Chapter 2
Boosting Distributed Constraint Networks

2.1 Introduction

In combinatorial tree-based search, finding a good labeling strategy is a difficult and tedious task which usually requires long and expensive preliminary experiments on a set of representative problem instances. Performing those experiments or defining realistic input samples is far from being simple for today's large scale real life applications. The previous observations are exacerbated in the processing of distributed constraint satisfaction problems (DisCSPs). Indeed, the distributed nature of those problems makes any preliminary experimental step difficult since constrained problems usually emerge from the interaction of independent and disconnected agents transiently agreeing to look after a set of globally consistent local solutions [FM02].

This work targets those cases where bad performance in the processing of a DisCSP can be prevented by choosing a good labeling strategy i.e., decide on an ordered set of variable and value pairs to branch on, and execute it in a beneficial order within the agents. In the following, we define a notion for the risks we have to face when choosing a strategy and present the new Multi-directional Search Framework or M-framework for the execution of distributed search. An M-portfolio executes several distributed search strategies in parallel and lets them compete to be the first to finish. Additionally, cooperation of the distributed searches is implemented with the aggregation of knowledge within agents. The knowledge gained from *all* the parallel searches is used by the agents for their local decision making in each single search. We present two principles of aggregation and employ them in communication-free methods.

Each DisCSP agent still has access to only a subset of the variables as usual but itself runs several copies of the search process on these variables under different *search contexts*, potentially integrating information across these different contexts. Since these contexts have different indirect information about other agents (based on the messages they have received), this indirectly allows aggregating information across different agents as well.

We apply our framework in two case studies where we define the algorithms M-ABT and M-IDIBT that improve their counterparts ABT [YDIK92] and IDIBT

Y. Hamadi, *Combinatorial Search: From Algorithms to Systems*,
DOI 10.1007/978-3-642-41482-4_2,

[Ham02b] by several orders of magnitude. With these case studies we can show the benefit of competition and cooperation for the underlying distributed search algorithms. We expect the M-framework to be similarly beneficial for other tree-based DisCSP algorithms [HR11, RH05]. The framework presented here may be applied to them in a straightforward way that is described in this chapter.

2.2 Previous Work

The benefit of cooperating searches executed in parallel was first investigated for CSP in [HH93]. They used multiple agents, each of which executed one monolithic search algorithm. Agents cooperated by writing/reading hints to/from a common blackboard. The hints were partial solutions or nogoods its sender had found and the receiver could reuse them in its efforts. In contrast to our work, this multi-agent system was an artifact created for the cooperation. Thus the overhead it produced, especially when not every agent could use its own processor, added directly to the overall performance. Another big difference between Hogg's work and ours is that DisCSP agents do not have a global view of the searches and can thus only communicate what's in their agent-view, which usually captures partial solutions for comparably few variables only.

Later the expected performance and the expected (randomization) risk in portfolios of algorithms was investigated in [GS97, GS01]. No cooperation between the processes was used here. In the newer paper the authors concluded that portfolios, provided there are enough processors, reduce the risk and improve the performance. When algorithms do not run in parallel (i.e., when it is not the case that each search can use its own processor) the portfolio approach becomes equivalent to random restarts [GSK98]. Using only one processor, the expected performance and risk of both are equivalent. In contrast to Gomes and Selman we cannot allocate search processes to CPUs. In DisCSP we have to allocate each agent, which participates in every search, to one process. Consequently, parallelism is in our setting and not an overhead prune artifact. We distribute our computations to the concurrent processes. However, this is done in a different way than in [GS01]; we do not assign each search to one process, but each search is temporarily performed in each process. Or from the other perspective, each agent participates in all the concurrent search efforts at the same time. Thus load-balancing is performed by the agents and not by the designer of the portfolio. In this work we consider agents that do this on a first-come-first-serve basis. Another major difference with Gomes and Selman's work is that we use cooperation (aggregation) between the agents.

Recent work on constraint optimization [CB04] has shown that letting multiple search algorithms compete and cooperate can be very beneficial without having to know much about the algorithms themselves. They successfully use various optimization methods on one processor which compete for finding the next best solutions. Furthermore they cooperate by interchanging the best known feasible solutions. However, this method of cooperation cannot be applied to our distributed

constraint satisfaction settings for two reasons: first, we do not have (or want) a global view to a current variable assignment, and second, we have no reliable metric to evaluate partial assignments in CSP.

Concurrent search in DisCSPs [ZM05, Ham02b, Ham02a] differs from M- in a significant way. These approaches also use multiple contexts in parallel to accelerate search. However, in the named works certain portions of the search space are assigned to search efforts. These works apply divide-and-conquer approaches. In the framework presented here we do not split the search space but let every context work on the complete problem. This makes a significant difference in the application of both concepts; M- is a framework while divide-and-conquer is a class of algorithms. M- requires algorithms to do the work while making use of available resources to try multiple things in parallel. Consequently concurrent search could be integrated in M- by letting multiple concurrent search algorithms (each hosting multiple concurrent searches) run in parallel.

In DisCSP research many ways to improve the performance of search have been found in recent years, including for example, [YD98, BBMM05, ZM05, SF05, MSTY05]. All of the named approaches can be integrated easily in the M-framework. The steps to take in order to do this are described in this chapter. The data structures have to be generalized to handle M contexts, and the search functions and procedures have to integrate an extra *context* parameter during their execution. Depending on the algorithm we may achieve heterogeneous portfolios in different ways. In this work we demonstrate the use of different agent topologies but other properties of algorithms can similarly be diversified in a portfolio. As described in the previous paragraph, the main difference between the work presented here and the named DisCSP research is that we do not provide but require a DisCSP algorithm to serve as input to create an instance of M-.

A different research trend performs "algorithm selection" [Ric76]. Here, a portfolio does not represent competing methods but complementary ones. The problem is then to select from the portfolio the best possible method in order to tackle some incoming instance. [XHHLB07, LBNA+03] applies the previous to combinatorial optimization. The authors use portfolios which combine algorithms with uncorrelated easy inputs. Their approach requires an extensive experimental step. It starts with the identification of the problem's features that are representative of runtime performances. These features are used to generate a large set of problem instances which allow the collection of runtime data for each individual algorithm. Finally, statistical regression is used to learn a real-valued function of the features which allows runtime prediction. In a real situation, the previous function predicts each algorithm's running time and the real instance is solved with the algorithm identified as the fastest one. The key point is to combine uncorrelated methods in order to exploit their relative strengths. The most important drawback here is the extensive offline step. This step must be performed for each new domain space. Moreover a careful analysis of the problem must be performed by the end user to identify key parameters. The previous makes this approach highly unrealistic in a truly distributed system made by opportunistically connected components [FM02]. Finally knowledge sharing is not applicable in this approach.

2.3 Technical Background

In this section we define some notions used later in the chapter. We briefly define the problem class considered, two algorithms to solve them and three metrics to evaluate the performance of these algorithms.

2.3.1 Distributed Constraint Satisfaction Problems

DisCSP is a problem solving paradigm usually deployed in multi-agent applications where the global outcome depends on the joint decisions of autonomous agents. Examples of such applications are distributed planning [AD97], and distributed sensor network management [FM02]. Informally, a DisCSP is represented by a set of variables, each of which is associated with a domain of values, and a set of constraints that restrict combinations of values between variables. The variables are partitioned amongst a set of agents, such that each agent owns a proper subset of the variables. The task is for each agent to assign a value to each variable it owns without violating the constraints.

Modeling a distributed problem in this paradigm involves the definition of the right decision variables (e.g., in [FM02] one variable to encode the orientation of the radar beam of some sensor) with the right set of constraints (e.g., in [FM02] at least three sensors must agree on the orientation of their beams to correctly track a target).

Solving a DisCSP is equivalent to finding an assignment of values to variables such that all the constraints are satisfied.

Formally, a DisCSP is a quadruplet $(\mathsf{X}, \mathsf{D}, \mathsf{C}, \mathsf{A})$ where:

1. X is a set of n variables $X_1, X_2, \ldots, X_n$.
2. D is a set of domains $D_1, D_2, \ldots, D_n$ of possible values for the variables $X_1, X_2, \ldots, X_n$ respectively.
3. C is a set of constraints on the values of the variables. The constraint $C_k(X_{k1}, \ldots, X_{kj})$ is a predicate defined on the Cartesian product $D_{k1} \times \cdots \times D_{kj}$. The predicate is true if the value assignment of these variables satisfies the constraint.
4. $\mathsf{A} = \{A_1, A_2, \ldots, A_p\}$ is a partitioning of X amongst p autonomous processes or agents where each agent A_k "owns" a subset of the variables in X with respect to some mapping function $f : \mathsf{X} \rightarrow \mathsf{A}$, s.t. $f(X_i) = A_j$.

A basic method for finding a global solution uses the distributed backtracking paradigm [YDIK92]. The agents are prioritized into a partial ordering graph such that any two agents are connected if there is at least one constraint between them. The ordering is determined by user-defined heuristics. Solution synthesis begins with agents finding solutions to their respective problems. The local solutions are then propagated to respective children i.e., agents with lower priorities. This propagation of local solutions from parent to child proceeds until a child agent is unable to find a local solution. At that point, a *nogood* is discovered. These elements record inconsistent combinations of values between local solutions, and can be represented as new constraints. Backtracking is then performed to some parent agent and the search proceeds from there i.e., the propagation of an alternative local solution or a

new backtrack. The detection and the recording of inconsistent states are the main features which distinguish distributed backtracking algorithms. This process carries on until either a solution is found or all the different combinations of local solutions have been tried and none of them can satisfy all the constraints. Since these algorithms run without any global management point, successful states—where each agent has a satisfiable local solution—must be detected through some additional termination detection protocol (e.g., [CL85]).

2.3.2 DisCSP Algorithms

As a case study to investigate the benefit of competition and cooperation in distributed search we applied our framework to the distributed tree-based algorithms IDIBT [Ham02b] and ABT [YDIK92].

IDIBT exploits the asynchronous nature of the agents in a DisCSP to perform parallel backtracking. This is achieved by splitting the solution space of the top priority agent into independent sub-spaces. Each sub-space combined with the remaining parts of the problem represents a new sub-problem or context. In each context, the same agent ordering is used. Globally, the search is truly parallel since two agents can simultaneously act in different sub-spaces. At the agent level, search contexts are interleaved and explored sequentially.

This divide-and-conquer strategy allows the algorithm to perform well when the value selection strategy is poorly informed. Besides this parallelization of the exploration, IDIBT uses a constructive approach to thoroughly explore the space by an accurate bookkeeping of the explored states. It does not add nogoods to the problem definition. However, it often requires the extension of the parent-child relation to enforce the completeness of the exploration.

In this work, IDIBT agents use exactly one context to implement (each) distributed backtracking. Please note that we also use contexts but in a different way. We only use them to implement our portfolio of variable orderings. In contrast to [Ham02b] we thus apply each of them to the complete search tree.

IDIBT requires a hierarchical ordering among the agents. Agents with higher priority will send their local solution through *infoVal* messages to agents with lower priority. In order to set up a static hierarchy among agents, IDIBT uses the DisAO algorithm [Ham02b]. In this chapter we do not use DisAO but define an order a priori by hand. However, the DisAO has an extra functionality which is essential for the correctness of IDIBT: it establishes extra links between agents which are necessary to ensure that every relevant backtrack message is actually received by the right agent. In order to prevent this pre-processing of the agent topology with DisAO we changed the IDIBT algorithm to add the required extra links between agents dynamically during search (similar to the processing of *addLink* messages in ABT). Finally we extended the algorithm to support dynamic value selection, which is essential for the aggregation described later in this chapter.

ABT is the most prominent tree-based distributed search algorithm. Just like IDIBT it uses a hierarchy to identify the receivers of messages that inform others of

currently made choices, of the need to backtrack or of the need to establish an extra link. In contrast to IDIBT, ABT uses a nogood store to ensure completeness.

In this work, we used ABT in its original version where the hierarchy of agents is given a priori.

Note that even if IDIBT is used with a single context in our experiments, that does not make it similar to ABT. Indeed, IDIBT does not record nogood, while ABT does. This makes a huge difference between these algorithms.

2.3.3 Performance of DisCSP Algorithms

The performance of distributed algorithms is comparably hard to capture in a meaningful way. The challenge is to find a metric which includes the complexity of the locally executed computations and the need for communication while taking into account the work that can practically be done in parallel. The community has proposed different metrics which meet these requirements.

Non-concurrent Constraint Checks Constraint checks (cc) is an established metric to express the effort of CSP algorithms. It is the number of queries made to constraints whether they are satisfied with a set of values or not. Non-concurrent Constraint Checks (nccc) [GZG+08] apply this metric to a concurrent context. nccc counts the constraint checks which *cannot* be made concurrently. When two agents A and B receive information about a new value from another agent C, they then can check their local consistency independently and thus concurrently. Assuming this costs 10 constraint checks each, it will be 20 cc but only 10 nccc. However, when agent C needs 10 cc to find this value, this is not independent of A and B and will result in 20 nccc and 30 cc respectively.

Sequential Messages Counting messages (mc) is an established method to evaluate the performance of distributed systems. The number of messages is relevant because their transportation often requires much more time than local computations. Analogously to counting cc in distributed systems we also have to distinguish the messages that can be sent concurrently [Lam78]. This also applies to DisCSP [SSHF00]. If an agent C informs two agents A and B of its new value then it uses two messages. However, the two mc will only count as one sequential message (smc) because both are independent and can be sent in parallel. When agent A now replies to this message then we will have two smc (and three mc), because the reply is dependent on the message sent by C. The metric thus refers to the longest sequence of messages that is sent for the execution of the algorithm.

Parallel Runtime Runtime is a popular metric in practice today. It expresses in a clear and easily understandable way the actual performance of an algorithm. Its drawback is that it is hardly comparable when using different hardware. In multitasking operating systems we usually use CPU time in order to capture just the time the considered process requires. Again, in concurrent systems this metric cannot be

applied so easily. We have multiple processes and CPUs which share the workload. In order to capture parallel runtime (pt) we have to track dependencies of computations and accumulate the dependent runtime required by different processes. The longest path through such dependent activities will be the required parallel time. In simulators of distributed systems which run on one processor we can capture the pt in the same way. With every message we transmit the pt required so far. The receiver will add the time it needs to process the message and pass the sum on with the next (dependent) message.

2.4 Risks in Search

Here we present two definitions of *risk* in search. Both kinds of risks motivate our work. We want to reduce the risk of poor performance in DisCSP. The first notion, called *randomization risk*, is related to the changes in performance when the same non-deterministic algorithm is applied multiple times to a single problem instance. The second notion, called *selection risk*, represents the risk of selecting the wrong algorithm or labeling strategy, i.e., one that performs poorly on the considered problem instance.

2.4.1 Randomization Risk

In [GS01] "risk" is defined as the standard deviation of the performance of one algorithm applied to one problem multiple times. This risk increases when more randomness is used in the algorithms. With random value selection, for example, it is high, and with a completely deterministic algorithm it will be close to zero. In order to prevent confusion we will refer to this risk as the randomization risk (R-risk) in the rest of the chapter.

Definition 2.1 The R-risk is the standard deviation of the performance of one algorithm applied multiple time to one problem.

In asynchronous and distributed systems we are not able to eliminate randomness completely. Besides explicitly intended randomness (e.g., in value selection functions) it emerges from external factors including the CPU scheduling of agents or unpredictable times for message passing [ZM03].

Reducing the R-risk leads in many cases to trade-offs in performance [GSK98], such that the reduction of this risk is in general not desirable. For instance, we would in most cases rather wait between one to ten seconds for a solution than waiting seven to eight seconds. In the latter case the risk is lower but we do not have the chance to get the best performance.

Moreover, increasing randomization and thus the R-risk is known to reduce the phenomena of heavy-tail behavior in search [Gom03]. Heavy-tailedness exposes

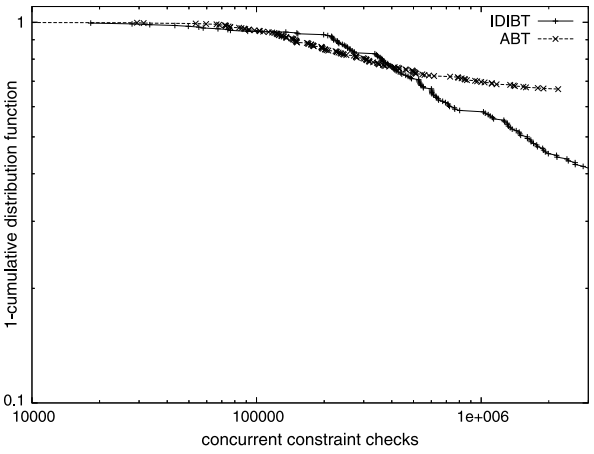

Fig. 2.1 Heavy-tail behavior of IDIBT and ABT

the phenomena that wrong decisions made early during search may lead to extensive thrashing and thus unacceptable performance. In a preliminary experiment we could detect this phenomenon in DisCSP with the algorithms ABT and IDIBT. We used lexicographic variable and value selection to solve 20 different quasigroup completion problems [GW]. A quasigroup is an algebraic structure resembling a group in the sense that "division" is always possible. Quasigroups differ from groups mainly in that they need not be associative.

The problems were encoded in a straightforward model: N^2 variables, one variable per agent, no symmetry breaking, binary constraints only. We solved problems with a 42 % ratio of pre-assigned values, which is the peak value in the phase transition for all orders, i.e., we used the hardest problem instances for our test. Each problem was solved 20 times resulting in a sample size of 400. With ABT we solved problems of order 6 and with the faster IDIBT problems of order 7. Randomness resulted from random message delays and the unpredictable agent activation of the simulator.

The results of this experiment are presented in Fig. 2.1. We can observe a linear decay of the cumulative distribution function of ABT on a log-log scale. For IDIBT, since this algorithm is more efficient than ABT, the linear decay is not visible, but would have been apparent at a different scale, i.e., for the processing of larger problems. The cumulative distribution function of x gives us the probability (y-axis) that the algorithm will perform worse than x. It can be seen that the curves display a Pareto distribution having a less than exponential decay. A Pareto distribution or power law probability distribution is seen in many natural phenomena (wealth distribution, sizes of sand particles, etc.); it implies that the phenomenon under consideration distributes a particular characteristic in an unbalanced way, e.g., 80–20 rule, which says that 20 % of the population controls 80 % of the wealth.

This hyperbolic (i.e., less than exponential) decay is identified on the log-log scale when the curves look linear. This is a common means of characterizing a heavy-tail [Hil75]. Thus, we could (for the first time) observe heavy-tails for both considered DisCSP algorithms in these experiments.

In order to diminish the heavy-tail Gomes and Selman propose the use of random restarts during search. With this technique we interrupt thrashing and restart search once the effort does not seem promising anymore. Nowadays, restart is an essential part of any modern tree-based SAT solver [BHZ06], and is also successfully applied to large scale CP applications [OGD06].

With a central control this decision to restart can be based on information gained from a global view on the search space e.g., overall number of fails or backtrack decisions. In DisCSP we do not have such a global view and could thus only decide locally either to restart or to keep trying. However, the local view may not be informed enough for this decision. In these algorithms different efforts are concurrently made on separate sets of variables. Thus we must face the risk that while one effort may thrash and identify the need to restart, another effort may have almost solved its sub-problem. Furthermore, stopping and restarting a distributed system is costly since it involves extra communication. It requires a wave of messages to tell all agents to stop. After that, global quiescence has to be detected before a restart can be launched. Thus, we do not consider restarts to be very practical for DisCSP.

In [GS01] the authors incorporate random restarts in a different way. When we use a portfolio of algorithms performing random searches in parallel then this can be equivalent to starting all of these algorithms one after each other in a restart setting. They showed that, if one processor is available, the use of portfolios of algorithms or labeling strategies has performance equivalent to the application of random restarts. When we use a portfolio of random searches, running in parallel on the same computational resources, then the expected value of the performance is the same as running these random searches one after each other using random restarts. If we have more than one processor, the performance may increase.

In this chapter we make use of this in order to reduce heavy-tail behavior in DisCSP. We use portfolios as a surrogate of random restarts to reduce the risk of extensive thrashing paralyzing the algorithm. This will reduce the risk of very slow runs and thus reduce the R-risk as well, and improve the mean runtime. The randomness may result from random value selection or from the distribution itself (message transportation and process activation). As we will show in Sect. 2.6 we can avoid heavy-tailedness with this new technique.

2.4.2 Selection Risk

The risk we take when we select a certain algorithm or a heuristic to be applied within an algorithm to solve a problem will always be that this is the wrong choice. For most problems we do not know in advance which algorithm or heuristic will be the best, and may select one which performs much worse than others. We'll refer to this risk as the selection risk (S-risk).

Definition 2.2 The S-risk of a set of algorithms/heuristics A is the standard deviation of the performance of each $a \in A$ applied the same number of times to one problem.

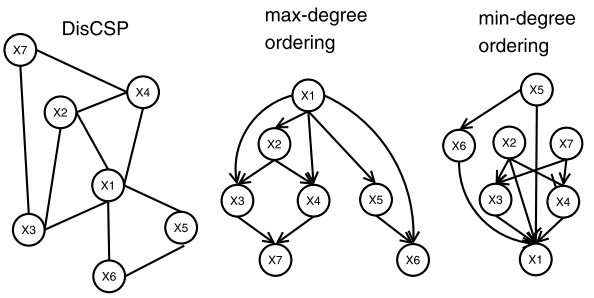

Fig. 2.2 DisCSP (*left*) and agent topologies implied by the variable orderings max-degree (*middle*) and min-degree (*right*)

We investigated the S-risk emerging from the chosen agent ordering in IDIBT in a preliminary experiment on small, fairly hard random problems (15 variables, 5 values, density 0.3, tightness 0.4). These problems represent randomly generated CSPs where the link density between variables is set to 30 %, whereas the tightness density of each constraint is set to 40 %, i.e., 40 % of the value combinations are disabled in each constraint. We used one variable per agent and could thus implement variable orderings in the ordering of agents. We used lexicographic value selection and four different static variable ordering heuristics: a well-known "intelligent" heuristic (namely maxDegree), its inverse (which should be bad) and two different blind heuristics. As expected, we could observe that the intelligent heuristic dominates on average but that it is not always the best. It was the fastest in 59 % of the tests, but it was also the slowest in 5 % of the experiments. The second best heuristic (best in 18 %) was also the second worst (also 18 %). The "anti-intelligent" heuristic turned out to be the best of the four in 7 %. The differences between the performances were quite significant with a factor of up to 5. Applied to the same problems, ABT gave very similar results with a larger performance range of up to factor 40.

2.5 Boosting Distributed Constraint Satisfaction

In DisCSP the variable ordering is partially implied by the agent topology. Neighboring agents will have to be labeled directly one after the other. For example, if each agent hosts one variable then for each constraint a connection between two agents/variables must be imposed. From this follows that the connected variables are labeled directly one after the other because they communicate along this established link. In other topologies where we have inner and outer constraints, naturally only the outer constraints must be implemented as links between agents and we have free choice of variable selection inside the nodes.

For the inter-agent constraints we have to define a direction for each link. This direction defines the priority of the agents [YDIK92] and thus the direction in which backtracking is performed. It can be chosen in any way for each of the existing connections. In Fig. 2.2 we show two different static agent topologies emerging from two different variable ordering heuristics in DisCSP.

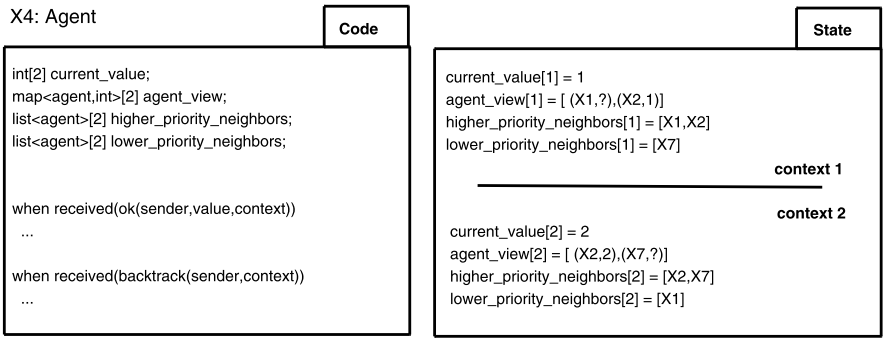

Fig. 2.3 Two contexts for the agent hosting X_4 from Fig. 2.2 resulting from two variable orderings

2.5.1 Utilizing Competition with Portfolios

The idea presented in this chapter is that several variable orderings and thus several agent topologies are used by concurrent distributed searches. We refer to this idea as the M-framework for DisCSP. Applied to an algorithm X it defines a DisCSP algorithm M-X which applies X multiple times in parallel. Each search operates in its usual way on one of the previously selected topologies. In each agent the multiple searches use separate contexts to store the various pieces of information they require. These include, for example, adjacent agents, their current values, their beliefs about the current values of other agents, etc.

In Fig. 2.3 we show how an agent hosting variable X_4 from Fig. 2.2 could employ the two described variable orderings. The figure shows the internal information, and the associated pseudo code. On the right part of the figure, we can see that this agent hosts two different current values, one for each search, and two different agent-views which contain its beliefs about the values of higher-priority agents. The set of these higher-priority agents depends on the chosen topology and thus on the chosen variable ordering. The figure also shows on the left the pseudo code associated with some tree-based search algorithm. There, the functions and procedures are augmented with an extra *context* parameter, which is used to access the right subset of data.

In an M- search, *different* search efforts can be made in parallel. Each message will refer to a context and will be processed in the scope of this context. The first search to terminate will deliver the solution or report failure. Termination detection has thus to be implemented for each of the contexts separately. This does not result in any extra communication, as shown for the multiple contexts of IDIBT in [Ham02b].

With the use of multiple contexts we implement a portfolio of heuristics which is known to reduce the heavy-tail of CSP [GS01]. As we will show in our experiments this is also beneficial for DisCSP. In contrast to random restarts we do not stop any search although it may be stuck due to bad early choices. We rather let such efforts run while concurrent efforts may find a solution. As soon as a solution is detected in one of the contexts all searches are stopped.

Additionally, we can reduce the S-risk by adding more diversity to the portfolio. Assuming we do not know anything about the quality of orderings, the chance of including a good ordering in a set of M different orderings is M times higher than selecting it for execution in one search. When we know intelligent heuristics we should include them but the use of many of them will reduce the risk of bad performance for every single problem instance (cf. experiment in Sect. 2.4.2). Furthermore, the expected performance is improved with the M-framework since always the best heuristic in the portfolio will deliver the solution or report failure. If we have a portfolio of orderings M where the expected runtime of each $m \in M$ is $t(m)$, then ideally (if no overhead emerges), the system terminates after $\min(\{t(m)|m \in M\})$.

2.5.2 Utilizing Cooperation with Aggregation

Besides letting randomized algorithms compete such that overall we are always "as good as the best heuristic" the M-framework can also use cooperation. Cooperation through knowledge sharing is a very powerful concept which allows a collection of agents to perform even better than the best of them. As suggested by Reid Smith, $Power = Knowledge^{Shared}$, where the exponent represents the number of agents whose knowledge is brought to the problem [Buc06]. With this, M-portfolios may be able to accelerate the search effort even more by providing it with useful knowledge others have found. Cooperation is implemented in the aggregation of knowledge *within* the agents. The agents use the information gained from one search context to make better decisions (value selection) in another search context. This enlarges the amount of knowledge on the basis of which local decisions are made.

In distributed search, the only information that agents can use for aggregation is their view of the global system. With multiple contexts, the agents have multiple views, and thus more information available for their local reasoning. Since all these views are recorded by each individual agent within its local knowledge base, sharing inter-context information is costless. It is just a matter of reading in the local knowledge base what has been decided for context c, in order to make a new decision in context c'. In this setting, the aggregation yields no extra communication costs (i.e., no message passing). It is performed locally and does not require any messages or accesses to some shared blackboard.

2.5.3 Categories of Knowledge

In order to implement aggregation we have to make two design decisions: first, which knowledge is used, and second, how it is used. As mentioned before, we use knowledge that is available for free from the internally stored data of the agents. In particular this may include the following four categories:

- *Usage*. Each agent knows the values it currently has selected in each search context.

Table 2.1 Methods of aggregation

	Diversity	Emulation
Usage	*minUsed*: the value which is used the least in other searches	*maxUsed*: the value which is used most in other searches
Support	–	*maxSupport*: the value that is most supported by constraints w.r.t. current agent-views
Nogoods	*differ*: the value which is least included in nogoods	*share:* always use nogoods of all searches
Effort	*minBt*: a value which is not the current value of searches with many backtracks	*maxBt*: the current value of the search with most backtracks

- *Support*. Each agent can store for each search context currently known values of other agents (agent-view) and the constraints that need to be satisfied with these values.
- *Nogoods*. Each agent can store for each search context partial assignments that are found to be inconsistent.
- *Effort*. Each agent knows for each search context how much effort in terms of the number of backtracks it has already invested.

2.5.4 Interpretation of Knowledge

The interpretation of this knowledge can follow two orthogonal principles: *diversity* and *emulation*. Diversity implements the idea of traversing the search space in different parts simultaneously in order not to miss the part in which a solution can be found. The concept of emulation implements the idea of cooperative problem solving, where agents try to combine (partial) solutions in order to make use of work which others have already done.

With these concepts of providing and interpreting knowledge we can define the portfolio of aggregation methods shown in Table 2.1. In each box we provide a name (to be used in the following) and a short description of which value is preferably selected by an agent for a search.

2.5.5 Implementation of the Knowledge Sharing Policies

The implementation of each knowledge sharing policy is rather simple since it only requires regular lookups to other contexts in order to make a decision. More concretely,

- *minUsed*, *maxUsed*. Each value of the initial domain of a local variable is associated to a counter. This counter is updated each time a decision for that variable is

made in any search context. Each counter takes values between 0 and the number of contexts. For each variable, pointers to the min (resp. max) used variables are incrementally updated. During a decision, *minUsed* selects the value which is the least used in other contexts, while *maxUsed* selects the one most used.

- *maxSupport*. Each value of the initial domain of a local variable is associated to a counter. This counter stores the number of supports each value has in other contexts. In order to illustrate this policy, let us consider an example with an inter-agent constraint $X \leq Y$ where X and Y have initial domains $\{a, b, c\}$. Now let us assume that two different agents own the variables, and that the M-framework uses three contexts where $Y = a$ in the first one, and $Y = b$ in the second one. If the agent owning X has to decide about its value in the third context, it will have the following values for the maxSupport counters: $maxSupport(a) = 2$, $maxSupport(b) = 1$, $maxSupport(c) = 0$. It will then select the value a since this value is the most supported w.r.t. its current agent-views. Note that implementing a *minSupport* policy would be straightforward with the previous counters. We did not try that policy, since it does not really make sense from a problem solving point of view.
- *differ*. Each value of the initial domain of a local variable is associated to a counter. This counter is increased each time a nogood which contains a particular value is recorded by ABT in any search context. During a decision, the value with the lowest counter is selected.
- *share*. With this policy, each nogood learnt by ABT is automatically reused in other search contexts.
- *minBt*, *maxBt*. The number of local backtracks performed by the agent in each of the contexts is recorded. Each time a value has to be selected for a particular variable, *minBt* forces the selection of the value used for the same variable in the search with the least number of backtracks. Inversely, *maxBt* forces the selection of the value used in the search with the largest number of backtracks.

As we can see, even the most complex policies only require the association of counters to domains values. These counters aggregate information among search contexts at the agent level. They are updated during decision in any particular context, and used to make better decisions in any other context. Updating these counters can be done naively or incrementally, for instance with the help of some bookkeeping technique.

2.5.6 Complexity

Before presenting the empirical evaluation of M-, we discuss its costs hereafter.

Space The trade-off in space for the application of M- is linear in the number of applied orderings. This is obvious for our implementation (see Fig. 2.3). Thus, it clearly depends on the size of the data structures that need to be duplicated for the contexts. This will include only internal data structures which are related to the

state of the search. M- does not duplicate the whole agent. For instance, the data structures for communication are jointly used by all the concurrent search efforts as shown in Fig. 2.3.

It turned out in our experiments that this extra space requirement is very small. We observed that the extra memory needed with a portfolio of size ten applied to IDIBT is typically only about 5–10 %. For ABT the extra memory when using 10 instead of one context differed depending on the problem. For easy problems, where few nogoods need to be stored the extra memory consumption was about 5–20 %. For hard problems we could observe up to 1,000 % more memory usage of the portfolio. This clearly relates to the well-known space trade-off of nogood recording.

Network Load The trade-off in network load, that is the absolute number of messages, is linear in the portfolio size. When using M parallel contexts that perform one search effort each, we will in the worst case have M times more messages. However, on average this may be less because not all of the M searches will terminate. As soon as one has found a solution the complete system will stop and $M - 1$ search efforts will omit the rest of their messages.

Furthermore, the absolute number of messages is not the most crucial metric in DisCSP. As described earlier, sequential messages are more appropriate. The sequential messages do not increase in complexity because the parallel search efforts are independent of each other such that the number of sequential messages (smc) is the maximum of the smc of all searches in the worst case. On average, however, it will be the smc of the search that is best. Consequently, the smc-complexity when using M-X is the same as the smc-complexity of X.

Using aggregation will not increase the number of required messages because this is performed internally by the agents.

Algorithm Monitoring The complexity of monitoring M-X is the same as it is necessary for the algorithm X. This includes starting the agents and termination detection. Since the number of agents is not increased when using M- we do not need any extra communication or computation for these tasks.

Time The trade-off in computational costs increases with the use of M-. Similar to the increase in absolute messages we have a linear increase in constraint checks. However, looking at non-concurrent constraint checks (nccc), the complexity of X and M-X is the same provided there is no aggregation. The derivation of this conclusion can be made analogously to the derivation concerning smc.

When we use aggregation, however, there may be an increase in computational costs of the agents. Depending on the effort an agent puts in using information it gets from other contexts, this may also increase the number of nccc. This will be analyzed in the next section.

Therefore, the overall cost of M-X is the same as the worst-case complexity of X when we use the concurrent metrics. On average, however, M- will be “as good as the best search heuristic” or even “better than the best” when knowledge sharing techniques are implemented. This will be presented in the next section.

2.6 Empirical Evaluation

For the empirical evaluation of the M-framework we processed more than 180,000 DisCSPs with M-IDIBT and M-ABT. We solved random binary problems (15 variables, 5 values), n-queens problems with n up to 20 and quasigroup completion problems with up to 81 agents.

All tests were run in a Java multi-threaded simulator where each agent implements a thread. The common memory of the whole process was used to implement message channels. Agents can send messages to channels where they are delayed randomly for one to 15 milliseconds. This was done to simulate real world contingencies in messages deliveries. After this delay they may be picked up by their addressee. All threads have the same priority such that we have no influence on their activation and on the computational resources assigned to them by the JVM or the operating system.

In this simulator we implemented the metrics described in Sect. 2.3.3. The absolute number of messages (mc), constraint checks (cc) and backtracks (bt) were counted locally and accumulated after termination of the algorithm. The more sophisticated metrics which reflect the parallelism were computed during the execution of the algorithms. Whenever a message is passed from A to B then A will include its current value of nccc and smc. The receiver takes the maximum of the value and its locally stored values, adds the costs it is now accruing and passes the result on with the next message it sends. After termination of the algorithm we select the maximum of all these values among all agents. Note that there has been recent research which has tried to define alternative performance metrics for DisCSP and DCOP (optimization) problems (see [SLS+08, GZG+08]).

2.6.1 Basic Performance

In Fig. 2.4 we show the median[1] numbers of messages sent and the runtime to find one solution by different sized portfolios on fairly hard instances (density 0.3, tightness 0.4) of random problems (sample size 300). These problems represent randomly generated CSPs where the link density between variables is set to 30 %, whereas the tightness density of each constraint is set to 40 %, i.e., 40 % of the value combinations for the underlying constraint are disabled. No aggregation was used in these experiments. The best known[2] variable ordering (maxDegree) was used in each portfolio, including those of size 1, which are equivalent to the basic algorithms. In the larger portfolios we added instances of lex, random and minDegree and further instances of all four added in this order. For example, 6-ABT would use

[1] We decided to use the median instead of the mean to alleviate the effects of messages interleaving. Indeed, interleaving can give disparate measures which can be pruned by the median calculation.

[2] We made preliminary experiments to determine this.

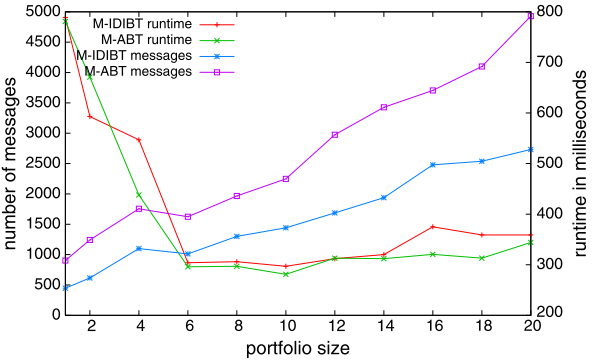

Fig. 2.4 Communication and runtime in M-portfolios

the orders (maxDeg, lex, rand, minDeg, maxDeg, lex). It can be seen that with increasing portfolio size there is more communication between agents. The absolute number of messages rises. In the same figure we show the runtime. It can be seen that the performance improves up to a certain point when larger portfolios are used. In our experimental setting this point is reached with size 10. With larger portfolios no further speed up can be achieved which would offset the communication cost and computational overhead. The same behavior can be observed when considering smc or nccc.

2.6.2 Randomization Risk

The randomization risk is defined as the standard deviation within each sample in our experimental setup. To evaluate it we applied M-IDIBT with homogeneous portfolios 30 times each to a set of 20 hard random problem instances $\langle 15, 5, 0.3, 0.5\rangle$. All portfolios used the same deterministic value selection function and variable ordering (both lexicographic) in all searches. For each problem instance we considered the standard deviation of the 30 runs. Then we took the average of these standard deviations over all 20 problem instances for each portfolio size. This gave us the R-risk that emerges exclusively from the distribution. The results for portfolios sized, 1 to 8 can be seen in Fig. 2.5. It can be seen that all three relevant performance measures (nccc, smc, and pt) decrease with portfolio size increased from 1 to 2. This means the randomization risk decreases when we apply the M-framework. Beyond 2 there is only a slight decrease.

In order to check the influence of the M-framework on the heavy-tail behavior we repeated the experiment described in Sect. 2.4.1 (quasigroup completion of order 6 for ABT and order 7 for IDIBT with 42 % preassigned values, sample size 800) with portfolios of size 10. In Fig. 2.6 we show the cumulative distribution function of the absolute number of backtracks when applying M-ABT and M-IDIBT to the quasigroup completion problems on a log-log scale. It can be seen that both curves decrease in more than a linear manner. As described earlier this implies the non-heavy-tailedness of the runtime distribution of these algorithms.

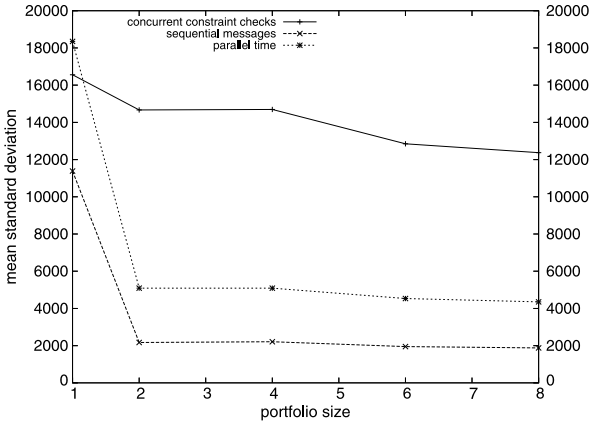

Fig. 2.5 Randomization risk emerging from message delays and thread activation

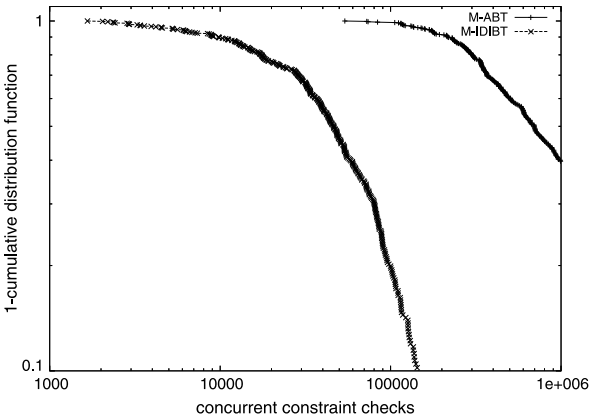

Fig. 2.6 No heavy-tails with M-ABT and M-IDIBT

2.6.3 Selection Risk

To evaluate the selection risk we used a similar experimental setting as before but with heterogeneous variable orderings in the portfolios. We chose to use M different random variable orderings in a portfolio of size M. This would reduce the effects we get from knowledge about variable selection heuristics. The value selection was the same (lexicographic) in all experiments in order to reduce the portion of R-risk as widely as possible and to expose the risk emerging from the selection of a particular variable ordering. In this setting we would get an unbiased evaluation of the risk we take when choosing variable orderings. The mean standard deviation of the parallel runtime for M-ABT and M-IDIBT is shown in Fig. 2.7 on a logarithmic scale. It can be seen that the risk is reduced significantly with the use of portfolios. With portfolio size 20, for instance, the S-risks of M-IDIBT and M-ABT are 344 and 727 times smaller than the ones of IDIBT and ABT, respectively.

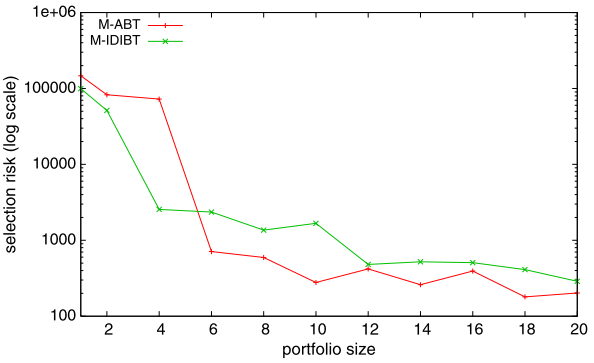

Fig. 2.7 S-risk (standard-dev of the parallel runtime) including the R-risk emerging from distribution

Table 2.2 Performance of aggregation methods for M-IDIBT

	Hard randoms			Quasigroups		
	smc	nccc	pt	$\frac{smc}{1000}$	$\frac{nccc}{1000}$	pt
minUsed	367	2196	1.563	102	1625	448
maxUsed	379	**2118**	**1.437**	40	**635**	182
minBt	392	2281	1.640	104	1330	367
maxBt	433	2541	1.820	43	694	171
maxSupp	**57**	5718	1.922	**1.9**	3727	**143**
random	409	2406	1.664	73	1068	298

2.6.4 Performance with Aggregation

The benefit of aggregation, which is implemented with the different value selection heuristics, is presented in Table 2.2. Each column in the table shows the median values of at least 100 samples solved with M-IDIBT with a portfolio of size 10 applied to 30 different hard random and quasigroup completion problems.

In the table we refer to the aggregation methods introduced in Table 2.1, the bottom line shows the performance with random value selection (and thus no aggregation). When we consider the parallel runtime, it seems that the choice of the best method depends on the problem. For the quasigroup, aggregation based on the emulation principle seems to be better, but not so on random problems.

Interestingly, message passing operations present a different picture. It can be seen that *maxSupport* uses by far the least messages. These operations are reduced by a factor of 7 (resp. 38) for random (resp. quasigroup) problems. However, when we consider parallel time, it cannot outperform the others significantly since our implementation of this aggregation method is relatively costly.[3] However, message passing is the most critical operation in real systems because of either long latencies or high energy consumption (e.g., ad hoc networks [FM02]). This makes the

[3]Bookkeeping could definitely help to reduce the amount of constraint checks in the computation of *maxSupport*.

maxSupport aggregation method really promising. Indeed, there is a clear correlation between the amount of messages sent and the amount of local computations, especially when agents host complex sub-problems. In these situations, since every incoming message may trigger the search of a new solution for the local problem, it is important to restrict message passing.

The performance of *maxSupport* can be explained as follows. It benefits from the efforts in other contexts by capitalizing on compatible values i.e., support relations. As a result this aggregation strategy effectively mixes the partial solutions constructed in the different contexts. It corresponds to an effective juxtaposition of partial solutions.

2.6.5 Scalability

In order to evaluate the relevance of the M-framework we investigated how it scales in larger and more structured problems. For this we applied good configurations found in the previous experiments to the quasigroup completion problem as described earlier in Sect. 2.4.1 (straightforward modeling with binary constraints, most difficult instances with 42 % pre-assignment).

Table 2.3 shows the experimental results of distributed search algorithms on problems of different orders (each column represents an order). ABT and IDIBT used the domain/degree (domDeg) variable ordering [BR96], which was tested best in preliminary experiments. In the larger portfolios we used domain/degree and additional heuristics including maxDegree, minDomain, lex and random. In all portfolios aggregation with the method *maxUsed* was applied.[4] For each order (column) we show the median parallel runtime (in seconds) to solve 20 different problems (once each) and the number of solved problems. When less than 10 instances could be solved within a time-out of two hours we naturally cannot provide meaningful median results. In the experiments with M-ABT we have also observed runs which were aborted because of memory problems in our simulator. For order 8 these were about one third of the unsolved problems, for order 9 this problem occurred in all unsuccessful tests. This memory problem arising from the nogood storage of ABT was addressed in [BBMM05] and is not the subject of this research.

From the successful tests it can be seen that portfolios improve the median performance of IDIBT significantly. In the problems of order 7 a portfolio of 10 was 28 times faster than the regular IDIBT. Furthermore, portfolios seem to become more and more beneficial in larger problems as the portfolio of size 10 seems to scale better than the smaller one. ABT does not benefit in the median runtime but the reduced risk makes a big difference. With the portfolio of size 10, we could solve 17 instances of order 7 problems whereas the plain algorithm could only solve one.

[4] We decided to use this method since it was shown to minimize nccc on previous tests (see Table 2.2).

Table 2.3 Median parallel runtime (pt) and instances solved (out of 20) of quasigroup completion problems with 42 % pre-assigned values

	5	6	7	8	9
ABT	**0.3**, **20**	–, 8	–, 1	–, 0	–, 0
M-ABT, size 5	0.5, 20	5.9, 19	35.8, 14	–, 2	–, 0
M-ABT, size 10	0.6, 20	**6.1**, 20	**40.6**, 17	–, 8	–, 1
IDIBT	1.8, 20	12.4, 20	234, 20	4356, 16	–, 5
M-IDIBT, size 5	**0.2**, **20**	**0.9**, **20**	9.3, 20	709, 20	–, 6
M-IDIBT, size 10	0.3, 20	1.7, 20	**8.2**, **20**	**339**, **20**	–, 8

Table 2.4 Idle times of agents in DisCSP

Problem class	Idle time of agents			
	ABT	IDIBT	M-ABT	M-IDIBT
Easy random	87 %	92 %	56 %	47 %
Hard random	92 %	96 %	39 %	59 %
n-queens	91 %	94 %	48 %	52 %
Hard quasigroups	87 %	93 %	28 %	59 %

2.6.6 Idle Time

To complete the presentation of our experimental results let us consider time utilization in distributed search. It appears that agents in both considered classical algorithms under-use available resources. This is documented in the first two columns of Table 2.4 for various problem classes. The numbers represent the average idle times (10–100 samples) of the agents. In our simulator we captured the idle times of each agent separately. Each agent accumulates the time it waits for new messages to be processed. Whenever an agent finishes processing one message and has no new message received it starts waiting until something arrives in its message channel. This waiting time is accumulated locally. After termination of the algorithm we take the mean of these accumulated times of all agents to compute the numbers shown in Table 2.4.

We can observe that ABT (Asynchronous BackTracking) and IDIBT (Interleaved Distributed Intelligent BackTracking) are most of the time idle. This idleness comes from the inherent disbalance of work in DisCSPs. Indeed, it is well known that the hierarchical ordering of the agents makes low-priority agents (at the bottom) more active than high-priority ones. Ideally the work should be balanced. Thus, ideally one agent on the top of the hierarchy in context 1 should be in the bottom in context 2, e.g., see agent in charge of variable X_1 in Fig. 2.2. Obviously, since we use well-known variable ordering heuristics we cannot enforce such a property. However, the previous is an argument for M-, which can use idle time "for free" in order to perform further computations in concurrent search efforts. This effect is

shown in the last two columns of the table, where the M-framework with a portfolio of size 10 is applied to the same problems. These algorithms make better use of computational resources. Certainly it is not a goal to reduce idleness to a minimum since the performance of our algorithm also depends in the response times of the agents, which may become very long with low idleness. However, without having studied this intensively we are convinced that a mean idleness of more than 90 % is not necessary for fast responses.

2.7 Summary

We have presented a generic framework for the execution of DisCSP algorithms. It was tested on two standard methods but any tree-based distributed search should easily fit in the M-framework. The framework executes a portfolio of cooperative DisCSP algorithms with different agent orderings concurrently until the first of them terminates. In real (truly distributed) applications, our framework will have to start with the computation of different orderings. The generic Distributed Agent Ordering heuristic (DisAO) [HBQ98] could easily be generalized at no extra message passing cost to concurrently compute several distributed hierarchies. The main idea is to simultaneously exchange multiple heuristic evaluations of a sub-problem instead of one.

Heterogeneous portfolios are shown to be very beneficial. They improve the performance and reduce the risk in distributed search. With our framework we were able to achieve a speed up of one order of magnitude while reducing the risk by up to three orders of magnitude compared to the traditional execution of the original algorithm. The chances of extensive thrashing due to bad early decisions (so-called heavy-tails) are significantly diminished.

A portfolio approach seems to make better use of computational resources by reducing the idle time of agents. This is the first of two special advantages of the application of portfolios in DisCSP: we do not have to artificially introduce parallelism and the related overhead but can use idle resources instead. The M-framework can be seen as a solution to the classical "work imbalance" flaw of tree-based distributed search.

We analyzed and defined distributed cooperation (aggregation) with respect to two orthogonal principles, *diversity* and *emulation*. Each principle was applied without overhead within the limited scope of each agent's knowledge. This is the second special advantage of using portfolios in DisCSP: aggregation made at the agent level yields no communication costs and preserves privacy [GGS07]. Our experiments identified the emulation-based *maxSupport* heuristic as the most promising one. It is able to efficiently aggregate partial solutions, which results in a large reduction in message passing operations.

In the next chapter we will see that the ideas developed here can be applied in the context of parallel satisfiability.

http://www.springer.com/978-3-642-41481-7

Combinatorial Search: From Algorithms to Systems

Hamadi, Y.

2013, XIII, 139 p. 30 illus., 16 illus. in color., Hardcover
ISBN: 978-3-642-41481-7