertices. To support this hypothesis, Max-Sum was run on synthetical graphs generated by the method described in section 6.3. All the following evaluations were performed with a graph coloring problem

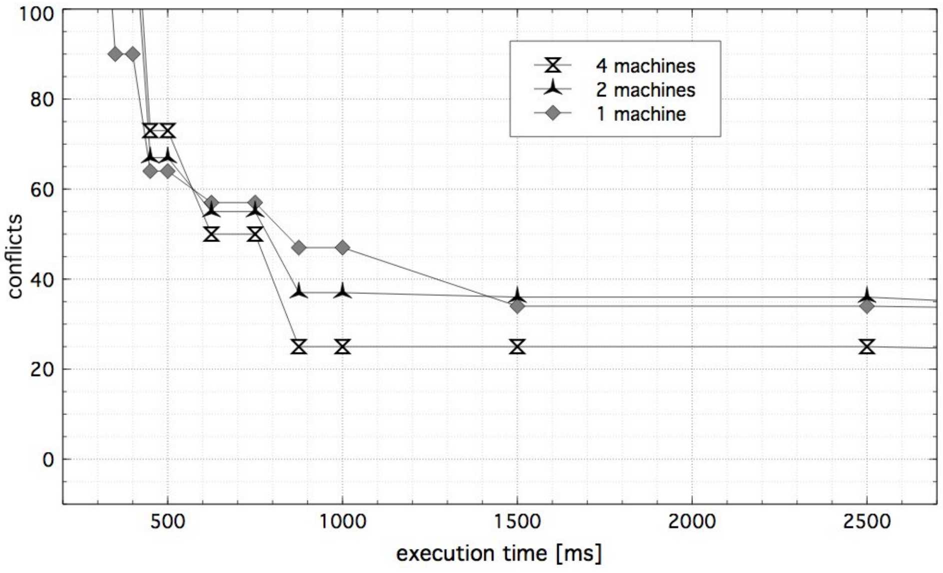


Figure 6.9: Conflicts over time for synchronous Max-Sum on a 300 node graph, depending on the number of distributed machines.

involving 3 colors.

100 agents: Three different versions of the 100 node graph were generated: One with an average degree of $deg = 2$, one with $deg = 3$, and one with $deg = 4$. On each of the graphs, an evaluation of the conflicts over time was performed. The results are presented in figure 6.10.

At the first intervals (250 ms to 450 ms) the behavior is as predicted by the theoretical complexity analysis: The results on the graphs with higher degrees deliver higher numbers of conflicts. As the execution goes on, the differences become smaller. At the end, the values for all three graphs are on almost the same level.

200 agents:

The results for 200 nodes (c.f. figure 6.11) show almost the same behavior as the results in figure 6.10. Again, one can see a weak influence of the degree on

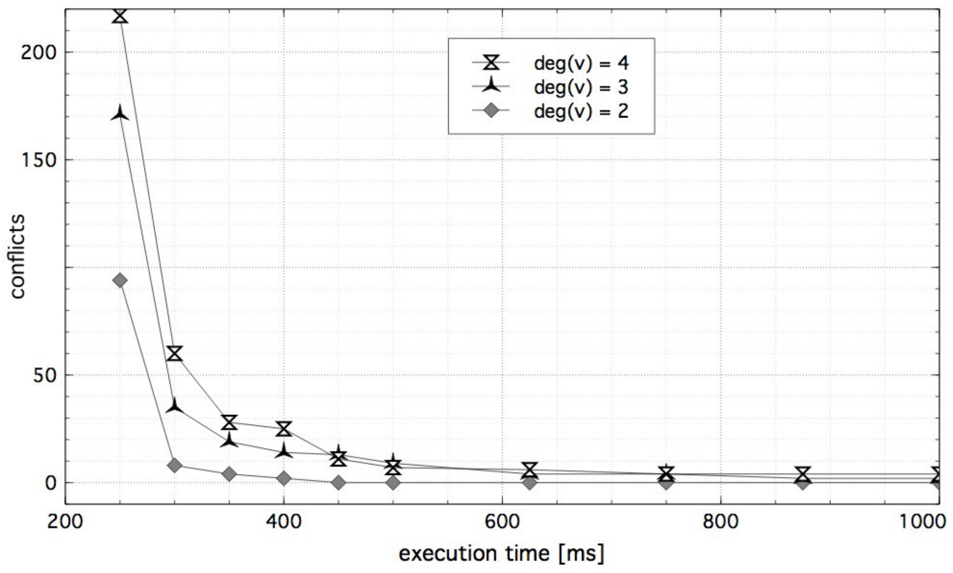


Figure 6.10: Conflicts over time for asynchronous Max-Sum on 100 node graphs with different vertex degrees.

the number of conflicts, but since the complexity analysis found an exponential complexity in the worst case, one would expect a greater impact of the degree on the results. After taking a closer look, one can see that the computed theoretical complexity holds only for the messages $R_{m\to n}(x_n)$ from function vertices to variable vertices. For the other type of messages, $Q_{n\to m}(x_n)$ from variable vertices to function vertices, the complexity is is linear in terms of vertex degree: $\mathcal{O}(|V|)$. Every second message computation during the execution of the Max-Sum algorithm will be of the type $Q_{n\to m}(x_n)$. Hence, the high computation complexity of $R_{m\to n}(x_n)$ will have impact only on 50 percent of the messages being computed. This is why the overall influence of the degree of the graphs is not as big as expected.

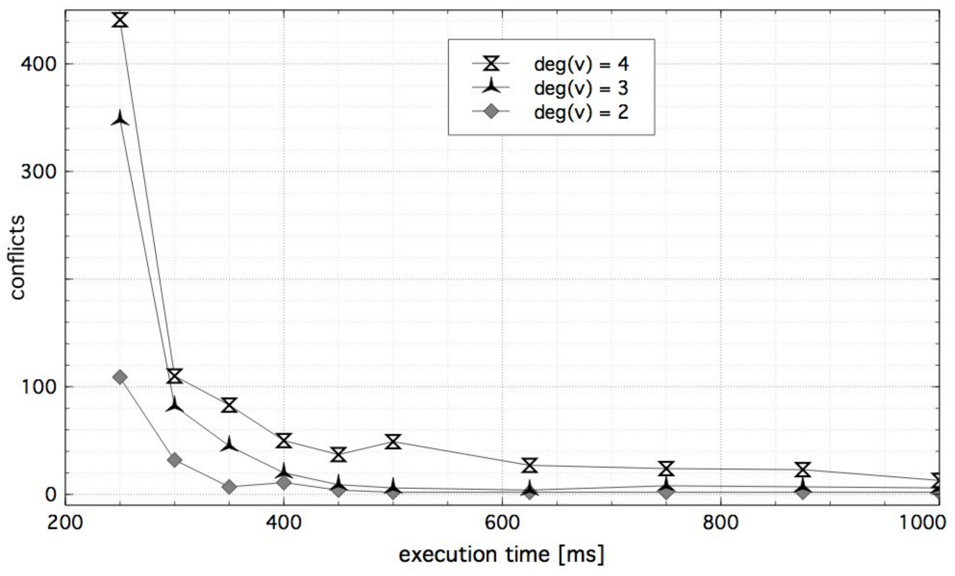


Figure 6.11: Conflicts over time for asynchronous Max-Sum on 200 node graphs with different vertex degrees.

7 Conclusion

As this is the last chapter of this thesis, the following sections aim to take a distant look at what has been done in this work, to mention aspects that may benefit from further investigation and to review aspects of the work in a critical way in order to identify strengths and weaknesses of the performed work and the results.

7.1 Summary and Conclusions

After reviewing related literature and research work and giving an overview over the current state of the art in the field of DCOPs, the thesis introduced the Max-Sum algorithm in a detailed and formal way. To improve the understanding of the algorithm, a hands-on example was illustrated.
These rather theoretical parts were followed by the main contribution of this thesis, the implementation of the Max-Sum algorithm in a graph processing framework. Signal/Collect [Stutz et al., 2010] was chosen as a graph processing environment for the implementation. Signal/Collect proved to be very useful and powerful during the implementation of the algorithm. The concept of a factor graph used in the formal definition of the Max-Sum algorithm could be mapped very easily through Signal/Collect's extendable vertex types. The message-passing approach of Max-Sum fits perfectly into the signaling and collecting scheme of the framework. Also,

the possibility to switch between synchronous and asynchronous execution modes proved to be a very essential and useful help. Another feature of the Signal/Collect framework that came in handy during the implementation of the benchmark infrastructure was the aggregation operation that provided a concise and simple way to measure statistics over the processed graph and algorithm. The score-guided computation of Signal/Collect allowed to install convergence detection in just a few lines of code. All told, Signal/Collect helped a lot in the implementation and was clearly the right choice for this kind of implementation. An aspect of the Max-Sum algorithm that posed difficulties was the fact that in contrast to most local DCOP algorithms, the Max-Sum algorithm never exchanges the actual states of the vertices. It exchanges messages in a certain format using the internal state of the vertices. This fact causes difficulties when measuring the number of conflicts and other statistics.

Having completed the implementation as the main building block of the thesis, the second important contribution of the thesis was ready to be processed: Benchmarking the Max-Sum algorithm against the DSA-A, DSA-B and Best-Response algorithms. The benchmarks should deliver empirical insights about the quality of solution and the performance of the Max-Sum algorithm. The quality was measured through the number of conflicts versus execution cycles and execution time, while the performance was measured by recording the execution time and the execution cycles until the computation converges. This benchmark is mainly a reproduction of the results in [Farinelli et al., 2008], but implemented in a modern graph processing framework. The first benchmark, evaluating the number of conflicts per step on a synchronous execution of the 4 candidate algorithms revealed very similar results to those found in [Farinelli et al., 2008]. In short, Max-Sum shows unsteady behavior on small ADOPT graphs, but it outperforms all of its contestants on the other ADOPT graphs. To analyze potential differences between asynchronous and synchronous computations of the Max-Sum algorithm, the conflicts over time (in milliseconds) were measured for executions of the Max-Sum algorithm in synchronous and asynchronous mode. This kind of analysis benchmark was not performed in related work and hence is a contribution of this thesis. On small graphs, the synchronous version delivered better results. On all other used graphs, both versions showed very similar results and there is no difference in the solution quality of the asynchronous and synchronous algorithms. After defining and implementing appropriate convergence mechanisms, the performance benchmarks were run analogous to the benchmark in [Farinelli et al., 2008]. First, the Max-Sum algorithm was evaluated against DSA-A, DSA-B and Best-Response algorithms in synchronous mode by measuring the execution steps until the computation converged. The results showed that there is no big difference in performance between the DSA variants and the Max-Sum algorithm. The Max-Sum algorithm was slightly weaker on most graphs. The Best-Response algorithm showed the best performance of the contestants in this benchmark. As a contribution of this thesis, a similar benchmark was run to compare the synchronous and asynchronous versions of the Max-Sum algorithm. The results show a significant difference between the performance of the synchronous and the asynchronous Max-Sum. The asynchronous Max-Sum version converges much faster, outperforming its contestant by a factor between 2.43 and 4.13. This underlines again the qualities of the Signal/Collect framework, providing a very simple way to switch between synchronous and asynchronous execution modes and comparing them.

The third goal besides the implementation and benchmark on a common machine was to distribute and benchmark the algorithm, to see its behavior when it is run on a cluster with multiple powerful machines. For this purpose, the implementation

was adapted in order to be distributable. This demanded for fully distributed statistics measuring and aggregation, as well as a fully distributed approach to spread global information or constants at the vertices and edges. These demands were very challenging and took quite an effort to satisfy them. The distributed statistics measuring was implemented by introducing a special type of messages exchanging the necessary statistics information among the edges and by implementing or adapting the necessary methods to send and receive those messages.
With a distributable implementation, contributions in form of distributed benchmarks were enabled. These distributed benchmarks were performed and presented in chapter 6. Again, the conflicts over the execution time were evaluated on different graphs for the synchronous and the asynchronous Max-Sum modes. In contrast to the analogous benchmark in chapter 5, there were clear differences in number of conflicts between synchronous and asynchronous Max-Sum. The asynchronous mode achieved generally lower numbers of conflicts. In order to create a link between the performance of the Max-Sum algorithm in terms of conflicts and the number of machines used, section 6.4.2 presented an evaluation of conflicts over time depending on the number of machines on which the algorithm was distributed. The assumption was that a higher number of machines used leads to a higher computation performance and hence to a faster reduction of conflicts in the evaluation. While on the smallest graph there was no observable influence, the assumption showed to be true as the graph size increased. The same analysis was performed also for the synchronous case. For the synchronous execution mode, the results were not as clear as in the asynchronous case. Especially on small graphs, the number of machines used seems to have no influence at all. Again, as the graph size increases, the influence increases slightly too. But clearly not to the same extent as in the asynchronous case. Since the theoretical complexity of the algorithm depends highly on the av-

erage vertex degree of the processed graphs, the thesis contributes an evaluation of the solution quality depending on the average vertex degrees. The results show that there is a slight impact of the average vertex degree on the solution quality and performance, but it is not as high since the complex computation explained in the complexity analysis is only performed for every second message during the execution.

7.2 Limitations and Future Work

This section aims to mention some of the points of the work that posed special difficulties, or should be critically examined. Also, further thoughts and ideas that didn't find their way into this thesis will be discussed.

As mentioned in section 5.6.1, Max-Sum showed poor results and unsteady behavior in the conflicts over steps benchmark on the 10 node ADOPT graph.

[Farinelli et al., 2008] base this behavior on the loop structure of the smaller ADOPT graphs, and they present a solution to this issue, which was also explained in detail in section 5.6.1. This suggested improvement was not considered in the implementation after this issue had been discovered during the benchmark. There are several reasons for this. One of them is the fact that this work was an ambitious project and hence had to be clearly limited in terms of scope, and time resources. Second, as the results in [Farinelli et al., 2008] show, modifying the utility function of the algorithm to adress the mentioned issue does in fact improve the performance of the algorithm (in terms of execution cycles) by a factor of around 4. At the same time, this modification leads to a decrease in solution quality of about the factor that the performance was improved. So essentially, it is a trade-off between solution quality and performance. This is a question where this thesis does not answer any results.

Hence, this might be an interesting hook for future work.

The results of the benchmarks on the time to convergence or the conflicts over time (wether on a distributed execution or not) have to be interpreted carefully. Those time values in milliseconds are not meant to be understood as *absolute* performance indicators. They serve rather as *relative* comparison tools between the synchronous and asynchronous versions of the Max-Sum algorithm and hence should be interpreted as such.

In the introduction of this thesis, the work on local and efficient DCOP algorithms was motivated through several scenarios such as large graph processing, distributed problems and especially embedded computing applications such as sensor networks. In the context of this application field, the resource efficiency, as well as the performance are of high importance. The Max-Sum algorithm showed good results in some of the performance benchmarks (c.f. chapter 5) but on the other side, its computation, especially the computation of the messages from function vertices to variable vertices is complex. As shown in section 4.3.4, the backtracking procedure to compute the maxima for the messages is quite involved and has a rather high worst-case complexity. This may raise the question wether the Max-Sum algorithm would be suitable for an application on embedded devices with low computational power and small memory resources. To answer this question, the results in this thesis are not sufficient, since they were produced on a graph processing framework intended for applications with a large scale on potentially powerful machines. To get significant empirical data about the adequacy of the Max-Sum algorithm for embedded devices, evaluations should be performed on appropriate hardware. [Farinelli et al., 2008] describe a hardware implementation of the Max-Sum algorithm on low-power chip systems.

During the experiments, the Max-Sum algorithm showed a certain sensitivity de-

pending on the initial preferences. This would be an interesting open question for further research. A question might be, how large is the impact of a certain configuration of the initial preferences on the convergence of the Max-Sum algorithm? Is there a graph-preference configuration that leads to non-convergence? This and similar questions would demand for future work.
Because of the limited scope and time resources of this thesis, the benchmarks in chapter 5 were performed on 3 graphs from the ADOPT repository. For more expressive results in future work, the benchmarks should consider a greater variety of input graphs from different sources with different structures. The same holds for the graphs used in chapter 6. A future work on this subject might achieve interesting results and insights by performing distributed evaluations on significantly larger scale. Especially with respect to a potential real-world use case, a benchmark on very large graphs would be beneficial.

References

[Aji and McEliece, 2000] Aji, S. M. and McEliece, R. J. (2000). The generalized distributive law. *Information Theory, IEEE Transactions on*, 46(2):325–343.

[Bernstein, 2012] Bernstein, A. (2012). Large-scale graph computation. Distributed Systems Lecture Notes, Dynamic and Distributed Information Systems Group, University of Zurich.

[Brown, 1951] Brown, G. W. (1951). Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13(1):374–376.

[Chapman et al., 2011] Chapman, A., Rogers, A., Jennings, N., and Leslie, D. (2011). A unifying framework for iterative approximate best-response algorithms for distributed constraint optimization problems. *The Knowledge Engineering Review*, 26:4(411-444).

[Cooper et al., 2007] Cooper, M. C., de Givry, S., and Schiex, T. (2007). Optimal soft arc consistency. In *Proc. of IJCAI*, volume 7, pages 68–73.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

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

Algorithm. *Proceedings of 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, (639-646).

[Fudenberg and Levine, 1998] Fudenberg, D. and Levine, D. K. (1998). *The Theory of Learning in Games*. MIT Press.

[Kearns et al., 2001] Kearns, M., Littman, M. L., and Singh, S. (2001). Graphical models for game theory. Technical report, Syntek Capital and ATT Labs Research.

[Kschischang et al., 2001] Kschischang, F. R., Frey, B. J., and Loeliger, H.-A. (2001). Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, 47(2):498–519.

[Low et al., 2010] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2010). Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990.

[Maheswaran et al., 2005] Maheswaran, R. T., Pearce, J. P., and Tambe, M. (2005). Distributed algorithms for dcop: A graphical-game-based approach. Technical report, University of Southern Californa, Los Angeles.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA. ACM.

[Modi, 2003] Modi, P. J. (2003). *Distributed Constraint Optimization For Multiagent Systems*. PhD thesis, University of Southern Californa.

[Modi et al., 2003] Modi, P. J., Shen, W.-M., Tambe, M., and Yokoo, M. (2003).

An asynchronous complete method for distributed constraint optimization. In *AAMAS*, volume 3, pages 161–168.

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

[Pearl, 1988] Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Petcu and Faltings, 2005] Petcu, A. and Faltings, B. (2005). A scalable method for multiagent constraint optimization. Technical report, Articial Intelligence Laboratory, Ecole Polytechnique Federale de Lausanne (EPFL).

[Schaller, 1997] Schaller, R. R. (1997). Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59.

[Sedgewick and Wayne, 2011] Sedgewick, R. and Wayne, K. (2011). *Algorithms, 4th Edition.* Addison-Wesley.

[Shoham and Leyton-Brown, 2009] Shoham, Y. and Leyton-Brown, K. (2009). *Multiagent Systems, Algorithmic, Game-Theoretic and Logical Foundations.* Cambridge University Press, Cambridge.

[Stutz et al., 2010] Stutz, P., Bernstein, A., and Cohen, W. (2010). Signal/collect: Graph algorithms for the (semantic) web. In *The Semantic Web–ISWC 2010*, pages 764–780. Springer.

[Tanenbaum and Van Steen, 2007] Tanenbaum, A. and Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms.* Pearson Prentice Hall.

[van Leeuwen et al., 2002] van Leeuwen, P., Hesselink, H., and Rohling, J. (2002). Scheduling Aircraft Using Constraint Satisfaction. *Electronic Notes in Theoretical Computer Science*, 76(252-268).

[Waldock et al., 2008] Waldock, A., Nicholson, D., and Rogers, A. (2008). Cooperative control using the max-sum algorithm.

[Weiser, 1991] Weiser, M. (1991). The computer for the 21st century. *Scientific American.*

[Young, 1993] Young, H. P. (1993). The evolution of conventions. *Econometrica: Journal of the Econometric Society*, pages 57–84.

[Zhang et al., 2005] Zhang, W., Wang, G., Xing, Z., and Wittenburg, L. (2005). Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1):55–87.

[Zhang and Xing, 2002] Zhang, W. and Xing, Z. (2002). Distributed breakout vs. distributed stochastic: a comparative evaluation on scan scheduling. *Proceedings of the AAMAS-02 workshop on Distributed Constraint Reasoning*, (192-201).

List of Algorithms

List of Listings

List of Figures

List of Tables