University of Zurich[UZH]

Bachelor Thesis July 16, 2013

Benchmarking Algorithms for Distributed Constraint Optimization Problems in Signal/Collect

Robin Hafen
of St. Gallen SG, Switzerland

Student-ID: 08-919-565
robin.hafen@uzh.ch

Advisor: **Mihaela Verman**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zürich
http://www.ifi.uzh.ch/ddis

Abstract

Many real world problems, such as network congestion control, can be mapped to the concept of a *distributed constraint optimization problem (DCOP)*. By analyzing a class of DCOP algorithms known as *local iterative approximate best-response (LIBR)* algorithms, [Chapman et al., 2011b] constructed a framework enabling the study and modular design of new hybrid algorithms. In [Chapman et al., 2011a], several classical, as well as new hybrid algorithms, were benchmarked in a series of graph coloring experiments. It was found that the modular approach to algorithm design allowed the creation of new, better performing algorithms.

In this thesis a similar approach was taken: selected existing LIBR algorithms, such as the *distributed stochastic algorithm* and *distributed simulated annealing*, were implemented and benchmarked using a graph processing framework called *Signal/Collect* [Stutz et al., 2010], with which no such benchmark has ever been conducted. As a further contribution, an existing, non-distributed algorithm from computer science literature called *tabu search* [Nurmela, 1993] was modularized and distributed in the same manner.

Zusammenfassung

Viele Probleme in der Informationstechnologie, wie zum Beispiel das Stausteuerungsproblem in Computernetzwerken, lassen sich als verteilte Bedingungsoptimierungsprobleme darstellen. [Chapman et al., 2011b] erstellten ein Rahmenwerk zur Analyse und modularen Konstruktion von neuen Hybridalgorithmen zum Lösen solcher Probleme. In [Chapman et al., 2011a] wurden mehrere solche Hybridalgorithmen in einer Serie von Graphfärbungsexperimenten auf ihre Leistungseigenschaften untersucht. Es wurde festgestellt, dass die modulare Konstruktionsweise die Erstellung von besseren Algorithmen ermöglicht.

In dieser Arbeit wurde ein ähnlicher Ansatz gewählt: Ausgewählte klassische Algorithmen wie der *distributed stochastic algorithm* und *distributed simulated annealing* sowie mehrere Hybridalgorithmen wurden anhand eines Programmierungsmodells namens *Signal/Collect* [Stutz et al., 2010] in einem bislang nicht existierenden Benchmark-Test evaluiert. Als ein weiterer Beitrag wurde ein zentraliserter Algorithmus namens *tabu search* in der gleichen Weise modularisiert und getestet.

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Structure . . . 2

2 Background and Related Work **3**
2.1 Distributed Constraint Optimization . . . 3
2.1.1 Formal Description . . . 3
2.1.2 Local Iterative Approximate Best-Response Algorithms . . . 5
2.1.3 DCOPs as Potential Games . . . 5
2.2 Signal/Collect . . . 6
2.2.1 Overview . . . 6
2.2.2 A Closer Look . . . 7
2.2.3 Related Technologies . . . 9

3 Algorithms **13**
3.1 Modularizing LIBR Algorithms . . . 13
3.2 Considerations in Algorithm Design . . . 15
3.3 Presentation of Algorithms . . . 16
3.3.1 Distributed Stochastic Algorithm . . . 16
3.3.2 Spatial Adaptive Play . . . 18
3.3.3 Greedy Spatial Adaptive Play . . . 19
3.3.4 Maximum-Gain Messaging . . . 19
3.3.5 Weighted Regret Matching with Inertia . . . 21
3.3.6 Joint Strategy Fictitious Play with Inertia . . . 22
3.3.7 Distributed Simulated Annealing . . . 23
3.3.8 Tabu Search . . . 24

4 Benchmark **29**
4.1 Design . . . 29
4.2 Results – Part One . . . 31
4.2.1 Distributed Stochastic Algorithm . . . 31
4.2.2 (Greedy) Spatial Adaptive Play . . . 34
4.2.3 Maximum-Gain Messaging . . . 36

4.2.4 Weighted Regret Matching with Inertia 37
4.2.5 Maximum-Gain WRM 38
4.2.6 Joint Strategy Fictious Play with Inertia 40
4.2.7 Distributed Simulated Annealing 42
4.2.8 Tabu Search 44
4.3 Results – Part Two 45

5 Discussion **51**

6 Limitations and Future Work **55**

7 Conclusion **57**

1

Introduction

1.1 Motivation

In classical optimization, a problem of interest is generally modeled as a function of specific parameters. The process of optimization then consists of finding a value assignment to these parameters such that the value of the function is maximal. Classical optimization is a well studied subject and therefore a plethora of techniques exist for solving even computationally demanding problems.

However, many problems, e.g. network congestion control, involve geographically dislocated variables and thus the classical approach to optimization becomes unwieldy: in order to utilize classical optimization techniques, the values of all variables values have to be available to some central entity. In a widely distributed setting, a central aggregation of information is, however, often impractical due to the high costs of communication.

If such a problem can be formulated as a *distributed constraint optimization problem (DCOP)*, distributing the process of optimization across multiple nodes in a computer network becomes feasible. Since a DCOP can be divided into multiple subproblems, a multi-agent system, in which each agent controls the value assignment of one variable, allows solving the problem at hand in a completely distributed manner.

[Chapman et al., 2011b] proposed a framework to modularize a class of DCOP algorithms known as *local iterative best-response (LIBR)* algorithms. By identifying the components of existing algorithms, they were able to create and analyze new hybrid algorithms.

Given the contributions of said framework, the goal of this thesis was to analyze the performance of existing LIBR algorithms using a graph processing framework called *Signal/Collect*[1] [Stutz et al., 2010]. By abstracting a problem to a graph wherein vertices and edges form the computational components, Signal/Collect allows for a clean implementation of a multi-agent system and is, therefore, a promising environment for the execution of LIBR algorithms.

The second goal was to apply the previously mentioned modularization process to existing centralized algorithms. This enabled the creation of a distributed version of a centralized optimization algorithm called *tabu search* [Nurmela, 1993].

[1] http://uzh.github.io/signal-collect/

The implementations of the algorithms and all resources related to the benchmark are made available online[2] under the Apache license[3] version 2.0.

1.2 Structure

The content of this thesis is structured as follows:
In chapter 2, a formal introduction to the topic of DCOPs is given. Additionally, the programming model Signal/Collect is introduced and set in the context of related technologies. The distributed version of tabu search, as well as the other hereinafter benchmarked algorithms, are presented in chapter 3. Chapter 4 then provides a description of the benchmark and shows the results obtained from it. Following the presentation of the result, chapter 5 contains interpretations of all findings and discusses the insights they provide. The thesis closes with a description of potential future work in chapter 6 and a concluding statement in chapter 7.

[2] https://github.com/hafenr/benchmarking-libr-algs-signal-collect
[3] http://www.apache.org/licenses/LICENSE-2.0.html

2

Background and Related Work

The following sections provide an overview of the two main topics to which the content of this thesis relates. First, the concept of a *constraint problem (CP)* and the more specific notion of a *distributed constraint optimization problem* will be explained. Second, a short overview of the graph processing framework *Signal/Collect*, which will later be used to evaluate various algorithms in the context of DCOPs, will be given.

For each of the two topics important terminology will be established and references to existing works will be provided.

2.1 Distributed Constraint Optimization

The notion of a constraint problem can be applied to problems of various kinds. In general, a constraint problem involves many different variables and the solution of the problem is constrained by a set of requirements involving these variables.

It is important to differentiate between *constraint satisfaction problems (CSPs)* and *constraint optimization problems (COPs)*. In the former case, the goal is to find a value assignment for each of the involved variables, such that all constraints are met. In the latter case, however, there must exist some numeric measure of "how good" a given value assignment for the variables is. The goal then becomes finding a set of values that yields the highest measure of "goodness".

This chapter progresses with a more complete formal description of constraint optimization problems. Additionally, an outline of related work that has been done in this field will be provided.

2.1.1 Formal Description

As defined by [Arshad and Silaghi, 2004], a constraint satisfaction problem can formally be described as:

A formal construct $\mathcal{P}$ involving

- a set of variables $V = \{x_1, x_2, \cdots, x_n\}$, each of which has an associated domain from the

- set of domains $D = \{d_1, d_2, \cdots, d_n\}$, and
- a set of constraints $C = \{c_1, c_2, \cdots, c_k\}$, of which each involves a subset of V.

In the case of a constraint optimization problem, a problem like this – involving n variables – can be regarded as finding the global optimum of a function from a n-dimensional domain to the set of real numbers. The target value then corresponds to some measure of "how good" or "how bad" a given assignment of values to the variables of the problem is. This measure is, of course, dependent on the context in which a solution is to be found. In existing literature, the terms *cost function* (less is better) or *utility function* (less is worse) are often used. A straightforward approach in the context of COPs is to define a utility function as the number of satisfied constraints depending on the value assignment of the variables. Before explaining what a *distributed* constraint optimization problem is, I will try to picture it with a simplified example of congestion control.

Example: In a network of routers, each node has a limit of amount of data it can process without beginning to lose packages and causing long delays. The goal is to find the amount of data each of the routers has to process, such that the least amount of routers deteriorate in quality of service, while still maximizing the system throughput[1].

In a non-distributed setting, where the whole state of the system is available to a central entity, a problem such as this could readily be modeled as a classical optimization problem. In the case of a computer network, however, such an approach would require that each of the nodes has to inform a centralized entity about its current state. A solution attempt like this is suboptimal due to various reasons:

For example, the reliance on a centralized entity is known to be a bad design decision in a distributed system. Furthermore, the number of long distance messages that were needed to be exchanged is by itself bad for system throughput. Hence, the nodes themselves should be able to communicate with their direct neighbors and decide how much data they accept for processing. The variables – corresponding to the maximum load of each router – therefore have to be controlled by local entities.

The previously established formal description of a constraint problem $\mathcal{P}$ must therefore be augmented by:

- a set of n agents $I = \{1, 2, \cdots, n\}$, each of which controls exactly one variable $x_i \in V$ and each variable is controlled by exactly one agent $i \in I$.

It has to be noted that in existing literature an agent is often defined as being in control of a subset of all variables. However, the extended definition in which each agent controls one variable maps more naturally to the way Signal/Collect operates, while not introducing any loss in generality [Chapman et al., 2011a]. An extensive study of distributed constraint satisfaction is given by [Yokoo et al., 1992].

[1] The term *throughput* refers to the average rate of successful message deliveries over a communication channel. *System throughput* is the analogous term for all communication channels in a computer network.

2.1.2 Local Iterative Approximate Best-Response Algorithms

The class of algorithms that is suited for DCOPs is known as *distributed constraint optimization algorithms*. [Chapman et al., 2011b] split this class into three subclasses. The subclass that is of special importance in the context of this thesis is known as the one of *local iterative approximate best-response (LIBR)* algorithms. These algorithms are defined as having two restrictions that make them interesting for solving DCOPs in a completely distributed manner:

- First, agents in these algorithms may communicate only with other agents if there exists a constraint involving the variables of both agents.
- Second, an agent is allowed to send only messages that contain information about himself (e.g., the current value of the variable he controls).

These restrictions ensure that the usage of such algorithms leads to relatively small communication costs. If the goal is to find the optimal solution to a DCOP, algorithms from another subclass known as *distributed complete* may provide better results. However, these algorithms are generally more complex and computationally demanding. Furthermore, as pointed out by the authors of these definitions, in many applications it is often more important to find a value assignment for the variables that yields a utility that is "good enough" in a small amount of time than to find the global optimum.

2.1.3 DCOPs as Potential Games

[Chapman et al., 2011b] took a concept from game theory[2] known as *potential games* to construct a framework for studying and designing LIBR algorithms.

From the game theoretic point of view, a potential game can informally be described as a set of players wherein each player – or agent – tries to maximize its own utility by playing a certain strategy. The decision of what strategy the agent plays depends on what strategies other players employ. Additionally, no player will play a strategy that lowers the global utility, because it is defined as the sum of utilities over all players. There may exist certain strategy configurations such that no agent has an incentive to change its strategy since no increase in global utility can be achieved. Such configurations are known as *Nash equilibria* and can be seen as the maxima of the local utility functions.

Furthermore, for each potential game, there exists a finite sequence of state changes such that the global utility increases with every step – ultimately resulting in a Nash equilibrium. This property is known as the *finite improvement property* and was extensively analyzed by [Monderer and Shapley, 1996]. The finite improvement property assures that an algorithm will ultimately find a solution corresponding to a Nash equilibirum, provided that players change states *only* if the change does not result in a worse utility.

In the previously established formulation of a DCOP, each agent $i \in I$ was defined as being in control of a variable $x_i \in V$. With the intent of uniting all concepts presented

[2]Game theory refers to the study of decision making using mathematical models.

thus far, the formal construct of a DCOP is extended by associating each agent i with a utility function

$$u_i : \prod_{j \in N_i \,\cup\, \{x_i\}} d_j \to \mathbb{R}$$

where N_i corresponds to the set of variables that are involved in a constraint with x_i and d_j denotes, as defined earlier, the domain of the variable x_j. These utility functions therefore associate each agent's local variable configuration with some real number – henceforth simply called the "utility".

2.2 Signal/Collect

One of the goals of this thesis is to analyze the performance of selected LIBR algorithms using the programming model Signal/Collect together with its official implementation[3]. As stated in the introduction, Signal/Collect provides a natural way of modeling a multi-agent system and thus lends itself as a suited environment for the execution of LIBR algorithms. Due to the choice of this specific framework, a small introduction to its programming model and some of the features of its implementation is in order.

The following sections therefore try to provide insight on how Signal/Collect operates and why its model of computation is suited for the execution of LIBR algorithms. In the remainder of this chapter, some notable other programming models are presented in order to provide rationale for choosing Signal/Collect over related technologies.

The current implementation of Signal/Collect is written in the *Scala*[4] programming language. Implementation-specific code will therefore be written in Scala-like pseudocode using `typewriter` font. Algorithms that can be understood in a more general context are, however, presented in typical imperative pseudocode.

2.2.1 Overview

Signal/Collect was originally created to provide a way to oppose the immense growth of available data specifically in the context of the *Semantic Web*. [Stutz et al., 2010] pointed out that some graph problems had to be shoehorned to programming models that were not specifically designed for such tasks. As a concrete example, the pipeline-like processing of key-value pairs employed by *MapReduce*[5] was mentioned. In contrast, Signal/Collect was created from the ground up to provide a more natural way of handling web-related data, which is inherently graph-like in structure.

In Signal/Collect, a *compute graph* G, consisting of a set of vertices V and a set of their connecting edges E, forms the basis of a computation.

The general idea is that vertices can send messages – called *signals* – along their connecting edges. Neighboring vertices receive these signals, use the therein stored

[3] The framework Signal/Collect is licensed under the Apache License 2.0 and is available at http://uzh.github.io/signal-collect/.

[4] http://www.scala-lang.org/

[5] http://research.google.com/archive/mapreduce.html

information to compute some value, and, again, pass new messages along their edges. This process can be divided into two consecutive phases:

The signal phase in which vertices send messages to their direct neighbors.

The collect phase in which vertices gather all received signals to compute some value, possibly altering their internal state.

Given this short overview of Signal/Collect, some details about the actual implementation are necessary since many of the later presented algorithms make use of its features. For the following sections, it has to be kept in mind that the terms "actor", "player", "vertex", and "node" can be understood as representing the same concept. Similarly, the terms "message" and "signal" may be regarded as synonyms.

2.2.2 A Closer Look

As previously mentioned, a computation in Signal/Collect consists of two consecutive phases – signal and collect – which are repeated until a certain termination condition is met.
A vertex has the following properties[6]:

id: a unique identifier in the graph.

state: the current state representing an intermediate result in the computation.

mostRecentSignalMap: a map from the ids of all neighboring vertices to the most recent signals they sent to this vertex.

Additionally, a vertex must implement the abstract method `collect: State`[7] which, when invoked by the framework, alters the state of the vertex. Of course, an actual change of state happens only if `collect` returns a state different from the previous one. The second basic unit of computation is the edge, which has among others the following properties:

sourceVertex: the source vertex where it originates.

targetVertex: the target vertex to which it connects.

Edges further have to implement the abstract method `signal: Signal`, which is also invoked by the framework and returns the message that is to be passed to the vertex `targetVertex`.

An important fact regarding vertices and edges is that Signal/Collect allows typed graphs. This means that each vertex and edge has a specific type, which is definable by means of inheritance. Since the framework is defined on the interfaces provided by

[6]This list is not exhaustive, but should provide enough information to picture how a computation in Signal/Collect works.
[7]The colon in this notation indicates the type of the function's value and can be read as "has type".

the type-parameterized base classes `Vertex[+Id,State]` and `Edge[+TargetId]`[8], graphs with different vertex and edge types can be constructed, thus allowing great flexibility when modeling a DCOP. Furthermore, a graph in Signal/Collect does not have to be static. During a computation, the topology of the compute graph may be altered at will by adding and removing edges or vertices. This feature is a strong point of Signal/Collect, considering the fact that the problems to be modeled are often very dynamic in nature.

Modes of execution

Signal/Collect was designed with the possibility of both synchronous and asynchronous execution in mind. For a program in Signal/Collect an *execution mode* can be specified. These execution modes include `Synchronous`, `PureAsynchronous`, and `OptimizedAsynchronous`. In synchronous executions, the signal and collect phases are globally synchronized. During each phase all vertices signal or collect in parallel. Such a programming model is known as the *bulk synchronous parallel (BSP)* model and was introduced by [Valiant, 1990].

In contrast, asynchronous execution mode does not provide globally synchronized signal and collect phases. While the signal and collect phases of each vertex are still consecutive, no guarantee exists that any two vertices are in the same phase at the same time. The default execution mode, `OptimizedAsynchronous`, is a slight alteration of `PureAsynchronous` in that it consists of one single globally synchronized step which is followed by asynchronous execution.

As will be shown later, the underlying execution modes of Signal/Collect can be used to construct new execution modes, each tailored for a specific family of LIBR algorithms.

Termination conditions

A Signal/Collect computation terminates when certain conditions are met. These conditions can be set prior to starting the computation by means of specifying the maximum number of signal and collect steps (in the case of synchronous execution) or a by certain time limit (in the case of asynchronous execution). Another possibility is to utilize the methods `scoreSignal: Double` and `scoreCollect: Double` along with their associated thresholds `signalThreshold` and `collectThreshold`. These functions can be implemented inside the vertex and edge class, respectively. Prior to invoking the `collect` method of a vertex, its `scoreCollect` function is called. If its returned value is below the associated threshold, the vertex will not collect. Analogously, an edge will not signal if its score is below `signalThreshold`. The rescoring of a vertex is done when a new signal is received. An edge is rescored if the `collect` method of its source vertex was invoked[9]. Automatic termination is triggered when there are no more active vertices

[8] In Scala, types stated in brackets are type parameters. The `+` sign in front of a type parameter marks it as covariant. This basically means that given two types `A`, `B`, a parameterized type `F[+T]` with a covariant type parameter `T`, and a subtype relation denoted `A <: B` (read "`A` is a subtype of `B`"), the assertion `F[A] <: F[B]` holds.

[9] Altering the graphs topology during runtime triggers rescoring as well. Additionally, the user may trigger rescoring manually. These mechanisms, however, are not essential in the context of this thesis.

and edges, i.e., all scoring functions returned values below their respective thresholds. All afterwards presented algorithms use an implementation of `scoreSignal` identical to the one in listing 2.1. Additionally, `signalThreshold` is set to a number greater than 0.

Listing 2.1: The `scoreSignal` function

```
override def scoreSignal: Double =
  if (allConstraintsSatisfiedWith(currentState) && !stateHasChanged)
    0
  else
    1
```

Another possible way of forcing termination, which is also used in the implementation of the hereinafter presented algorithms, is the use of *global termination conditions*: after a certain interval of time (or steps), termination can be triggered by repeatedly checking if, given the current state of the graph, a user-definable predicate holds. To make use of this functionality, the user has to define what information is to be extracted from each vertex and how this information is to be aggregated. The "current state of the graph" can therefore be defined in the context of a specific algorithm.

2.2.3 Related Technologies

There exist various other frameworks with different programming paradigms for parallelized information processing. The reasons that motivate the use of Signal/Collect in the context of DCOPs are best explained by highlighting how Signal/Collect compares to other frameworks. For this reason, an overview of two other frameworks will be given.

MapReduce

MapReduce is a programming model made famous by its use in the data center infrastructure of Google. Apart from the *Google File System* [Ghemawat et al., 2003] and *BigTable* [Chang et al., 2008], MapReduce[10] is one of the core components of Google's massive data processing infrastructure[11] [cnet2008google, 2008].

As the name suggests, the algorithm of MapReduce mainly consists of two functions called `Map` and `Reduce`, whose names are inspired by the functions `map` and `reduce`[12] which exist in many functional programming languages. The algorithm consists of multiple consecutive steps:

Upon initialization, a large and possibly widely distributed data set (the problem) is split into separate chunks by a centralized entity – the so called *master node*. These chunks get assigned to a number of worker nodes that may split the data sets even further. After the process of splitting, worker nodes process the – now much smaller –

[10] https://developers.google.com/appengine/docs/python/dataprocessing/overview

[11] Although patented by Google, open-source implementations of MapReduce and associated technologies exist in the form of Apache's Hadoop project (http://hadoop.apache.org/).

[12] In some programming languages the function corresponding to `reduce` is called `fold`.

data sets by applying a user-defined `Map` function to them. The data returned by `Map` is then sent back to the master node which sorts it and distributes it again among multiple *reducer nodes*. These nodes process their assigned data by the, also user-defined, `Reduce` function. Eventually, the MapReduce system aggregates the final data to produce the output of the algorithm.

Modelling a DCOP as a graph of interacting agents leads to an inherently non-linear data flow. The pipeline-like way of data processing that MapReduce employs is thus a rather unnatural way of approaching such a problem. According to [Malewicz et al., 2010], using MapReduce on large graphs ”can lead to suboptimal performance and usability issues”. Furthermore, the reliance on a central entity like the master node of MapReduce is disadvantageous in the context of completely distributed algorithms.

In contrast, the already graph-based programming model of Signal/Collect maps much closer to the concept of a DCOP and should therefore be better suited for the task at hand.

Pregel

Pregel is a scalable platform designed by Google for processing large graphs. It was designed to have a degree of fault-tolerance that allows it to run on ”clusters of thousands of commodity computers” [Malewicz et al., 2010].

The most basic unit inside the compute graph of Pregel is – much like in Signal/Collect – a vertex. In accordance with general graph theory, each vertex may have several edges which connect it to neighboring vertices. Pregel uses a model of computation similar to the bulk synchronous parallel model, which is also employed by the synchronous execution mode of Signal/Collect. The synchronous part of a Pregel computation is given by a sequence of iterations called *supersteps*. During each superstep S, vertices can – as in Signal/Collect – pass messages along their edges which are received at superstep $S + 1$.

User-definable functions decide for each vertex and superstep what messages are to be sent, how to process the received messages sent at $S - 1$, and how the state of each vertex should be changed. These functions are invoked in parallel for each vertex at each superstep. In contrast to the `signal` and `collect` functions, the behavior of changing state and messaging neighbors is therefore merged into a single procedure.

The separation of these functionalities into distinct components eases the modular construction of algorithms. This fact is a strong plus factor for the usage of Signal/Collet since the modular construction of LIBR algorithms is a core subject in this thesis.

Also note that – in contrast to Signal/Collect – Pregel does not support custom vertex types. Although a typed graph can be emulated in Pregel by means of simple programming constructs, the inheritance-based approach of Signal/Collect enables a conciser and type safe way of modularizing algorithms.

The topic of termination detection highlights a further similarity between the two models: in Pregel vertices can ”vote” to halt upon which they are deactivated until they receive a new message. When all vertices in the graph are deactivated, the algorithm

terminates. This is similar to the functionality of the scoring functions `scoreSignal` and `scoreCollect` used in Signal/Collect.

In Pregel, subsets of vertices get assigned to *partitions* which are then, in turn, assigned to threads. The assignment of a vertex to a partition is done (by default) purely by id. The assignment of partitions to actual machine workers (computers) can also be customized to, for example, exploit vicinity between certain machine workers.

As previously mentioned, Pregel tries to achieve a high degree of fault tolerance. Its main mechanism for providing fault-tolerance relies on *checkpointing*: at the beginning of each superstep, the workers running the partitions get instructed to save their state to persistent storage. By consecutively pinging each worker thread, the system detects crashed partitions and – in case they will not reboot – will assign the dead partition to a new worker thread.

The creators of Pregel argue that the synchronicity of their programming model makes reasoning about the correctness of a program easier and also eliminates all possibilities for race conditions. Due to the similar nature of the two frameworks, these statements can readily be applied to Signal/Collect as well.

It has to be noted that the algorithms benchmarked in this thesis solely use the synchronous execution mode of Signal/Collect. However, in contrast to Pregel, Signal/-Collect provides the possibility of asynchronous execution. This fact is a further strong point since asynchronicity enables the construction of a whole new set of algorithms that may be studied in potential future works.

3

Algorithms

If the previously established definition of a DCOP is simplified by the change that, now,

- each constraint involves exactly two variables $x_i, x_j \in V$, where $i \neq j$ and further states that $x_i \neq x_j$,

the problem can be understood as a loop-free graph in which pairs of agents and the variables they control correspond to vertices and constraints can be seen as undirected edges. The problem that arises by the introduction of this change is the famous *vertex coloring problem*, in which an assignment of colors to vertices is to be found such that no two adjacent vertices are colored alike.

As a further simplification, the domains of all variable d_i are equal to some set D. In the context of vertex coloring, this means that every vertex has the same finite set of colors it can be colored with. For purposes of easier implementation, colors are represented by integers. Therefore, the domain for each variable can be defined as a set of integers $\{0, 1, 2, \ldots, |D| - 1\}$.

3.1 Modularizing LIBR Algorithms

As mentioned in section 2.1.3, [Chapman et al., 2011b] created an analytical framework for studying and designing LIBR algorithms. An important aspect of this framework is that it allows the specification of an algorithm in terms of its components or modules. These modules are as follows:

- First, a *target function*, which evaluates a given state according to a utility function. The value returned by this function is then referred to as the *payoff* of this state. In its most simple version, the target function corresponds to the utility of a state assumed the neighboring vertices do not alter their state. In this case, the target function is known as *immediate payoff*.

- Second, a *decision rule* that returns – given some evaluated states – the state that matches some specific criterion.

- Third, an *adjustment schedule* that decides if a vertex should compute a new state.

These modules can be expressed naturally in Scala in terms of *traits*, which can be used as a way to extend a class with certain functionalities. Compared to other programming constructs, traits are similar to *mixins* or *interfaces* that allow the specification of behavior.

The modules defined above are further extended by a function called `prospectiveStates` that enables a vertex to consider only a subset of its domain as possible new states. Listing 3.1 outlines how these modules fit together. The colon notation is to be read as "has type".

Listing 3.1: The components of a LIBR algorithm

```
trait TargetFunction {
  def utility(state: State): Utility = {
    return numberOfConstraintsSatisfiedWith(state)
  }

  def evaluate(state: State): (State, Payoff) = {
    return (state, utility(state))
  }

  def evaluateAll(states: List[State]): List[(State, Payoff)] = {
    val evaluatedStates = for (state <- states) yield {
      evaluate(state)
    }
    return evaluatedStates
  }
}

trait DecisionRule {
  def decisionRule(currentEvaluatedState: (State, Payoff),
                   otherEvaluatedStates: List[(State, Payoff)]): State
}

trait AdjustmentSchedule {
  def shouldComputeNewState: Boolean
}

trait ProspectiveStates {
  def prospectiveStates(allStates: List[State]): List[State]
}
```

In section 2.2.1 it was explained that vertices in Signal/Collect compute their states by means of their `collect` function and that edges use their `signal` function to pass signals between adjacent vertices.

Having defined the components of LIBR algorithms in terms of extendable traits, the implementation of `collect` and `signal` can be generalized to the one shown in listing 3.2 and 3.3, respectively.

Listing 3.2: The generalized collect function

```
def collect: State = {
  if (shouldComputeNewState) {
    val statesToEvaluate = prospectiveStates(domain)
    val evaluatedOtherStates = evaluateAll(statesToEvaluate)
    val evaluatedCurrentState = evaluate(state)
    val chosenState = decisionRule(evaluatedCurrentState,
       evaluatedOtherStates)
    return chosenState
  } else {
    return state
  }
}
```

In the case of `signal`, the standard behavior is to simply inform adjacent vertices of their neighbors' states.

Listing 3.3: The generalized signal function

```
def signal: Signal = return sourceVertex.state
```

3.2 Considerations in Algorithm Design

Making an optimization algorithm distributed poses several difficulties that have to be addressed when designing such an algorithm. Two of the most important characteristics of LIBR algorithms are described below.

- If neighboring agents are allowed to change their state at the same time, they can not know how their state change will affect the global utility, i.e., the number of constraints satisfied. Consider a scenario involving two neighboring vertices whose domains consist of the colors green and red. If both the vertices are colored green at time t, both of them want to change their color. Since – from their point of view – a change to color red would satisfy their constraint, both of them change to color red. At time $t+1$ both vertices are colored red and in the same situation as before. This oscillatory process can go on indefinitely and is known as *thrashing*. Reducing the number of vertices that are allowed to change their state simultaneously can minimize the risk of thrashing. However, less simultaneously computing vertices also lead to less utilized computing power. This issue is approached differently by many of the algorithms presented in section 3.3.

- Another important characteristic of an algorithm is a property known as *anytime*. An algorithm is said to be anytime if it can be stopped at any instant during its runtime and it will return the best solution found up to that point in time. The famous optimization algorithm *gradient ascent*[1] or its twin *gradient descent* are

[1] Gradient ascent is a simple optimization algorithm that gets its name from the fact that it tries to

anytime since they – by design – always move in the right direction and therefore never deteriorate in solution quality they produce. Many of the simpler anytime algorithms are prone to get stuck in local optima. For this reason, many algorithms employ stochastic mechanisms that help them to get out of optima at the cost of not being anytime. With non-distributed optimization algorithms, a straightforward approach would be to have the algorithm remember the best solution it found thus far, which – in case it is terminated prematurely – will be returned. However, in the context of DCOPs this is not easily done. For this claim I provide three reasons: Some vertex i can not know if the solution that was best for itself is also the best for some vertex $j \neq i$. The approach of having the vertices negotiating which solution was best falls short due to the prohibitively large amount of messages it would require. Also, designating special vertices as "book keepers" may work but would defeat the purpose of *distributed* constraint optimization due to the existence of centralized entities.

In the following sections, the algorithms whose performance will later be analyzed are presented and their respective components identified. In cases where it serves the purpose of clarity, the components are outlined in pseudocode. It has to be kept in mind that these components are implemented by simply subclassing the previously mentioned traits and overriding their respective functions.

3.3 Presentation of Algorithms

The following algorithms were introduced in various research papers but were never mapped to the framework presented in [Chapman et al., 2011b] using Signal/Collect. The specific algorithms were chosen on account of their good performance identified in existing works, historic significance, and/or their property of representing a non-standard approach to optimization.

3.3.1 Distributed Stochastic Algorithm

[Zhang and Wittenburg, 2002] found that the family of *distributed stochastic algorithms (DSA)* showed better performance in vertex coloring problems than an algorithm known as *distributed breakout*, which was previously considered the top contender in this discipline.

A plethora of DSA variants has been proposed in existing literature. The benchmark of this thesis includes DSA-A, as well as DSA-B, a variant which promises good performance on vertex coloring problems, as shown by [Zhang et al., 2002].

DSA-A and DSA-B both use their utility function to evaluate prospective new states without adding additional complexity. As mentioned before, the utility of a given state for some vertex i is equal to the number of constraints in which i is involved that were

find the maximum of a function by simply following its gradient, i.e., the direction of its greatest rate of increase.

satisfied if i was to choose said state and its neighborhood would not change. In the context of vertex coloring this corresponds to the number of neighbors with different colors.

This kind of target function is called immediate payoff because the payoff of some state is simply the utility an agent would achieve with said state assumed the state configuration of its neighborhood does not change.

In the case of DSA-B, the procedure that is used to decide which of the evaluated states to choose is known as `argmaxB`. `argmaxB` is – as the name implies – related to the mathematical operator *argmax* which yields the set of argument values for which the value of a given function is maximal. The main differences between the two are: First, `argmaxB` yields only one value. Second, in the case that multiple such argument values are found, `argmaxB` selects one of these values randomly. An outline of the `argmaxB` procedure is shown in algorithm 1.

DSA-A uses a slightly different member of the argmax family, namely `argmaxA` (algorithm 2). In contrast to `argmaxB`, `argmaxA` chooses a new state only if its payoff is *strictly* greater than the payoff of the current state. In the case that multiple such states are found, the state is – just like in `argmaxB` – chosen randomly.

Algorithm 1 The decision rule argmaxB

```
procedure DECISIONRULE_ARGMAXB(currentPayoff, otherPayoffs)
    allPayoffs ← {currentPayoff} ∪ otherPayoffs
    pMax ← { p ∈ allPayoffs | p = max(otherPayoffs) ∧ p ≥ currentPayoff }
    if pMax = ∅ then
        return stateOf(currentPayoff)
    else
        return stateOf(randomElementIn(pMax))
    end if
end procedure
```

Algorithm 2 The decision rule argmaxA

```
procedure DECISIONRULE_ARGMAXA(currentPayoff, otherPayoffs)
    pMax ← { p ∈ otherPayoffs | p = max(otherPayoffs) ∧ p > currentPayoff }
    if pMax = ∅ then
        return stateOf(currentPayoff)
    else
        return stateOf(randomElementIn(pMax))
    end if
end procedure
```

All variants of DSA use an adjustment schedule known as a *parallel random schedule* which allows a vertex to compute a new state only with a certain probability p – known as the *degree of parallel executions*. In [Chapman et al., 2011b], the parameter p is

defined slightly differently: instead of being the probability of a vertex computing a new state, p is defined as the probability of a vertex changing to an already computed state. The altered definition is used in this thesis since I consider the process of evaluating a possibly large amount of states, only to forbid the vertex to acquire a subsequently chosen state to be wasteful in terms of computational resources. Using this altered definition, a parallel random schedule can easily be implemented using the synchronous execution mode of Signal/Collect and a procedure similar to the one shown in algorithm 3. As mentioned in the beginning of this chapter, thrashing poses a difficulty in algorithm design. Restricting the number of vertices that may change their state at the same time by a probabilistic process can lessen the risk of thrashing. Its occurrence, however, can not be ruled out completely.

Algorithm 3 The parallel random adjustment schedule

```
procedure shouldComputeNewState
    if randomElementIn( [0,1) ) < p then
        return true
    else
        return false
    end if
end procedure
```

3.3.2 Spatial Adaptive Play

SAP [Young, 2001] too uses immediate payoff as its target function. Its decision rule is, however, quite unlike the `argmax*` family. SAP decides which state to pick according to a probability distribution which generally assigns states with higher payoff a higher probability. Although the probability of such an event is relatively small, even a state with a payoff lower than the payoff of the current state can be chosen as the new state.

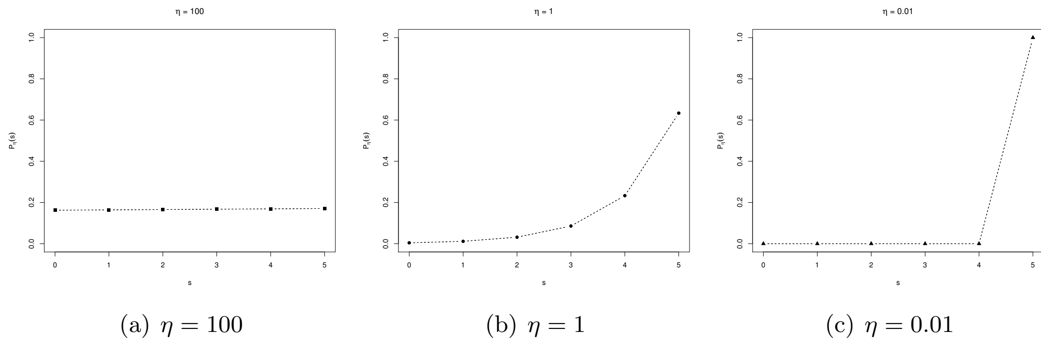
The probability mass function[2] which describes said distribution is called a *multinomial logit function* and is given by:

$$P_\eta(s) = \frac{e^{\frac{1}{\eta}u_i(s)}}{\sum_{k \in d_i} e^{\frac{1}{\eta}u_i(k)}}, \quad s \in d_i$$

where d_i is the set of considered states (the domain of the vertex) and u_i the utility function over it. The parameter η determines the shape of the distribution substantially, as can be seen in figure 3.1.

The higher the value of η is, the closer the distribution is to an uniform one. In contrast, if η gets closer to 0 the probability that non-maximal payoffs are chosen gets continuously smaller. Therefore, at $\eta = +\infty$ the decision rule of SAP randomly chooses

[2] A probability mass function $p_X(x)$ of some random variable X yields the probability that its associated random variable acquires a value equal some x, i.e., $p_X(x) = P(X = x)$.



(a) $\eta = 100$ (b) $\eta = 1$ (c) $\eta = 0.01$

Figure 3.1: The multinomial logit function with various values for η.
The y-axes indicates the probability of choosing a state dependent on the payoff associated with that state. Note that the dotted lines are solely for purposes of clarity.

its values, regardless of their payoff. At $\eta = 0$, however, the decision rule resembles the behaviour of `argmax*` in which only the value with the highest payoff is chosen.
The value η can either be set to a constant value or gradually be altered over time. In the following benchmark η is kept constant at various values. The mechanic of gradually lowering η is employed by an algorithm named *distributed simulated annealing*, which is presented shortly after.

As an adjustment schedule, SAP uses a *sequential random schedule*. Again, this schedule uses Signal/Collect's synchronous execution mode. In contrast to the parallel random schedule employed by the DSA family of algorithms, only one vertex is randomly chosen and allowed to compute a new state at each globally synchronized collect step.

3.3.3 Greedy Spatial Adaptive Play

As previously shown, by lowering the parameter η in the decision rule of SAP, its behaviour approaches that of `argmax*`. To analyze how an immediate payoff algorithm like DSA performs with a sequential random schedule, a "greedy"[3] version of SAP (henceforth called gSAP) is added to this list of algorithms.

The excellent performance of gSAP was underlined by the findings of [Chapman et al., 2011a] which further qualifies it as a good candidate for benchmarking in Signal/Collect.

3.3.4 Maximum-Gain Messaging

Up until this point, all of the presented algorithms simply passed their newly computed state along their edges during a `signal` step. Maximum-gain messaging (MGM), however, uses two different types of signals.

As the name of the algorithm suggests, a vertex in MGM informs its neighbors about the maximum payoff – or gain – it could achieve if it was allowed to acquire some state.

[3]The term greedy refers to the fact that, unlike SAP, gSAP tries to maximize its payoff each `collect` step without considering states that would result in a lower utility.

A vertex is allowed to change its state only if it could achieve the greatest payoff of all its neighbors. This mechanism ensures that no thrashing can occur since no two neighboring vertices can change their state at the same time. Furthermore, the fact that each vertex only changes to states of higher payoff leads to the algorithm being anytime [Maheswaran et al., 2004].

Although the maximum-gain mechanic is an adjustment schedule, the functionality was implemented in terms of overriding `signal` and `collect`. However, the same effect could have been achieved by providing a special version of `shouldComputeNewState`. As can be seen in algorithm 4, MGM uses the immediate payoff target function to evaluate its prospective states and `argmaxB` as its decision rule. The modularity is therefore retained.

Algorithm 4 The maximum-gain schedule

```
procedure mgmCollect
    if phase = maxGainExchange then
        phase ← stateInform                                   ▷ change phase
        gains ← gains(recievedSignals) + myGain
        maxGains ← maxValues(gains)
        maxGain ← breakTiesByVertexId(maxGains)
        if maxGain.id = myId then          ▷ vertex ids are sent along with messages
            return candidateState               ▷ acquire the computed best state
        else
            return currentState                         ▷ keep the current state
        end if
    else if phase = stateInform then
        phase ← maxGainExchange                               ▷ change phase
        neighborStates ← states(recievedSignals)
        payoffs ← { utility(st, neighborStates) | st ∈ domain }
        myGain ← argmaxB(payoffs)                   ▷ the chosen highest payoff
        candidateState ← stateOf(myGain)       ▷ the state which yields myGain
        return currentState
    end if
end procedure

procedure mgmSignal                          ▷ invoked for each outgoing edge
    if sourceVertex.phase = maxGainExchange then
        return sourceVertex.myGain         ▷ send the computed gain to neighbor
    else if sourceVertex.phase = stateInform then
        return currentState                ▷ send the computed gain to neighbor
    end if
end procedure
```

3.3.5 Weighted Regret Matching with Inertia

An approach of giving an algorithm a notion of memory is having each vertex remember a measure of "regret" for not having taken a certain state in the past. This measure is calculated as the difference between the payoff of a state that was not chosen and the payoff of the actually chosen state. For each state in its domain the algorithm remembers the average regret. Each time the algorithm evaluates new states using its target function, the average regret for each state is updated. These averages may be interpreted as the regret of a vertex for not having chosen this state during the last `collect` step.

The mechanic of having an algorithm remembering regrets is known as *regret matching*, of which several variants have been proposed in existing literature. [Marden et al., 2007] proposed a modification to regret matching called *weighted regret matching (WRM)* in which the regret of past state changes is weighed by some pre-defined factor. Since the regrets are updated each collect step, weighing past steps by some factor $M \in [0, 1)$ results in the fact that regrets from the distant past have less impact on the decision of a vertex since their contribution to the total average regret was more often multiplied by some value $\in [0, 1)$.

The target function of WRM-I is given in algorithm 5.

Algorithm 5 The target function of WRM-I

procedure EVALUATE_WRM-I(state)
  $i \leftarrow$ currentStep
  currentRegret $\leftarrow$ utility(state) - utility(currentState)
  discountedRegret $\leftarrow M \cdot$ currentRegret $+ (1 - M) \cdot$ pastRegret(state)
  finalRegret $\leftarrow \max(0, \text{discountedRegret})$
  pastRegret(state) $\leftarrow \frac{1}{i}$(discountedRegret $+ (i - 1) \cdot$ pastRegret(state))
  return (state, finalRegret)
end procedure

In the actual implementation, the weighing factor is called `fadingMemory`. This name resembles the fact that a higher value of `fadingMemory` results in stronger disregard of past events. If `fadingMemory` had a value of 1, only the current regret would influence the decision of a vertex.

After evaluating its potential new states by means of its regret mechanic, the algorithm chooses the next state according to the `argmaxBI` function (algorithm 6). `argmaxBI` is almost identical to `argmaxB` except for the fact that there exists some probability – called *inertia* – that the current state is kept even if a better state has been found. Since WRM was extended by inertia, it is called WRM-I. The reason for employing a mechanic like inertia has to do with the adjustment schedule employed by WRM-I:

The adjustment schedule WRM-I uses is known as the *flood schedule*. This schedule uses the synchronous execution mode of Signal/Collect and always allows the vertices to compute a new and possibly better state. Although a flood schedule does not add anything to the synchronous execution mode of Signal/Collect, the terms are used separately

to comply with existing terminology.

A flood schedule is essentially the same as a parallel random schedule with $p = 1$. As was mentioned in the description of DSA, the parameter p has to be adjusted to minimize thrashing while still allowing a high enough degree of parallel computation. In the case of WRM-I, however, a parallel random schedule can not be used since if the vertices were forbidden to collect, they could not update their average regrets. By using inertia as defined in the context of `argmaxBI`, thrashing can also be minimized while still allowing the vertex to conduct its computation.

Algorithm 6 The decision rule `argmaxBI`

```
procedure decisionRule_argmaxBI(currentPayoff, otherPayoffs)
    allPayoffs ← {currentPayoff} ∪ otherPayoffs
    pMax ← { p ∈ allPayoffs | p = max(otherPayoffs) ∧ p ≥ currentPayoff }
    if pMax = ∅ then
        return stateOf(currentPayoff)
    else if randomElementIn( [0, 1) ) < I then
        return stateOf(currentPayoff)
    else
        return stateOf(randomElementIn(pMax))
    end if
end procedure
```

As another way of countering the risk of multiple vertices acquiring a new state at the same time, a hybrid algorithm called MGM-WRM is benchmarked as well. Like its name suggests, MGM-WRM leaves the mechanic of inertia out in favor of the maximum-gain adjustment schedule. To test how this schedule would affect convergence speed, [Chapman et al., 2011a] benchmarked an algorithm very similar to MGM-WRM. The difference between both algorithms is merely that theirs did not make use of the `fadingMemory` mechanic. The benchmark of MGM-WRM should therefore provide insight on how both of these algorithms differ in performance.

3.3.6 Joint Strategy Fictitious Play with Inertia

In a family of algorithms known as *fictitious play (FP)*, each player keeps track of its neighbors' histories of state changes. While evaluating possible new states by means of its target function, FP takes these historic records into account.

In FP, an agent assumes that, first, its neighbors behave independently from each other. Second, it assumes that they play randomly according to the frequency of states they were in in the past. It has to be noted that these assumptions are clearly false but they are made to approximate the behavior of neighboring vertices.

A complete description of FP is provided by [Marden et al., 2009]. To understand how FP works, the following description shall suffice. The payoff for some state s is computed as follows:

If the neighbors of some vertex i are denoted by $N_i = \langle 1, 2, \cdots, J \rangle$ and each of those neighbors has a domain d_j for $j \in N_i$, then the possible state configurations in the neighborhood of i are: $C_i = d_1 \times \cdots \times d_J$. With this in mind, the payoff $p_i(s)$ for some possible state s is given by:

$$p_i^t(s) = \sum_{c \in C_i} u_i(s, c) \prod_{j \in N_i} P(\text{"}j \textit{ is in state } c_j \textit{ at time } t\text{"})$$

where, in the context of this thesis, $u_i(s, c)$ is the number of constraints satisfied with the theoretical state configuration c of the neighborhood. The fact that each vertex has to store the relative frequencies of states for each one of his neighbors is often not feasible for larger systems. As a less computationally demanding version of FP, an algorithm called *joint-strategy fictitious play with inertia (JSFP-I)* is analyzed in this benchmark.

[Marden et al., 2009] introduced JSFP-I as a lighter version of FP, alleviating the "informational and computational burden" that is an inherent problem with most learning algorithms that have some kind of memory. This problem, of course, can be severe in large-scale systems where the average degree of the graph is high.

As an earlier try to counter the computational demand of FP, [Lambert et al., 2005] designed a variation of FP called *sampled FP* in which each node only recorded the history of a subset of past state frequencies for each neighbor. However, it was found that for fast convergence, the sample size needed grew impractically fast with respect to number of nodes in the graph under consideration.

In contrast to FP and sampled FP, JSFP-I only keeps track of the frequencies of its neighbors' "joint strategies" which translates to a state configurations of neighboring vertices.

In terms of actual implementation, this means that each vertex has to store a hash table which associates for each possible state configuration of its neighborhood a frequency value that gets updated every time a new state configuration is observed.

Using the same notation as before, the target function of FP can therefore be simplified to:

$$p_i^t(s) = \sum_{c \in C_i} u(s, c) P(\text{"}\textit{the neighbors of } i \textit{ have the configuration } c \textit{ at time } t\text{"})$$

As shown by [Chapman et al., 2011a], this equation is equivalent with having each vertex updating a record of the average utility of each state. At the beginning of each collect step, the vertices update their record of average utilities. The set of all average utilities are then passed together with their respective states to the decision rule. Much like WRM-I, JSFP-I uses a flood schedule. Due to the same reasons WRM-I is in need of a mechanic that could minimize thrashing, JSFP-I as well has to make use of inertia. Thus the decision rule `argmaxBI` complements JSFP-I.

3.3.7 Distributed Simulated Annealing

As shown previously, the shape of the probability distribution that underlies the decision rule of SAP can be altered by changing the value of η. *Distributed simulated annealing*

(DSAN) makes use of the fact that if η is gradually lowered during the runtime of the algorithm, the decision rule gets "greedier" until it reaches – at $\eta = 0$ – the behaviour of the `argmax*` family. If DSAN considers a state that would be worse than its current state, there is still a certain probability than DSAN acquires this state. This probability is dependent on η.

The term "simulated annealing" stems from a process similarly called "annealing" used in material sciences. By heating metals above a certain temperature, molecules break their intramolecular bonds and then – while gradually cooling off – reform them, resulting in an often more "fine-tuned" molecular structure.

Simulated annealing is a popular heuristic in optimization problems, since it allows – intuitively speaking – an algorithm to explore its search space without greedily climbing the next local maximum. As the algorithm progresses, η – which is often called *temperature* – is lowered according to function of the current iteration number[4] to speed up its convergence.

A distributed version of simulated annealing for solving DCOPs has been introduced by [Arshad and Silaghi, 2004]. In its introductory paper, DSAN was matched against various DSA variants and exhibited – especially on heavily constrained problems – good performance. Also shown in its authors' experiments, the η parameter has a huge influence on its behavior: they found that in order to compete with DSA, η had to reach values close to 0 relatively quickly. Also, if η was too high initially, the algorithm was too random in its decision what values to pick next and was not able to produce good solutions as quickly as DSA. They found that $\eta_k^c(i) = \frac{c}{i^k}$ where i is the number of the current iteration and c is some constant, is a good temperature function if c is close to 1 and k is set to 2.

In contrast to all other algorithms presented here, DSAN has a different `prospectiveStates` component. Whereas other algorithms evaluate their whole domain, DSAN only evaluates one state that is chosen randomly from its domain. This design decision was made since, first, the authors of DSAN specified the algorithm this way and, second, the choice of a different `prospectiveStates` component allows for more insight on how these components work together.

The evolution of the probability for choosing a state of a given payoff depending on c can be seen in figure 3.2. Note that the target function of DSAN returns simply the utility of the current state minus the utility of the considered state – denoted Δ in the previously mentioned figure. The complete decision rule of DSAN is outlined in algorithm 4.8.

Regarding its adjustment schedule, DSAN, just like many other algorithms, uses a parallel random schedule to counter thrashing.

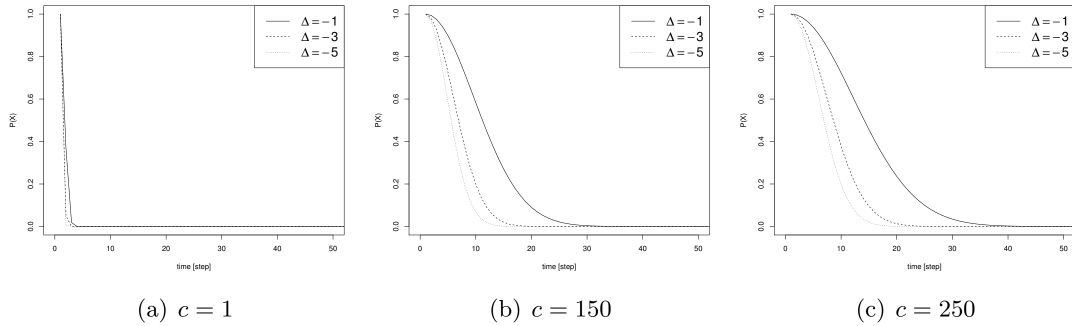
3.3.8 Tabu Search

As a novel addition to the set of algorithms that were modularized and distributed, an algorithm known as *tabu search (TS)* is presented. Tabu search was first introduced

[4] Other measures such as the quality of the current solution are also possible.

Algorithm 7 The decision rule of DSAN

procedure DECISIONRULE_DSAN(payoff)
  if payoff < 0 **then**
    $p \leftarrow e^{\text{payoff} \ / \ \eta^{c}_{k=2}(\text{currentStepNumber})}$
    if randomElementIn($[0, 1)$) < p **then**
      return stateOf(payoff)
    else
      return currentState
    end if
  else
    return stateOf(payoff)
  end if
end procedure



(a) $c = 1$ (b) $c = 150$ (c) $c = 250$

Figure 3.2: The probability of DSAN choosing a state with negative Δ depending on the number of iterations.

by [Glover and McMillan, 1986] and amongst others analyzed by [Nurmela, 1993], on whose descriptions this implementation is based.

Tabu search is related to the family of optimization algorithms that use an `argmax*` decision rule. A problem with algorithms that use the `argmaxB` decision rule is that they allow state changes even if they do not increase the utility. This leads to the possibility of them "jumping" back and forth between states with the same utility. This problem is solved in tabu search by introducing a *tabu list* that stores previously made moves. If the algorithm changes its state s_i to s_j where $i \neq j$, the inverse of this move, $s_j \to s_i$, is added to the tabu list. If the algorithm wants to change back from s_j to s_i, the tabu criterion forbids it and the algorithm has to consider another new state.

This mechanism was slightly altered in the herein used implementation: Instead of just adding the reversed move, both moves, i.e., $s_i \to s_j$ and $s_j \to s_i$, are added to the tabu list. The reason for this decision is the fact that in the paper on which TS is based, the algorithm considered only one state that is close to its current state. However, to test whether TS performed better than its closest relative, DSA-B, TS had to have the

same `prospectiveStates` component. Therefore, if the algorithm was to add only the reversed move, a vertex could still change its state in a cyclic manner: $1 \to 2 \to 3 \to 1 \to 2 \to \ldots$ and so forth.

[Nurmela, 1993] mentioned that the tabu criterion may be too restrictive. For this reason, they extended the classical TS algorithm by providing the possibility of defining an *aspiration level*. If the payoff for some state is higher than the aspiration level, the move is allowed even if it is contained in the tabu list. In preliminary experiments conducted, it was found that defining an aspiration level had indeed a significant impact on the performance of the algorithm. The mechanic of the aspiration level is included by means of an overridable function called `overruleTabuList`. Two of the more promising implementations for `overruleTabuList` that were tested are: A procedure that permitted a move only if its application resulted in a payoff that was higher than the one gotten during the most recent application of the same move. Inspired by the process of simulated annealing, another version overruled the tabu criterion with a probability of $\frac{1}{i}$, where i corresponds to the current iteration number. However, a time-invariant function that overruled the tabu criterion with a time-invariant probability of $\frac{1}{10}$ led to the highest increase in performance.

Algorithm 8 shows the decision rule as well as the default version of `overruleTabuList` of TS. As can be seen by the implementation of TS, a parameter called `stepsToRemember` is used to restrict the number of moves that are kept in the tabu list, thus disabling the tabu criterion for moves made in the (distant) past. Like DSA-B, tabu search uses immediate payoff as its target function as well as a parallel random schedule.

Algorithm 8 The decision rule of TS

```
function overruleTabuList(move)
    if randomElementIn( [0, 1) ) < 0.1 then
        return TRUE
    else
        return FALSE
    end if
end function
procedure decisionRule_ts(currentPayoff, otherPayoffs)
    bestAllowedMoves ← ()
    for all p in otherPayoffs do
        move ← (currentState, stateOf(p))
        isBetterOrEqualCurrent ← p ≥ currentPayoff
        isHigherThanCandidate ← bestAllowedMoves = () ∨ p ≥ bestAllowedMoves_0
        tabuListIsOverruled ← overruleTabuList(move)
        notInTabuList ← ¬contains(tabuList, move)
        if isBetterOrEqualCurrent ∧ isHigherThanCandidate then
            if notInTabuList ∨ tabuListIsOverruled then
                if p > bestAllowedMoves_0 then
                    bestAllowedMoves ← (move)
                else
                    append(bestAllowedMoves, move)
                end if
            end if
        end if
    end for
    if bestAllowedMoves = () then
        return currentState
    else
        chosenMove ← randomElementIn(bestAllowedMoves)
        append(tabuList, chosenMove)
        append(tabuList, reverse(chosenMove))
        if size(bestAllowedMoves) > stepsToRemember then
            removeFirst(tabuList)
        end if
        return targetState(chosenMove)
    end if
end procedure
```

4

Benchmark

It was shown in chapter 3 that the presented algorithms are customizable by different parameters. These parameters include, for example, inertia or the degree of parallel executions. Due to the relatively large amount of algorithms, each with its different possible parameter settings, the best configuration for each algorithm had to be determined before a more comprehensive benchmark could be started. For this reason, a preliminary benchmark was conducted. The following sections present the effects that different parameter values had on their respective algorithms.

The results of the second, more comprehensive benchmark, in which the top contenders of each algorithm family competed against each other, can be found in the second half of this chapter.

4.1 Design

The benchmark to test the effects that values for the various parameters was designed as follows:

All graphs on which the algorithms were run were made of 80 vertices. Since a DCOP consists of many local subproblems involving neighboring vertices, larger graphs with similar topologies should not affect the performance of an algorithm. This was also asserted by preliminary tests. Further, to account for uncertainty due to the probabilistic nature of some algorithms, 5 repetitions were conducted on respectively 10 graphs with slightly different topologies.

Every graph had a mean degree of approximately 14 and thus about 558 edges. The individual degrees of all vertices in the graph were approximately normally distributed[1]. The reason for using a normal degree distribution is that it provides an even field for all algorithms. In contrast, if one consider a graph with a degree distribution following a power law[3], then certain vertices are more involved in the computation than others.

[1] All graphs were generated by a sequence of normally distributed degrees with mean 14 using the *igraph* package developed for the R programming language[2].

[3] In a graph with a degree distribution following a power law, the fraction of vertices with degree k is given by $k^{-\gamma}$, where $2 < \gamma < 3$. In such graphs, most vertices have a very small degree whereas a small number of vertices disproportionally high degrees and thus form central hubs.

With a graph density[4] D of approximately 0.18 the graphs are rather sparse in structure.

$$D = \frac{2|E|}{|V|(|V|-1)} \approx 0.18$$

The domain size of each algorithm was set to 8 and the maximum iteration number to 250. In preliminary tests it was found that these values allowed algorithms that converged more slowly to find the global optimum while still providing results that allowed a clear distinction between runs of algorithms with different parameter configurations.

The metrics to evaluate the performance of each algorithm are the same as used by [Chapman et al., 2011a] in their benchmark of LIBR algorithms. These metrics are as follows:

First, to visualize the convergence behavior of an algorithm, the average solution quality at each time step (iteration number) t – denoted Q_t – was recorded and visualized. Q_t is calculated as the sum over the actual utilities of the vertices divided by the number of constraints they are involved in.

$$Q_t = \frac{1}{|V|} \sum_{i=1}^{|V|} \frac{u_i^t}{|N_i|}$$

Q_t was measured after every single signal/collect cycle. Since all vertices collect simultaneously, their view of their neighbors' states does not correspond to the graphs actual state. For this reason, each vertex was queried about the utility of its *previous* state given its current view of its neighborhood.

In addition to the plots, the averaged metrics for each algorithm are given in table form. The average solution quality over the whole runtime of an algorithm – denoted by $Q_{..}$ – favors algorithms that converge quickly but may ultimately find a lower-quality solution than other algorithms.

$$Q_{..} = \frac{1}{T+1} \sum_{\tau=0}^{T} Q_\tau$$

To highlight algorithms that converge more slowly but find higher-quality solutions eventually, the solution quality at the termination point of an algorithm is measured and denoted by Q_T. Additionally, the ratio of runs in which the algorithm reached the global optimum and the ratio of runs in which it found a Nash equilibrium are denoted by R_{opt} and R_{nash}, respectively. Of the runs in which the algorithm reached the global optimum/any Nash equilibrium, the average number of steps needed to do so is recorded in the form of N_{opt} and N_{nash}. It has to be noted that R_{nash} is calculated by the number of steps after which every vertex in the graph announced that it was not able to satisfy more constraints despite not having satisfied all of its constraints.

As a last measure, the average number of messages each vertex received is given by the metric M_{avg}.

[4] The graph density is a measure of how densely connected a graph is. A completely connected undirected graph has graph density $D = 1$.

For each algorithm, these metrics were averaged over all its individual runs and are therefore denoted by an additional horizontal bar. The best values are written in boldface. This excludes, however, metrics involving Nash equilibria since the information they provide is dependent on other values. For example, a low value for $\overline{N}_{nash}$ does not necessarily hint to a good performance since a low value might be due to an early confinement in a local maxima. Furthermore, the occurrence of ' - ' indicates that the algorithm did not converge in any of its runs and therefore no value for the respective metric could be calculated.

Where appropriate, qualitative comparisons to the benchmark conducted in [Chapman et al., 2011a] are made. It has to be kept in mind that the therein used graphs were of much sparser structure.

4.2 Results – Part One

4.2.1 Distributed Stochastic Algorithm

In section 3.3.1 the family of DSA algorithms was introduced. All versions of DSA use a parallel random adjustment schedule which is parameterizable by its *degree of parallel executions* p – the probability of a vertex computing a potential new state.

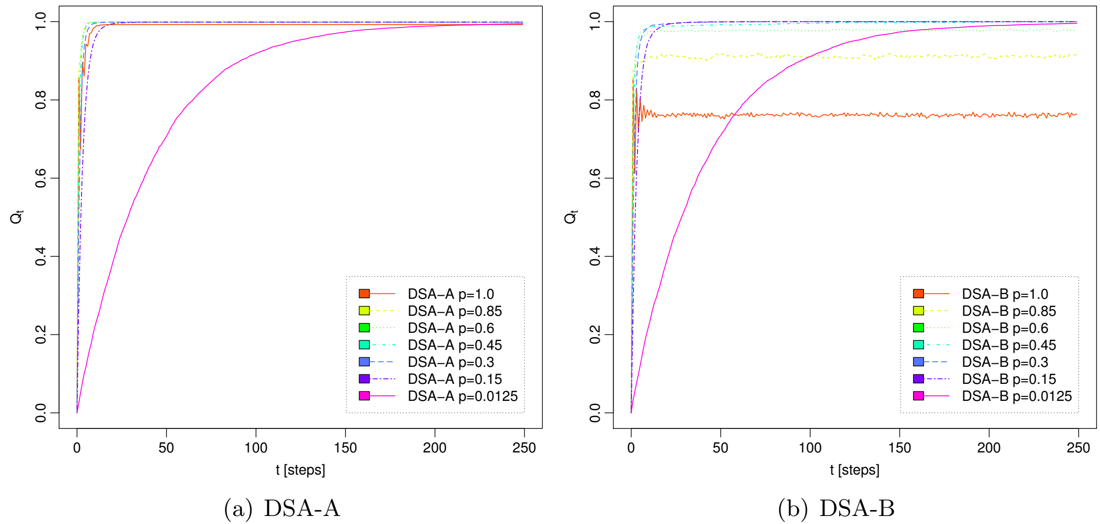
DSA-A

The results of DSA-A are given in table 4.1 and figure 4.1(a). In terms of average solution quality per step, the variant with $p = 0.0125$ is clearly very different from the others. Only after about 150 steps starts DSA-A $p = 0.0125$ approaching $\overline{Q}_{..} = 1$. It has to be noted that in the case of $p = 0.0125$, p equals $\frac{1}{|V|}$ and thus leads to only one vertex being allowed to compute a new state on average. In contrast, the second smallest p value, 0.15, led to much faster convergence. A relatively small difference of $\Delta p = 0.1375$ seems to make a big impact in terms of convergence speed. The slow convergence of DSA-A $p = 0.0125$ also resulted in a rather low average solution quality. On the other hand, higher p values resulted in very high but also very similar average solution qualities. On the other end of the spectrum, DSA-A $p = 1.0$ with its low $\overline{R}_{opt}$ almost never converged to the global optimum.

After $\approx$ 1.5 steps, all vertices of this variant reported that they could not increase their utility anymore and thus got stuck in a state configuration from which they were not able to escape. This observation is in coherence with the low number of average messages exchanged per cycle: since the vertices did not change their state, no state update was issued to their neighbors. Due to the fact that $p = 1$ resulted in all vertices collecting simultaneously, this behavior was to be expected.

The best performing variants were the ones with $0.3 \leq p \leq 0.80$ of which especially 0.6 and 0.8 stood out due to their low average number of steps they needed to reach the global optimum. [Chapman et al., 2011a] found that $p = 0.4$ led to the best overall performance of DSA-A. They further reported that a high degree of thrashing occurred

with $p > 0.8$. A reasonable assumption might be that the combination of domain size and mean degree used in this benchmark leads to a smaller fraction of vertices acquiring the same state at the same time, thus reducing the effect of thrashing.



(a) DSA-A (b) DSA-B

Figure 4.1: DSA parameterized by p

Table 4.1: Experimental results for DSA-A

p	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
1.0	0.9910	0.9930	0.04	1.00	13.5000	1.4808	**2.4672**
0.8	**0.9991**	**0.9994**	**0.68**	1.00	**8.4706**	7.3333	7.1797
0.6	0.9983	0.9986	0.50	1.00	8.6800	7.9067	5.8198
0.45	0.9982	0.9986	0.50	1.00	11.0000	9.7600	5.3787
0.3	0.9977	0.9988	0.54	1.00	15.9630	14.7662	4.9789
0.15	0.9930	0.9987	0.60	1.00	27.5667	26.9500	4.6355
0.0125	0.8454	0.9948	0.10	0.18	228.4000	218.6429	5.0033

DSA-B

The plot of DSA-B, shown in figure 4.1(b), is quite different from DSA-A. Unlike its counterpart, DSA-B $p = 1$ was not able to reach a high solution quality at termination point and deteriorated in oscillatory motions – the characteristic trait of thrashing. Just like in the case of DSA-A, the vertices of DSA-B $p = 1$ reported that an increase in utility is impossible right from the start.

Like with DSA-A, DSA-B $p = 0.0125$ converged the slowest – although a little faster than in the case of DSA-A. Whereas in DSA-A all variants with higher p values had approximately equal average solution qualities, the variants of DSA-B exhibit a noticeable

pattern: starting from $p = 1$, $\overline{Q}_{..}$ and $\overline{Q}_T$ increase while p decreases until, at $p < 0.3$, this effect starts to reverse. Additional tests confirm that $p = 0.3$ proves to be the variant with the best performance in case of DSA-B.

Table 4.2**:** Experimental results for DSA-B

p	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
1.0	0.7616	0.7642	-	1.00	-	1.0000	12.7852
0.85	0.9109	0.9164	-	0.56	-	122.1786	9.4624
0.6	0.9771	0.9779	0.10	0.68	85.0000	90.2820	5.5487
0.45	0.9950	0.9996	0.98	1.00	90.1429	69.6263	**4.7708**
0.3	**0.9978**	**1.0000**	**1.00**	1.00	**32.9200**	30.4600	5.3845
0.15	0.9938	**1.0000**	**1.00**	1.00	36.5200	34.1400	5.4567
0.0125	0.8448	0.9958	0.18	0.24	219.5556	207.6667	5.1536

4.2.2 (Greedy) Spatial Adaptive Play

In section 3.3.2 Spatial Adaptive Play was introduced. As was shown in figure 3.1, the decision rule of SAP is parameterizable by a parameter η which influences the shape of the probability distribution that underlies its decision of what state to pick next.

The fact that the sequential random schedule used by SAP and gSAP only allows exactly one vertex to collect each cycle is clearly visualized by the slow convergence in figure 4.2. This also resulted in a low average solution quality as well as a high number for $\overline{N}_{opt}$ (table 4.3). At about $\overline{Q}_t = 0.7$, the different variants started to diverge clearly. Variants with lower η ascended to solutions of better quality whereas higher values of η did not show any signs of improvement. SAP $\eta = 0.1$ performed the best and reaches the best possible global utility in the majority of runs, although it requires almost 180 steps to do so.

Compared the other non-parameterized algorithm, MGM, gSAP performed worse on almost all accounts except for the number of messages exchanged. Furthermore, the best SAP variant, $\eta = 0.1$, generally outperformed gSAP. These observation stand in marked contrast to the results obtained in [Chapman et al., 2011a] where SAP and gSAP were considered the best performing algorithms. Such a big difference can only be attributed to different benchmark parameters.

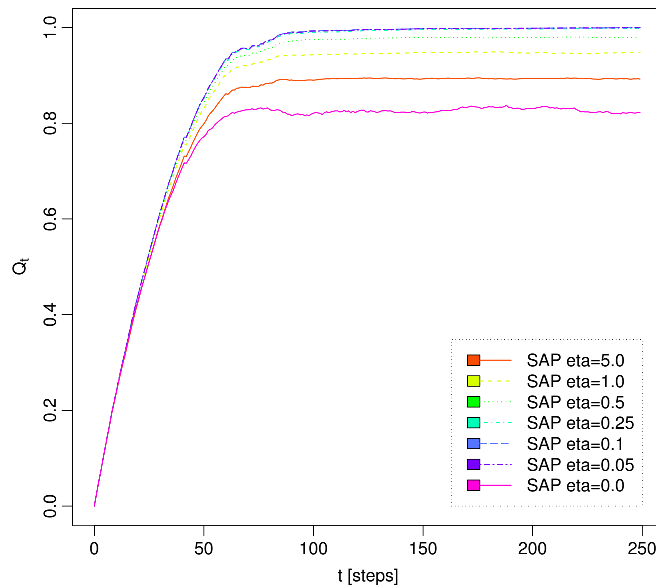


Figure 4.2: SAP with various values for η.

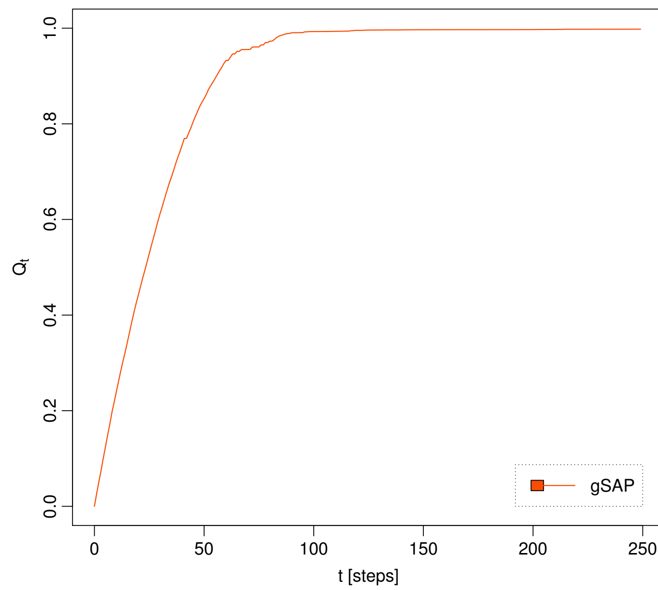


Figure 4.3: gSAP

Table 4.3: Experimental results for SAP

η	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
5.0	0.8133	0.8923	-	-	-	-	10.3120
1.0	0.8577	0.9476	-	-	-	-	7.7370
0.5	0.8826	0.9797	-	-	-	-	5.2145
0.25	0.8962	0.9983	0.52	0.62	195.5000	184.4386	**4.7642**
0.1	**0.8980**	**0.9996**	**0.84**	0.92	180.6429	170.1818	5.0864
0.05	0.8978	0.9995	0.78	0.96	**178.6410**	172.3333	5.0816
0.0	0.7613	0.8228	-	-	-	-	11.2266

Table 4.4: Experimental results for gSAP

$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
0.8970	0.9979	0.32	0.94	163	159.9841	4.7907

4.2.3 Maximum-Gain Messaging

The maximum-main messaging algorithm is one of the few algorithms that is not parameterized. In contrast to algorithms employing a signaling mechanism that simply forwards states, MGM needs two cycles for each state change. This characteristic is visualized by the stairs-like line in figure 4.4. It is also this feature that is in part responsible for the rather low average solution quality of MGM. Still, the algorithm reached the global optimum in half of all conducted runs and did so in approximately 32 cycles. The relatively slow convergence of MGM was also observed by [Chapman et al., 2011a]. A big difference to other algorithms that do not employ the message-intensive maximum-gain mechanic is – as expected – the big number of average messages per cycle.

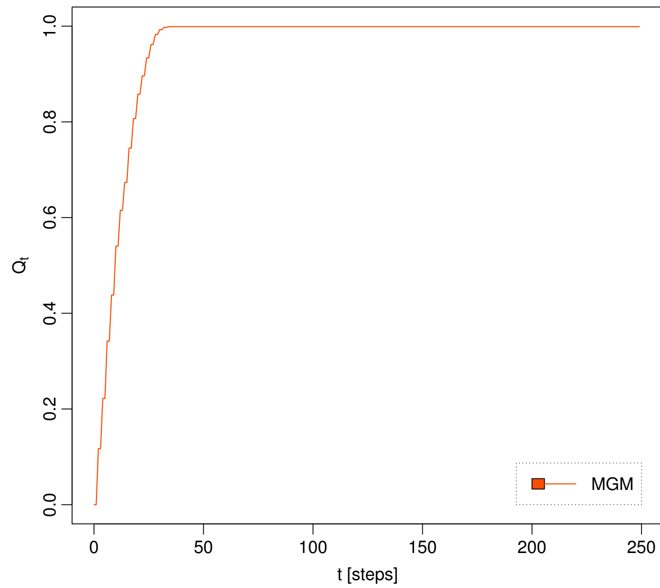


Figure 4.4: Maximum-Gain Messaging Algorithm

Table 4.5: Experimental results for MGM

$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
0.9631	0.9989	0.5	1.00	32.52	32.0933	13.2403

4.2.4 Weighted Regret Matching with Inertia

WRM-I is parameterizable by its inertia and `fadingMemory` value. In preliminary tests it was found that WRM-I showed the best results with an inertia value around 0.4. The effect of varying its memory parameter can be seen in figure 4.5. $M = 0.0$ does not weigh the current regret more strongly than past ones. Table 4.6 shows that this resulted in a rather low average solution quality and a convergence ratio of 0. All other variants performed relatively similar with the big exception of their $\overline{N}_{opt}$ value. High values for `fadingMemory` led to the fastest convergence. Everything considered, $I = 0.8$ showed the best results.

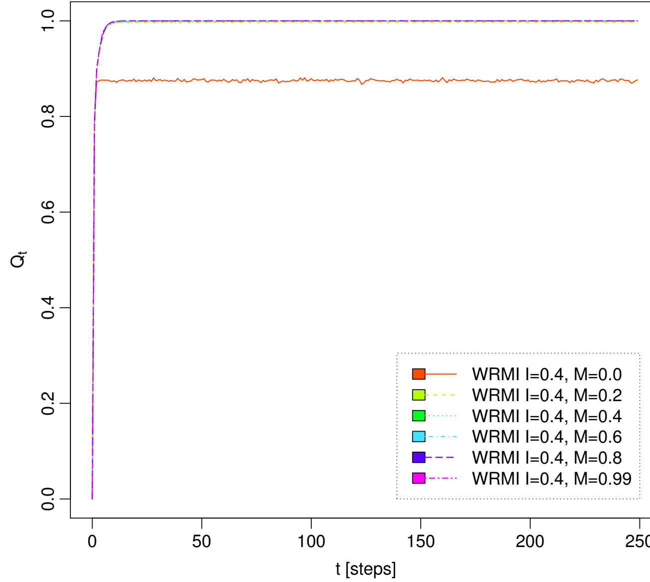


Figure 4.5: WRM-I with $I = 0.4$ and different values for M

Table 4.6: Experimental results for WRM-I

I, M	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
0.4, 0.0	0.8751	0.8765	-	-	-	-	12.1258
0.4, 0.2	0.9968	0.9975	0.36	0.36	13.2222	13.2222	**4.4429**
0.4, 0.4	0.9987	0.9999	0.98	1.00	67.5713	67.7071	4.4628
0.4, 0.6	0.9992	**1.0000**	**1.00**	1.00	18.4000	18.1000	7.2375
0.4, 0.8	**0.9993**	**1.0000**	**1.00**	1.00	**11.6000**	10.9000	8.6856
0.4, 0.99	0.9992	0.9999	0.98	1.00	11.9799	11.2828	8.3889

4.2.5 Maximum-Gain WRM

Figure 4.6 and table 4.7 show the effects a maximum-gain adjustment schedule had on WRM. As was the case with standard WRM-I, a `fadingMemory` value of 0 led to thrashing. Variants with values below 0.5 had generally problems reaching the global optimum often. However, $0.7 \leq M \leq 0.9$ proved to be exceptional in terms of convergence ratio. Although the steps needed to reach the best possible global utility were higher than in the case of standard MGM, they got to it in almost every single run. This observation is different from the one made in [Chapman et al., 2011a]. The maximum-gain regret matching algorithm they analyzed does not weigh current regrets stronger than past ones. They found that it scored worse than standard MGM on all accounts except for M_{avg}. The results of MGM-WRM therefore suggest that letting past regrets influence the decision rule only slightly (i.e., a high value for M) can have a positive effect on MGM.

Of all variants, $M = 0.9$ can be regarded as the top contender since it had the best combination of low $\overline{N}_{opt}$ and high $\overline{R}_{opt}$.

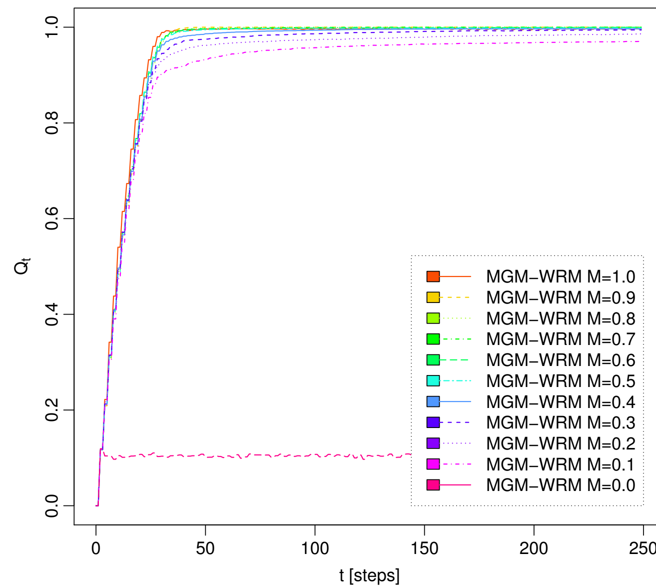


Figure 4.6: MGM-WRM with different values of M

Table 4.7: Experimental results for MGM-WRM

M	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
1.0	0.9606	0.9961	0.66	1.00	41.3636	40.6747	13.2450
0.9	0.9595	0.9999	0.98	1.00	**39.3265**	38.8788	13.3523
0.8	0.9589	**1.0000**	**1.00**	1.00	54.2000	54.1700	13.2638
0.7	**0.9597**	**1.0000**	**1.00**	1.00	81.3200	81.2700	13.2018
0.6	0.9588	0.9998	0.88	0.90	127.0000	127.0786	13.1357
0.5	0.9580	0.9983	0.42	0.42	73.8571	73.8571	13.1135
0.4	0.9542	0.9967	0.24	0.24	118.3333	118.3333	13.0646
0.3	0.9483	0.9942	0.10	0.10	178.2000	178.2000	13.0496
0.2	0.9380	0.9861	-	-	-	-	**13.0472**
0.1	0.9194	0.9703	-	-	-	-	**13.0472**
0.0	0.1042	0.1011	-	-	-	-	**13.0472**

4.2.6 Joint Strategy Fictious Play with Inertia

JSFP-I was run with different values of inertia in the same conditions as the other algorithms. Figure 4.7 shows quite clearly that inertia did not affect the speed of convergence very much. Except for the extreme case like JSFP-I $I = 0.9$, all lines overlap. Looking at the numbers in table 4.8, there is, however, a notable difference in terms of steps needed to reach $Q_t = 1$. Without exceptions, lower inertia values led to a lower value of $\overline{N}_{opt}$, i.e., faster convergence. Although no inertia meant the fastest convergence, the actual number of runs in which the algorithm reached $Q_t = 1$ is the lowest. The version of FP benchmarked in [Chapman et al., 2011a] is equivalent with JSFP-I $I = 0$ and – like here – performed very strongly in terms of convergence speed. They also found that FP had a rather low ratio of complete convergence. The results obtained with JSFP-I $I = 0.1$ suggest that a small value of inertia helps in terms of the number of runs in which the algorithm reaches $Q_t = 1$.

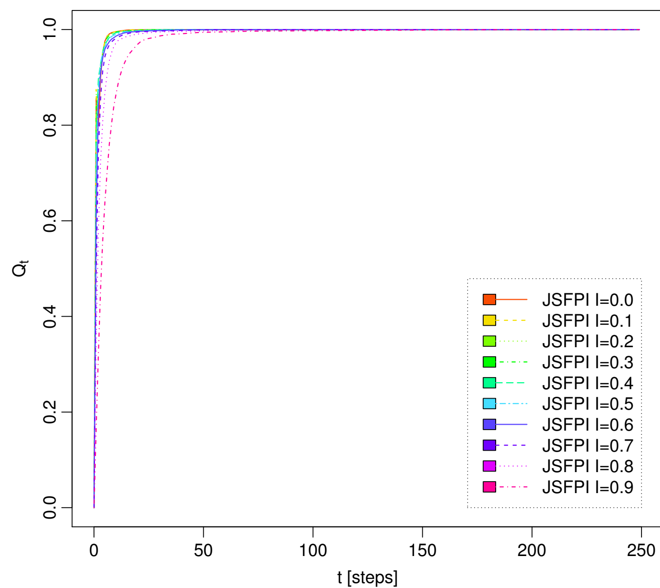


Figure 4.7: JSFP-I with different values of I

Table 4.8: Experimental results for JSFP-I

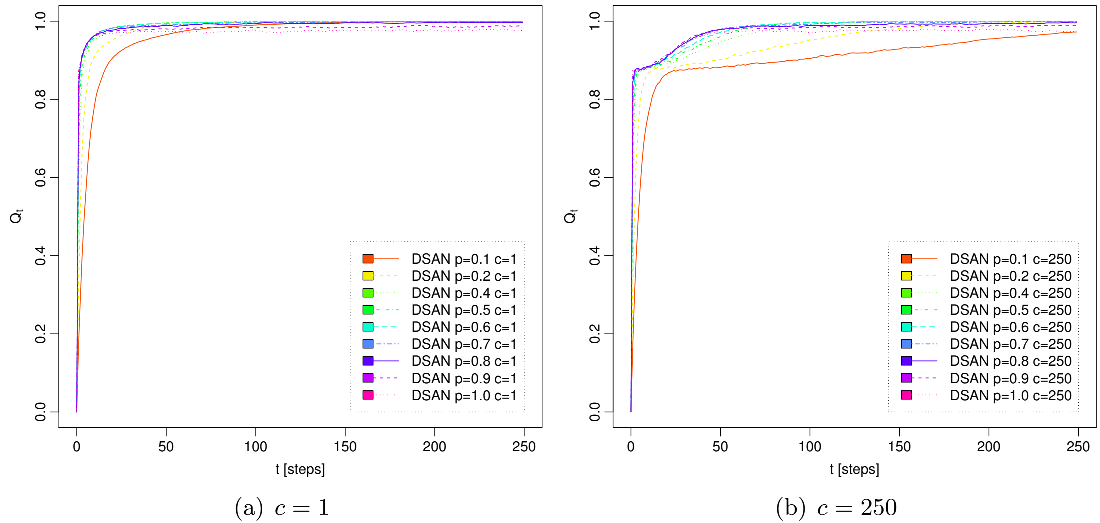
I	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
0.0	0.9988	0.9997	0.86	1	**16.0698**	7.9677	6.7180
0.1	0.9991	**1.0000**	**1.00**	1	17.9600	17.0200	7.0262
0.2	0.9990	**1.0000**	0.98	1	19.2653	19.1111	6.6643
0.3	0.9991	**1.0000**	**1.00**	1	21.0600	20.8800	6.4566
0.4	0.9989	**1.0000**	0.98	1	24.7551	24.7071	6.0112
0.5	0.9988	**1.0000**	**1.00**	1	28.5800	28.3600	5.6438
0.6	0.9983	0.9999	0.96	1	36.3125	36.1224	5.1151
0.7	0.9975	0.9999	0.96	1	44.1458	43.9184	4.8040
0.8	0.9950	**1.0000**	**1.00**	1	75.2600	72.5400	4.0953
0.9	**0.9852**	0.9999	0.98	1	136.3469	137.0808	**3.6988**

4.2.7 Distributed Simulated Annealing

DSAN is parameterized by its value of p and by its time-dependent temperature parameter. As mentioned in section 3.3.7, [Arshad and Silaghi, 2004] found that $\eta^c_{k=2}(i) = \frac{c}{i^2}$, where i is the time measured in steps and c is some constant, performed best. Its authors reported that $c \approx 1$ yielded the best results. To test how c affects the performance of DSAN with the herein used graphs, two extreme cases were tested. First, c was set to 1. Second, c was set to the maximum number of iterations (250). The second parameter, p, was varied for each value of c. Results for all runs can be found in figure 4.8 and table 4.9. It has to be kept in mind that these results can not be compared to the ones obtained in [Arshad and Silaghi, 2004], since the graphs and domain sizes they used are different from the ones in this benchmark.

In the case of $c = 1$, DSAN $p > 0.2$ ascends quickly to $Q_t = 1$ whereas the variants with $p \leq 0.2$ do so only much slower. This behavior was to be expected and is in coherence with all other algorithms. The effect of a higher value for c is clearly seen in figure 4.8(b): After the big boost in solution quality gained from the initial state change of all vertices, Q_t increased only slowly for about 20 cycles. During this time its decision rule acquired states with a worse than its current utility with a relatively high probability.

DSAN converged faster with a low value of c while still having similar convergence ratios. With both values of c, a value of p between 0.4 and 0.7 performed best. Everything considered, DSAN $p = 0.6,\ c = 1$ is the top contender of this family.



(a) $c = 1$ (b) $c = 250$

Figure 4.8: DSAN with different values for p and c

Table 4.9**:** Experimental results for DSAN with different c and p

p, c	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
0.1,1	0.9704	0.9996	0.84	0.96	176.7619	171.0889	4.7792
0.2,1	0.9879	**1.0000**	**1.00**	1.00	118.1200	105.3700	4.6956
0.4,1	0.9942	0.9999	0.98	1.00	78.2245	70.7778	4.6143
0.5,1	0.9948	**1.0000**	**1.00**	1.00	78.5800	70.7200	4.4672
0.6,1	**0.9949**	0.9999	0.98	1.00	**73.7551**	66.5758	4.6239
0.7,1	0.9931	0.9992	0.94	0.98	102.5745	94.6771	**4.2270**
0.8,1	0.9908	0.9968	0.80	0.90	103.0500	100.1882	4.2898
0.9,1	0.9820	0.9894	0.38	0.58	159.5263	151.3125	4.2856
1.0,1	0.9732	0.9755	0.18	0.26	149.7778	158.5454	5.1483
0.1,250	0.9091	0.9724	-	-	-	-	8.6320
0.2,250	0.9519	0.9988	0.66	0.78	215.4242	213.2917	6.6159
0.4,250	0.9752	0.9999	0.98	1.00	141.2653	136.8485	6.4099
0.5,250	0.9798	**1.0000**	**1.00**	1.00	117.7800	112.7000	6.5281
0.6,250	**0.9818**	0.9999	0.98	1.00	**110.4694**	106.5859	6.3454
0.7,250	0.9817	0.9995	0.96	1.00	139.0208	133.5510	5.5985
0.8,250	0.9789	0.9960	0.64	0.82	140.9062	141.3014	5.4067
0.9,250	0.9748	0.9888	0.42	0.60	141.3333	131.9216	**5.3354**
1.0,250	0.9664	0.9734	0.14	0.18	127.5714	129.2500	5.7583

4.2.8 Tabu Search

Figure 4.9 shows how the `tabuList` criterion affects DSA-B. The amount of steps to remember was chosen very high so that its effect is most visible. After the initial state change, all variants were roughly in the same position as they were in the case of DSA-B. Whereas all variants of DSA-B with a high value for p quickly went into oscillatory motions, TS managed to increase its utility regardless of p. Albeit the increase is very slow, the fact that TS forbids vertices to repetitively change their states in the same direction is clearly visible. The average solution quality in table 4.10 supports this observation as well. In general, however, DSA-B performed better than TS if p was chosen around 0.15.

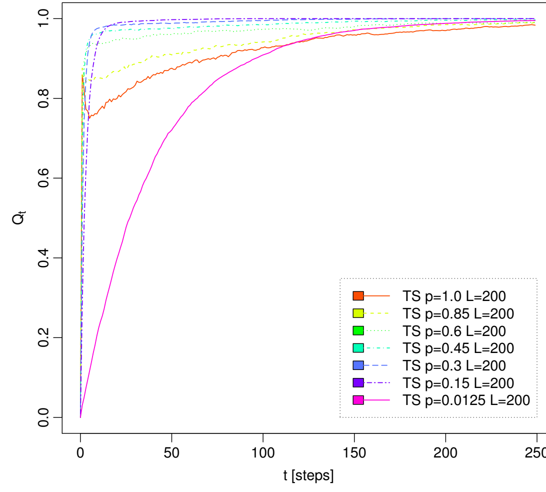


Figure 4.9**:** TS

Table 4.10**:** Experimental results for TS with 200 steps to remember

p	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
1.0	0.9231	0.9836	0.12	1.00	212.1667	23.6250	7.7621
0.85	0.9467	0.9898	0.38	0.48	207.2632	203.2558	6.8919
0.6	0.9734	0.9912	0.46	0.56	186.6087	183.7451	5.3339
0.45	0.9862	0.9969	0.76	0.88	156.0000	149.3293	4.4918
0.3	**0.9936**	**0.9999**	0.98	1.00	100.1224	92.6465	**4.2243**
0.15	0.9928	**1.0000**	**1.00**	1.00	**54.5800**	52.1100	4.8334
0.0125	0.8456	0.9952	0.16	0.18	237.5000	237.5294	5.0893

4.3 Results – Part Two

With the previous benchmark, the best parameter values for each algorithm were identified. The algorithms that performed best are:

- SAP $\eta = 0.1$
- DSAN $p = 0.6, c = 1$
- DSA-A $p = 0.8$
- DSA-B $p = 0.15$
- MGM-WRM $M = 0.9$
- WRM-I $i = 0.4, M = 0.8$
- JSFP-I $I = 0.1$
- MGM
- gSAP
- TS $p = 0.2, R = 100$

For the more comprehensive benchmark, the algorithms were run in three different scenarios: The first two are similar to the previously done benchmark. Identical to the first benchmark is the graph size of 80 and the normal degree distribution with mean 14. For the first scenario of the new benchmark the differences are an increase in the number of graphs from 10 to 50 and 10 instead of 5 repetitions. Furthermore, the maximum iteration number was reduced from 250 to 100. These parameters allow a clearer and more precise distinction between the top contenders of each algorithm family.

To test how the algorithms perform in a harder problem, the second scenario is equivalent to the first one but involves a reduction in domain size from 8 to 7.

The third scenario is quite different from the first two and tries to analyze how the algorithms behave in an extreme environment which is represented by a fully connected graph. To reduce the computational effort while still conducting the same number of repetitions, this graph contains only 50 vertices. The domain size of each vertex is chosen to be equal to $|V|$ to allow complete convergence.

It has to be noted that the specific parameter configurations listed above serve as a rough approximation of what parameter values lead to the best performance for each algorithm. The values that have to be chosen depend, of course, on the structure of the problem. While the first benchmark is almost identical with scenario 1 and quite similar to scenario 2, the scenario featuring a fully connected graph is very different. However, to avoid the immense computational costs of evaluating each algorithm in all configurations and scenarios, this approximation was necessary.

The results of the first scenario are shown in figure 4.10 and table 4.11. As expected, the results are very similar to the first benchmark. SAP and gSAP were the slowest to converge and thus had the worst average solution quality. In contrast to gSAP, SAP had, however, a rather high convergence ratio. Compared to the first benchmark, gSAP improved by about 0.15 in terms of convergence ratio.

The second pair of algorithms that performed very similarly were MGM and MGM-WRM, recognizable by their stairs-like line. MGM managed an increase of about 0.1 in $\overline{R}_{opt}$. Still, MGM-WRM achieved – as before – a much higher convergence ratio even though it had a slightly higher $\overline{N}_{opt}$.

The rest of the algorithms reached high solution qualities very quickly. Both algorithms using `argmaxB`, DSA-B and TS, needed relatively many cycles to reach the global optimum. A notable observation is that TS improved its optimal convergence ratio by almost 0.4, although it was also about 10 steps slower to reach $Q_t = 1$. DSA-A converged the quickest if it did converge. However, it got seemingly often stuck in local maxima. WRM-I and JSFP-I performed the best, as recognizable by their low number of steps needed and high convergence ratio. Especially WRM-I stood out as the clear winner since it had almost the same convergence speed as DSA-A but managed to reach the global optimum in almost all runs.

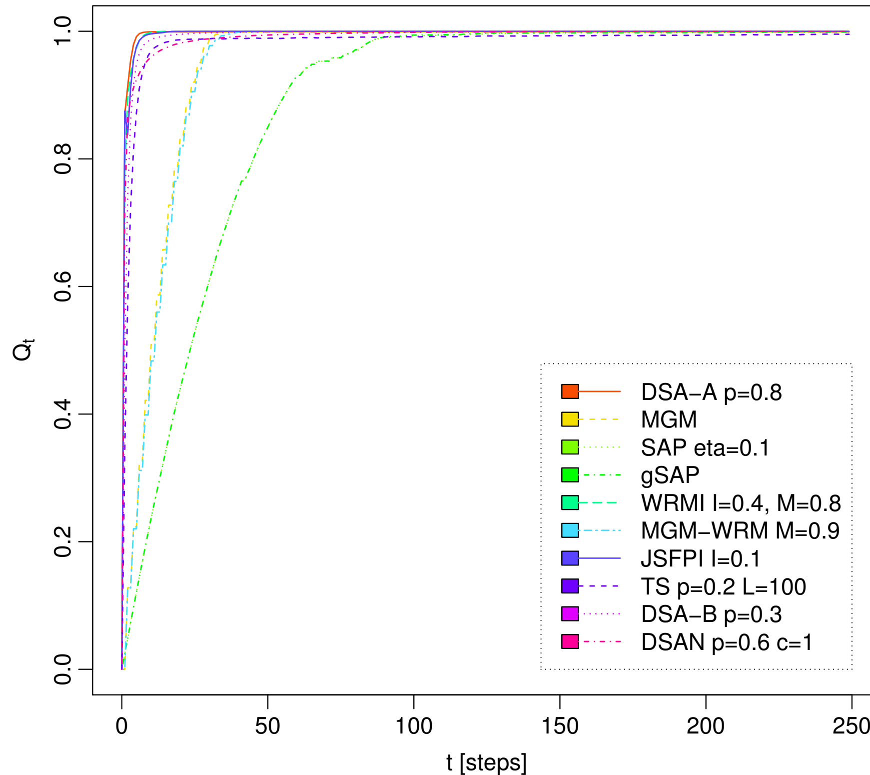


Figure 4.10: Interfamily benchmark, scenario 1

Figure 4.11 and table 4.12 show the results of the second scenario with domain size equal to 7. Except for the ones using `argmaxB`, all algorithms performed worse in this run. The algorithms that were negatively affected by the smaller domain size lost about 0.3 - 0.5 in terms of convergence ratio. This is to be expected since a decrease in domain size results in a smaller solution space because fewer state configurations exist that satisfy all constraints. Also, for many algorithms the reduction in the number of possible paths that lead to the best possible utility meant a higher chance of getting stuck somewhere “along the road”. For this reason, the greediest algorithm, DSA-A, was affected the most by this change. An important observation is that DSA-B performs almost equally

Table 4.11: Experimental results for the interfamily benchmark, scenario 1

algorithm	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
DSA-A (0.8)	0.9989	0.9992	0.6720	**1.000**	8.4643	7.3900	7.1812
MGM	0.9615	0.9990	0.5840	1.000	33.0274	32.7601	13.2525
SAP (0.1)	0.8970	0.9995	0.8000	0.9680	166.1350	160.0113	5.2454
gSAP	0.8969	0.9987	0.4960	0.9760	158.2742	152.1413	5.0783
WRM-I (0.4, 0.8)	**0.9993**	0.9999	0.9800	1.000	**11.7592**	11.1051	8.4952
MGM-WRM (0.9)	0.9601	0.9999	0.9760	1.000	40.8770	39.5162	13.3318
JSFP-I (0.1)	0.9991	0.9999	0.9840	1.000	16.5000	16.2056	7.1668
TS (0.2,100)	0.9885	0.9953	0.6040	0.8200	114.3245	110.3371	**4.2200**
DSA-B (0.3)	0.9979	**1.0000**	**1.0000**	1.000	32.5160	29.4980	5.4861
DSAN (0.6,1)	0.9946	0.9999	0.9913	1.000	82.5504	72.5502	4.3827

well in this run. Also, TS – the other algorithm using `argmaxB` – shows an increase of about 0.3 in $\overline{R}_{opt}$ and is about 30 cycles faster to reach the best possible utility. However, the fact that `argmaxB` allows the exploration of states even if they did not increase the utility has its downsides. This is best visualized by the results obtained in the scenario featuring the fully connected graph:

As can be seen in table 4.13, DSA-B and TS both reached a high solution quality very quickly but failed to satisfy all constraints in every single run. Right before $Q_t = 1$ both algorithms deteriorated in oscillatory movement. Like in the previous benchmarks, MGM and MGM-WRM both converged steadily but only very slowly. As expected, the number of messages exchanged in the case of a maximum-gain schedule as roughly equal to the mean degree of the graph. At the other end of the spectrum, DSA-A clearly outperformed every other algorithm in terms of convergence speed. Another notable contender is JSFP-I that, although slower than DSA-A, exchanged the fewest messages.

Table 4.12: Experimental results for the interfamily benchmark, scenario 2

algorithm	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
DSA-A (0.8)	0.9961	0.9966	0.140	1	**9.5714**	7.5649	**3.1283**
MGM	0.9585	0.9957	0.100	1	34.6800	32.9164	13.0673
SAP (0.1)	0.8932	0.9978	0.320	0.832	206.9375	188.8750	4.6202
gSAP	0.8924	0.9948	0.080	0.856	178.7500	176.1197	4.4455
WRM-I (0.4, 0.8)	**0.9979**	0.9990	0.592	1	16.5338	13.9146	5.4960
MGM-WRM (0.9)	0.9583	0.9988	0.532	1	52.0977	44.9008	13.1657
JSFP-I (0.1)	0.9975	0.9989	0.568	1	22.8310	19.7270	4.7713
TS (0.2, 100)	0.9932	0.9998	0.984	1	84.3537	67.5081	4.3341
DSA-B (0.3)	0.9976	**1.0000**	**1.000**	1	34.4520	28.5420	5.2656
DSAN (0.6,1)	0.9938	0.9999	0.975	1.000	93.1026	79.9620	4.1656

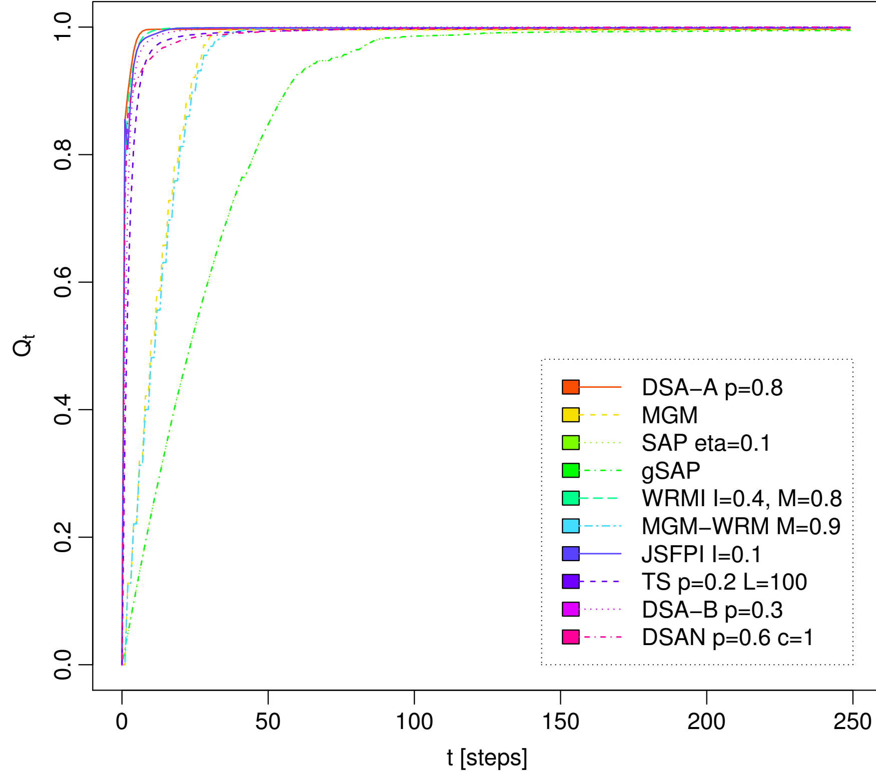


Figure 4.11: Interfamily benchmark, scenario 2

Table 4.13: Experimental results for the interfamily benchmark, scenario 3

algorithm	$\overline{Q}_{..}$	$\overline{Q}_T$	$\overline{R}_{opt}$	$\overline{R}_{nash}$	$\overline{N}_{opt}$	$\overline{N}_{nash}$	$\overline{M}_{avg}$
DSA-B (0.3)	0.9852	0.986	-	-	-	-	29.86326
DSA-A (0.8)	**0.9998**	**1.00**	**1**	1	**12.6**	9.4	21.06578
MGM	0.7888	**1.00**	**1**	1	99.0	99.0	49.75232
SAP (0.1)	0.8905	**1.00**	**1**	1	93.6	93.6	19.25057
gSAP	0.8904	**1.00**	**1**	1	84.0	84.0	19.96314
WRM-I (0.4,0.8)	0.9956	**1.00**	**1**	1	30.2	29.8	18.10620
MGM-WRM (0.9)	0.7888	**1.00**	**1**	1	99.0	99.0	49.75232
JSFP-I (0.1)	0.9958	**1.00**	**1**	1	64.6	63.4	**10.62861**
TS (0.2, 150)	0.9844	0.99	-	-	-	-	26.34560
DSAN (0.6,1)	0.9898	0.9894	-	-	-	-	25.4845

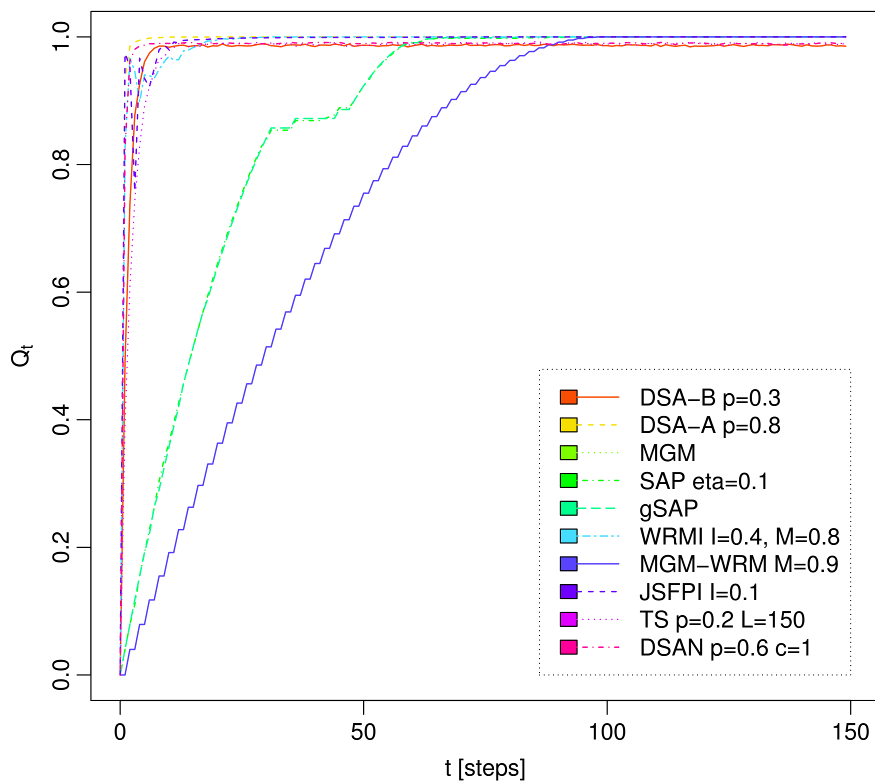


Figure 4.12: Interfamily benchmark, scenario 3

5

Discussion

In the previous chapter the results for multiple benchmarks were presented: First, the parameters with which each algorithm performed best were identified. In a second phase, the algorithms were run with different domain sizes on 50 graphs with normally distributed degrees and, ultimately, on a completely connected graph with $|V| = 50$. The implications these results have on the design of LIBR algorithms are discussed in context of their respective components.

Prospective States

Almost all algorithms that were tested used a function called `CompleteSearch` as their prospective states component. As the name implies, `CompleteSeach` lets a vertex evaluate all states in its domain. Depending on the problem at hand, this function could, however, just as well select only a subset of possible states for evaluation. To adhere to the description of its author, DSAN only evaluated a single random state each cycle. DSAN with $c = 1$ is otherwise very similar to DSA-B. Comparing the two shows clearly how the evaluation of a single state leads to slower convergence. However, it is important to note that such an evaluation strategy also allows for a higher degree of parallel executions since connected vertices are automatically less likely to acquire the same state. DSAN generally outperformed algorithms like SAP and gSAP that rely on a complete search but refrain from parallel execution.

Target Function

Several target functions were used to evaluate the set of prospective states. The simplest target function, immediate payoff, proved to be very effective and was employed by some of the best performing algorithms like DSA-B. In the case of JSFP-I, a target function employing memory led to very fast convergence. The ability of a vertex to predict how its neighbors are likely to change their states, enabled them to "evade" each other and thus minimizing thrashing. The observation of fast convergence is also in coherence with the ones made in [Chapman et al., 2011a].

In the case of WRM-I current regrets were weighed higher according to the chosen value of `fadingMemory`. As it turned out, the factor had to be close to one to allow fast convergence. The biggest contribution of this mechanic to the behavior of the algorithm

was that it allowed the algorithm to break ties between states with otherwise equal regret in a non-probabilistic manner. By this process, a vertex was forced in a specific direction and thus made it possible for its neighbors to adapt and change their states accordingly.

Decision Rule

The decision rules that were tested ranged from very greedy to rather probabilistic. The greediest decision rule, `argmaxA` was employed by DSA-A. In contrast, SAP and DSAN allowed states to be taken that would not immediately maximize their utility. A high degree of leniency when choosing states was found to be detrimental in both convergence ratio, as well as average solution quality. DSA-A, with its greedy optimization, managed to reach much better state configurations in much shorter time. However, its straight race to the next maximum also got it often caught in state configurations, from which it was not able to escape. Allowing vertices to take states with sub-optimal payoff proved to be a good way to escape local maxima if the probability for doing so is not too high: SAP with $\eta = 0.1$ had almost always a higher convergence ratio than DSA-A. Furthermore, as pointed out by [Chapman et al., 2011a], SAP should theoretically achieve the highest solution quality if it is allowed to run for a long time. However, if the goal is to find the solution with the highest quality regardless of the time it takes, other DCOP algorithms such as those from the distributed complete class may be better suited.

By restricting the domain size to 7, the total number of state configurations that could satisfy all constraints was reduced. As a further consequence, the number of local maxima was increased because each vertex had fewer options of changing the global state configuration. These changes affected the performance of DSA-A the most in terms of convergence ratio. It has to be noted, however, that if DSA-A found a path leading to the global optimum, it needed only very few steps to reach it. In the case of a fully connected graph, no state configurations exist that could trap an algorithm like DSA-A. Since every vertex knows all its neighbors, they can react to all their changes very quickly. For problems with such a structure, the straightforward way of optimization employed by DSA-A is unmatched by any of the other herein tested algorithms.

The effect of *argmaxB* is best analyzed by looking at DSA-B. The fact that `argmaxB` lets vertices change to states that yield a payoff equal to the current one, enabled DSA-B to have the highest convergence ratio on the more lightly constrained problems. If the graph was in a state where many vertices evaluate prospective states as having equal utility, the vertices of DSA-A would simply stop collecting. In contrast, because of the the way `argmaxB` works, the vertices of DSA-B would just randomly change states. This can be a good thing if new graph configurations may lead to better initial positions for reaching the global optimum. However, as was seen in the case of a fully connected graph with $|V| = 50$ and an equally large domain size, if a vertex of DSA-B has many choices of what state to pick next, its behavior gets very unpredictable for its neighbors. This ultimately leads to state change loops and no convergence at all. The addition of a `tabuList` did not solve this problem. Its effect was, however, clearly positive in terms of average solution quality. It can not be ruled out that a combination of an appropriate aspiration level and value for `stepsToRemember` would greatly decrease the

looping behavior of DSA-B.

Adjustment Schedule

The degree of thrashing and convergence speed is heavily related to what adjustment schedule was employed. More complex schedules like the sequential random schedule of SAP or the maximum-gain schedule were not able to outperform simpler alternatives like the flood schedule. In the case of a sequential random schedule, allowing only one vertex at a time to alter its state led to very slow convergence. Especially in problems involving a large number of variables, such an adjustment schedule may perform poorly. However, gSAP may still convince due to its anytime property and guarantee of no thrashing. As previously mentioned, [Chapman et al., 2011a] found the SAP family to be very well performing. They stated that the reasons for their good performance may lie in their benchmark setup – the same argument can be applied here.

A maximum-gain schedule led to quite consistent results. By using a decision rule that disallows changes to states of inferior utility, the usage of a maximum-gain schedule provides anytime behavior as well as the guarantee of no thrashing. These properties, however, come at the cost of a high number of messages exchanged. If the cost of exchanging messages is high, such a schedule is a bad choice for problems corresponding to dense graphs. Still, many problems are sparse in structure and could be reliably optimized by a MGM-like algorithm.

The best overall performance was provided by the parallel random schedule or, the very similar, flood schedule combined with a decision rule making use of inertia. The values for I and p had to be chosen for each algorithm separately. With all algorithms, however, values close to 1 or 0 led to either a high degree of thrashing or slow convergence. [Chapman et al., 2011a] stated:

> [...] the random parallel adjustment schedule limits the effects of thrashing, however this trait is only useful in algorithms that use immediate payoffs as a target function, as other target functions use averaging techniques (i.e. beliefs or average regrets) to eliminate thrashing and cycling.

Since the parallel random schedule has the same means of countering the occurrence of thrashing as the use of inertia, this contradicts the results made here. JSFP-I – an algorithm employing a memory-based target function – performed *better* with the addition of a small inertia value than with $I = 0$.

Generally speaking, DSA-B, already known for its strong performance in vertex coloring benchmarks, proved again to be a very strong performer. Its shortcomings were, however, quite obvious when it was faced with a densely connected graph and a high domain size. The introduction of a mechanic like the one employed by tabu search was not able to constrain the chaotic behavior of DSA-B in these cases. The use of different aspiration levels, may, however, still make a tabu list a valuable addition to algorithms like DSA-B. Other algorithms that could be considered as having performed the best were WRM-I and especially JSFP-I.

6

Limitations and Future Work

As shown in previous chapters, many algorithms for solving DCOP problems either already exist or may be newly constructed by combining components identified by [Chapman et al., 2011b] and the herein presented `ProspectiveStates` module.

Furthermore, almost all of these components are parameterizable, thus introducing even more possibilities for variation. This versatility comes at a price, however. The sheer number of possible variations make it difficult to thoroughly analyze algorithms with the goal of detecting algorithms that may prove better at solving DCOPs than others. Furthermore, to test their general performance, the algorithms have to be run on a large amount of different graph topologies. With the amount of repetitions needed for achieving statistically reliable results, the computational intensity increases even further.

Nevertheless, it is my belief that modularizing algorithms is a powerful way for detecting potential synergies between such components and therefore enabling the possibility of discovering useful search heuristics. For this reason, I believe that given a computer system with enough computing power, an evaluation system could be constructed that tries to discover such synergies in a completely automated way. If new algorithms were discovered, a formal investigation of their properties could then follow. Of course, many parameters have no single best value that proves to be the best in all scenarios. Factors such as graph density, graph topology, domain size, etc., heavily influence the best values for each parameter. However, if the experiment was to be chosen in an appropriate manner, data mining techniques could allow to identify non-obvious relations between problem structure and parameter values.

Signal/Collect proved to be a convenient environment for realizing the modularization of DCOP algorthms. The possibility of creating typed graphs should further enhance the possibilities for creating even more variations. Also, in this thesis only adjustment schedules that map to the bulk synchronous model of computation were explored. The asynchronous execution mode of Signal/Collect provides an additional, vast domain of DCOP algorithms to explore.

7

Conclusion

The combination of the framework presented in [Chapman et al., 2011b] and the choice of Signal/Collect as an environment to execute LIBR algorithms proved to be very powerful. It was possible to implement algorithms in a modularized and concise manner, while still providing high performance.

As a further success, the algorithm tabu search was split into its components and made ready for distributed computing. Although tabu search was not able to outperform algorithms like DSA in its current configuration, its high degree of extensibility allows for vast improvements.

Of all possible DCOP algorithms, only a small fraction has been evaluated here. The amount of hybrid algorithms that could be constructed from individual components is almost endless. However, it is also this versatility that provides a huge range of research opportunities. With the contribution of the first comprehensive benchmark of LIBR algorithms in Signal/Collect, this thesis positions itself as a starting point for many more research projects.

References

[Arshad and Silaghi, 2004] Arshad, M. and Silaghi, M. C. (2004). Distributed simulated annealing. *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, 112.

[Chang et al., 2008] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

[Chapman et al., 2011a] Chapman, A. C., Rogers, A., and Jennings, N. R. (2011a). Benchmarking hybrid algorithms for distributed constraint optimisation games. *Autonomous Agents and Multi-Agent Systems*, 22(3):385–414.

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

[cnet2008google, 2008] cnet2008google (2008). Google spotlights data center inner workings.

[Ghemawat et al., 2003] Ghemawat, S., Gobioff, H., and Leung, S.-T. . T. (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM.

[Glover and McMillan, 1986] Glover, F. and McMillan, C. (1986). The general employee scheduling problem. an integration of ms and ai. *Computers & operations research*, 13(5):563–573.

[Lambert et al., 2005] Lambert, T. J., Epelman, M. A., and Smith, R. L. (2005). A fictitious play approach to large-scale optimization. *Operations Research*, 53(3):477–489.

[Maheswaran et al., 2004] Maheswaran, R. T., Pearce, J. P., and Tambe, M. (2004). Distributed algorithms for dcop: A graphical-game-based approach. *Proc. Parallel and Distributed Computing Systems PDCS*, pages 432–439.

[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*, pages 135–146. ACM.

[Marden et al., 2007] Marden, J. R., Arslan, G., and Shamma, J. S. (2007). Regret based dynamics: convergence in weakly acyclic games. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 42. ACM.

[Marden et al., 2009] Marden, J. R., Arslan, G., and Shamma, J. S. (2009). Joint strategy fictitious play with inertia for potential games. *Automatic Control, IEEE Transactions on*, 54(2):208–220.

[Monderer and Shapley, 1996] Monderer, D. and Shapley, L. S. (1996). Potential games. *Games and economic behavior*, 14(1):124–143.

[Nurmela, 1993] Nurmela, K. J. (1993). *Constructing combinatorial designs by local search*. Citeseer.

[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.

[Valiant, 1990] Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

[Yokoo et al., 1992] Yokoo, M., Ishida, T., Durfee, E. H., and Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 614–621. IEEE.

[Young, 2001] Young, H. P. (2001). *Individual strategy and social structure: An evolutionary theory of institutions*. Princeton University Press.

[Zhang et al., 2002] Zhang, W., Wang, G., and Wittenburg, L. (2002). Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance. In *Proceedings of AAAI Workshop on Probabilistic Approaches in Search*.

[Zhang and Wittenburg, 2002] Zhang, W. and Wittenburg, L. (2002). Distributed breakout revisited. In *Proceedings of the National Conference on Artificial Intelligence*, pages 352–358. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

List of Figures

3.1 The multinomial logit function with various values for η. 19
3.2 The probability of DSAN choosing a state with negative Δ depending on the number of iterations. 25

4.1 DSA parameterized by p 32
4.2 SAP with various values for η. 34
4.3 gSAP 35
4.4 Maximum-Gain Messaging Algorithm 36
4.5 WRM-I with $I = 0.4$ and different values for M 37
4.6 MGM-WRM with different values of M 38
4.7 JSFP-I with different values of I 40
4.8 DSAN with different values for p and c 42
4.9 TS 44
4.10 Interfamily benchmark, scenario 1 46
4.11 Interfamily benchmark, scenario 2 48
4.12 Interfamily benchmark, scenario 3 49

List of Tables

4.1 Experimental results for DSA-A 32
4.2 Experimental results for DSA-B 33
4.3 Experimental results for SAP 35
4.4 Experimental results for gSAP 35
4.5 Experimental results for MGM 36
4.6 Experimental results for WRM-I 37
4.7 Experimental results for MGM-WRM 39
4.8 Experimental results for JSFP-I 41
4.9 Experimental results for DSAN with different c and p 43
4.10 Experimental results for TS with 200 steps to remember 44
4.11 Experimental results for the interfamily benchmark, scenario 1 47
4.12 Experimental results for the interfamily benchmark, scenario 2 47
4.13 Experimental results for the interfamily benchmark, scenario 3 48

List of Algorithms

1 The decision rule argmaxB 17
2 The decision rule argmaxA 17
3 The parallel random adjustment schedule 18
4 The maximum-gain schedule 20
5 The target function of WRM-I 21
6 The decision rule `argmaxBI` 22
7 The decision rule of DSAN 25
8 The decision rule of TS 27