Youssef Hamadi

Combinatorial Search: From Algorithms to Systems

Springer

Combinatorial Search: From Algorithms to Systems

Youssef Hamadi

Combinatorial Search: From Algorithms to Systems

Springer

Youssef Hamadi
Microsoft Research Cambridge
Cambridge, UK

ISBN 978-3-642-41481-7 ISBN 978-3-642-41482-4 (eBook)
DOI 10.1007/978-3-642-41482-4
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013953692

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

To solve a problem as efficiently as possible, a user selects a type of solver (MIP, CP, SAT), then defines a model and selects a method of resolution. The model expresses the problem in a way understandable for the solver. The method of resolution can be complete (one is certain not to miss solutions) or incomplete (it uses a heuristic, i.e., a method that favors the chances of finding a solution but offers no completeness guarantee).

Since solvers exist, researchers try to simplify the task of the end user, helping her in these keys steps: the creation of the model, and the finding of a method of resolution. In this book, Youssef Hamadi helps the user on the second point by presenting ways to automatically select and adjust resolution strategies.

This book proposes several methods for both SAT and CP solvers. Firstly, the author demonstrates the benefit of parallelism through the duplication of search strategies. In the best case, this can provide super linear speed up in the resolution process. In most cases, this results in a more robust resolution method, to the point that such a solver is never beaten by a solver using the best method. The solver ManySAT, co-developed by Mr. Hamadi, is based on this idea and has won numerous prizes in SAT competitions. Its fame goes far beyond the SAT solving domain and this line of work is now a reference for the domain.

Any resolution method must be guided by the user through the definition of a resolution strategy which typically defines the next decision to be made, i.e., which variable must be assigned to which value? This book considers the automatic learning of the parameters of resolution strategies. It shows how to extract knowledge from the information available during search. The difficulty is to determine the relevant information and decide how they can be exploited. A particularly novel approach is proposed. It considers the successive resolutions of similar problems to gradually build an efficient strategy.

This is followed by the presentation of Autonomous Search, a major contribution of the book. In that formalism, the solver determines itself the best way to find solutions. This is a very important topic, which has often been approached too quickly, and which is finally well defined in this book. Many researchers should benefit from this contribution.

This book is fun to follow and the reader can understand the continuity of the proposed approaches. Youssef Hamadi is able to convey his passion and conviction. It is a pleasure to follow him on his quest for a fully automated resolution procedure. The problem is gradually understood and better resolved through the book.

The quality, diversity and originality of the proposed methods should satisfy many readers and this book will certainly become a reference in the field. I highly recommend its reading.

Nice, France Jean-Charles Régin
September 2013

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Combinatorial search algorithms are typically concerned with the solving of NP-hard problems. Such problems are not believed to be solvable in general. In other words there is no known algorithm that efficiently solves all instances of NP-hard problems. However, tractability results from complexity theory along decades of experimental analysis suggest that instances coming from practical application domains can often be efficiently solved. Combinatorial search algorithms are devised to efficiently explore the usually large solution space of these instances. They rely on several techniques able to reduce the search space to feasible regions and use heuristics to efficiently explore these regions.

Combinatorial search problems can be cast into general mathematical definitions. This involves finding a finite set of homogeneous objects or variables whose state must satisfy a finite set of constraints and preferences. Variables have a domain of potential values, and constraints or preferences are used to either restrict or order combinations of values between variables. Dedicated algorithms are able to efficiently enumerate combinations or potential solutions over these definitions.

There are several mathematical formalisms used to express and tackle combinatorial problems. This book will consider the Constraint Satisfaction Problem (CSP) and the Propositional Satisfiability problem (SAT), two successful formalisms at the intersection of Artificial Intelligence, Operations Research, and Propositional Calculus. Despite the fact that these formalisms can express exactly the same set of problems, as proved by complexity theory, they can be differentiated by their *practical* degree of expressiveness. CSP is able to exploit more general combinations of values and more general constraints; SAT on the other hand focuses on Boolean variables, and on one class of constraints. These degrees of expressiveness offer different algorithmic trade-offs. SAT can rely on more specialized and finely tuned data structures and heuristics. On the other hand, algorithms operating on CSP modeling have to trigger different classes of constraints and variables and therefore have to deal with the associated overhead. These algorithms or constraint solvers, if different, are based on the same principles. They apply search space reduction through inference techniques, use activity-based heuristics to guide their exploration, diversify their search through frequent restarts, and often learn from their mistakes.

Y. Hamadi, *Combinatorial Search: From Algorithms to Systems*,
DOI 10.1007/978-3-642-41482-4_1, © Springer-Verlag Berlin Heidelberg 2013

This book focuses on Knowledge Sharing in combinatorial search, the capacity to generate and exploit meaningful information during search. Meaningful information is made of redundant constraints, heuristic hints, and performance measures. It can be used at different levels to drastically improve the performance of a constraint solver. Typically, information can be shared between multiple constraint solvers simultaneously working on the same instance, or information can help solvers to achieve good performance while solving a large set of related instances.

In the first case, multiple constraint solvers are working on the same instance, and information sharing has to be performed at the expense of the underlying search effort, since a solver has to stop its main effort to prepare and communicate the information to other solvers. On the other hand, not sharing information can incur a cost for the whole system by having solvers potentially exploring the unfeasible spaces discovered by other solvers.

In the second case, sharing performance measures can be done with little overhead, and the goal is to be able to acutely tune a constraint solver in relation to the characteristics of a new instance. This corresponds to the selection of the most suitable algorithm for solving a given instance [Ric75].

The book contains two main parts. In Chaps. 2, 3, and 4, portfolios of distributed and parallel algorithms are presented. The reading of Chap. 2 is essential to understand the notions of selection and randomization risks in combinatorial search. These risks explain and motivate parallel portfolio solvers. Chapters 5 and 6 present the benefit of using learning mechanisms during or between search efforts. They can be read independently. Finally, Chap. 7 unifies the previous chapters into the new Autonomous Search framework.

Chapter 2 presents portfolios of distributed CSP algorithms [YDIK92] which demonstrate that competition and cooperation through knowledge sharing can improve the performance of existing distributed search techniques by several orders of magnitude. We show that a portfolio approach makes better use of computational resources by reducing the idle time of agents. It allows search agents to simultaneously work at different tree search levels and provides a solution to the classical work imbalance problem of distributed backtracking. This is achieved through the selective sharing of heuristic hints and decisions. It also shows the value of knowledge sharing to significantly speed up search and provide portfolios whose performance is better than any constituent algorithm.

The previous notions are then applied to the important problem of parallel propositional satisfiability in Chap. 3. This chapter presents the knowledge sharing aspects of ManySAT, the first parallel SAT portfolio built on lessons learned from portfolios of distributed CSP algorithms. In ManySAT different modern SAT solvers are organized around a cooperative framework to quickly solve a given instance. They exchange redundant constraints through advanced control mechanisms which adjust the level of cooperation in relation with the perceived relevance of the information exchanged.

Chapter 4 considers parallel local search algorithms for the problem of propositional satisfiability. This work is motivated by the demonstrated importance of

clause sharing in the performance of complete parallel SAT solvers. Unlike complete solvers, efficient local search algorithms for SAT are not able to generate redundant clauses during their execution. In our settings, each member of the portfolio shares its best configuration (i.e., which minimizes conflicting clauses) in a common structure. At each restart point, instead of classically generating a random configuration to start with, each algorithm aggregates the shared knowledge to carefully craft a new starting point. We present several aggregation strategies and evaluate them on a large set of problems. Our techniques improve the performance of a large set of local search algorithms.

In Chap. 5, our objective is to heuristically discover a simplified form of functional dependencies between variables called weak dependencies. Once discovered, these relations are used to rank branching decisions. Our method shows that these relations can be detected with some acceptable overhead during constraint propagation. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic. Experiments on a large set of problems show that, on average, the search trees are reduced by a factor of three while runtime is decreased by one third.

Chapter 6 presents Continuous Search (CS). In CS, we interleave two functioning modes. In exploitation mode, the instance submitted by the user is processed by the constraint solver; the current heuristics model is used to parameterize the solver depending on the instance at hand. In learning or exploration mode, CS reuses the last submitted instance, running other heuristics than the one used in exploitation mode in order to find which strategy would have been most efficient for this instance. New information is thus generated and exploited in order to refine the heuristics model, in a transparent manner: without requiring the user's input and by only using the idle computer's CPU cycles. CS acts like an autonomous search system able to analyse its performances and gradually correct its search strategies.

In Chap. 7, we leverage knowledge sharing mechanisms in the unified Autonomous Search framework. We define autonomous solvers as solvers that contain control in their search process, and study such autonomous systems w.r.t. their specific control methods. A control process includes a strategy that manages the modification of some of the solver's components and behavioral features after the application of some solving functions. The overall strategy to combine and use components and parameters can be based on learning that uses meaningful information from the current solving process and/or from previously solved instances. This chapter proposes a taxonomy of search processes w.r.t. their computation characteristics, and provides a rule-based characterization of autonomous solvers. This allows a formalizing of solver adaptation and modification with computation rules that describe the modification of the solver's component transformations.

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

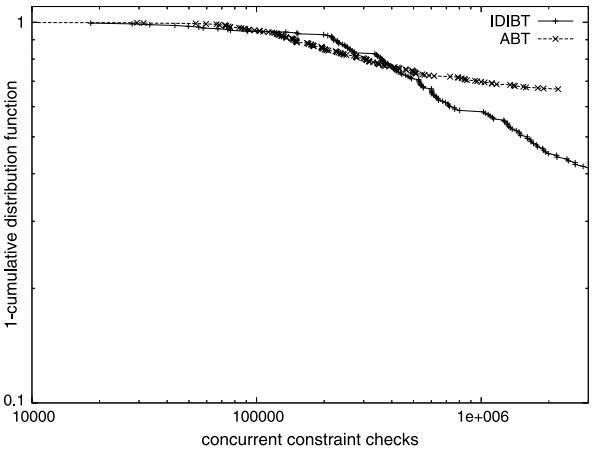


Fig. 2.1 Heavy-tail behavior of IDIBT and ABT

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.

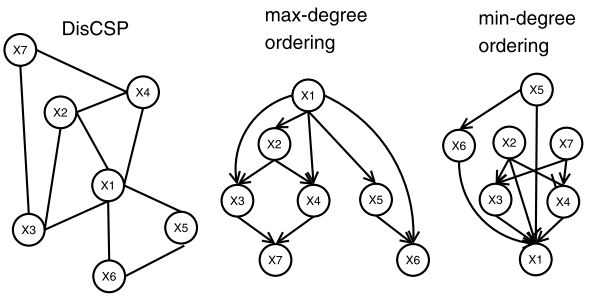


Fig. 2.2 DisCSP (*left*) and agent topologies implied by the variable orderings max-degree (*middle*) and min-degree (*right*)

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.

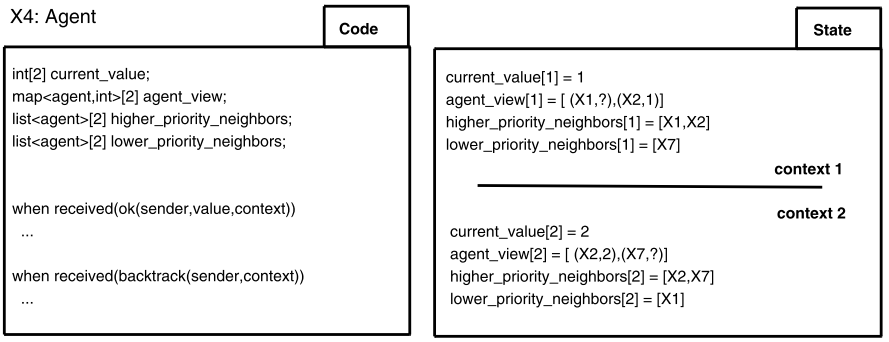


Fig. 2.3 Two contexts for the agent hosting X_4 from Fig. 2.2 resulting from two variable orderings

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.

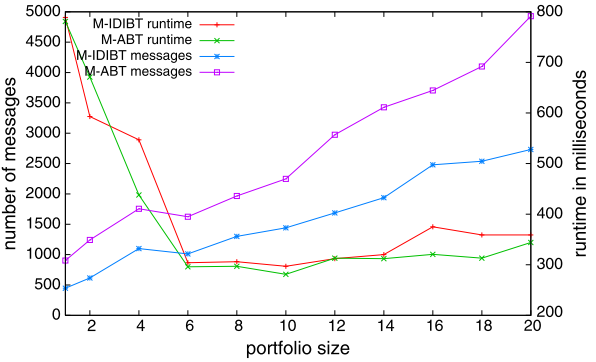


Fig. 2.4 Communication and runtime in M-portfolios

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.

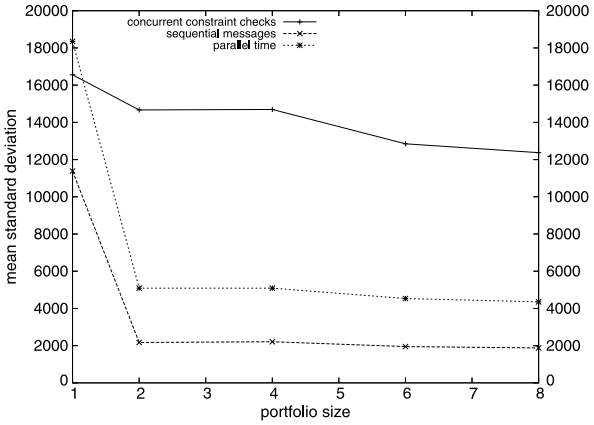


Fig. 2.5 Randomization risk emerging from message delays and thread activation

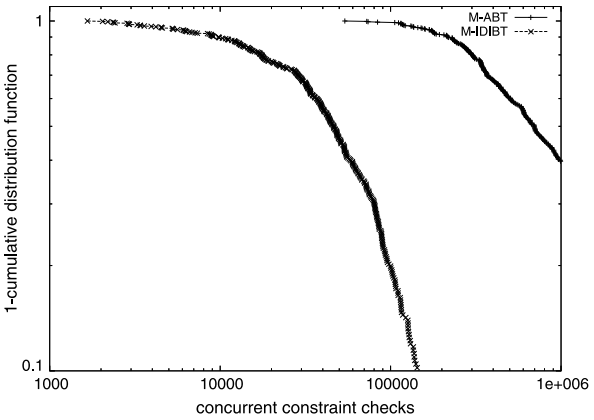


Fig. 2.6 No heavy-tails with M-ABT and M-IDIBT

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.

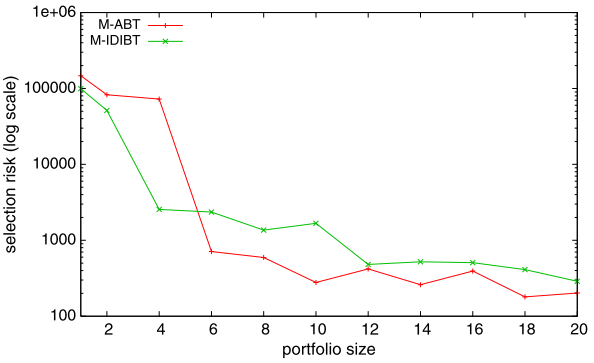


Fig. 2.7 S-risk (standard-dev of the parallel runtime) including the R-risk emerging from distribution

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.

Chapter 3
Parallel Tree Search for Satisfiability

3.1 Introduction

In the previous chapter, we have seen how a portfolio of algorithms, opportunistically exchanging knowledge about the problem, can be used to boost the performance of distributed search by several orders of magnitude. In this chapter, we are going to apply the same concepts to centralized search, i.e., to situations where the problem is fully expressed in one particular node or agent. More specifically, we are going to apply parallel portfolios to the important domain of propositional satisfiability.

In recent years, SAT solvers had a huge impact in their traditional hardware and software verification domains. Today, they are also gaining popularity in new fields like Automated Planning, General Theorem Proving or Computational Biology [Rin11, dMB08, CBH+07]. This widespread adoption is the result of the efficiency gains made during the last decade [BHZ06]. Indeed, many industrial problems with hundreds of thousands of variables and millions of clauses are now solved within a few minutes. This impressive progress can be related to both low-level algorithmic improvements and to the ability of SAT solvers to exploit the hidden structures of a practical problem.

However, many new applications with instances of increasing size and complexity are coming to challenge modern solvers, while at the same time it becomes clear that the gains traditionally given by low-level algorithmic adjustments are almost gone. As a result, a large number of industrial instances from recent competitions remain challenging for all the available SAT solvers. Fortunately, the previous comes at a time when the generalization of multicore hardware gives parallel processing capabilities to standard PCs. While in general it is important for existing applications to exploit new hardware, for SAT solvers, this becomes crucial.

Many parallel SAT solvers have been previously proposed. Most of them are based on the divide-and-conquer principle (see Sect. 3.2). They either divide the search space, using, for example, guiding paths, or the formula itself using decomposition techniques. The main problem behind these approaches is the difficulty of getting the workload balanced among the different processor units or workstations.

Y. Hamadi, *Combinatorial Search: From Algorithms to Systems*,
DOI 10.1007/978-3-642-41482-4_3,

Another drawback of these approaches arises from the fact that for a given large SAT instance with hundreds of thousands of variables it is very difficult to find the most relevant set of variables to divide the search space.

In the following, we detail ManySAT, a new parallel SAT solver, winner of the 2008 Sat Race.[1] The design of ManySAT takes advantage of the main weakness of modern solvers: their sensitivity to parameter tuning. For instance, changing the parameters related to the restart strategy or to the variable selection heuristic can completely change the performance of a solver on a particular problem class. In a multicore context, we can easily take advantage of this lack of robustness by designing a portfolio which will run different incarnations of a sequential solver on the same instance. Each solver would exploit a particular parameter set and their combination should represent a set of orthogonal yet complementary strategies. Moreover, individual solvers could perform knowledge exchange in order to improve the performance of the system beyond the performance of its individual components.

As we can see, the ManySAT approach is a direct application of our previous M-framework to SAT. Unlike in M-, ManySAT solves centralized problems and uses multiple resources to speed up processing. Here, an M- *context* corresponds to the full execution of a sequential SAT engine. In the portfolio, engines are differentiated with respect to their labeling strategies but also to various other features of SAT solvers [HJS09a, HJS09b, GHJS10, HJPS11, WHdM09, HMSW11, AH11, HW12, HW13].

3.2 Previous Work

We present here the most noticeable approaches related to parallel SAT solving.

PSATO [ZBH96] is based on the SATO (SAtisfiability Testing Optimized) sequential solver [ZS94]. Like SATO, it uses a *trie* data structure to represent clauses. PSATO uses the notion of *guiding paths* to divide the search space of a problem. These paths are represented by a set of unit clauses added to the original formula. The parallel exploration is organized in a master/slave model. The master organizes the work by assigning guiding paths to workers which have no interaction with each other. The first worker to finish stops the system. The balancing of the work is organized by the master.

In [JLU05] a parallelization scheme for a class of SAT solvers based on the DPLL procedure is presented. The scheme uses a dynamic load-balancing mechanism based on work-stealing techniques to deal with the irregularity of SAT problems. PSatz is the parallel version of the well-known Satz solver.

Gradsat [CW06] is based on zChaff. It uses a master-slave model and the notion of guiding paths to split the search space and to dynamically spread the load between clients. Learned clauses are exchanged between all clients if they are smaller than

[1] http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/index.html.

a predefined limit on the number of literals. A client incorporates a foreign clause when it backtracks to level 1 (top level).

In [BSK03], the authors use an architecture similar to Gradsat. However, a client incorporates a foreign clause if it is not subsumed by the current guiding path constraints. Practically, clause sharing is implemented by *mobile agents*. This approach is supposed to scale well on computational grids.

Nagsat [FS02] is a parallel SAT solver which exploits the heavy-tailed distribution of random 3-SAT instances. It implements *nagging*, a notion taken from the DALI theorem prover. Nagging involves a master and a set of clients called *naggers*. In Nagsat, the master runs a standard DPLL algorithm with a static variable ordering. When a nagger becomes idle, it requests a *nagpoint* which corresponds to the current state of the master. Upon receiving a nagpoint, it applies a transformation (e.g., a change in the ordering of the remaining variables), and begins its own search on the corresponding sub-problem.

In [BS96] the input formula is dynamically divided into disjoint sub-formulas. Each sub-formula is solved by a sequential SAT solver running on a particular processor. The algorithm uses optimized data structures to modify Boolean formulas. Additionally, workload balancing algorithms are used to achieve a uniform distribution of workload among the processors.

MiraXT [LSB07], is designed for shared memory multiprocessors systems. It uses a divide-and-conquer approach where threads share a unique clause database which stores the original and the learnt clauses. When a new clause is learnt by a thread, it uses a lock to safely update the common database. Read access can be done in parallel.

PMSat uses a master-slave scenario to implement a classical divide-and-conquer search [GFS08]. The user of the solver can select among several partitioning heuristics. Learnt clauses are shared between workers, and can also be used to stop efforts related to search spaces that have been proven irrelevant. PMSat runs on networks of computer through an MPI implementation.

In [CS08], the authors use a standard divide-and-conquer approach based on guiding paths. However, it exploits the knowledge on these paths to improve clause sharing. Indeed, clauses can be large with respect to some static limit, but when considered with the knowledge of the guiding path of a particular thread, a clause can become small and therefore highly relevant. This allows pMinisat to extend the sharing of clauses since a large clause can become small in another search context.

3.3 Technical Background

In this section, we first recall the basis of the most commonly used DPLL search procedure. Then, we introduce some computational features of modern SAT solvers. A brief description of multicore-based architectures is given. Finally, we present the principle of the AIMD feedback control-based algorithm used by advanced versions of ManySAT to manage knowledge sharing.

3.3.1 DPLL Search

Most of the state-of-the-art SAT solvers are simply based on the Davis, Putnam, Logemann and Loveland procedure, commonly called DPLL [DLL62]. DPLL is a backtrack search procedure; at each node of the search tree, a *decision* literal is chosen according to some branching heuristics. Its assignment to one of the two possible values (true or false) is followed by an *inference* step that deduces and propagates some forced literal assignments such as unit and monotone literals. The assigned literals (the decision literal and the propagated ones) are labeled with the same decision level starting from 1 and increased at each decision (or branching) until finding a model or a conflict is reached. In the first case, the formula is found to be satisfiable, whereas in the second case, we backtrack to the last decision level and assign the opposite value to the last decision literal. After backtracking, some variables are unassigned, and the current decision level is decreased accordingly. The formula is found to be unsatisfiable when a backtrack to level 0 occurs. Many improvements have been proposed over the years to enhance this basic procedure, leading now to what is commonly called modern SAT solvers. We also mention that, some look-ahead based improvements are at the basis of other kinds of DPLL SAT solvers (e.g. Satz [LA97], Kcnfs [DD01], March-dl [HvM06]) particularly efficient on hard random and crafted SAT categories.

3.3.2 Modern SAT Solvers

Modern SAT solvers [MMZ+01, ES03a] are based on classical DPLL search procedures [DLL62] combined with (i) restart policies [GSK98, KHR+02], (ii) activity-based variable selection heuristics (VSIDS-like) [MMZ+01], and (iii) clause learning [MSS96], the interaction of these three components being performed through efficient data structures (e.g., watched literals [MMZ+01]). All the state-of-the-art SAT solvers are based on a variation in these three important components.

Modern SAT solvers are especially efficient on structured instances coming from industrial applications. VSIDS and other variants of activity-based heuristics [BGS99], on the other hand, were introduced to avoid thrashing and to focus the search: when dealing with instances of large size, these heuristics direct the search to the most constrained parts of the formula. Restarts and VSIDS play complementary roles since the first component reorders assumptions and compacts the assumptions stack while the second allows for more intensification. Conflict Driven Clause Learning (CDCL) is the third component, leading to non-chronological backtracking. In CDCL a central data structure is the *implication graph*, which records the partial assignment that is under construction together with its implications [MSS96]. Each time a dead end is encountered (say at level i) a conflict clause or nogood is learnt due to a bottom-up traversal of the implication graph. This traversal is also used to update the activity of related variables, allowing VSIDS to always select the most active variable as the new decision point. The learnt conflict clause, called

asserting clause, is added to the learnt database and the algorithm backtracks non-chronologically to level $j < i$.

Progress saving is another interesting improvement; initially introduced in [FD94], it was recently presented in the Rsat solver [PD07]. It can be seen as a new selection strategy of the literal polarity. More precisely, each time a backtrack occurs from level i to level j, the literal polarity of the literals assigned between the two levels is saved. Then, such a polarity is used in subsequent search trees. This can be seen as a partial component caching technique that avoids solving some components multiple times.

Modern SAT solvers can now handle propositional satisfiability problems with hundreds of thousands of variables or more. However, it is now recognized (see the recent SAT competitions) that the performances of the modern SAT solvers evolve in a marginal way. More precisely, on the industrial benchmarks category usually proposed at the annual SAT Race and/or SAT Competition, many instances remain open (not solved by any solver within a reasonable amount of time). These problems which cannot be solved even using a three hour time limit are clearly challenging to all the available SAT solvers. Consequently, new approaches are clearly needed to solve these challenging industrial problems.

3.3.3 Multicore Architectures

We can abstract a multicore architecture as a set of processing units which communicate through a shared memory. In theory, access to the memory is uniform, i.e., can be done simultaneously. Practically, the use of cache mechanisms in processing units creates coherence problems which can slow down the memory accesses.

Our work is built on this shared memory model. The communication between the DPLL solvers of a portfolio is organized through lock-less queues that contain the lemmas that a particular core wants to exchange.

3.3.4 AIMD Feedback Control-Based Algorithm

The Additive Increase/Multiplicative Decrease (AIMD) algorithm is a feedback control algorithm used in TCP congestion avoidance. The problem solved by AIMD is to guess the communication bandwidth available between two communicating nodes. The algorithm performs successive probes, increasing the communication rate w linearly as long as no packet loss is observed, and decreasing it exponentially when a loss is encountered. More precisely, the evolution of w is defined by the following $AIMD(a, b)$ formula:

- $w = w - a \times w$, if loss is detected
- $w = w + \frac{b}{w}$, otherwise

Different proposals have been made in order to prevent congestion in communication networks based on different numbers for a and b. Today, AIMD is the major

component of TCP's congestion avoidance and control [Jac88]. On probe of network bandwidth, increasing too quickly will overshoot limits (underlying capacities). On notice of congestion, decreasing too slowly will not be reactive enough.

In the context of ManySAT, it is important to exchange knowledge between solvers. We will see that AIMD-based control policies can be used to achieve a particular throughput or a particular throughput of maximum quality. Since any increase in the size limit can potentially generate a very large number of new clauses, AIMD's slow increase can help us to avoid a quick overshoot of the throughput. Similarly, in case of overshooting, aggressive decrease can help us to quickly reduce clause sharing by a very large amount.

3.4 ManySAT: A Parallel SAT Solver

ManySAT is a parallel portfolio of several DPLL engines which includes all the classical features like two-watched-literal, unit propagation, activity-based decision heuristics, lemma deletion strategies, and clause learning. In addition to the classical first UIP scheme [ZMMM01], it incorporates a new technique which extends the implication graph used during conflict analysis to exploit the satisfied clauses of a formula [ABH+08]. In the following, we describe and motivate a set of important parameters used to differentiate the different solvers in the portfolio.

3.4.1 Restart Policies

Restart policies represent an important component of modern SAT solvers. Contrary to the common belief, in SAT restarts are not used to eliminate the heavy-tailed phenomena [GSK98, GSCK00] since after restarting SAT solvers dive in the part of the search space that they just left. In SAT, restarts policies are used to compact the assignment stack and improve the order of assumptions.

Different restart policies have been previously presented. Most of them are static, and the cutoff value follows different evolution schemes (e.g. arithmetic, geometric, Luby). To ensure the completeness of the SAT solver, in all these restart policies, the cutoff value in terms of the number of conflicts increases over the time. The performance of these different policies clearly depends on the considered SAT instances. More generally, rapid restarts (e.g. Luby) perform well on industrial instances; however, on hard SAT instances slow restarts are more suitable. Generally, it is hard to say in advance which policy should be used on which problem class [Hua07].

Our objective was to use complementary restart policies to define the restart cutoff x_i.

We decided to use the well-known Luby policy [LSZ93], and a classical geometric policy, $x_i = 1.5 \times x_{i-1}$ with $x_1 = 100$ [ES03a]. The Luby policy was used with a unit factor set to 512. In addition, we decided to introduce two new policies. A very slow arithmetic one, $x_i = x_{i-1} + 16000$ with $x_1 = 16000$, and a new dynamic one.

New Dynamic Restart Policy The early work on dynamic restart policy goes back to 2008. Based on the observation that frequent restarts significantly improve the performance of SAT solvers on industrial instances, Armin Biere presents in [Bie08] a novel adaptive restart policy that measures the "agility" of the search process dynamically, which in turn is used to control the restart frequency. The agility measures the average number of recently flipped assignments. Low agility enforces frequent restarts, while high agility tends to prohibit restarts.

In [RS08], the authors propose applying restarts according to measures local to each branch. More precisely, for each decision level d a counter $c(d)$ of the number of conflicts encountered under the decision level d is maintained. When backtracking to the decision level d occurs, if the value $c(d)$ is greater than a given threshold, the algorithm restarts.

Considering CDCL-based SAT solvers, it is now widely admitted that restarts are an important component when dealing with industrial SAT instances, whereas on crafted and random instances they play a marginal role. More precisely, in the industrial (resp. crafted) category, rapid (resp. long) restarts are more appropriate. It is important to note that on hard SAT instances, learning is useless. Indeed, on such instances, conflict analysis generally leads to a learnt clause which includes at least one literal from the level just before the current conflict level. In other words the search algorithm usually back-jumps to the level preceding that of the current conflict. For example, if we consider the well-known Pigeon-hole problem, learning from conflicts will produce a clause which includes at least one literal from each level. It is also obvious from this example, that learning does not achieve important back-jumps in the search tree. The algorithm usually carries out a chronological backtracking.

In the following, we define a new dynamic restart policy based on the evolution of the average size of back-jumps. First, such information is a good indicator of the decision errors made during search. Secondly, it can be seen as an interesting measure of the relative hardness of the instance. Our new policy is designed in such a way that, for high (resp. low) fluctuation of the average size of back-jumps (between the current and the previous restart), it delivers a low (resp. high) cutoff value. In other words, the cutoff value of the next restart depends on the average size of back-jumps observed during the two previous and consecutive runs. We define it as, $x_1 = 100$, $x_2 = 100$, and $x_{i+1} = \frac{\alpha}{y_i} \times |\cos(1 - r_i)|$, $i \geq 2$, where $\alpha = 1200$, y_i represents the average size of back-jumps at restart i, $r_i = \frac{y_{i-1}}{y_i}$ if $y_{i-1} < y_i$, $r_i = \frac{y_i}{y_{i-1}}$ otherwise.

From Fig. 3.1, we can observe that the cutoff value in terms of the number of conflicts is low in the first restarts and high in the last ones. This means that the fluctuation between two consecutive restarts is more important at the beginning of the resolution process. Indeed, the activity of the variables is not sufficiently accurate in the first restarts, and the sub-problem on which the search focuses is not sufficiently circumscribed.

The dynamic restart policy, presented in this section is implemented in the first version of ManySAT [HJS08] presented at the parallel track of the SAT Race 2008.

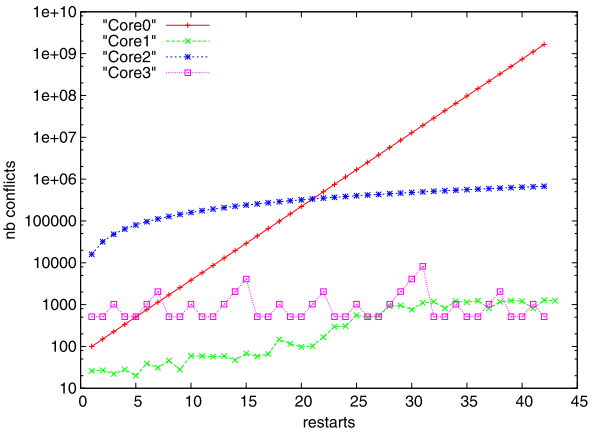


Fig. 3.1 Restart strategies

3.4.2 Heuristic

We decided to increase the random noise associated to the VSIDS heuristic [MMZ+01] of core 0 since its restart policy is the slowest one. Indeed, that core tends to intensify the search, and slightly increasing the random noise allows us to introduce more diversification.

3.4.3 Polarity

Each time a variable is chosen, one needs to decide if such a variable might be assigned true (positive polarity) or false (negative polarity). Different kinds of polarity have been defined. For example, Minisat usually chooses the negative polarity, whereas Rsat uses progress saving. More precisely, each time a backtrack occurs, the polarity of the assigned variables between the conflict and the back-jumping level are saved. If one of these variables is chosen again its saved polarity is preferred. In CDCL-based solvers, the chosen polarity might have a direct impact on the learnt clauses and on the performance of the solver.

The polarity of the core 0 is defined according to the number of occurrences of each literal in the learnt database. Each time a learnt clause is generated, the number of occurrences of each literal is increased by 1. Then to maintain a more constrained learnt database, the polarity of l is set to *true* when $\#occ(l)$ is greater than $\# occ(\neg l)$, and to *false* otherwise. For example, by setting the polarity of l to *true*, we bias the occurrence of its negation $\neg l$ in the next learnt clauses.

This approach tends to balance the polarity of each literal in the learnt database. By doing so, we increase the number of possible resolvents between the learnt clauses. If the relevance of a given resolvent is defined as the number of steps needed to derive it, then a resolvent between two learnt clauses might lead to more relevant clauses in the database.

As the restart strategy in core 0 tends to intensify the search, it is important to maintain a learnt database of better quality. However, for rapid restarts as in cores 1 and 3, progress saving is most suitable for saving the work accomplished. For core 2, we decided to apply a complementary polarity (*false* by default as in Minisat).

3.4.4 Learning

Learning is another important component which is crucial for the efficiency of modern SAT solvers. Most of the known solvers use similar CDCL approaches associated with the first UIP (Unique Implication Point) scheme.

In our parallel SAT solver ManySAT, we used a new learning scheme obtained using an extension of the classical implication graph [ABH+08]. This new notion considers additional arcs, called inverse arcs. These are obtained by taking into account the satisfied clauses of the formula, which are usually ignored by classical conflict analysis. The new arcs present in our extended graph allow us to detect that even some decision literals admit a reason, something which is ignored when using classical implication graphs. As a result, the size of the back-jumps is often increased.

Let us illustrate this new extended conflict analysis using a simple example. We assume that the reader is familiar with the classical CDCL scheme used in modern SAT solvers (see [MSS96, MMZ+01, ABH+08]).

Let $\mathcal{F}$ be a CNF formula and ρ a partial assignment, given below:

- $\mathcal{F} \supseteq \{c_1, \ldots, c_9\}$
- $(c_1)\ x_6 \vee \neg x_{11} \vee \neg x_{12}$
- $(c_2)\ \neg x_{11} \vee x_{13} \vee x_{16}$
- $(c_3)\ x_{12} \vee \neg x_{16} \vee \neg x_2$
- $(c_4)\ \neg x_4 \vee x_2 \vee \neg x_{10}$
- $(c_5)\ \neg x_8 \vee x_{10} \vee x_1$
- $(c_6)\ x_{10} \vee x_3$
- $(c_7)\ x_{10} \vee \neg x_5$
- $(c_8)\ x_{17} \vee \neg x_1 \vee \neg x_3 \vee x_5 \vee x_{18}$
- $(c_9)\ \neg x_3 \vee \neg x_{19} \vee \neg x_{18}$
- $\rho = \{\langle \ldots \neg x_6^1 \ldots \neg x_{17}^1 \rangle \langle (x_8^2) \ldots \neg x_{13}^2 \ldots \rangle \langle (x_4^3) \ldots x_{19}^3 \ldots \rangle \ldots \langle (x_{11}^5) \ldots \rangle\}$

The sub-sequence $\langle (x_8^2) \ldots \neg x_{13}^2 \ldots \rangle$ of ρ expresses the set of literals assigned at level 2 with the decision literal mentioned in parenthesis and the set of propagated literals (e.g. $\neg x_{13}$). The current decision level is 5. The classical implication graph $\mathcal{G}_{\mathcal{F}}^{\rho}$ associated to $\mathcal{F}$ and ρ is shown in Fig. 3.2 with only the plain arcs.

In the sequel, $\eta[x, c_i, c_j]$ denotes the *resolvent* between a clause c_i containing the literal x and a clause c_j containing the literal $\neg x$. In other words $\eta[x, c_i, c_j] = c_i \cup c_j \backslash \{x, \neg x\}$. Also a clause c subsumes a clause c' iff $c \subseteq c'$.

The traversal of the graph $\mathcal{G}_{\mathcal{F}}^{\rho}$ allows us to generate three asserting clauses corresponding to the three possible UIPs (see Fig. 3.2). Let us illustrate the resolution process leading to the first asserting clause Δ_1 corresponding to the first UIP.

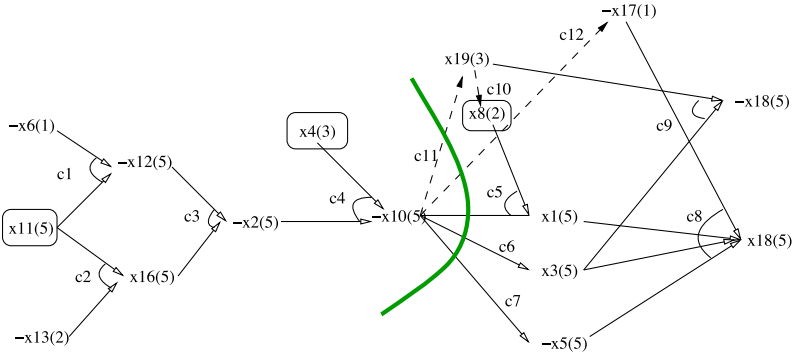


Fig. 3.2 Implication graph/extended implication graph

- $\sigma_1 = \eta[x_{18}, c_8, c_9] = (x_{17}^1 \vee \neg x_1^5 \vee \neg x_3^5 \vee x_5^5 \vee \neg x_{19}^3)$
- $\sigma_2 = \eta[x_1, \sigma_1, c_5] = (x_{17}^1 \vee \neg x_3^5 \vee x_5^5 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$
- $\sigma_3 = \eta[x_5, \sigma_2, c_7] = (x_{17}^1 \vee \neg x_3^5 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$
- $\sigma_4 = \eta[x_3, \sigma_3, c_6] = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$

As we can see, σ_4 gives us a first asserting clause (that we'll also name Δ_1) because all of its literals are assigned before the current level except one (x_{10}), which is assigned at the current level 5. The intermediate clauses σ_1, σ_2 and σ_3 contain more than one literal of the current decision level 5, and $\neg x_{10}$ is a first UIP. If we continue such a resolution process, we obtain the two additional asserting clauses, $\Delta_2 = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee \neg x_4^3 \vee x_2^5)$, corresponding to a second UIP $\neg x_2^5$, and $\Delta_3 = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee \neg x_4^3 \vee x_{13}^2 \vee x_6^1 \vee \neg x_{11}^5)$, corresponding respectively to a third UIP ($\neg x_{11}^5$), which is the last UIP since it corresponds to the last decision literal in the partial assignment.

In modern SAT solvers, clauses containing a literal x that is implied at the current level are essentially ignored by the propagation. More precisely, because the solver does not maintain the information whether a given clause is satisfied or not, a clause containing x may occasionally be considered by the propagation, but only when another literal y of the clause becomes false. When this happens the solver typically skips the clause. However, in cases where x is true *and all the other literals are false*, an arc is revealed for free that could as well be used to extend the graph. Such arcs are those we exploit in our proposed extension.

To explain further the idea behind our extension, let us consider, again, the formula $\mathcal{F}$ and the partial assignments given in the previous example. We define a new formula $\mathcal{F}'$ as follows: $\mathcal{F}' \supseteq \{c_1, \ldots, c_9\} \cup \{c_{10}, c_{11}, c_{12}\}$ where $c_{10} = (\neg x_{19} \vee x_8)$, $c_{11} = (x_{19} \vee x_{10})$ and $c_{12} = (\neg x_{17} \vee x_{10})$.

The three added clauses are satisfied under the instantiation ρ. c_{10} is satisfied by x_8 assigned at level 2, c_{11} is satisfied by x_{19} at level 3, and c_{12} is satisfied by $\neg x_{17}$ at level 1. This is shown in the extended implication graph (see Fig. 3.2) by the dotted edges. Let us now illustrate the usefulness of our proposed extension. Let us consider again the asserting clause Δ_1 corresponding to the classical first UIP. We

can generate the following strong asserting clause: $c_{13} = \eta[x_8, \Delta_1, c_{10}] = (x_{17}^1 \vee \neg x_{19}^3 \vee x_{10}^5)$, $c_{14} = \eta[x_{19}, c_{13}, c_{11}] = (x_{17}^1 \vee x_{10}^5)$ and $\Delta_1^s = \eta[x_{17}, c_{14}, c_{12}] = x_{10}^5$. In this case we backtrack to the level 0 and we assign x_{10} to *true*.

As we can see Δ_1^s subsumes Δ_1. If we continue the process we also obtain other strong asserting clauses $\Delta_2^s = (\neg x_4^3 \vee x_2^5)$ and $\Delta_3^s = (\neg x_4^3 \vee x_{13}^2 \vee x_6^1 \vee \neg x_{11}^5)$ which subsume respectively Δ_2 and Δ_3.

This first illustration gives us a new way to minimize the size of the asserting clauses.

Let us now explain briefly how the extra arcs can be computed. Usually unit propagation does not keep track of implications from the satisfiable sub-formula. In this extension the new implications (deductions) are considered. For instance in the previous example, when we deduce x_{19} at level 3, we *rediscover* the deduction x_8, which was a choice (decision literal) at level 2. Our proposal keeps track of these re-discoveries.

Our approach makes an original use of inverse arcs to back-jump farther, i.e., to improve the back-jumping level of the classical asserting clauses. It works in three steps. In the first step (1), an asserting clause, say $\sigma_1 = (\neg x^1 \vee \neg y^3 \vee \neg z^7 \vee \neg a^9)$, is learnt using the usual learning scheme, where 9 is the current decision level. As $\rho(\sigma_1) = \mathit{false}$, usually we backtrack to level 7. In the second step (2), our approach aims to eliminate the literal $\neg z^7$ from σ_1 using the new arcs of the extended graph. Let us explain this second and new processing. Let $c = (z^7 \vee \neg u^2 \vee \neg v^9)$ such that $\rho(z) = \mathit{true}$, $\rho(u) = \mathit{true}$ and $\rho(v) = \mathit{true}$. The clause c is an inverse arc i.e., the literal z assigned at level 7 is implied by the two literals u and v respectively assigned at levels 2 and 9. From c and σ_1, a new clause $\sigma_2 = \eta[z, c, \sigma_1] = (\neg x^1 \vee \neg u^2 \vee \neg y^3 \vee \neg v^9 \vee \neg a^9)$ is generated. We can observe that the new clause σ_2 contains two literals from the current decision level 9. In the third step (3), using classical learning, one can search from σ_2 for another asserting clause σ_3 with only one literal from the current decision level. Let us note that the new asserting clause σ_3 might be worse in terms of back-jumping level. To avoid this main drawback, the inverse arc c is chosen if the two following conditions are satisfied: (i) the literals of c assigned at the current level (v^9) have been already visited during the first step and (ii) all the other literals of c are assigned before level 7, i.e., level of z. In this case, we guaranteed that the new asserting clause achieves better back-jumping.

This new learning scheme is integrated on the SAT solvers of cores 0 and 3.

3.4.5 Clause Sharing

Unlike in the previously presented M-framework, knowledge in SAT is made of conflict clauses, and knowledge sharing is referred as clause sharing.

To start with, we can use a static clause sharing policy where each core exchanges a learnt clause if its size is less than or equal to 8. This decision is based on extensive tests with representative industrial instances. Figure 3.3 (resp. Fig. 3.4) shows for different limits e the performance of ManySAT on instances taken from the SAT

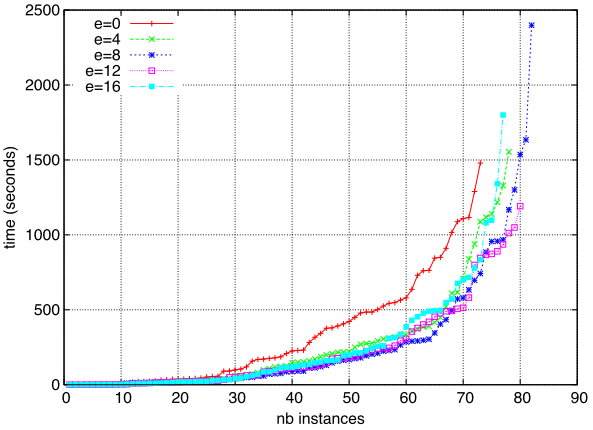


Fig. 3.3 SAT Race 2008: different limits for clause sharing

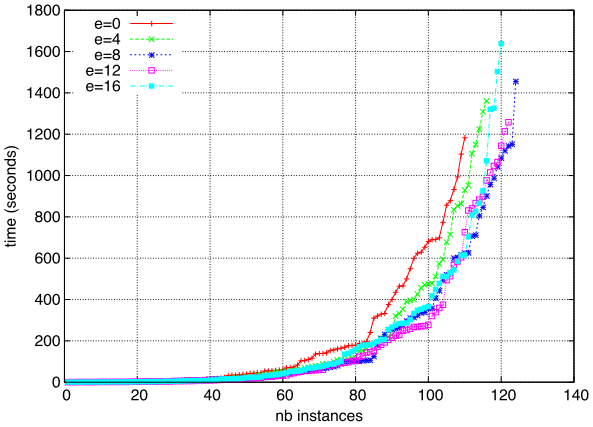


Fig. 3.4 SAT Competition 2007: different limits for clause sharing

Race 2008 (resp. SAT Competition 2007). We can observe that on each set of benchmarks a limit size of 8 gives the best overall performance.

The communication between the solvers of the portfolio is organized through lock-less queues which contain the lemmas that a particular core wants to exchange.

Each core imports unit clauses when it reaches level 0 (e.g., after a restart). These important clauses correspond to the removal of Boolean variables, and therefore are more easily enforced at the top level of the tree.

All the other clauses are imported on the fly, i.e., after each decision. Several cases have to be handled for the integration of a foreign clause c:

- c is false in the current context. In this case, conflict analysis has to start, allowing the search process to back-jump. This is clearly the most interesting case.
- c is unit in the current context. The clause can be used to enforce more unit propagation, allowing the process to reach a smaller fix-point or a conflict.
- c is satisfied by the current context. It has to be watched. To exploit such a clause in the near future, we consider two literals assigned at the highest levels.

- otherwise, c has to be watched. In this last case, the first two unassigned literals are watched.

The following example illustrates the different cases mentioned above.

Let $\mathcal{F}$ be a CNF formula and $\rho = \{\langle \dots \neg x_6^1 \dots \neg x_{17}^1 \rangle \langle (x_8^2) \dots \neg x_{13}^2 \dots \rangle \langle (x_4^3) \dots x_{19}^3 \dots \rangle \dots \langle (x_{11}^5) \neg x_{12}^5, x_{16}^5, \neg x_2^5, \dots, \neg x_{10}^5, x_1^5, \dots, x_{18}^5 \rangle\}$ a partial assignment. To make the shared clause c exploitable in near future, it must be watched in a certain way. Suppose that,

- $c = (x_{17}^1 \vee \neg x_{19}^3 \vee x_{10}^5) \in \mathcal{F}$. The clause c is false and the two literals $\neg x_{19}^3$ and x_{10}^5 are watched.
- $c = (x_{17}^1 \vee \neg x_{19}^3 \vee x_{30}) \in \mathcal{F}$. The clause c is unit and the two literals $\neg x_{19}^3$ and x_{30} are watched.
- $c = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_{10}^5) \in \mathcal{F}$. We watch the last satisfied literal $\neg x_{10}$ and another literal with the highest level from the remaining ones.
- $c = (x_{25} \vee \neg x_{34} \neg x_{29}) \in \mathcal{F}$. We watch any two literals from c.

3.4.6 Summary

Table 3.1 summarizes the choices made for the different solvers of the ManySAT portfolio. For each solver (core), we mention the restart policy, the heuristic, the polarity, the learning scheme and the size of shared clauses.

3.5 Evaluation

3.5.1 Performance Against a Sequential Algorithm

ManySAT was built on top of Minisat 2.02 [ES03a]. SatElite was applied systematically by each core as a pre-processor [EB05]. In all the figures, instances solved by Satellite in the preprocessing step are not included. In this section, we evaluate the performance of the solver on a large set of industrial problems. Figure 3.5 shows the improvement of performance provided by our solver when compared to the sequential solver Minisat 2.02 on the problems of the SAT Race 2008. It shows the performance of ManySAT running with respectively one, two, three and four cores. When more than one core is used, clause sharing is done up to clause size 8.

We can see that even the sequential version of ManySAT (single core) outperforms Minisat 2.02. This simply means that our design choices for core 1 represent a good combination to put in a sequential solver. Interestingly, with each new core, the performance increases both in speed and number of problems solved. This is the result of the diversification of the search but also the fact that clause sharing quickly boosts these independent search processes.

Table 3.1 ManySAT: different strategies

Strategies	Core 0	Core 1	Core 2	Core 3
Restart	Geometric $x_1 = 100$ $x_i = 1.5 \times x_{i-1}$	Dynamic (Fast) $x_1 = 100, x_2 = 100$ $x_i = f(y_{i-1}, y_i), i > 2$ if $y_{i-1} < y_i$ $f(y_{i-1}, y_i) = \frac{\alpha}{y_i} \times \lvert\cos(1 - \frac{y_{i-1}}{y_i})\rvert$ else $f(y_{i-1}, y_i) = \frac{\alpha}{y_i} \times \lvert\cos(1 - \frac{y_i}{y_{i-1}})\rvert$ $\alpha = 1200$	Arithmetic $x_1 = 16000$ $x_i = x_{i-1} + 16000$	Luby 512
Heuristic	VSIDS (3 % rand.)	VSIDS (2 % rand.)	VSIDS (2 % rand.)	VSIDS (2 % rand.)
Polarity	if $\#occ(l) > \#occ(\neg l)$ $l = true$ else $l = false$	Progr. saving	False	Progr. saving
Learning	CDCL (ext. [ABH+08])	CDCL	CDCL	CDCL (ext. [ABH+08])
Cl. sharing	Size ≤ 8	Size ≤ 8	Size ≤ 8	Size ≤ 8

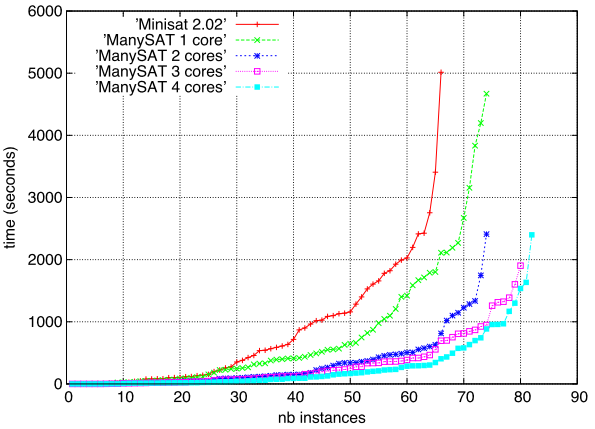


Fig. 3.5 SAT Race 2008: ManySAT $e = 8$, $m = 1 \ldots 4$ against Minisat 2.02

Table 3.2 SAT-Race 2008: comparative performance (number of problems solved)

	ManySAT	pMinisat	MiraXT
SAT	45	44	43
UNSAT	45	41	30

3.5.2 Performance Against Other Parallel SAT Solvers

We report here the official results of the 2008 Sat-Race. They can be downloaded from the competition website.[2] They demonstrate the performance of ManySAT as opposed to other parallel SAT solvers. These tests were done on 2× Dual Core Intel Xeon 5150 running at 2.66 GHz, with a time-out set to 900 seconds.

Table 3.2 shows the number of problems (out of 100) solved before the time limit for ManySAT, pMinisat [CS08], and MiraXT [LSB07], these solvers are described in the next section. We can see that ManySAT solves five more problems than pMinisat, which solves 12 more problems than MiraXT. Interestingly, the performance of our method is well balanced between SAT and UNSAT problems.

Table 3.3 shows the speed up provided by these parallel SAT algorithms as compared to the best sequential algorithm of the SAT Race 2008, Minisat 2.1. We can see that on average, ManySAT is able to provide a superlinear speed up of 6.02. It is the only solver capable of such performance. The second best provides on average a speed up of 3.10, far from linear. When we consider the minimal speed up we can see that the performance of the first two solvers is pretty similar. They decrease the performance against the best sequential solver of the 2008 SAT Race by up to a factor 4, while the third solver decreases the performance by a factor 25. Finally, the maximal speed up is given by ManySAT, which can be up to 250 times faster than Minisat 2.1. These detailed results show that the performance of the parallel solvers is usually better on SAT problems than on UNSAT ones.

[2] http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/.

Table 3.3 SAT Race 2008: parallel solvers against the best sequential solver (Minisat 2.1)

	ManySAT	pMinisat	MiraXT
Average speed up by SAT/UNSAT	6.02	3.10	1.83
	8.84/3.14	4.00/2.18	1.85/1.81
Minimal speed up by SAT/UNSAT	0.25	0.34	0.04
	0.25/0.76	0.34/0.46	0.04/0.74
Maximal speed up by SAT/UNSAT	250.17	26.47	7.56
	250.17/4.74	26.47/10.57	7.56/4.26

Table 3.4 SAT Race 2008: runtime variation of parallel solvers

	ManySAT	pMinisat	MiraXT
Average variation by SAT/UNSAT	13.7 %	14.7 %	15.2 %
	22.2 %/5.5 %	23.1 %/5.7 %	19.5 %/9.7 %

It is well known that parallel search is not deterministic. Table 3.4 gives the average runtime variation of each parallel solver. ManySAT exhibits a lower variation than the other techniques, but the small differences between the solvers do not allow us to draw any definitive conclusion.

3.6 Control-Based Clause Sharing

The clause sharing approach based on some predefined size limit has several flaws, the first and most apparent being that an overestimated value might induce a very large cooperation overhead, while an underestimated one might completely inhibit the cooperation. The second flaw comes from the observation that the size of learnt clauses tends to increase over time, leading to an eventual halt of the cooperation. The third flaw is related to the internal dynamic of modern solvers which tend to focus on particular sub-problems thanks to the activity/restart mechanisms. In parallel SAT, this can lead two search processes toward completely different sub-problems where clause sharing becomes pointless.

We propose a dynamic clause sharing policy which uses pairwise size limits to control the exchange between any pair of processing units. Initially, high limits are used to enforce the cooperation, and allow pairwise exchanges. On a regular basis, each unit considers the number of foreign clauses received from other units. If this number is below/above a predefined threshold, the pairwise limits are increased/decreased. This mechanism allows the system to maintain a throughput. It addresses the first two flaws. To address the last flaw related to the poor relevance of the shared clauses, we extend our policy to integrate the quality of the exchanges.

Each unit evaluates the quality of the received clauses, and the control is able to selectively increase/decrease the pairwise limits based on the underlying quality of the recently communicated clauses, the rationale being that the information recently received from a particular source is qualitatively linked to the information which could be received from it in the very near future. The evolution of the pairwise limits w.r.t. the throughput or quality criterion follows an AIMD (Additive-Increase-Multiplicative-Decrease) feedback control-based algorithm (see Sect. 3.3).

3.6.1 Throughput and Quality-Based Control Policies

In this section, we describe our dynamic control-based clause sharing policies which control the exchange between any pair of processing units through dynamic pairwise size limits.

The first policy controls the throughput of clause sharing. Each unit considers the number of foreign clauses received from other units. If this number is below/above a predefined throughput threshold, the pairwise limits are all increased/decreased using an AIMD feedback algorithm. The second policy is an extension of the previous one. It introduces a measure of the quality of foreign clauses. With this information, the increase/decrease of the pairwise limits becomes proportional to the underlying quality of the clauses shared by each unit. The first (resp. second) policy allows the system to maintain a throughput (resp. throughput of better quality).

We consider a parallel SAT solver with n different processing units. Each unit u_i corresponds to a SAT solver with clause learning capabilities. Each solver can work either on a sub-space of the original instance, as in divide-and-conquer techniques, or on the full problem, as in ManySAT. We assume that these different units communicate through a shared memory (as in multicore architectures).

In our control strategy, we consider a control-time sequence as a set of steps t_k with $t_0 = 0$ and $t_k = t_{k-1} + \alpha$ where α is a constant representing the time window defined in terms of the number of conflicts. The step t_k of a given unit u_i corresponds to the conflict number $k \times \alpha$ encountered by the solver associated to u_i. In the sequel, when there is no ambiguity, we sometimes write t_k simply as k. Then, each unit u_i can be defined as a sequence of states $S_i^k = (\mathcal{F}, \Delta_i^k, R_i^k)$, where $\mathcal{F}$ is a CNF formula, Δ_i^k the set of its proper learnt clauses and R_i^k the set of foreign clauses received from the other units between two consecutive steps $k-1$ and k. The different units achieve pairwise exchange using pairwise limits. Between two consecutive steps $k-1$ and k, a given unit u_i receives from all the other remaining units u_j, where $0 \leq j < n$ and $j \neq i$, a set of learnt clauses $\Delta_{j\to i}^k$ of length less or equal to a size limit $e_{j\to i}^k$ i.e., $\Delta_{j\to i}^k = \{c \in \Delta_j^k / \ |c| \leq e_{j\to i}^k\}$. Then, the set R_i^k can be formally defined as $\bigcup_{0\leq j<n,\ j\neq i} \Delta_{j\to i}^k$.

Using a fixed throughput threshold T of shared clauses, we describe our control-based policies which allow each unit u_i to guide the evolution of the size limit $e_{j\to i}$ using an AIMD feedback mechanism.

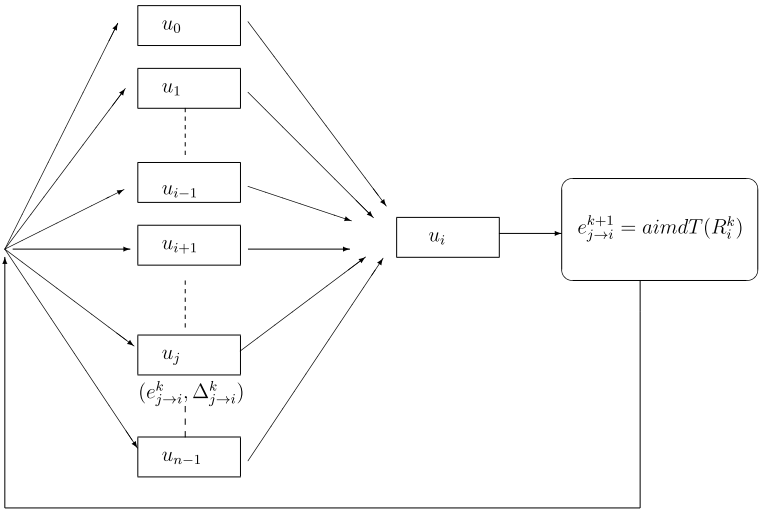


Fig. 3.6 Throughput-based control policy

Throughput-Based Control As illustrated in Fig. 3.6, at step k a given unit u_i checks whether the throughput is exceeded or not. if $|R^k_i| < T$ (resp. $|R^k_i| > T$) the size limit $e^{k+1}_{j\to i}$ is additively increased (resp. multiplicatively decreased).

More formally, the upper bound $e^{k+1}_{j\to i}$ on the size of clauses that a solver j shares with the solver i between k and $k+1$ is changed using the following AIMD function:

$$
\begin{aligned}
&aimdT\big(R^k_i\big)\{\\
&\forall j | 0 \leq j < n, j \neq i\\
&e^{k+1}_{j\to i} = \begin{cases} e^k_{j\to i} + \frac{b}{e^k_{j\to i}}, & \text{if } (|R^k_i| < T)\\ e^k_{j\to i} - a \times e^k_{j\to i}, & \text{if } (|R^k_i| > T) \end{cases}\\
&\}
\end{aligned}
$$

where a and b are positive constants.

Throughput and Quality-Based Control In this policy, to control the throughput of a given unit u_i, we introduce a quality measure $Q^k_{j\to i}$ (see Definition 3.1) to estimate the relative quality of the clauses received by u_i from u_j. In the throughput- and quality-based control policy, the evolution of the size limit $e^k_{j\to i}$ is related to the estimated quality.

Our quality measure is defined using the activity of the variables at the basis of the VSIDS heuristic [MMZ+01], another important component of modern SAT solvers. The variables with greatest activity represent those involved in most of the (recent) conflicts. Indeed, with each conflict, variables whose literals are used during

conflict analysis have their activity augmented. The most active variables are those related to the current part of the search space. Consequently, our quality measure exploits these activities to quantify the relevance of a clause learnt by unit u_j to the current state of a given unit u_i. To define our quality measure, suppose that, at any time of the search process, we have $\mathcal{A}_i^{max}$, the current maximal activity of u_i's variables, and $\mathcal{A}_i(x)$, the current activity of a given variable x.

Definition 3.1 (Quality) Let c be a clause and $\mathcal{L}_{\mathcal{A}_i}(c) = \{x/x \in c \text{ s.t. } \mathcal{A}_i(x) \geq \frac{\mathcal{A}_i^{max}}{2}\}$ the set of active literals of c with respect to unit u_i. We define $\mathcal{P}^k_{j\to i} = \{c/c \in \Delta^k_{j\to i} \text{ s.t. } |\mathcal{L}_{\mathcal{A}_i}(c)| \geq Q\}$ to be the set of clauses received by i from j between steps $k-1$ and k with at least Q active literals. We define the quality of clauses sent by u_j to u_i at a given step k as $Q^k_{j\to i} = \frac{|\mathcal{P}^k_{j\to i}|+1}{|\Delta^k_{j\to i}|+1}$.

Our throughput- and quality-based control policy changes the upper bound $e^{k+1}_{j\to i}$ on the size of clauses that a solver j shares with the solver i between k and $k+1$ using the following AIMD function:

$$
\begin{aligned}
&aimdTQ\big(R_i^k\big)\{\\
&\forall j | 0 \leq j < n, j \neq i\\
&e^{k+1}_{j\to i} = \begin{cases} e^k_{j\to i} + (\frac{Q^k_{j\to i}}{100}) \times \frac{b}{e^k_{j\to i}}, & \text{if } (|R_i^k| < T)\\ e^k_{j\to i} - (1 - \frac{Q^k_{j\to i}}{100}) \times a \times e^k_{j\to i}, & \text{if } (|R_i^k| > T) \end{cases}\\
&\}
\end{aligned}
$$

where a and b are positive constants.

As shown by the AIMD function of the throughput- and quality-based control policy, the adjustment of the size limit depends on the quality of shared clauses. Indeed, as it can be seen from the above formula, when the exchange quality between u_j and u_i ($Q^k_{j\to i}$) tends to 100 % (resp. 0 %), then the increase in the limit size tends to be maximal (resp. minimal) while the decrease tends to be minimal (resp. maximal). Our aim in this second policy is to maintain a throughput of good quality, the rationale being that the information recently received from a particular source is qualitatively linked to the information which could be received from it in the very near future.

3.6.2 Experiments

Our tests were done on Intel Xeon Quad core machines with 16 GB of RAM running at 2.3 GHz. We used a time-out of 1,500 seconds for each problem. ManySAT was used with 4 DPLL strategies, each one running on a particular core (unit). To

alleviate the effects of unpredictable thread scheduling, each problem was solved three times and the average was taken.

Our dynamic clause sharing policies were added to ManySAT and compared against ManySAT with its default static policy *ManySAT* $e = 8$, which exchanges clauses up to size 8. Note that since each pairwise limit is read by one unit, and updated by another one, our proposal can be integrated without any lock.

We have selected $a = 0.125$, $b = 8$ for aimdT and aimdTQ, associated to a time window of $\alpha = 10000$ conflicts. The throughput T is set to $\frac{\alpha}{2}$ and the upper bound Q on the number of active literals per clause c is set to $\frac{|c|}{3}$ (see Definition 3.1). Each pairwise limit $e_{j \to i}$ was initialized to 8.

The Table 3.5 presents the results on the 100 industrial problems of the 2008 SAT Race. The problem set contains families with several instances or individual instances.

From left to right we present, the family/instance name, the number of instances per family, results associated to the standard ManySAT, with the number of problems solved before time-out, and the associated average runtime. The right part reports results for the two dynamic policies. For each dynamic policy we provide $\bar{e}$, the average of the $e_{j \to i}$ observed during the computation. The last row provides for each method the total number of problems solved and the cumulated runtime. For the dynamic policies, it also presents the average of the $\bar{e}$ values.

At this point we have to stress that the static policy ($e = 8$) is optimal in the way that it gives the best average performance on this set of problems. We can observe that the static policy solves 83 problems while the dynamic policies aimdT and aimdTQ solve respectively 86 and 89 problems. Except on the *ibm_** and *manol_** families, the dynamic policies always exhibit a runtime better or equivalent to the static one. Unsurprisingly, when the runtime is significant but does not drastically improve over the static policy, the values of $\bar{e}$ are often close to 8, i.e., equivalent to the static size limit. When we consider the last row, we can see that the aimdT is faster than the aimdTQ. However, this last policy solves more problems. We can explain this as follows. The quality-based policy intensifies the search by favoring the exchange of clauses related to the current exploration of each unit. This intensification leads to the resolution of more difficult problems. However, it increases the runtime on easier instances where a more diversified search is often more beneficial. Overall these results are very good since our dynamic policies are able to outperform the best possible static tuning.

3.7 Summary

We have presented ManySAT, a portfolio-based parallel SAT solver which advantageously exploits multicore architectures. ManySAT is based on an understanding of the main weakness of modern sequential SAT solvers, their sensitivity to parameter tuning and their lack of robustness. As a result, ManySAT uses a portfolio of complementary sequential algorithms, and lets them cooperate in order to improve further the overall performance. This design philosophy of ManySAT, which

Table 3.5 SAT Race 2008: industrial problems

Family/instance	#inst	ManySAT $e = 8$		ManySAT aimdT			ManySAT aimdTQ		
		#Solved	Time(s)	#Solved	Time(s)	$\bar{e}$	#Solved	Time(s)	$\bar{e}$
ibm_*	20	19	**204**	19	218	7	19	286	6
manol_*	10	10	**117**	10	**117**	8	10	205	7
mizh_*	10	6	762	7	746	6	**10**	**441**	5
post_*	10	9	325	9	**316**	7	9	375	7
velev_*	10	8	585	8	**448**	5	8	517	7
een_*	5	5	2	5	2	8	5	2	7
simon_*	5	5	111	5	84	10	5	**59**	9
bmc_*	4	4	7	4	7	7	4	**6**	9
gold_*	4	1	1160	1	**1103**	12	1	1159	12
anbul_*	3	2	742	**3**	**211**	11	3	689	11
babic_*	3	3	2	3	2	8	3	2	8
schup_*	3	3	129	3	**120**	5	3	160	5
fuhs_*	2	2	90	2	**59**	11	2	77	10
grieu_*	2	1	783	1	**750**	8	1	**750**	8
narain_*	2	1	786	1	**776**	8	1	792	8
palac_*	2	2	20	2	**8**	3	2	54	7
aloul-chnl11-13	1	0	1500	0	1500	11	0	1500	10
jarvi-eq-atree-9	1	1	70	1	69	25	1	**43**	17
marijn-philips	1	0	1500	**1**	1133	34	1	**1132**	29
maris-s03-gripper11	1	1	11	1	11	10	1	11	8
vange-col-abb313gpia-9-c	1	0	1500	0	1500	12	0	1500	12
Total/(average)	100	83	10406	86	9180	(10.28)	89	9760	(9.61)

clearly contrasts with well-known parallel SAT solvers, is directly inspired by our work in the previous M-framework for distributed constraint satisfaction problems. The good performance obtained by ManySAT on industrial SAT instances clearly suggests that the parallel portfolio approach is more interesting than the traditional divide-and-conquer one.

We have also presented how knowledge sharing policies could be finely controlled through dynamic clause sharing policies which can adjust the size of shared clauses between any pair of processing units. The first policy controls the overall number of exchanged clauses whereas the second policy additionally exploits the relevance quality of shared clauses. This part has been described in [HJS09a].

As stated here, our four-core portfolio was carefully crafted in order to mix complementary strategies. If ManySAT could be run on dozens of computing units, what would be the performance? We have considered this question in a more general context in [BHS09]. This work presents the first study on scalability of constraint solving on 100 processors and beyond. It proposes techniques that are simple to apply and shows empirically that they scale surprisingly well. It proves that portfolio-based approaches can also scale up to several dozens of processors.

Finally, as stated in the introduction, SAT is now applied to other domains. One domain which particularly benefits from the recent advances in SAT is Satisfiability Modulo Theory [NOT06]. There, our ManySAT approach has been integrated with the Z3 SMT solver [dMB08], allowing it to achieve impressive speed ups on several classes of problems [WHdM09].

In the next chapter we will see that the parallel portfolio approach can be used to boost the performance of local search algorithms.

Chapter 4
Parallel Local Search for Satisfiability

4.1 Introduction

As we have seen in the previous chapter, complete parallel solvers for the propositional satisfiability problem have received significant attention. This push towards parallelism in complete SAT solvers has been motivated by their practical applicability. Indeed, many domains, from software verification to computational biology and automated planning, rely on their performance. On the other hand, since the application of local search solvers has been mainly focused on random instances, their parallelization has not received much attention so far. The main contribution of the parallelization of local search algorithms for SAT solving basically executes a portfolio of independent algorithms which compete without any communication between them. In our settings, each member of the portfolio shares its best configuration (i.e., one which minimizes the number of conflicting clauses) in a common structure. At each restart point, instead of classically generating a random configuration to start with, each algorithm aggregates the shared knowledge to carefully craft a new starting point. We present several aggregation strategies and evaluate them on a large set of instances. Our best strategies largely improve over a parallel portfolio of non-cooperative local searches. We also present an analysis of configuration diversity during parallel search, and find out that the best aggregation strategies are the ones which are able to maintain a good diversification/intensification tradeoff. This chapter extends the conference paper [AH11]. It is organized as follows. Section 4.2 describes previous work on parallel SAT and cooperative algorithms. Section 4.3 gives background material. Section 4.4 presents our methodology and our aggregation strategies, Sect. 4.5 evaluates them, and Sect. 4.6 presents some concluding remarks and future research directions.

Y. Hamadi, *Combinatorial Search: From Algorithms to Systems*,
DOI 10.1007/978-3-642-41482-4_4,

4.2 Previous Work

4.2.1 Incomplete Methods for Parallel SAT

In [ZHZ02] the authors proposed PSAT, a hybrid algorithm that starts with a standard DPLL algorithm to divide the problem space into sub-spaces. Each sub-space is then allocated to a given local search algorithm (Walksat). Experimental results reported in the paper suggested that PSAT greatly outperformed the sequential version of WSAT.

PGSAT [Rol02] is a parallel version of the GSAT algorithm. The entire set of variables is randomly divided into τ subsets and allocated to different processors. In this way, at each iteration, if no global solution has been obtained, the ith processor uses the GSAT score function (see Sect. 4.3) to select and flip the best variable for the ith subset. Another contribution to this parallel architecture is described in [RBB05] where the authors aim to combine PGSAT and random walk. Thus at each iteration, the algorithm performs a random walk step with a certain probability wp, that is, a random variable from an unsatisfied clause is flipped. Otherwise, PGSAT is used to flip τ variables in parallel at a cost of reconciling partial configurations to test if a solution is found.

gNovelty+-v2 [PG09] belongs to the portfolio approach. This algorithm executes n independent copies of the *gNovelty+-v2* algorithm (each one with a different random seed) in parallel, until at least one of them finds a solution or a given time-out is reached. This algorithm was the only parallel local search solver submitted to the *random* category of the 2009 SAT Competition.[1] Furthermore, in [Hoo98] and [CIR12] the authors present a detailed analysis of the runtime distribution of this parallel framework in the context of SAT and MaxSAT.

In [KSGS09], the authors studied the application of a parallel hybrid algorithm to deal with the MaxSAT problem. This algorithm combines a complete solver (Minisat) and an incomplete one (Walksat). Broadly speaking, both solvers are launched in parallel and Minisat is used to guide Walksat to promising regions of the search space by suggesting values for the selected variables. Other literature related to the application of the parallel portfolio approach without cooperation to the MaxSAT problem includes [PPR96] and [ARR02].

4.2.2 Cooperative Sequential Algorithms

In [HW93] a set of algorithms running in parallel exchange hints (i.e., partial valid solutions) to solve hard graph coloring instances. To this end, they share a blackboard where they can write a hint with a given probability q and read a hint with a given probability p.

[1] http://www.satcompetition.org/2009/.

Algorithm 4.1 Local Search For SAT (CNF formula F, Max-Flips, Max-Tries)

```
1: for try := 1 to Max-Tries do
2:    A := initial-configuration(F)
3:    for flip := 1 to Max-Flips do
4:       if A satisfies F then
5:          return A
6:       end if
7:       x := select-variable(A)
8:       A := A with x flipped
9:    end for
10: end for
11: return 'No solution found'
```

Although *Averaging in Previous Near Solutions* [SK93] is not a cooperative algorithm by itself, this method is used to determine the initial configuration for the ith restart in the GSAT algorithm. Broadly speaking, the initial configuration is computed by performing a bitwise average of variables of the best solution found during the previous restart ($restart_{i-1}$) and two restarts before ($restart_{i-2}$). That is, variables with the same values in both configurations are reused, and the extra set of variables is initialized with random values. Since over time, configurations with few conflicting clauses tend to become similar, all the variables are randomly initialized after a given number of restarts.

4.3 Technical Background

4.3.1 Local Search for SAT

The propositional satisfiability problem (SAT) is defined in Chap. 3. Algorithm 4.1 describes a well-known local search procedure for SAT solving. It starts with a random assignment for the variables (so-called configuration[2]), *initial-configuration* in line 2. The key point of local search algorithms is depicted in lines (3–9) where the algorithm flips the most appropriate candidate variable until a solution is found or a given number of flips is reached (Max-Flips). After this process the algorithm restarts itself with a new (fresh) random configuration.

As one may expect, a critical part of the algorithm is the variable selection function (line 7, *select-variable*), which indicates the next variable to be flipped in the current iteration of the algorithm. Broadly speaking, there are two main categories of variable selection functions. The first of these is motivated by the GSAT algorithm [SLM92] based on the following score function.

$$score(x) = breakcount(x) - makecount(x)$$

[2]In the following, we use the term configuration as a synonym for assignment for the variables.

Intuitively, *breakcount*(x) indicates the number of clauses that are currently satisfied but by flipping x become unsatisfied, and *makecount*(x) indicates the number of clauses that are unsatisfied but by flipping x become satisfied. In this way, local search algorithms select the variable with minimal score value (preferably with a negative value), because flipping this variable would most likely increase the chances of solving the instance.

The second category of variable selection functions is the Walksat-based one [SKC94b] which works as follows. First, the algorithm selects, uniformly at random, an unsatisfied clause c. Then, one variable appearing in c is selected according to a given heuristic function. The selected variable usually improves (i.e., decreases) the number of unsatisfied clauses in the formula.

Alternatively, some local search approaches aim at proving unsatisfiability. For instance, [FR04] and [CI96] integrate new clauses to the initial clause database (or problem definition). Broadly speaking, as soon as a local minimum is reached (none of the candidate variables reduces the number of unsatisfied clauses), two neighbor clauses of the form $(x_1 \vee X)$ and $(\bar{x}_1 \vee Y)$ are used to infer a new clause $(X \vee Y)$. Along the same lines, [ALMS09] introduces the concept of CDLS which adapts Conflict Driven Clause Learning to local search. Other local search literature to prove unsatisfiability includes [PL06].

The next section presents a more detailed description of the variable selection functions.

4.3.2 Refinements

This section reviews the main characteristics of state-of-the-art local search solvers for SAT solving. As pointed out before, these algorithms have been developed to deal with the variable selection function. In the following, we describe several well-known mechanisms for selecting the most appropriate variable to flip at a given state of the search.

TSAT [MSG97a] extends the *GSAT* algorithm [SLM92] by proposing the use of a tabu list. This list contains a set of recently flipped variables in order to avoid flipping the same variable for a given number of iterations. This way, the tabu list helps prevents search stagnation.

Novelty [MSK97] firstly selects an unsatisfied clause c and from c selects the best v_{best} and second best v_{2best} variable candidates. The former, v_{best}, is flipped iff it is not the most recently flipped variable in c. Otherwise v_{2best} is flipped with a given probability p and v_{best} with probability $1 - p$. Important extensions to this algorithm are *Novelty+* [Hoo99a], *Novelty++* [LH05], and *Novelty+p* [LWZ07].

Novelty+ [Hoo99a] with a given probability wp (random walk probability) selects a random variable from an unsatisfied clause and with probability $1 - wp$ uses *Novelty* as a backup heuristic.

Adaptive Novelty+ (AN+) [Hoo02b] uses an adaptive mechanism to properly tune the noise parameter (wp) of Walksat-like algorithms (e.g. *Novelty+*). wp is initially set to 0 and as soon as search stagnation is observed (i.e., no improvement

has been observed for a while) wp is incremented as follows: $wp := wp + (1 + wp) \times \phi$. On the other hand, whenever an improvement is observed wp is decreased as follows: $wp := wp - wp \times \phi/2$. This adaptive mechanism has shown impressive results, and was used to improve the performance of other local search algorithms in the context of SAT solving, e.g. TNM [WL09] and RSAPS [HTH02].

Scaling and Probabilistic Smoothing (SAPS) [HTH02] adds a weight penalty to each clause. These weights are initialized to 1 and updated during the search process. Generally speaking, SAPS maintains a list L which contains a set of variables whose objective value (i.e., sum of all unsatisfied clause weights) is maximal and greater than a given threshold $SAPS_{thresh}$. If L is not empty, SAPS selects, uniformly at random, one of the variables in L. Otherwise, a random variable is selected with a probability wp, and with probability $1 - wp$ SAPS performs a two-step procedure in order to scale and smooth clause penalties.

The scaling procedure updates the weight of all unsatisfied clauses as follows: $weight_i := weight_i \times \alpha$. The smoothing procedure updates, with a probability P_{smooth}, all clause penalties as follows: $weight_i := weight_i \times \rho + (1 - \rho) \times w$, where w indicates the average weight over all clauses and ρ is a parameter which remains fixed during the entire search process. SAPS uses five parameters (α, ρ, wp, P_{smooth}, and $SAPS_{thresh}$) that need to be tuned in order to achieve a top performance. Taking this into account, [HHHLB06] proposes a machine learning framework to identify the most suitable values for the parameters of the algorithm, and [HHLBS09] studies the application of paramILS, a parameter tuning algorithm, to identify promising parameters to solve a given benchmark family.

Reactive SAPS (RSAPS) [HTH02] extends SAPS by adding an automatic tuning mechanism to identify suitable values for P_{smooth}. This parameter is increased, i.e., $P_{smooth} := P_{smooth} + 2 \times \delta \times (1 - P_{smooth})$, if an improvement has been observed in the current iteration. The value of the parameter is decreased, i.e., $P_{smooth} := \delta \times P_{smooth}$, if no improvement has been observed after $\theta \times |\mathcal{C}|$ iterations. Moreover, $|\mathcal{C}|$ denotes the number of clauses in the problem; δ and θ are constants set to 0.1 and $1/6$.

Pure Additive Weighting Scheme (PAWS) [TPBF04], similarly to SAPS, has each clause associated with a weight penalty. However, in this case the weight scaling step is replaced with an additive one ($weight_i := weight_i + 1$). Moreover, when no variable provides an improvement in the objective function, a variable that does not degrade the objective is selected with a given probability P_{flat}. Finally, PAWS decreases weights after Max_{inc} increases.

Novelty++ [LH05] with a given probability dp (diversification probability) flips the most recently flipped variable from the selected unsatisfied clause. Otherwise, with probability $1 - dp$ the algorithm uses *Novelty* as a backup heuristic.

G^2WSAT [LH05] (G2) introduces the concept of promising decreasing variables. Broadly speaking, a variable is decreasing if flipping it reduces the overall number of failed clauses. Initially (line 2 in Algorithm 4.1), all variables are marked as promising decreasing; then the status of the variables is updated by observing the total gain (w.r.t. the objective) after flipping the variable (line 8 in Algorithm 4.1). That is, a variable becomes non-promising if flipping it increased the overall number

of failed clauses. In addition, all variables that become decreasing as a result of the flip are marked as promising decreasing.

Taking this into account, G2 selects the best promising variable. If there are no promising decreasing variables, the algorithm uses *Novelty++* as a backup heuristic. Similarly, $G^2WSAT{+}p$ (G2+p) also uses the concept of promising decreasing variables. However, in this case the algorithm selects the least recently flipped promising variable, and *Novelty+p* is used as a backup heuristic.

Novelty+p [LWZ07] introduces the concept of promising score (*pscore*) for a given variable as follows:

$$pscore(x) = score_A(x) + score_B\left(x'\right)$$

where A is the current assignment for the variables, B is the configuration after flipping x, and x' the best promising decreasing variable with respect to B. Similarly to *Novelty*, *Novelty+p* starts by selecting v_{best} and v_{2best} from an unsatisfied clause c. Afterwards, if v_{best} is the most recently flipped variable in c, then with a probability p the algorithm selects v_{2best} and with probability $1 - p$ it uses the promising score to select the next variable. Finally, if v_{best} is not the most recently flipped variable in c but was flipped after v_{2best}, then v_{best} is selected. Otherwise, the promising score is used to select the best variable.

Adaptive G^2WSAT (AG2) [LWZ07] aims to integrate the AN+ adaptive noise mechanism into the G^2WSAT algorithm. That is, the noise value is initially set to 0 and updated during the execution of the algorithm. Intuitively, the noise is decreased as soon as an improvement is observed in the objective function, and decreased if no improvement has been observed after a given number of iterations. *Adaptive* $G^2WSAT{+}p$ (AG2+p) uses $G^2WSAT{+}p$ with the same adaptive noise mechanism of AG2.

gNovelty+ [PTGS08] combines properties of four well-known algorithms: AN+, G2, PAWS, and SAPS. As in SAPS, clauses are associated with penalty weights whose initial value is 1. The algorithm starts by selecting, with a probability wp, a random variable from an unsatisfied clause; otherwise, with probability $1 - wp$, the G2 mechanism is used to select a variable from the list of promising decreasing variables. If this list is empty, *gNovelty+* selects the variable with best improvement in the objective function (sum of all unsatisfied clause penalties); ties are broken using the flip history. After selecting the most appropriate variable, clause penalties are updated, i.e., increasing by one unit the weight of unsatisfied clauses, and finally with a probability sp the weight of all clause penalties is decreased by one unit.

Two Noise Mechanisms (TNM) [WL09] interleaves the execution of two adaptive noise methods in order to solve a given SAT instance. The first is AG2+, the second is a new method in which the algorithm adds two new variables per clause; var_false and num_false. The former indicates the variable that most recently falsified each clause, while the latter indicates the number of times that var_false consecutively falsified its associated clause. If the best variable corresponds to var_false, then the noise value is set to 0; otherwise it is set to num_false. This adaptive noise mechanism is included into AG2 and named AG2′. Finally, the use of one method or another is defined according to a new parameter called γ.

4.4 Knowledge Sharing in Parallel Local Search for SAT

Our objective is to extend a parallel portfolio of state-of-the-art local search solvers for SAT with knowledge sharing and cooperation. Each algorithm is going to share with others the best configuration it has found so far with the cost (number of unsatisfied clauses) of the respective configuration in a shared pair $\langle M, C \rangle$.

$$M = \begin{pmatrix} X_{11} & X_{12} & \dots & X_{1n} \\ X_{21} & X_{22} & \dots & X_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ X_{c1} & X_{c2} & \dots & X_{cn} \end{pmatrix} \qquad C = [C_1, C_2, \dots, C_c]$$

where n indicates the total number of variables of the problem and c indicates the number of local search algorithms in the portfolio. In the following we are associating local search algorithms and processing cores. Each element X_{ji} in the matrix indicates the ith variable of the best configuration found so far by the jth core. Similarly, the jth element in C indicates the cost for the respective configuration in M. Notice that M is updated iff a better configuration is observed (i.e., a configuration with better cost).

These best configurations can be exploited by each local search procedure to build a new initial configuration. In the following, we propose seven strategies to determine the initial configuration (cf. function *initial-configuration* in Algorithm 4.1).

4.4.1 Using Best Known Configurations

In this section, we propose three methods to build the new initial configuration *init* by aggregating best known configurations. In this way, we define $init_i$ for all the variables $X_i, i \in [1..n]$ as follows:

1. *Agree*: if there exists a value v such that $v = X_{ji}$ for all $j \in [1..c]$ then $init_i = v$; otherwise the value is chosen uniformly at random.
2. *Majority*: if there exist two values v and v' such that $|\{X_{ji} = v | j \in [1..c]\}| > |\{X_{ji} = v' | j \in [1..c]\}|$ then $init_i = v$; otherwise the value is chosen uniformly at random.
3. *Prob*: $init_i = 1$ with probability $p_{ones} = \frac{ones}{c}$ and $init_i = 0$ with probability $1 - p_{ones}$, where $ones = |\{X_{ji} = 1 | j \in [1..c]\}|$.

4.4.2 Weighting Best Known Configurations

In contrast with our previous methods where all best known solutions are considered equally important, the methods proposed in this section use a weighting mechanism to consider the cost of best known configurations. The computation of the initial configuration *init* uses one of the following two weighting systems: *Ranking* and

Normalized Performance, where values from better configurations are most likely to be used.

Ranking This method sorts the configurations of the shared matrix from worst to best according to their cost. The worst ranked configuration gets weight of 1 (i.e., $RankW_1 = 1$), and the best ranked c (i.e., $RankW_c = c$).

Normalized Performance This method assigns weights ($NormW$) considering a normalized value of the number of unsatisfied clauses of the configuration:

$$NormW_j = \frac{|\mathcal{C}| - C_j}{|\mathcal{C}|}$$

Using the previous two weighting mechanisms, we define the following four additional methods to determine initial configurations.

To this end, we define $\Phi(val, Weight) = \sum_{k \in \{j | X_{ji} = val\}} Weight_k$.

1. *Majority RankW*: if there exist two values v and v' such that $\Phi(v, RankW) > \Phi(v', RankW)$ then $init_i = v$; otherwise the value is chosen uniformly at random.
2. *Majority NormalizedW*: if there exist two values v and v' such that $\Phi(v, NormW) > \Phi(v', NormW)$ then $init_i = v$; otherwise the value is chosen uniformly at random.
3. *Prob RankW*: $init_i = 1$ with probability $P_{Rones} = \frac{Rones}{Rones+Rzeros}$ and $init_i = 0$ with probability $1 - P_{Rones}$, where $Rones = \Phi(1, RankW)$ and $Rzeros = \Phi(0, RankW)$.
4. *Prob NormalizedW*: $init_i = 1$ with probability $P_{Nones} = \frac{Nones}{Nones+Nzeros}$ and $init_i = 0$ with probability $1 - P_{Nones}$, where $Nones = \Phi(1, NormW)$ and $Nzeros = \Phi(0, NormW)$.

4.4.3 Restart Policy

As mentioned earlier on, shared knowledge is exploited when a given algorithm is restarted. At this point, the current working configuration of a given algorithm is re-initialized according to a given aggregation strategy. However, it is important to restrict cooperation since it adds overheads, and more importantly, tends to generate similar configurations. As will be described in Sect. 4.5.4, a key point for a cooperative portfolio is to properly balance the *diversification* and *intensification* of the acquired knowledge. Too much diversification results in performance similar to that of a portfolio without cooperation, and too much intensification ends up in a parallel portfolio where all the algorithms explore very similar regions of the search space.

We propose a new restart policy to avoid re-initializing the working configuration again and again. This new policy re-initializes the working configuration for a given restart (i.e., every MaxFlips) if and only if performance improvements in best known solutions have been observed during the latest restart window. This new restart policy is formally described in the following definition. Let bc_{ki} denote the

cost (number of unsatisfied clauses) of the best known configuration produced by core i up to the $(k-1)$-th restart.

Definition 4.1 At a given restart k for a given algorithm i the working configuration is reinitialized iff there exists an algorithm q such that $bc_{kq} < bc_{(k-1)q}$, with $q \neq i$.

4.5 Experiments

This section reports on the experimental validation of the proposed aggregation strategies.

4.5.1 Experimental Settings

We conducted experiments using instances from the RANDOM category of the 2009 SAT competition. We removed instances whose status was reported as UNKNOWN in the competition and considered a collection of 359 satisfiable instances, which were divided in the competition into two main groups: large and medium, each group itself containing sets of k-SAT instances, where k indicates the number of literals for each clause. Large size represents a set of 88 3-SAT, 49 5-SAT, and 27 7-SAT instances, while medium size represents a set of 110 3-SAT, 40 5-SAT, and 45 7-SAT instances. The proportion clauses/variables in the large size group ranges from 33,600/8,000 to 42,000/10,000 for 3-SAT, 18,000/900 to 20,000/1,000 for 5-SAT, and 17,000/200 to 18,000/900 for 7-SAT, while in the medium size group the proportion ranges from 1530/360 to 2,380/560 for 3-SAT, 1,917/90 to 2,130/100 for 5-SAT, and 5,340/60–6,675/75 for 7-SAT.

We decided to build our parallel portfolio on top of UBCSAT 1.1, a well-known local search library that provides an efficient implementation of the latest local search for SAT algorithms [TH04]. We did preliminary experiments to extract from this library the eight algorithms which perform best on our set of problems. From that, we defined the following three baseline portfolio constructions where algorithms are independent searches without cooperation. The first one, *pcores-PAWS*, uses p copies of the best single algorithm (PAWS); the second portfolio, *4cores-No sharing*, uses the best subset of four algorithms (PAWS, G2+p, AG2, AG2+p); and the last one, *8cores-No sharing*, uses all the eight algorithms (PAWS, G2+p, AG2, AG2+p, G2, SAPS, RSAPS, AN+). All the algorithms were used with their default parameters and without any restart, since these techniques are equipped with important diversification strategies and usually perform better when the restart flag is switched off. For example, algorithms such as [WL09, HT07, WLZ08] have eliminated the restart mechanism of their default local search procedures.

This portfolio construction can be seen as the *best virtual portfolio* (BVP) on the entire set of instances. The portfolio, which selects a set of four and eight algorithms, maximizes the overall number of solved instances. Notice that the BVP might change from instance to instance and is not known in beforehand. However,

it is worth mentioning that this portfolio construction is a near-optimal one on instances of the 2007 SAT competition, confirming the robustness of this portfolio by considering all available algorithms of the library.

Moreover, we also consider the best (TNM) and second best (gNovelty+-v2) local search algorithms of the 2009 SAT competition; two complete local search solvers: CDLS [ALMS09] and clsHai04 [FR04]; and Walksat||Minisat [KSGS09]. Once again, all these solvers are used with their default parameters.

On the other hand, the knowledge aggregation mechanisms described in the previous section were built on top of a portfolio with four algorithms (same algorithms as *4cores-No sharing*) and a portfolio with eight algorithms (same algorithms as *8cores-No sharing*). There, we used the modified restart policy described in Sect. 4.4.3 with *MaxFlips* set to 10^6.

All tests were conducted on a cluster of eight Linux Mandriva machines with 8 GB of RAM, two Quad core (eight cores) 2.33 GHz Intel Xeon Processors E5345, and 128 KB L1 cache and 8 MB L2 cache for each processor. In all the experiments, we used a five-minute time-out (300 seconds) for each algorithm in the portfolio, so that for each experiment the total CPU time was set to $c \times 300$ seconds, where c indicates the number of algorithms in the portfolio.

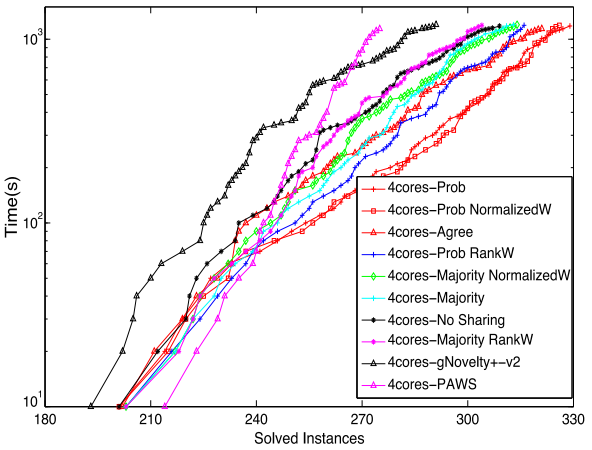
We ran each solver 10 times on each instance (each time with a different random seed) and reported two metrics. First, the *Penalized Average Runtime* (PAR-10) [HHLB10] which computes the average runtime over all instances, but where unsolved instances are considered as $10\times$ the cutoff time. Second, the runtime for each instance, which is calculated as the median across the 10 runs. Overall, our experiments for these 359 SAT instances took 1,914 days of CPU time.

4.5.2 Practical Performances with Four Cores

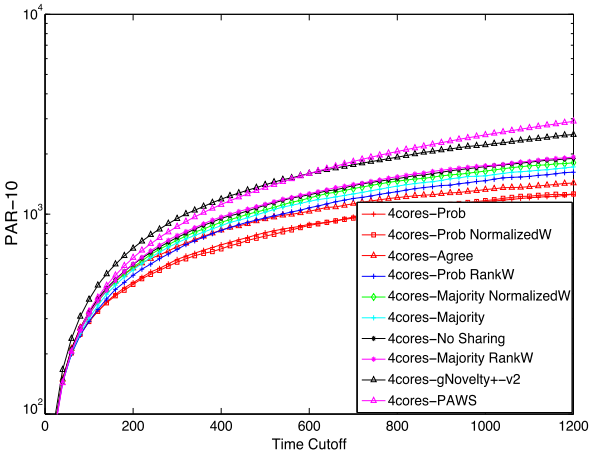
Figure 4.1(a) shows the results of each aggregation strategy using a portfolio with four cores, comparatively to the four core baseline portfolios. The *x-axis* gives the number of problems solved and the *y-axis* presents the cumulated runtime on a log-scale.

As expected, the portfolio with the top four best algorithms (*4cores-No Sharing*) performs better (solving 309 instances) than the one with four copies of the best algorithms (*4cores-PAWS*) (solving 275 instances). Additionally, Fig. 4.1(b) shows the performance when considering the PAR-10 metric. The *y-axis* (log-scale) shows the *Penalized Average Runtime* for a given time cutoff given on the *x-axis*. In this figure, it can be observed that the aggregation policies are also efficient when varying the time limit to solve problem instances.

The performance of the portfolios with knowledge sharing is quite good. Overall, it seems that adding a weighting mechanism can often hurt the performance of the underlying aggregation strategy. Among the weighting options, it seems that the Normalized Performance performs better. The best portfolio implements the *Prob* strategy without any weighting (solving 329 instances). This corresponds to a gain of 20 problems against the corresponding *4cores-No Sharing* baseline.



(a) Number of solved instances



(b) Penalized Average Runtime

Fig. 4.1 Performance using four cores in a given amount of time

A detailed examination of *4cores-Prob* and *4cores-No Sharing* is presented in Figs. 4.2 and 4.3. The first figure uses log-scales on each axis. These figures show, respectively, a runtime and a best configuration cost comparison. Notice that each number in Fig. 4.3 represents the sum of the overlapping points at that (x, y) location. In both figures, points below (resp. above) the diagonal line indicates that *4cores-Prob* performs better (resp. worse) than *4cores-No Sharing*. In the runtime comparison, we observe that easy instances are correlated as they require few steps to be solved, and for the remaining set of instances *4cores-Prob* usually exhibits a better performance. On the other hand, the second figure shows that when instances are not solved, the median cost of the best configuration (number of unsatisfied clauses) found by *4cores-Prob* is usually better than for *4cores-No Sharing*. In particular, *4cores-Prob* reports a better solution cost for 38 instances, while *4cores-No Sharing* was better for only six instances.

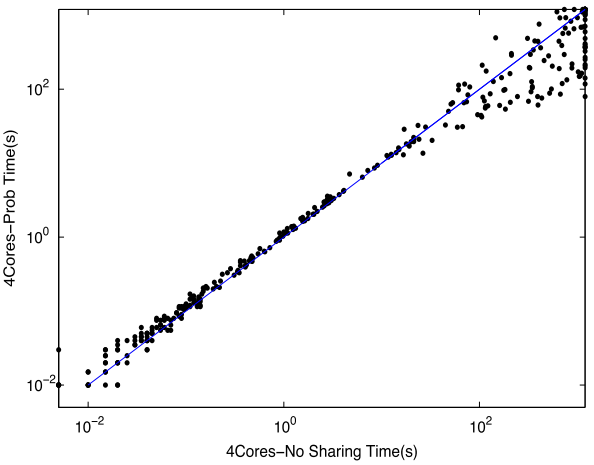


Fig. 4.2 Runtime comparison; *each point* indicates the runtime to solve a given instance using *4cores-Prob* (*y-axis*) and *4cores-No Sharing* (*x-axis*)

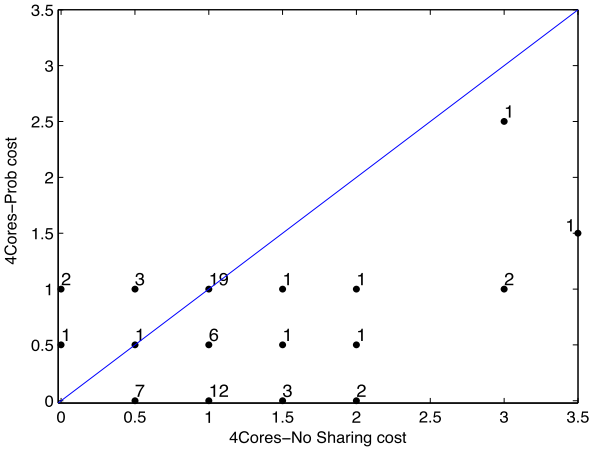
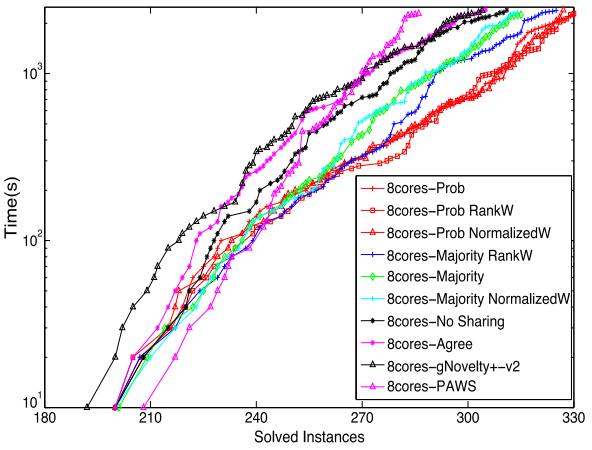


Fig. 4.3 Best configuration cost comparison on unsolved instances. *Each point* indicates the best configuration (median) cost of a given instance using *4cores-Prob* (*y-axis*) and *4cores-No Sharing* (*x-axis*)

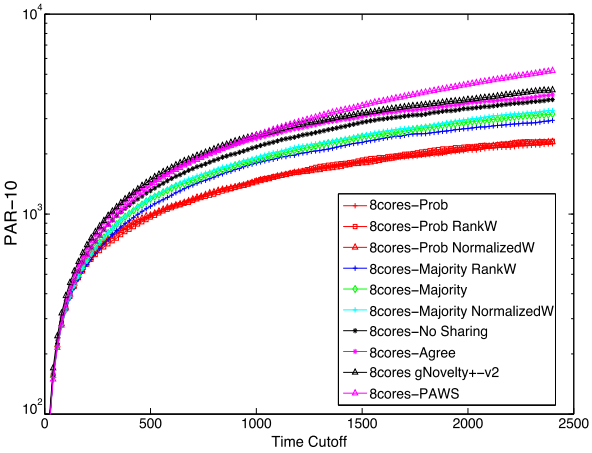
Table 4.1 summarizes all the experiments using four cores; each cell indicates the results for medium-size instances (top), large-size instances (middle), and the total overall instances (bottom) for each portfolio. We report the number of solved instances (#solved), the median time across all instances (median time), the *Penalized Average Runtime* (PAR), and the total number of instances that timed out in all the 10 runs (never solved). These results confirm that sharing best known configurations outperforms independent searches. For instance, *4cores-Prob* and *4cores-prob NormalizedW* solved respectively 20 and 17 more instances than *4cores-No Sharing*, and all the cooperative strategies (except *4cores-majority RankW*) exhibit better PAR. Interestingly, *4cores-PAWS* exhibited the best median runtime overall in the experiments with four cores; this fact suggests that PAWS by itself is able to quickly solve an important number of instances. Moreover, only two instances timed out in all the 10 runs for *4cores-Agree* and *4cores-prob NormalizedW* against seven for *4cores-No Sharing*. Notice that this table also includes *1core-PAWS*, the

Table 4.1 Overall evaluation using four cores. Each cell summarizes the results of medium-size instances (*top*), large-size instances (*middle*), and the total overall instances (*bottom*). The best strategy for each column is highlighted in *bold*

Strategy	#solved	Median time	PAR-10	Never solved
1core-TNM	195	0.07	1.48	0
	113	121.10	1007.91	8
	308	1.30	461.24	8
1core-gNovelty+-v2	195	0.11	2.38	0
	74	300.00	1637.09	38
	269	2.52	749.16	38
1core-PAWS	195	0.07	1.67	0
	54	300.00	1992.58	71
	249	1.76	911.17	71
4cores-gNovelty+-v2	195	0.28	3.87	0
	96	855.4	5771.75	34
	291	5.075	2501.73	33
4cores-PAWS	195	**0.08**	1.48	0
	80	**1200.00**	6379.67	61
	275	**1.63**	2915.19	61
4cores-No Sharing	195	0.11	1.84	0
	114	533.04	4159.15	7
	309	2.19	1901.00	7
4cores-Agree	195	0.12	1.70	**0**
	126	313.59	3131.19	**2**
	321	2.54	1431.33	**2**
4cores-Majority	195	0.11	1.95	0
	118	343.74	3773.63	11
	313	2.53	1724.94	11
4cores-Prob	**195**	0.11	**2.02**	0
	134	195.81	**2751.24**	4
	329	2.51	**1257.93**	4
4cores-Majority RankW	195	0.12	2.07	0
	109	518.39	4223.70	11
	304	2.47	1930.61	11
4cores-Majority NormalizedW	195	0.13	1.94	0
	119	447.06	3954.19	9
	314	2.48	1807.42	9
4cores-Prob RankW	195	0.12	1.97	0
	121	259.97	3546.78	7
	316	2.53	1621.33	7
4cores-Prob NormalizedW	195	0.12	2.03	**0**
	131	180.39	2759.74	**2**
	326	2.50	1261.82	**2**



(a) Number of solved instances



(b) Penalized Average Runtime

Fig. 4.4 Performance using eight cores in a given amount of time

best sequential local search on this set of problems. The PAR-10 score for *1core-PAWS* is lower than the other values of the table since this portfolio uses only one algorithm, therefore, the time-out is only 300 seconds, while four-core portfolios use a time-out of 1,200 seconds.

4.5.3 Practical Performances with Eight Cores

We now move on to portfolios with eight cores. The results of these experiments are depicted in Fig. 4.4, indicating the total number of solved instances within a given amount of time. As in previous experiments, we report the results of baseline portfolios *8cores-No Sharing* and *8cores-PAWS*, as well as the seven cooperative strategies. We observe that the cooperative portfolios (except *8cores-Agree*)

greatly outperform the non-cooperative ones in both the number of solved instances (Fig. 4.4(a)) and the PAR-10 metric (Fig. 4.4(b)). Indeed, as detailed in Sect. 4.5.4, *8cores-Agree* exhibits a poor performance mainly because the best known configurations stored in the shared data structure tend to be different from each other. Therefore, this policy tends to generate completely random starting points, and cannot exploit the acquired knowledge.

Table 4.2 summarizes the results of this experiment, and once again, it includes the best individual algorithm running in a single core. We can observe that *8cores-Prob*, *8cores-Prob RankW*, and *8cores-Prob NormalizedW* solve 24, 22, and 16 more instances than *8cores-No Sharing*. Furthermore, it shows that knowledge sharing portfolios are faster than individual searches, with a PAR-10 of 3,743.63 seconds for *8cores-No Sharing* against 2,247.97 for *8cores-Prob*, 2,312.80 for *8cores-Prob RankW*, and 2,295.99 for *8cores-Prob NormalizedW*. Finally, it is also important to note that only one instance timed out in all the 10 runs for *8cores-Prob NormalizedW*, against eight instances for *8cores-No Sharing*.

These experimental results show that *Prob* (four and eight cores) exhibited the overall best performance. We attribute this to the fact that the probability component of this method balances the exploitation of best solutions found so far with the exploration of other values for the variables; therefore, the algorithm diversifies the search by exploring new starting configurations.

4.5.4 Analysis of the Diversification/Intensification Trade-off

Maintaining an appropriate balance between *diversification* and *intensification* of the acquired knowledge is an important step of the proposed cooperative portfolios to improve performance. In this chapter, *diversification* (resp. *intensification*) refers to the ability of generating different (resp. similar) initial configurations at each restart.

Figure 4.5 aims to analyze the balance between *diversification* and *intensification*. The *x-axis* gives the pairwise average Hamming distance (*HamDis*) of all pairs of algorithms in a portfolio after a given number of flips (*y-axis*) for a typical SAT instance.[3] Notice that some lines are of different sizes because some strategies required fewer flips to solve the instance. *HamDis* is formally described as follows:

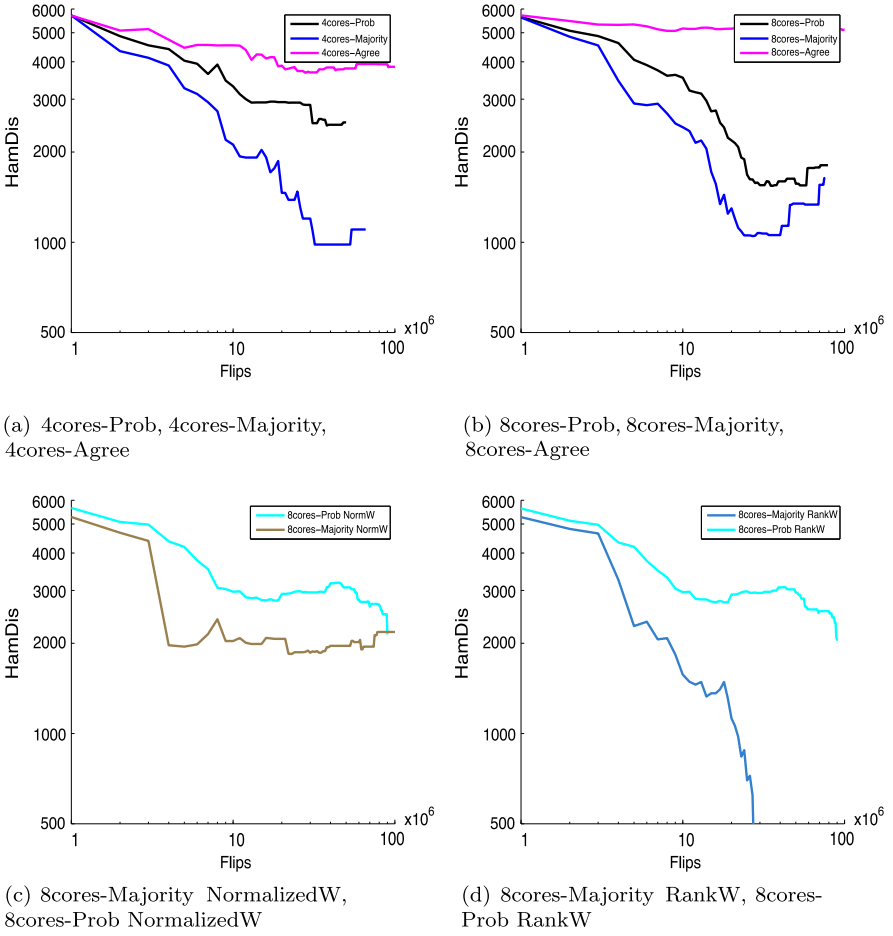
$$HamDis = \frac{\sum_{i=1}^{c} \sum_{j=i+1}^{c} Hamming(X_i, X_j)}{c(c-1)/2}$$

where X_i and X_j indicate the best configurations found so far for the ith and jth algorithms in the portfolio; c represents the number of algorithms in the portfolio; and $Hamming(X_i, X_j)$ corresponds to the number of variables in X_i and X_j assigned to different values, that is, $Hamming(X_i, X_j) = |\{k : X_{ik} \neq X_{jk}\}|$.

[3] We performed the same experiment on several instances and observed similar behavior.

Table 4.2 Overall evaluation using eight cores. Each cell summarizes the results of medium-size instances (*top*), large-size instances (*middle*), and the total overall instances (*bottom*). The best strategy for each column is highlighted in *bold*

Strategy	#solved	Median time	PAR-10	Never solved
1core-TNM	195	0.07	1.48	0
	113	121.10	1007.91	8
	308	1.30	461.24	8
1core-gNovelty+-v2	195	0.11	2.38	0
	74	300.00	1637.09	38
	269	2.52	749.16	38
1core-PAWS	195	0.07	1.67	0
	54	300.00	1992.58	71
	249	1.76	911.17	71
8cores-gNovelty+-v2	195	0.305	3.72	0
	109	1164.98	6113.02	33
	304	4.66	4173.14	33
8cores-PAWS	195	**0.07**	1.52	0
	91	**1482.08**	11411.41	56
	286	**2.00**	5213.84	56
8cores-No Sharing	195	0.125	1.86	0
	116	937.64	8192.69	8
	311	2.33	3743.63	8
8cores-Agree	195	0.15	1.92	0
	110	1251.10	8649.17	17
	305	2.48	3952.19	17
8cores-Majority	195	0.13	2.11	0
	120	650.56	6921.42	6
	315	2.47	3163.02	6
8cores-Prob	**195**	0.16	**2.40**	0
	140	373.86	**4918.01**	2
	335	2.45	**2247.97**	2
8cores-Majority RankW	195	0.14	2.06	0
	130	409.70	6444.05	4
	325	2.39	2944.92	4
8cores-Majority NormalizedW	195	0.14	2.15	0
	119	638.37	7218.16	9
	314	2.54	3298.60	9
8cores-Prob RankW	195	0.13	2.07	0
	138	299.74	5060.32	2
	333	2.55	2312.80	2
8cores-Prob NormalizedW	195	0.14	2.47	**0**
	132	397.59	5023.04	**1**
	327	2.47	2295.99	**1**



(a) 4cores-Prob, 4cores-Majority, 4cores-Agree

(b) 8cores-Prob, 8cores-Majority, 8cores-Agree

(c) 8cores-Majority NormalizedW, 8cores-Prob NormalizedW

(d) 8cores-Majority RankW, 8cores-Prob RankW

Fig. 4.5 Pairwise average Hamming distance (*x-axis*) vs. Number of flips every 10^6 steps (*y-axis*) to solve the *unif-k3-r4.2-v16000-c67200-S2082290699-014.cnf* instance

Figure 4.5(a) shows the *diversification-intensification* analysis using four cores. Among the cooperative strategies *4cores-Majority* exhibits a premature convergence due to its reduced diversification; while *4cores-Agree* shows a slow convergence due to its increased diversification. In contrast to these two methods, *4cores-Prob* is balancing *diversification*, and *intensification*. This phenomenon helps to explain the superiority shown by this method in Sect. 4.5.2.

A similar observation is drawn from the experiments with eight cores presented in Fig. 4.5(b). However, in this case *8cores-Agree* dramatically increases diversification, which actually degrades its overall performance when compared against its counterpart portfolio with four cores (see Table 4.2). Additionally, Fig. 4.5(c) shows the behavior of *8cores-Majority NormalizedW* and *8cores-Prob NormalizedW*, and Fig. 4.5(d) shows the behavior of *8cores-Majority RankW* and *8cores-Prob RankW*.

Table 4.3 *Diversification-Intensification* analysis using four cores over the whole set of benchmarks

Strategy	$\overline{HamIns}$
4cores-PAWS	38.2
4cores-No Sharing	**39.0**
4cores-Agree	35.0
4cores-Majority	31.7
4cores-Prob	33.1
4cores-Majority RankW	**25.9**
4cores-Majority NormalizedW	27.1
4cores-Prob RankW	30.8
4cores-Prob NormalizedW	32.8

Table 4.4 *Diversification-Intensification* analysis using eight cores over the whole set of benchmarks

Strategy	$\overline{HamIns}$
8cores-PAWS	38.3
8cores-No Sharing	**39.5**
8cores-Agree	38.3
8cores-Majority	30.8
8cores-Prob	33.4
8cores-Majority RankW	**29.3**
8cores-Majority NormalizedW	29.5
8cores-Prob RankW	33.1
8cores-Prob NormalizedW	33.8

From these two last figures, it can be observed that *Majority*-based strategies provide less diversification than the *Prob*-based ones.

Now we switch our attention to Tables 4.3 and 4.4, where we extend our analysis to all problem instances. To this end, we launched an extra run for each portfolio strategy to compute *HamIns*, an average over all values of *HamDis* during the run on each instance. *HamIns* is formally defined as follows:

$$HamIns(i) = \frac{\overline{HamDis}(i)}{total\text{-}vars(i)} \times 100$$

where $\overline{HamDis}(i)$ computes the mean over all *HamDis* values achieved when solving i, and $total\text{-}vars(i)$ indicates the number of variables involved in i. We use $\overline{HamIns}$ to denote the mean *HamIns* over all the instances that required at least 10^6 flips to be solved. Notice that instances requiring fewer flips do not employ cooperation because the first restart is not reached.

As can be observed, *prob*-based strategies have shown the best performance as they exhibit a better trade-off between *diversification* and *intensification* than *Agree*-(resp. *Majority*-) based strategies. For instance, excluding *4cores-agree*, which is known to provide more diversification than intensification, *4cores-Prob* shows the

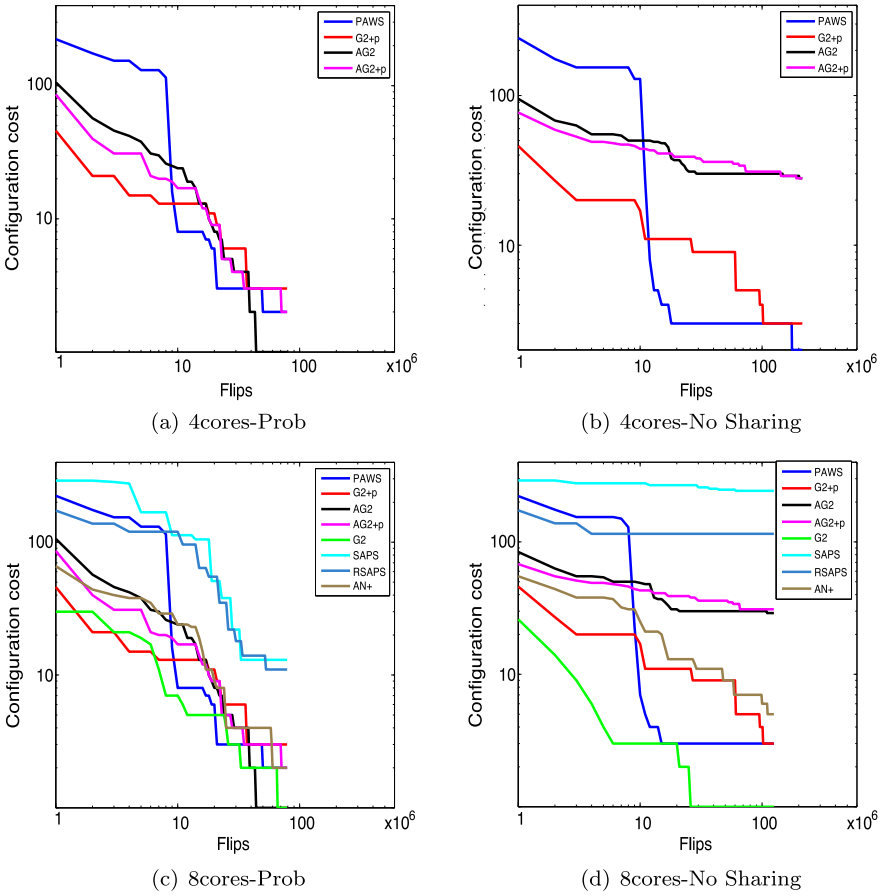


Fig. 4.6 Individual algorithm performance to solve the *unif-k3-r4.2-v16000-c67200-S2082290699-014.cnf* instance

highest $\overline{HamIns}$ variation among all cooperative portfolios using four cores. Moreover, *Majority*-based strategies are bad for diversification as they might tend to start with a configuration similar to the one given by the best single algorithm. It is also worth mentioning that our baseline portfolios *4cores-PAWS* and *4cores-No Sharing* exhibit the highest values, which is not surprising as no cooperation is allowed. Notice that algorithms in non-cooperative portfolios are independent from each other; for this reason each algorithm defines its own search trajectory.

On the other hand, a similar observation is seen in the case of eight cores. However, it is worth mentioning that *8cores-agree* gives too much diversification, degrading the overall performance when compared against its counterpart with four cores (see Tables 4.1 and 4.2).

Finally, Fig. 4.6 shows a trace of the best configuration cost found so far for each algorithm in the portfolio to solve a typical instance. The *x-axis* shows the

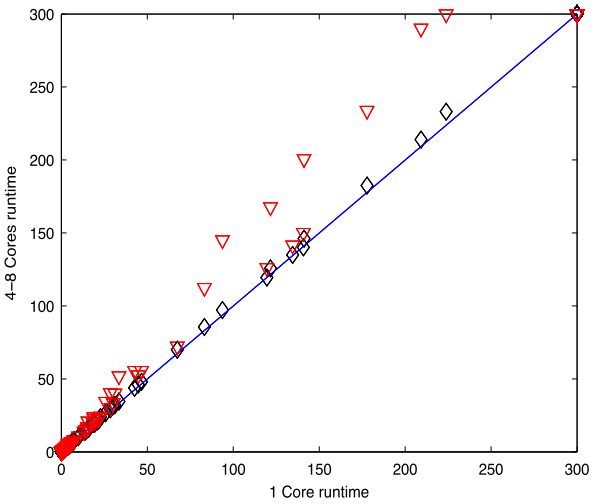


Fig. 4.7 Runtime comparison using parallel local search portfolios made of respectively one, four, and eight identical copies of PAWS (same random seed and no cooperation). *Black diamonds* indicate the performance of four cores vs. one core. *Red triangles* indicate the performance of eight cores vs. one core, *points above the blue line* indicate that one core is faster

best solution for each algorithm vs. the number of flips (*y-axis*). The right part of the figure shows the performance of individual searches using four and eight cores without cooperation, while the left part depicts the performance of *4cores-Prob* and *8cores-Prob*. As expected, non-cooperative algorithms exhibit different behaviors. For instance, Fig. 4.6(d) shows that SAPS and RSAPS are still far from the solution after reaching the time-out, while Fig. 4.6(c) shows that by using cooperation all algorithms (including SAPS and RSAPS) are pushed to promising areas of the search, i.e., assignments with few unsatisfied clauses.

4.5.5 Analysis of the Limitations of the Hardware

In this section, we wanted to assess the inherent slowdown caused by increased cache and bus contingency when more processing cores are used at the same time. Indeed, having an understanding of this slowdown help us to assess the real benefits of parallel search. To this end, we decided to run our PAWS baseline portfolio using the same random seed for each independent algorithm in the portfolio using one, four, and eight cores. Since all the algorithms are executing the same search, this experiment measures the slowdown caused by hardware limitations. The results are presented in Fig. 4.7.

The first case executes a single copy of PAWS with a time-out of 300 seconds, the second case executes four parallel copies of PAWS with a time-out of 1,200 seconds (4×300) and the third case executes eight parallel copies of PAWS with a time-out of 2,400 seconds (8×300).

Finally, we estimate the runtime of each instance as the median across 10 runs (each time with the same seed) divided by the number of cores. In this figure, it can be observed that the performance overhead is almost not distinguishable between one and four cores (black diamonds). However, the overhead between 1 and

8 cores increases for difficult instances (red triangles). As a final observation from this figure, we would like to point out that 111 points overlap at (300, 300).

This simple test can help us to assess the remarkable performance of our aggregation techniques. Indeed, on eight cores, the best technique is able to solve 86 more problems than the sequential search. This is achieved despite the slowdown caused by cache and bus contingencies revealed by this experiment.

4.6 Summary

In this work, our objective was to integrate knowledge sharing strategies in parallel local search for SAT. We were motivated by recent developments in parallel DPLL solvers. We decided to restrict the information shared to the best configuration found so far by the algorithms in a portfolio. From that we defined several simple knowledge aggregation strategies along a specific restart policy which creates a new initial assignment for the variables when a fixed cutoff is reached and when the quality of the shared information has been improved.

Extensive experiments were done on a large number of instances taken from the 2009 SAT competition. They showed that adding the proposed sharing policies improves the performance of a parallel portfolio. This improvement is exhibited in both,the number of solved instances and the *Penalized Average Runtime* (PAR). It is also reflected in the best configuration cost of problems which could not be solved within the time limit.

We believe that our work represents a very first step in the incorporation of knowledge sharing strategies in parallel local search for SAT. Further work will investigate the use of additional information to exchange, such as: tabu list, the age and score of a variable, information on local minima, etc. We also intend to investigate the best way to integrate this extra knowledge while solving a given problem instance. To this end, we plan to explore the taxonomies of cooperative search described in [CT10]. Moreover, as said earlier, state-of-the-art local searches for SAT perform better when they do not restart. Incorporating extra information without forcing the algorithm to restart is likely to be important.

Along those lines, we plan to equip the local search algorithms used in this chapter with clause learning, as described in [CI96] and [ALMS09], to exchange learnt clauses, borrowing ideas from portfolios for complete parallel SAT solvers.

A longer-term perspective regards the use of machine learning to identify the best subset of algorithms to solve a given instance.

Chapter 5
Learning Variable Dependencies

5.1 Introduction

The relationships between the variables of a combinatorial problem are key to its resolution. Among all the possible relations, explicit constraints are the most straightforward and are widely used. For instance, they are used to support classical look-ahead and look-back schemes. During look-ahead, they can restrict the maintenance of some level of consistency to some locality. During look-back, they can improve the backtracking by jumping to related and/or guilty decisions. These relationships are also used in dynamic variable ordering (DVO) to relate the current variable selection to past decisions (e.g., [Bre79]), or to give preference to the most constrained parts of the problem.

Recently, the notion of backdoor has been proposed. A backdoor can be informally defined as a subset of the variables such that, once assigned values, the remaining instance simplifies to a computationally tractable class. Backdoors can be explained by the presence of a particular relation between variables, e.g., functional dependencies. Unfortunately, detecting backdoors can be computationally expensive [DGS07], and their exploitation is often restricted to restart-based strategies as in modern SAT solvers [WGS03].

In this work, our objective is to heuristically discover a simplified form of functional dependencies between variables called *weak dependencies*. Once discovered, these relations are used to rank the importance of each variable. Our method assumes that these relations can be detected with low overhead during constraint propagation. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic [AH09].

5.2 Previous Work

In [BHLS04] the authors have proposed *dom-wdeg*, a heuristic which gives priority to variables frequently involved in failed constraints. It adds a weight to each con-

Y. Hamadi, *Combinatorial Search: From Algorithms to Systems*,
DOI 10.1007/978-3-642-41482-4_5, © Springer-Verlag Berlin Heidelberg 2013

straint which is updated (i.e, incremented by 1) each time the constraint fails. Using this value, variables are ranked according to domain size, and associated weight. X_i, the selected variable, minimizes dom-wdeg$(X_i) = |X_i| / \sum_{c \in prop(X_i)} weight(c)$. As shown in the previous section, domFD is superior to dom-wdeg on many problems. Interestingly, while dom-wdeg can only learn from conflicts, domFD can also learn from successful branchings. This is an important difference between these two techniques.

In [Ref04], Refalo proposes the *impact* dynamic variable-value selection heuristic. The rationale here is to maximize the reduction of the remaining search space. In this context an *impact* is computed taking into account the reduction of the search space due to an instantiated variable. Impact also considers values, and can therefore select the best instantiation instead of the best variable. With domFD, a variable is well ranked if its instantiation has generated several other instantiations. This is equivalent to an important pruning of the search space. In that respect domFD is close to impact. However, its principle is the dynamic exploitation of functional dependencies, not the explicit quantification of search space reductions. More generally, since DVO heuristics are all based on some understanding of the *fail-first* principle they are all aiming at an important reduction of the search space.

To improve SAT solving, [OGMS02] proposes a new pre-processing step that exploits the structural knowledge that is hidden in a CNF formula. It delivers a hybrid formula made of clauses together with a set of equations of the form $y = f(x_1, \ldots, x_n)$. This set of functional dependencies is then exploited to eliminate clauses and variables, while preserving satisfiability. This work detects real functions while our heuristic observes weak dependencies. Moreover, it uses a preprocessing step while we perform our learning during constraint propagation.

5.3 Technical Background

In this section, we briefly introduce definitions and notation used hereafter.

Definition 5.1 A Constraint Satisfaction Problem (CSP) is a triple (X, D, C) where:

- $X = \{X_1, X_2, \ldots, X_n\}$ represents a set of n variables.
- $D = \{D_1, D_2, \ldots, D_n\}$ represents the set of associated domains, i.e., possible values for the variables.
- $C = \{C_1, C_2, \ldots, C_m\}$ represents a finite set of constraints.

Each constraint C_i is associated to a set of variables $vars(C_i)$, and is used to restrict the combinations of values between these variables. Similarly, each variable X_i is related to a set of constraints $prop(X_i)$. The arity of a constraint C_i corresponds to $|vars(C_i)|$, and the degree of a variable X_i corresponds to $|prop(X_i)|$.

Fig. 5.1 Classic propagation engine

```
1: Q = {p1, p2, ...}
2: while Q ≠ {} do
3:     p = choose(Q);
4:     run(p);
5:     for all X_i ∈ vars(p) s.t. D_i was narrowed do
6:         schedule(Q, p, X_i);
7:     end for
8: end while
```

Solving a CSP involves the finding of a solution, i.e., an assignment of values to variables such that all the constraints are satisfied. If a solution exists, the problem is stated as satisfiable, and as unsatisfiable otherwise.

A depth-first search backtracking algorithm can be used to tackle CSPs. At each step a value is assigned to some variable. Each assignment is combined with a look-ahead process called constraint propagation which can reduce the domains of the remaining variables. Constraint propagation is usually based on some constraint network property which determines its locality and therefore its computational cost. Arc-consistency is widely used, and the result of its combination with backtrack search is called MAC, for Maintaining Arc-Consistency [SF94].

Figure 5.1 describes a classic constraint propagation engine [SC06]. In this algorithm, constraints are managed as propagators[1] in a propagation queue, Q. This structure represents the set of propagators that need to be revised. Revising a propagator corresponds to the enforcement of some consistency level on the domains of the associated variables.

Initially, Q is set to the entire set of constraints. This is used to enforce the arc-consistency property before the search process. During depth-first exploration, each decision is added to an empty queue, and propagated through this algorithm.

The function *choose* removes a propagator $p \in Q$, *run* applies the filtering algorithm associated to p, and *schedule* adds $prop(X_i)$ to Q. The algorithm terminates when the queue is empty. A fix-point is reached and more propagations can only appear as the result of a tree-based decision.

Definition 5.2 $f(X, y)$ is a functional dependency between the variables in the set X and the variable y if and only if for each combination of values in X there is precisely one value for y satisfying f.

Many constraints of arity k can be seen as functional dependencies between a set of $k-1$ variables and some remaining variable y. For instance, the arithmetic constraint $X + Y = Z$ gives the dependencies $f(\{X, Y\}, Z)$, $f(\{X, Z\}, Y)$, and $f(\{Y, Z\}, X)$. There are also many exceptions like the constraint $X \neq Y$, where in the general case, one variable is not functionally dependent of the other one.

[1] In the following, we will use this as a synonym for constraints.

5.4 Exploiting Weak Dependencies in Tree-Based Search

5.4.1 Weak Dependencies

Our objective is to take advantage of functional dependencies during search. We propose to heuristically discover a weaker form of relation called *weak dependency* between pairs of variables. A weak dependency is observed when a variable gets instantiated as the result of another instantiation. Our new DVO heuristic records these weak dependencies and exploits them to prioritize the variables during the search process.

Definition 5.3 During constraint propagation with the algorithm presented in Fig. 5.1, we call (X, Y) a weak dependency if the two following conditions hold:

1. Y is instantiated as the result of the execution of a propagator p.
2. p is inserted in Q as the result of the instantiation of X.

Property 5.1 *Weak dependency relations* (X, Y) *can be recorded as the result of the execution of a propagator* p iff $X \in vars(p)$ *and* $Y \in vars(p)$.

The proof is straightforward if we consider the algorithm presented in Fig. 5.1.

5.4.2 Example

To illustrate our definition, we consider the following set of constraints:

- $p_1 \equiv X_1 + X_2 < X_3$
- $p_2 \equiv X_1 \neq X_4$
- $p_3 \equiv X_4 \neq X_5$

With the domains, $D_1 = D_2 = D_4 = D_5 = \{0, 1\}$ and $D_3 = \{1, 2\}$.

The initial filtering does not remove any value and the search process has to be started. Assuming that the search is started on X_1 with value 1, the propagator $X_1 = 1$ is added to Q, and after its execution the domain D_1 has narrowed, it is necessary to schedule p_1 and p_2.

Running p_1 sets X_2 to 0, and X_3 to 2, and gives the weak dependencies (X_1, X_2) and (X_1, X_3). Afterwards, p_2 sets X_4 to 0, which corresponds to (X_1, X_4). Finally, the narrowing of D_4 schedules p_3, which sets X_5 to 1, and gives the weak dependency (X_4, X_5).

Weak dependencies are binary; therefore they only roughly approximate functional dependencies. For example, with the constraint $X + Y = Z$ they will never record $(\{X, Y\}, Z)$. On the other hand weak dependencies exploit the current domains of the variables and can record relations which are not true in general but hold in particular cases. For instance, the propagator p_3 above creates (X_4, X_5). This represents a real functional dependency since the domains of the variables are binary and equal.

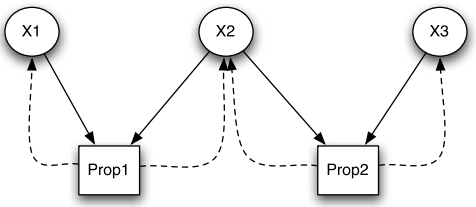


Fig. 5.2 Variables and propagators

```
1: enqueue(Q, prop(X_i));
2: if |D_i| = 1 then
3:     dependencies(p.assigned, X_i);
4:     for all p' in prop(X_i) do
5:         p'.assigned.add(X_i);
6:     end for
7: end if
```

Fig. 5.3 Schedule(Queue Q, Propagator p, Variable X_i)

5.4.3 Computing Weak Dependencies

We can represent weak dependencies as a weighted digraph relation among the variables of the problem, where the nodes of the graph are the variables and the edges indicate weak dependency relations between two variables, i.e, when there is an edge between two variables X and Y, the direction of the edge shows the relation and its weight indicates the number of observed occurrences of that relation.

In a propagation-centered approach [LS07] each variable has a list of dependent propagators and each propagator knows its variables (see Fig. 5.2).

In this way, once the domain of a variable is narrowed it is necessary to schedule its associated propagators in the propagator pool. Since we are interested in capturing weak dependencies, we have to track the reasons for constraint propagation. More specifically, when a propagator gets activated as the result of the direct assignment of some variable, we need to keep a reference to that variable. Since the assignment of several variables can activate a propagator, we might have to keep several references.

A modified *schedule* procedure is shown in Fig. 5.3. The algorithm starts by enqueueing all the propagators associated to a given variable X_i in the propagators pool. If the propagator p was called as the result of the assignment of X_i ($|D_i| = 1$), a weak dependency is created between each variable of the set *p.assigned* and X_i. Variables from this set are the ones whose assignment was the reason for propagating p. After that, a reference to X_i is added to its propagators $prop(X_i)$. This is done to ensure that if these propagators assign other variables, a subsequent call to the schedule procedure will be able to create dependencies between X_i and these variables.

5.4.4 The domFD Dynamic Variable Ordering

In the previous section, we have seen that a generic constraint propagation algorithm can be modified to compute weak dependencies. As we pointed out above, weak

dependencies can be seen as a weighted digraph relation among the variables. Using this graph, we propose to define a function $FD(X_i)$ which computes the out-degree weight of a variable X_i taking into account only uninstantiated variables.

$$FD(X_i) = \sum_{X_j \in \Gamma^+(X_i)} weight(X_i, X_j) \tag{5.1}$$

where $\Gamma^+(x)$ (resp. $\Gamma^-(x)$) represents the set of outgoing (resp. ingoing) edges from (resp. to) x in the graph of dependencies. It is also important to note that when there is no outgoing edge associated to X_i we assume $FD(X_i) = 1$.

Given the definition of *FD*, we define *domFD*, a new DVO heuristic based on both, the observed weak dependencies of the problem and the well-known fail-first *mindom* heuristic:

$$domFD(X_i) = \frac{|X_i|}{FD(X_i)} \tag{5.2}$$

Then, the heuristic selects the variable whose *domFD* value is minimal.

5.4.5 Complexities of domFD

Space We know from Property 5.1 that dependencies are created between variables which share a constraint. Therefore, computing the weak dependency graph requires in the worst case a space proportional to the space used for the representation of the problem. Assuming n variables and m constraints, the space is proportional to $n + m$.

Time The computation of weak dependencies is tightly linked to constraint propagation. The original schedule procedure only enqueues the propagators related to X_i in Q; therefore its original cost is $O(m)$. Our new procedure creates dependencies each time a variable gets instantiated. Dependencies between variables can be recorded as the result of the instantiation of one or several variables. In the latter case, up to $n - 1$ dependencies can be created since the instantiation of up to $n - 1$ variables can be responsible for the scheduling of the current propagator (line 3 in the algorithm of Fig. 5.3). Once dependencies are created, the propagators associated to X_i need to reference it. Here the cost is bounded by m. Overall, the time complexity of the new schedule procedure is $O(n + m)$.

We now have to consider the cost of maintaining the weak dependency graph. Since our heuristic only considers the weights related to the variables which are not instantiated we have to disconnect variables from the graph when they get a value, and we have to reconnect them when the search backtracks. This can be done incrementally.

Practically, we do not have to physically remove a variable from the dependency graph; we can just offset the weight of the recorded dependencies between other

variables and that variable. For instance, when X_i gets instantiated as the result of a tree decision or as the result of constraint propagation, we only need to update the out degrees of variables $X_j \in \Gamma^-(X_i)$. The update is done by decreasing their associated counter $X_j.FD$ by $weight(X_j, X_i)$. These counters represent the number of times the weak dependency (X_j, X_i) was observed during the search process. During backtracking, X_i gets back its domain, and we just have to "reconnect" the associated $X_j \in \Gamma^-(X_i)$ by adding $weight(X_j, X_i)$ to $X_j.FD$. Since a variable can be linked to m propagators, an update of the dependency graph costs $O(m)$. In the worst case, each branching holds no propagation and therefore at each node, the cost of updating the dependency graph is $O(m)$.

Finally, selecting the variable which minimizes *domFD* can cost an iteration over n variables if no special data structure is used.

Now if we consider all the operations, constraint propagation with the new schedule procedure, disconnecting a single variable, and selection of the variable which minimizes *domFD*, we have $O(n+m)$ instead of $O(m)$ initially.

5.5 Experiments

In this section, we propose to study the performance of *domFD* when compared to *dom-wdeg*, a recently introduced heuristic able to focus on the difficult parts of a problem [BHLS04].

In dom-wdeg, the priority is given to variables which are frequently involved in failed constraints. A weight is added to each constraint and updated (i.e, incremented by 1) each time a constraint fails. Using this value variables are selected based on their domain size and their total associated weight. X_i, the selected variable, minimizes $\text{dom-wdeg}(X_i) = |X_i| / \sum_{c \in prop(X_i)} weight(c)$.

This heuristic is used in the Abscon solver, which appeared to be the most robust in a recent CSP competition,[2] where it finished one time first, three times second, 3 times third, and three times fourth, when compared against 15 other solvers.

To compare domFD against the powerful dom-wdeg, we implemented them in Gecode-2.0.1 [Gec06] and used them to tackle several problems. Since Gecode is now widely used, we decided to take from the Internet problems already encoded for the Gecode library. We paid attention to the fact that overall our problems cover a large set of Gecode's constraints.

We used 35 instances coming from nine different benchmark families. They involve satisfaction, counting, and optimization problems. They were solved using the default Gecode's branch-and-prune strategy, and a modified restart technique based on the default strategy. In the tests, the value selection ordering was Gecode's `INT_VAL_MIN`, which returns the minimal value of a domain. All the experiments were performed on a MacBook Pro 2.4 GHz Intel Core 2 Duo, under Ubuntu Linux 7.10 and gcc version 4.0.1. A time-out (TO) of 10 minutes was used for each experiment.

[2] http://www.cril.univ-artois.fr/CPAI06/round2/results/ranking.php?idev=6.

5.5.1 The Problems

In the following, we list the different benchmark families. When they are described on www.csplib.org, we only present the number in the library. Note that for all problems (except Quasigroup) the model and its implementation are the ones proposed in the Gecode examples.[3]

- Quasigroup, *qwh*, problem 3 of CSPLib.
- Golomb ruler, *gol-rul*, problem 6 of CSPLib.
- All-interval, *all-int*, problem 7 of CSPLib.
- Nonogram, *nono*, problem 12 of CSPLib.
- Magic-square, *magic-squ*, problem 19 of CSPLib.
- Langford number, *lfn*, problem 24 of CSPLib.
- Sports league tournament, *sport-lea*, problem 26 of CSPLib.
- Balanced Incomplete Block Design, *bibd*, problem 28 of CSPLib.
- Crowded-chess, *crow-ch*; this problem consists in arranging n queens, n rooks, $2n-1$ bishops and k knights on an $n \times n$ chessboard, so that queens cannot attack each other, no rook can attack another rook and no bishop can attack another bishop. Note that two queens (in general two pieces of the same type) are attacking each other even if there is a bishop (in general another piece of different type) between them.

When an instance is solved, the number of nodes in the tree(s), the number of fails and the time in seconds are reported. If the 10 minutes time-out is reached, TO is reported.

5.5.2 Searching for All Solutions or for an Optimal Solution

The first part of Table 5.1 presents results related to the finding of all the solutions of all-interval problems of order 11 to 14. We can observe that the trees generated with domFD are usually far smaller than the ones generated by dom-wdeg. Most of the time, domFD runtime is also better. However, the time per node is more important for our heuristic. For instance, on all-int-14, dom-wdeg does 89,973 nodes/s while domFD runs at 54,122 nodes/s.

The second part of the table presents results for the optimal Golomb rulers of orders 10 to 12. Here, we can observe that order 10 is easier for dom-wdeg, but tree sizes are comparable. Orders 11, and 12 give advantage to domFD, with far smaller search trees and better runtimes. As before, the time per node is more important for our heuristic (31,771 vs 35,852 on gol-rul-11).

[3] Available from http://www.gecode.org/gecode-doc-latest/group__ExProblem.html.

Table 5.1 All solutions and optimal solution

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	Time (s)	#nodes	#failures	Time (s)
all-int-11	100844	50261	0.93	52846	26262	**0.81**
all-int-12	552668	276003	6.92	211958	105648	**3.45**
all-int-13	2.34M	1.17M	**26.13**	1.64M	821419	29.74
all-int-14	15.73M	7.86M	**174.83**	11.27M	5.63M	208.23
gol-rul-10	93732	46866	**1.97**	102910	51449	2.70
gol-rul-11	2.77M	1.38M	77.26	1.77M	889633	**55.71**
gol-rul-12	12.45M	6.22M	404.92	6.97M	3.48M	**266.28**

Table 5.2 First solution, branch-and-prune strategy

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	Time (s)	#nodes	#failures	Time (s)
qwh-30-316-1	1215	603	**0.22**	234	115	0.32
qwh-30-316-2	48141	24063	8.09	10454	5220	**3.62**
qwh-30-316-3	6704	3347	**1.11**	2880	1437	1.15
bibd-7-3-2	100	39	**0.01**	65	28	**0.01**
bibd-7-3-3	383	180	0.03	96	42	**0.01**
bibd-7-3-4	–	–	TO	132	56	**0.03**
lfn-3-9	168638	84316	6.16	7527	3760	**0.26**
lfn-2-19	–	–	TO	1.64M	822500	**43.05**
lfn-3-10	2.21M	1.10M	87.15	12440	6218	**0.46**
nono-5	1785	879	0.12	491	239	**0.11**
nono-8	17979	8983	3.54	1084	537	**0.54**
nono-9	248	115	**0.04**	120	58	0.12

5.5.3 Searching for a Solution with a Classical Branch-and-Prune Strategy

Experiments related to the finding of a first solution are presented in Table 5.2. They show results for, respectively, quasigroups, balance incomplete block design, Langford numbers, and nonograms.

Quasigroups Three instances of order 30 with 316 unassigned positions were produced with the generator presented in [AGKS00]. On these instances, domFD always generates smaller search trees. When this difference is large enough e.g., second instance, the runtime is also better.

Balance Incomplete Block Design Our heuristic always explores smaller trees, which allows better runtimes. Interestingly the third instance is solved in 0.03 seconds by domFD while dom-wdeg cannot solve it in 10 minutes.

Langford Numbers On these problems, domFD is always superior to dom-wdeg. For instance, lfn-3-10 can be solved by both heuristics but the performance of domFD is far better: 190 times faster.

Nonograms Table 5.2 shows results for the nonogram problem. Three instances of orders 5, 8, and 9 were generated. Here again, the trees are systematically smaller with domFD and when the difference is large enough runtimes are always better.

5.5.4 Searching for a Solution with a Restart-Based Branch-and-Prune Strategy

Restart-based searches are very efficient since they can alleviate the effects of early bad decisions. Therefore, it is important to test our new heuristic with a restart strategy.

A restart is done when some cutoff limit in the number of fails is met, i.e., at some node in a tree. There, the actual domFD graph is stored and used to start the next tree-based search. This allows the early selection of well-ranked variables. The same technique is used with dom-wdeg, and the next search tree can branch early on well-ranked variables.

This part presents results with a restart-based branch-and-prune strategy where the cutoff value used to restart the search was initially set to 1,000, and the cutoff increase policy to $\times 1.2$. The same 10 minutes timeout was used.

Table 5.3, presents the results for magic square, crowded chess, sports league tournament, quasigroup, and bibd problems.

Magic Square Instances of orders 5 to 11 were solved. Clearly, domFD is the only heuristic able to solve large orders within the time limit. For example, dom-wdeg cannot deal with orders greater than 8, while our technique can. The reduction in the search tree sizes is very significant, e.g., on mag-squ-8, dom-wdeg develops 35.18M nodes and domFD 152,466, which allows it to be more than 100 times faster.

Crowded Chess As before, domFD can tackle large problems while dom-wdeg cannot.

Sports League Tournament If we exclude the last instance, domFD is always better than dom-wdeg.

Quasigroups Here, on most problems, domFD generates smaller search trees, and can return a solution more quickly. On the hardest problem (order 35), domFD is nearly two time faster.

Table 5.3 First solution, restart-based strategy

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	Time (s)	#nodes	#failures	Time (s)
mag-squ-5	2239	1113	**0.02**	3025	1505	0.06
mag-squ-6	33238	16564	0.32	4924	2440	**0.08**
mag-squ-7	9963	4868	**0.20**	33422	16614	0.86
mag-squ-8	35.18M	17.59M	460.40	152446	75987	**4.51**
mag-squ-9	–	–	TO	66387	32951	**1.64**
mag-squ-10	–	–	TO	83737	41607	**2.17**
mag-squ-11	–	–	TO	8.52M	4.26M	**374.62**
crow-ch-7	2029	1002	**0.04**	3340	1656	0.22
crow-ch-8	16147	8036	0.67	2041	1002	**0.14**
crow-ch-9	129827	64788	**6.15**	228480	114089	37.97
crow-ch-10	–	–	TO	1134052	566761	**263.01**
sports-lea-14	4746	2327	0.68	4814	2359	**0.65**
sports-lea-16	28508	14073	4.05	3913	1912	**0.61**
sports-lea-18	546475	272510	101.70	51680	25549	**10.72**
sports-lea-20	182074	90355	**36.69**	2.07M	1.03M	514.18
qwh-30-316-1	1215	603	**0.22**	234	115	0.32
qwh-30-316-2	118348	59104	20.06	8828	4397	**2.7**
qwh-30-316-3	8944	4451	1.68	3114	1552	**1.01**
qwh-35-405	2.38M	1.19M	562.62	475053	237369	**236.05**
bibd-7-3-2	100	39	**0.01**	65	28	**0.01**
bibd-7-3-3	383	180	0.03	96	42	**0.01**
bibd-7-3-4	6486	3210	0.79	132	56	**0.03**

Balanced Incomplete Block Design Here domFD performs very well, with both smaller search trees and small runtime.

5.5.5 Synthesis

Table 5.4 summarizes the performance of the heuristics. These results were generated by only taking into account the problems which can be solved by both domFD and dom-wdeg i.e., we removed six instances which cannot be solved by dom-wdeg.

We can observe that the search trees generated by domFD are on the average three times smaller. The difference in the number of fails is similar. Finally, even if domFD is two times slower on the time per node, it is 31 % faster overall.

Table 5.4 Synthesis of the experiments

Heuristic	*average*			
	#nodes	#failures	Time (s)	Nodes/s
dom-wdeg	2.14M	1.07M	56.99	37664
domFD	717202	358419	39.53	18139

Technically, our integration into Gecode is quite straightforward and not particularly optimized. For instance we use *Leda*,[4] an external library to maintain the graph, while a bespoke light class with the right set of features should be used. The way we record weak dependencies is also not optimized and requires extra data structures whose accesses could be easily improved, e.g., the *assigned* list of variables shown in the algorithm of Fig. 5.3. For all this, we think that it must be possible to increase the speed of our heuristic by some factor.

We also did some experiments to see if the computation of domFD could be cheaply approximated. We used a counter with each variable to record the number of times that variable was at the origin of a weak dependency. This represents an approximation of domFD since the counter considers dependencies on instantiated variables. Unfortunately, this fast approximation is always beaten by domFD on large instances.

5.6 Summary

In this work, our goal was to heuristically discover a simplified form of functional dependencies between variables called *weak dependencies*. Once discovered, these relations are used to rank the branching variables. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic.

Experiments done on 35 problems coming from nine benchmark families showed that on the average domFD reduces search trees by a factor of 3 and runtime by 31 % when compared against dom-wdeg, one of the best dynamic variable ordering heuristics. domFD is also more expensive to compute since it puts some overhead on the propagation engine. However, it seems that our implementation can be improved, for example, by using incremental data structures to record potential dependencies in the propagation engine.

Our heuristic learns from successes, allowing a quick exploitation of the solver's work. In a way, this is complementary to dom-wdeg which learns from failures. Moreover, both techniques rely on the computation of *mindom*. Combining their respective strengths seems obvious but is not straightforward.

[4] www.algorithmic-solutions.com.

Chapter 6
Continuous Search

6.1 Introduction

In the previous chapters, we have seen that portfolios of algorithms can positively impact the robustness of search. In Chap. 2, our portfolio was using multiple variable ordering heuristics whose executions were interleaved at the agent level. In Chaps. 3 and 4, we moved to fully fledged parallelism with portfolios of parallel CDCL and local search solvers competing and cooperating to tackle a given SAT instance. Finally, in Chap. 5 we have seen that we can incrementally learn an ordering of the variables based on their observed interactions.

The present chapter shows how to improve performance by considering a set of instances. It considers a situation where a given Constraint Programming engine is used to successively solve problems coming from a given application domain. The objective is to incrementally learn a predictive model able to accurately match instance features to good solver's parameters. The learning is possible thanks to the relative coherence of the instances, and the goal is to eventually achieve top performance for the underlying application domain.

In Constraint Programming, properly crafting a constraint model which captures all the constraints of a particular problem is often not enough to ensure acceptable runtime performance. Additional tricks, e.g. adding redundant and channeling constraints, or using some global constraint (depending on your constraint solver) which can efficiently do part of the job, are required to achieve efficiency. Such tricks are far from being obvious, unfortunately; they do not change the solution space, and users with a classical mathematical background might find it hard to see why adding redundancy helps.

For this reason, users are often left with the tedious task of tuning the search parameters of their constraint solver, and this, again, is both time consuming and not necessarily straightforward. Parameter tuning indeed appears to be conceptually simple ((i) try different parameter settings on representative problem instances, (ii) pick up the setting yielding best average performance). Still, most users would easily consider instances which are not representative of their problem, and get misled.

Y. Hamadi, *Combinatorial Search: From Algorithms to Systems*,
DOI 10.1007/978-3-642-41482-4_6, © Springer-Verlag Berlin Heidelberg 2013

The goal of the work presented in this chapter is to allow any user to eventually get their constraint solver achieving a top performance on their problems. The proposed approach is based on the original concept of Continuous Search (CS), gradually building a heuristics model tailored to the user's problems, and mapping a problem instance onto some appropriate parameter setting. A main contribution compared to the state-of-the art (see [SM08] for a recent survey; more in Sect. 6.4) is relaxing the requirement of a large set of representative problem instances having to be available beforehand to support offline training. The heuristics model is initially empty (set to the initial default parameter setting of the constraint solver) and it is enriched according to a lifelong learning approach, exploiting the problem instances submitted by the user to the constraint solver.

Formally, CS interleaves two functioning modes. In production or exploitation mode, the instance submitted by the user is processed by the constraint solver; the current heuristics model is used to parameterize the constraint solver depending on the instance at hand. In learning or exploration mode, CS reuses the last submitted instance, running other heuristics than the one used in production mode in order to find which heuristics would have been most efficient for this instance. CS thus gains some expertise relative to this particular instance, which is used to refine the general heuristics model through machine learning (Sect. 6.3.2). During the exploration mode, new information is thus generated and exploited in order to refine the heuristics model, in a transparent manner: without requiring the user's input and by only using the idle computer's CPU cycles.

We claim that the CS methodology is realistic (most computational systems are always on, especially production ones, and most systems waste a large amount of CPU cycles) and compliant with real-world settings, where the solver is critically embedded within large and complex applications. The CS computational cost must be balanced against the huge computational cost of offline training [XHHLB07, HHHLB06, HH05]. Finally, lifelong learning appears to be a good way to construct an efficient and agnostic heuristics model and to be able to adapt to new modeling styles or new classes of problem [AHS10, AHS09].

6.2 Related Work

This section briefly reviews and discusses some related works devoted to heuristic selection within CP and SAT solvers.

SATzilla [XHHLB07] is a well-known SAT portfolio solver which is built upon a set of features. Roughly speaking SATzilla includes two kinds of basic features: general features such as number of variables, number of propagators, etc. and local search features which actually probe the search space in order to estimate the difficulty of each problem-instance for a given algorithm. The goal of SATzilla is to learn a runtime prediction function by using a linear regression model. Along the same lines, Haim et al. in [HW09] build the portfolio taking into account several restart policies for a set of well-known SAT solvers.

CPHydra [OHH+08] is a portfolio approach based on case-based reasoning; it maintains a database with all solved instances (so-called *cases*). Later on, once a new instance I arrives, a set of similar cases C is computed, and based on C it builds a switching policy selecting a set of CSP solvers that maximizes the possibilities of solving I within a given amount of time.

The approach most similar to the presented one is that of [SM07], who likewise apply machine learning techniques to perform online combination of heuristics into search tree procedures. Unfortunately, this work requires an important number of training instances to build a model with a good generalization property.

In [CB05] low knowledge is used to select the best algorithm in the context of optimization problems; this work assumes a black-box optimization scenario where the user has no information about the problem or even about the domain of the problem, and the only known information is the output (i.e., solution cost for each algorithm in the portfolio). Unfortunately this mechanism is only applicable to optimization problems and cannot be used to solve CSPs.

The purpose of *The Adaptive Constraint Engine* (ACE) [EFW+02] is to unify the decisions of several heuristics in order to guide the search process. In this way, each heuristic votes for a possible variable/value decision to solve a CSP. Afterwards, a global controller selects the most appropriate variable/value pair according to previously (offline) learnt weights associated to each heuristic. The authors however did not present any experimental scenario taking into account any restart strategy, although these nowadays are an essential part of constraint solvers.

Combining Multiple Heuristics Online [SGS07] and Portfolios with Deadlines [WvB08] are designed to build a scheduler policy in order to switch the execution of *black-box* solvers during the resolution process. However, in these papers the switching mechanism is learnt/defined beforehand, while our approach relies on the use of machine learning to switch the execution of heuristics on the fly.

Finally, in [AST09] and [HHLBS09] the authors studied the automatic configuration problem whose objective is to find the best parameters of a given algorithm in order to efficiently solve a class of problems.

6.3 Technical Background

6.3.1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (*CSP*) is a triple (X, D, C) where X represents a set of variables, D a set of associated domains (i.e., possible values for the variables) and C a finite set of constraints.

Solving a CSP involves finding a solution, i.e., an assignment of values to variables such that all constraints are satisfied. If a solution exists the problem is stated as satisfiable, and as unsatisfiable otherwise. A depth-first search backtracking algorithm can be used to tackle CSPs. At each step of the search, an unassigned variable

X and a valid value v for X are selected; the exploration of variables/values is combined with a look-ahead strategy able to narrow the domains of the variables and reduce the remaining search space through constraint propagation. Restarting the search engine [GSK98, KHR+02] helps to reduce the effects of early mistakes in the search process. A restart is done when some cutoff limit in the number of failures (backtracks) is met (i.e., at some point in the search tree); before restarting the search each heuristic stores its ranking metrics in order to start the next tree-based search.

In this work, we consider five well-known variable selection heuristics. *min-dom* [HE79] selects the variable with the smallest domain, *wdeg* [BHLS04] selects the variable which is involved in the highest number of failed constraints, *dom-deg* selects the variable which minimizes the ratio $\frac{dom}{deg}$, *dom-wdeg* [BHLS04] selects the variable which minimizes the ratio $\frac{dom}{wdeg}$ and *impacts* [Ref04] selects the (variable, value) pair which maximizes the reduction of the remaining search space. While only deterministic heuristics will be considered, the proposed approach can be extended to randomized algorithms by following the approach proposed in [HHHLB06].

6.3.2 Supervised Machine Learning

Supervised machine learning exploits data labeled by the expert to automatically build hypotheses emulating the expert's decisions [Vap95]. Only the binary classification case will be considered in the following. Formally, a learning algorithm processes a training set $\mathcal{E} = \{(x_i, y_i), x_i \in \Omega, y_i \in \{1, -1\}, i = 1 \ldots n\}$ made of n examples (x_i, y_i), where x_i is the example description (e.g. a vector of values, $\Omega = \mathbb{R}^d$) and y_i is the associated label; example (x, y) is referred to as positive (respectively, negative) iff y is 1 (resp., -1). The learning algorithm outputs a hypothesis $f : \Omega \mapsto Y$ associating to each example description x a label $y = f(x)$ in $\{1, -1\}$. Among ML applications are pattern recognition, ranging from computer vision to fraud detection [LB08], game playing [GS07], or autonomic computing [RBea05].

Among the prominent ML algorithms are *Support Vector Machines* (SVMs) [CST00]. Linear SVM considers real-valued positive and negative instances ($\Omega = \mathbb{R}^d$) and constructs the separating hyperplane which maximizes the margin, i.e., the minimal distance between the examples and the separating hyperplane. The margin maximization principle provides good guarantees about the stability of the solution and its convergence towards the optimal solution when the number of examples increases.

The linear SVM hypothesis $f(x)$ can be described from the sum of the scalar products of the current instance x and some of the training instances x_i, called support vectors:

$$f(x) = \langle w, x \rangle + b = \sum \alpha_i \langle x_i, x \rangle + b$$

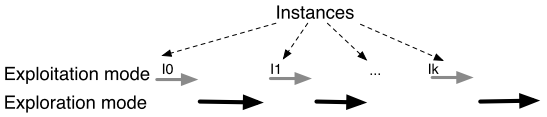


Fig. 6.1 Continuous search scenario

The SVM approach can be extended to non-linear spaces, by mapping the instance space Ω into a more expressive feature space $\Phi(\Omega)$. This mapping is made implicit through the so-called *kernel trick*, by defining $K(x, x') = \langle \Phi(x), \Phi(x') \rangle$; it preserves all good SVM properties provided the kernel is positive definite. Among the most widely used kernels are the Gaussian kernel ($K(x, x') = \exp\{-\frac{\|x-x'\|^2}{\sigma^2}\}$) and the polynomial kernel ($K(x, x') = (\langle x, x' \rangle + c)^d$). More complex separating hypotheses can be built on such kernels,

$$f(x) = \sum \alpha_i K(x_i, x) + b$$

using the same learning algorithm core as in the linear case. In all cases, a new instance x is classified as positive (respectively negative) if $f(x)$ is positive (resp. negative).

6.4 Continuous Search in Constraint Programming

The Continuous Search paradigm, illustrated in Fig. 6.1, considers a functioning system governed from a heuristics model (which could be expressed, e.g., as a set of rules, a knowledge base, or a neural net). The goal of continuous search is to exploit the problem instances submitted to the system in a two-step process:

1. Exploitation mode: unseen problem instances are solved using the current heuristics model.
2. Exploration mode:
   (a) these instances are solved with other heuristics, yielding new information. This information associates to the description x of the example (accounting for the problem instance and the heuristics) a Boolean label y (the heuristics improves/does not improve on the current heuristics model);
   (b) the training set $\mathcal{E}$, augmented with these new examples (x, y), is used to revise or relearn the heuristics model.

The Exploitation or production mode (step 1) aims at solving new problem instances as quickly as possible. The Exploration or learning mode (steps 2 and 3) aims at learning a more accurate heuristics model.

Definition 6.1 A continuous search system is endowed with a heuristics model, which is used as is to solve the current problem instance in production mode, and which is improved using the previously seen instances in learning mode.

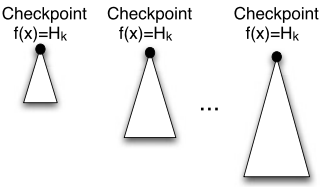


Fig. 6.2 *dyn-CS*: selecting the best heuristic at each restart point

Initially, the heuristics model of a continuous search system is empty, that is, it is set to the default settings of the search system. In the proposed CS-based constraint programming, the default setting is a given heuristic noted *DEF* in the following (Sect. 6.5). Assumedly, *DEF* is a reasonably good strategy on average; the challenge is to improve on *DEF* for the particular types of instances which have been encountered in production mode.

6.5 Dynamic Continuous Search

The Continuous Search paradigm is applied to a restart-based constraint solver, defining the *dyn-CS* algorithm. After a general overview of *dyn-CS*, this section details the different modules thereof.

Figure 6.2 depicts the general scheme of *dyn-CS*. The constraint-based solver involves several restarts of the search. A restart is launched after the number of backtracks in the search tree reaches a user-specified threshold. The search stops after a given time limit. Before starting the tree-based search and after each subsequent restart, the description x of the problem instance is computed (Sect. 6.5.1). We will call checkpoints the calculations of these descriptions.

In production mode, the heuristics model f is used to compute the heuristic $f(x)$ to be applied for the entire checkpoint window, i.e., until the next restart. Not to be confused with the *choice point* which selects a variable/value pair at each node in the search tree, *dyn-CS* selects the most promising heuristic at a given checkpoint and uses it for the whole checkpoint window. In learning mode, other combinations of heuristics are applied (Sect. 6.5.4) and the eventual result (depending on whether the other heuristics improved on heuristics $f(x)$) leads to building training examples (Sect. 6.5.3). The augmented training set is used to relearn the heuristics model $f(x)$.

6.5.1 Representing Instances: Feature Definition

At each checkpoint (or restart), the description of the problem instance is computed, including static and dynamic features.

While a few of these descriptors had already been used in SAT portfolio solvers [HHHLB06, XHHLB07], many descriptors had to be added as CSPs are more diverse than SAT instances: SAT instances only involve Boolean variables and clauses,

contrasting with CSPs using variables with large domains, and a variety of constraints and pruning rules.

Static Features Static features encode the general description of a given problem instance; they are computed once for each instance as they are not modified along the resolution process. The static features also allow one to discriminate between types of problems, and different instances.

- *Problem definition* (four features): Number of variables, constraints, variables assigned/not assigned at the beginning of the search.
- *Variable size information* (six features): `Size prod`, `sum`, `min`, `max`, `mean` and `variance` of the variable domain size.
- *Variable degree information* (eight features): `min`, `max`, `mean` and `variance` of the variable degree (resp. variable domain/degree).
- *Constraint information* (six features): The degree (or arity) of a given constraint c is represented by the total number of variables involved in c. Likewise the size of c is represented by the product of its corresponding variable domain sizes. Taking into account this information, the following features are computed `min`, `max`, `mean` of constraint sizes and degrees.
- *Filtering cost category* (eight features): Each constraint c is associated a category.[1] In this way, we compute the number of constraints for each category. Intuitively each category represents the implementation cost of the filtering algorithm. $Cat = \{Exponential, Cubic, Quadratic, Linear\ expensive, Linear\ cheap, Ternary, Binary, Unary\}$, where *Linear expensive* (resp. *cheap*) indicates the complexity of a linear equation constraint and the last three categories indicate the number of variables involved in the constraint. More information about the filtering cost category can be found in [Gec06].

Dynamic Features Two kinds of dynamic features are used to monitor the performance of the search effort at a given checkpoint: global statistics describe the progress of the overall search process; local statistics check the evolution of a given strategy.

- *Heuristic criteria* (15 features): each heuristic criterion (e.g., *wdeg*, *dom-wdeg*, *impacts*) is computed for each variable; their `prod`, `min`, `max`, `mean` and `variance` over all variables are used as features.
- *Constraint weight* (12 features): likewise report the `min`, `max`, `mean` and `variance` of all constraint weights (i.e., constraint *wdeg*s). Additionally the `mean` for each filtering cost category is used as a feature.
- *Constraint information* (three features): `min`, `max` and `mean` of constraint *run-prop*, where *run-prop* indicates the number of times the propagation engine has called the filtering algorithm of a given constraint.

[1] Out of eight categories, detailed in http://www.gecode.org/doc-latest/reference/classGecode_1_1PropCost.html.

- *Checkpoint information* (33 features): for every checkpoint$_i$ relevant information from the previous checkpoint$_{i-1}$ (when available) is included into the feature vector. From checkpoint$_{i-1}$ we include the total number of nodes and maximum search depth. From the latest non-failed node, we consider the total number of assigned variables, the satisfied constraints, the sum of variables *wdeg* (resp. sizes and degree) and the product of variable degrees (resp. *domain*, *wdeg* and *impact*) of unassigned variables. Finally, using the previous 11 features the `mean` and `variance` is computed taking into account all visited checkpoints.

  The attributes listed above include a collection of 95 features.

6.5.2 Feature Pre-processing

Feature pre-processing is an important first step in machine learning [WF05]; it can significantly improve the predictive accuracy of the learned hypothesis. Typically, the descriptive features detailed above are on different scales; the number of variables and/or constraints can be high while the *impact* of (variable, value) is between 0 and 1. A data normalization step, scaling down feature values in $[-1, 1]$ (*MinMax-normalization*), is used.

Although selecting the most informative features might improve the performance, in this work we do not consider any feature selection algorithm, and only features that are constant over all examples are removed as they offer no discriminant information.

6.5.3 Learning and Using the Heuristics Model

The selection of the best heuristic for a given problem instance is formulated as a binary classification problem, as follows. Let $\mathcal{H}$ denote the set of k candidate heuristics, two particular elements in $\mathcal{H}$ being *DEF* (the default heuristics yielding reasonably good results on average) and *dyn-CS*, the (dynamic) ML-based heuristics model initially set to *DEF*.

Definition 6.2 Each training example $p_i = (x_i, y_i)$ is generated by applying some heuristic h ($h \in \mathcal{H}, h \neq$ *dyn-CS*) at some checkpoint in the search tree of a given problem instance. Description x_i ($\in \mathbb{R}^{97}$) is made of the static feature values describing the problem instance, the dynamic feature values computed at this checkpoint and describing the current search state, and two additional features: *checkpoint-id* gives the number of checkpoints up to now and *cutoff-information* gives the cutoff limit of the next restart. The associated label y_i is positive iff the associated runtime (using heuristic h instead of *dyn-CS* at the current checkpoint) improves on the heuristics model-based runtime (using *dyn-CS* at every checkpoint); otherwise, label y_i is negative.

If the problem instance cannot be solved i.e., time out during the exploration and exploitation modes whatever the heuristic used, it is discarded (since the associated training examples do not provide any relevant information).

In production mode, the hypothesis f learned from the above training examples (their generation is detailed in next subsection) is used as follows:

Definition 6.3 At each checkpoint, for each $h \in \mathcal{H}$, the description x_h and the associated value $f(x_h)$ are computed. If there exists a single h such that $f(x_h)$ is positive, it is selected and used in the subsequent search effort. If there exist several heuristics with positive $f(x_h)$, the one with maximal value is selected.[2] If $f(x_h)$ is negative for all h, the default heuristic *DEF* is selected.

6.5.4 Generating Examples in Exploration Mode

The Continuous Search paradigm uses the idle computer's CPU cycles to explore different heuristic combinations on the last seen problem instance, and see whether one could have done better than the current heuristics model on this instance. The rationale for this exploration is that improving on the last seen instance (albeit meaningless from a production viewpoint since the user already got a solution) will deliver useful indications as to how to best deal with further similar instances. In this way, the heuristics model will expectedly be tailored to the distribution of problem instances actually dealt with by the user.

The CS exploration proceeds by slightly perturbing the heuristics model. Let $dyn\text{-}CS^{-i,h}$ denote the policy defined as: use heuristics model *dyn-CS* at all checkpoints except the i-th one, and use heuristic h at the i-th-checkpoint.

Algorithm 6.1 describes the proposed Exploration mode for Continuous Search. A limited number (10) of checkpoints in the *dyn-CS*-based resolution of instance $\mathcal{I}$ are considered (line 2); for each checkpoint and each heuristic h (distinct from *dyn-CS*), a lesion study is conducted, applying h instead of *dyn-CS* at the i-th checkpoint (heuristics model $dyn\text{-}CS^{-i,h}$); the example (described from the i-th checkpoint and h) is labeled positive iff $dyn\text{-}CS^{-i,h}$ improves on *dyn-CS*, and added to the training set $\mathcal{E}$; once the exploration mode for a given instance is finished, the hypothesis model is updated by retraining the SVM, including the feature pre-processing, as stated in Sect. 6.5.2.

6.5.5 Imbalanced Examples

It is well-known that one of the heuristics often performs much better than the others for a particular distribution of problems [CB08]. Accordingly, negative train-

[2] The rationale for this decision is that the margin, i.e., the distance of the example w.r.t. the separating hyperplane, is interpreted as the confidence of the prediction [Vap95].

Algorithm 6.1 Exploration-time (instance: $\mathcal{I}$)

```
1: E = {} //initialize the training set
2: for all i in checkpoints(I) do
3:    for all h in H do
4:       Compute x describing the current checkpoint i and heuristic h
5:       if h ≠ dyn-CS then
6:          Launch dyn-CS^{-i,h}
7:          Define y = 1 iff dyn-CS^{-i,h} improves on dyn-CS and −1 otherwise
8:          E ← E ∪ {x, y}
9:       end if
10:   end for
11: end for
12: return E
```

ing examples considerably outnumber the positive ones (it is difficult to improve on the winning heuristics). This phenomenon, known as *imbalanced distribution*, might severely hinder the SVM algorithm [AKJ04]. Two simple ways of enforcing a balanced distribution in such cases, intensively examined in the literature and considered in earlier work [AHS09], are to oversample examples in the minority class (generating additional positive examples by Gaussianly perturbing the available ones) and/or undersample examples in the majority class.

Another option is to use prior knowledge to rebalance the training distribution. Formally, instead of labeling an example positive (resp, negative) iff the associated runtime is strictly less (resp. greater) than that of the heuristic model, we consider the difference between the runtimes. If the difference is less than some tolerance value dt, then the example is relabeled as positive.

The number of positive examples and hence the coverage of the learned heuristics model increase with dt; in the experiments (Sect. 6.6), dt is set to 1 minute iff *time-exploitation* (time required to solve a given instance in production mode) is greater than 1 minute; otherwise dt is set to *time-exploitation*.

6.6 Experimental Validation

This section reports on the experimental validation of the proposed Continuous Search approach. All tests were conducted on Linux Mandriva 2009 boxes with 8 GB of RAM and 2.33 GHz Intel processors.

6.6.1 Experimental Settings

The presented experiments consider 496 CSP instances taken from different repositories.

- *nsp*: 100 *nurse scheduling* instances from the MiniZinc[3] repository.
- *bibd*: 83 *Balance Incomplete Block Design* instances from the XCSP[4] repository, translated into Gecode using Tailor.[5]
- *js*: 130 *Job Shop* instances from the XCSP repository.
- *geom*: 100 *Geometric* instances from the XCSP repository.
- *lfn*: 83 *Langford number* instances, translated into Gecode using global and channelling constraints.

The learning algorithm used in the experimental validation of the proposed approach is a Support Vector Machine with Gaussian kernel, using the libSVM implementation with default parameters.[6] All considered CSP heuristics (Sect. 6.3) are home-made implementations integrated in the Gecode 2.1.1 [Gec06] constraint solver. Our *dyn-CS* technique was used as a heuristics model on top of the heuristics set $\mathcal{H} = \{dom\text{-}wdeg, wdeg, dom\text{-}deg, min\text{-}dom, impacts\}$, taking *min-value* as the value selection heuristic. The cutoff value used to restart the search was initially set to 1,000 and the cutoff increase policy to $\times 1.5$; the same cutoff policy is used in all the experimental scenarios.

Continuous Search was assessed comparatively to the best two dynamic variable ordering heuristics on the considered problems, namely *dom-wdeg* and *wdeg*. It must be noted that Continuous Search, being a lifelong learning system, will depend on the curriculum, that is the order of the submitted instances. If the user "pedagogically" starts by submitting informative instances first, the performance in the first stages will be better than if untypical and awkward instances are considered first. For the sake of fairness, the performance reported for Continuous Search on each problem instance is the median performance over 10 random orderings of the CSP instances.

6.6.2 Practical Performances

Figure 6.3 highlights the Continuous Search results on Langford number problems, comparatively to *dom-wdeg* and *wdeg*. The x-axis gives the number of problems solved and the y-axis presents the cumulated runtime. The (median) *dyn-CS* performance (gray line) is satisfactory as it solves 12 more instances than *dom-wdeg* (black line) and *wdeg* (light gray line). The dispersion of the *dyn-CS* results depending on the instance ordering is depicted from the set of dashed lines. Note that traditional portfolio approaches such as [HHHLB06, SM07, XHHLB07] do not present such performance variations as they assume a complete set of training examples to be available beforehand.

[3] http://www.g12.cs.mu.oz.au/minizinc/download.html.

[4] http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html.

[5] http://www.cs.st-andrews.ac.uk/~andrea/tailor/.

[6] http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

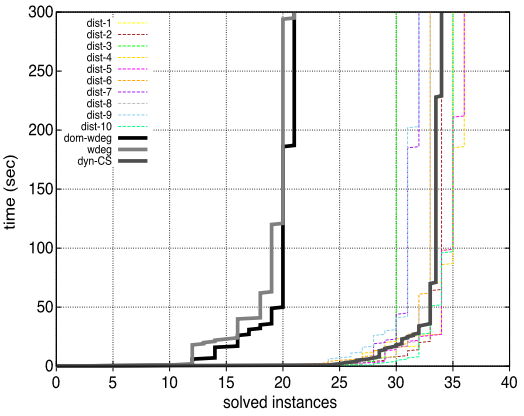


Fig. 6.3 Langford number (lfn): Number of instances solved in less than 5 minutes with *dyn-CS*, *wdeg*, and *dom-wdeg*. *Dashed lines* illustrate the performance of *dyn-CS* for a particular instance ordering

Figure 6.4 depicts the performance of *dyn-CS*, *dom-wdeg* and *wdeg* on all other problem families, respectively (bibd, js, nsp, and geom). On the bibd (Fig. 6.4(a)) and js (Fig. 6.4(b)) problems, the best heuristic is *dom-wdeg*, solving three more instances than *dyn-CS*. Note that *dom-wdeg* and *wdeg* coincide on bibd since all decision variables are Boolean.

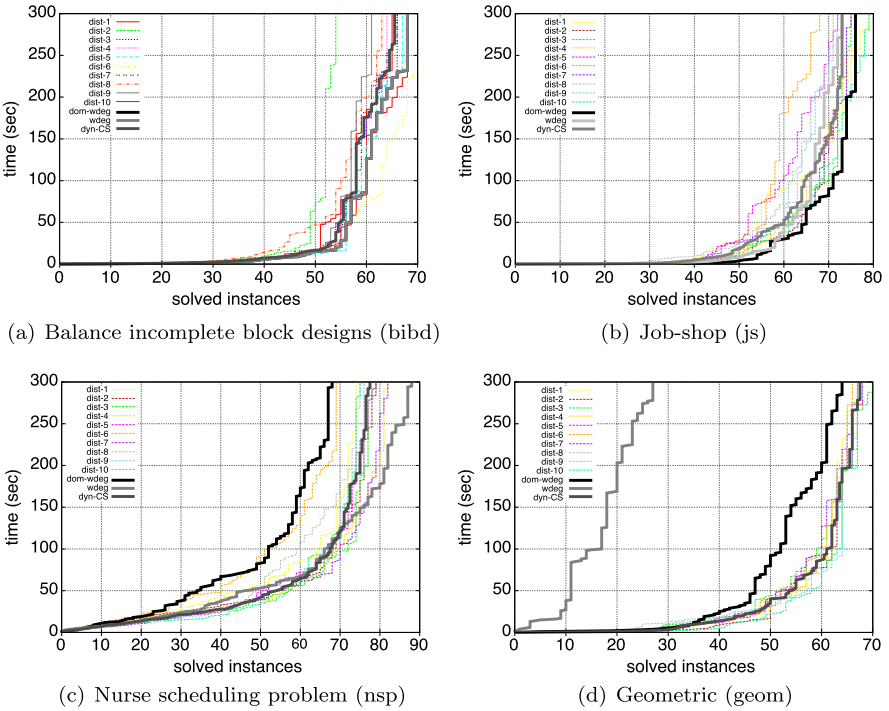
On nsp (Fig. 6.4(c)), *dyn-CS* solves nine more problems than *dom-wdeg*, but is outperformed by *wdeg* by 11 problems. On geom (Fig. 6.4(d)), *dyn-CS* improves on the other heuristics, solving respectively three more instances and 40 more instances than *dom-wdeg* and *wdeg*.

These results suggest that *dyn-CS* is most often able to pick up the best heuristics on a given problem family, and sometimes able to significantly improve on the best of the available heuristics.

All experimental results are summarized in Table 6.1, reporting for each considered heuristic the number of instances solved (#sol), the total computational cost for all instances (time, in hours), and the average time (avg-time, in minutes) per instance, over all problem families. These results confirm that *dyn-CS* outperforms *dom-wdeg* and *wdeg*, solving respectively 18 and 41 instances more out of 315. Furthermore, it shows that *dyn-CS* is slightly faster than the other heuristics, with an average time of 2.11 minutes, against respectively 2.39 for *dom-wdeg* and 2.61 for *wdeg*. It is also worth mentioning that the total CPU time required to complete the exploration (or learning) mode after solving a given instance was on average no longer than two hours.

Additionally, a random heuristic selection scenario was also experimented with (i.e., executing 10 times each instance with a uniform heuristic selection and reporting the median value over the 10 runs). The random selection strategy was able to solve 278 out of 496 instances, 19 instances less than *dom-wdeg* and 37 instances less than *dyn-CS*.

Another interesting lesson learned from the experiments concerns the difficulty of the underlying learning problem, and the generalization error of the learned hypothesis. The generalization error in the Continuous Search framework is estimated by 10-fold Cross Validation on the whole training set (including all train-



(a) Balance incomplete block designs (bibd)

(b) Job-shop (js)

(c) Nurse scheduling problem (nsp)

(d) Geometric (geom)

Fig. 6.4 Number of instances solved in less than 5 minutes

Table 6.1 Total solved instances

Problem	*dom-wdeg*			*wdeg*			*dyn-CS*		
	#Sol	Time (h)	Avg-time (m)	#Sol	Time (h)	Avg-time (m)	#Sol	Time (h)	Avg-time (m)
nsp	68	3.9	2.34	**88**	**2.6**	**1.56**	77	2.9	1.74
bibd	**68**	**1.8**	**1.37**	68	1.8	1.37	65	2.0	1.44
js	**76**	**4.9**	**2.26**	73	5.1	2.35	73	5.2	2.4
lfn	21	5.2	3.75	21	5.3	3.83	**33**	**4.1**	**2.96**
geom	64	3.9	2.34	27	6.8	4.08	**67**	**3.3**	**1.98**
Total	297	19.7	2.39	274	21.6	2.61	**315**	**17.5**	**2.11**

ing examples generated in exploration mode). Table 6.2 reports on the predictive accuracy of the SVM algorithm (with same default settings) on all problem families, with an average accuracy of 67 %. As could have been expected, the predictive accuracy is correlated to the performance of Continuous Search: the problems with best accuracy and best performance improvement are geom and lfn.

Table 6.2 Predictive accuracy of the heuristics model

bibd	nsp	geom	js	lfn
63.2 %	58.8 %	76.9 %	63.6 %	73.8 %

Table 6.3 Total solved instances

Problem	#Sol	Time (h)	Problem	#Sol	Time (h)
nsp-geom‡	55	4.1	lfn-bibd‡	23	5.3
nsp-geom†	67	3.4	lfn-bibd†	63	2.3

To give an idea of order, 62 % predictive accuracy was reported in the context of SATzilla [XHHLB07], aimed at selecting of the best heuristic in a portfolio.

A direct comparison of the predictive accuracy might however be biased. On the one hand SATzilla errors are attributed to the selection of some near-optimal heuristics, by the authors; on the other hand, Continuous Search would involve several selection steps (in each checkpoint) and could thus compensate for earlier errors.

6.6.3 The Power of Adaptation

Our second experimental test combines instances from different domains in order to show how CS is able to adapt to changing problem distribution. Indeed, unlike classical portfolio-based approaches which can only be applied if the training and exploitation sets come from the same domain, CS can adapt to changes and provide top performance even if the problems change.

In this context, Table 6.3 reports the results on the geom (left) and bibd (right) problems by considering the following two scenarios. In the first scenario, we are going to emulate a portfolio-based search which would use the wrong domain to train. In *nsp-geom*‡, CS incrementally learns while solving the 100 nsp instances, and then solves one by one the 100 geom instances. However, when switching to this second domain, incremental learning is switched off, and checkpoint adaptation uses the model learnt on nsp. In the second scenario, *nsp-geom*†, we solve nsp, then geom instances one by one, but this time we keep the incremental learning on when switching from the first domain to the second one—as if CS was not aware of the transition.

As we can see in the first line of the table, training on the wrong domain gives poor performance (55 instances solved in 4.1 hours). In contrast, the second line shows that CS can recover from training on the wrong domain thanks to its incremental adaptation (solving 67 instances in 3.4 hours). The right part of the table reports similar results for the bibd problem.

As can be observed in nsp-geom† and lfn-bibd†, CS successfully identifies the new distribution of problems, solving respectively the same number and two fewer instances than geom and bibd when CS is only applied to this domain starting from

scratch. However, the detection of the new distribution introduces an overhead in the solving time (see results for single domain in Table 6.1).

6.7 Summary

The main contribution of the presented approach, the Continuous Search framework, aims at designing a heuristics model tailored to the user problem distribution, allowing it to get top performance from the constraint solver. The representative instances needed to train a good heuristics model are not assumed to be available beforehand; they are gradually built and exploited to improve the current heuristics model, by stealing the idle CPU cycles of the computing system. Metaphorically speaking, the constraint solver uses its spare time to play against itself and gradually improve its strategy over time; further, this expertise is relevant to the real-world problems considered by the user, all the more so as it directly relates to the problem instances submitted to the system.

The experimental results suggest that Continuous Search is able to pick up the best of a set of heuristics on a diverse set of problems, by exploiting the incoming instances; in two out of five problems, Continuous Search swiftly builds up a mixed strategy, significantly overcoming all baseline heuristics. With the other classes of problems, its performance is comparable to the best two single heuristics. Our experiments also showed the capacity of adaptation of CS. Moving from one problem domain to another one is possible thanks to its incremental learning capacity. This capacity is a major improvement over classical portfolio-based approaches which only work when offline training and exploitation use instances from the same domain.

Chapter 7
Autonomous Search

7.1 Introduction

The selection and the correct setting of the most suitable algorithm for solving a given problem was already investigated many years ago [Ric75]. The proposed abstract model was suggested to extract features in order to characterize the problem, to search for a suitable algorithm in the space of available algorithms and then to evaluate its performances with respect to a set of measures. These considerations are still valid and this general problem can indeed be considered at least from two complementary points of view:

- selecting solving techniques or algorithms from a set of available techniques
- tuning an algorithm with respect to a given instance of a problem

To address these issues, the proposed approaches include tools from different computer science areas, especially from machine learning. Moreover, they have been developed to answer the algorithm selection problem in various fields as described in the recent survey of Smith-Miles [SM08].

In this chapter, we will focus on the application of this general question to constraint satisfaction and optimization problems. In this particular area, the problem of finding the best configuration in a search space of heuristic algorithms is also related to the recent notion of Hyper-heuristics [BHK+09a, BKN+03, CKS02]. Hyper-heuristics are methods that aim at automating the process of selecting, combining, generating, or adapting several simpler heuristics (or components of such heuristics) to efficiently solve computational search problems. Hyper-heuristics are also defined as "heuristics to choose heuristics" [CS00] or "heuristics to generate heuristics" [BEDP08]. This idea was pioneered in the early 1960s with the combination of scheduling rules [FT63, CGTT63]. Hyper-heuristics that manage a set of given available basic search heuristics by means of search strategies or other parameters have been widely used for solving combinatorial problems (see Burke et al. [BHK+09a] for a recent survey).

From a practical point of view, Burke et al. [BHK+09b] proposed a comprehensive classification of hyper-heuristics considering two dimensions: the nature of the

Y. Hamadi, *Combinatorial Search: From Algorithms to Systems*,
DOI 10.1007/978-3-642-41482-4_7, © Springer-Verlag Berlin Heidelberg 2013

heuristics and the source of the feedback for learning. They thus distinguish between heuristics that select heuristics from a pre-existing set of search heuristics and heuristics that generate new heuristics from basic components. Concerning the feedback, they identify three categories: online learning, offline learning, and no learning. The distinction between online and offline processes was previously proposed in order to classify parameter settings in evolutionary algorithms [EHM99], distinguishing parameter tuning (offline) from parameter control (online).

As classical offline mechanisms, we may mention *portfolio* algorithms [HHHLB06, XHHLB08], where previously acquired knowledge is used in order to select the suitable solving method with regard to a given problem instance. Gagliolo et al. [GS08] use reinforcement learning-based techniques for algorithm selection.

Online control of heuristics has been widely addressed, for instance in adaptive strategies in evolutionary computation [Thi07, Kra08], in adaptive neighborhood selection for local search [HR06, CB01, PR08], or in constraint programming solvers [EFW+02].

When considering parameter setting, the space of possible algorithms is the set of possible configurations of a given algorithmic scheme induced by the possible values of its parameters that control its computational behavior. Parameter tuning of evolutionary algorithms has been investigated for many years (we refer the reader to the book [LLM07] for a recent survey). Adaptive control strategies were also proposed for other solving approaches such as local search [Hoo02a, PK01]. Offline mechanisms are also available for tuning parameters, such as in the work of Hutter et al. [HHLBS09], which proposes to use a local search algorithm in order to automatically find a good (i.e., efficient) configuration of an algorithm in the parameter space. Including this work, a more complete view of the configuration of search algorithms is presented in the Ph.D. thesis of Hutter [Hut09]. Revac [NE07, NE06] is a method that uses information theory to identify the most important parameters and calibrate them efficiently. We may also mention that racing techniques [BSPV02, YG04, YG05, YG07] can be used to choose suitable parameter settings when facing multiple choices.

Another important research community that focuses on very related problems has been established under the name *Reactive Search* by Battiti et al. [BBM08, BB09]. After focusing on local search with the seminal works on reactive tabu [BT94] or adaptive simulated annealing [Ing89], this community is now growing through the dedicated *Learning and Intelligent OptimizatioN (LION)* conference.

It clearly appears that these approaches share common principles and purposes and have been developed in parallel in different but connected communities. Their foundations rely on the fact that, since the solving techniques and search heuristics are more and more sophisticated and the problem structures more and more intricate, the choice and the correct setting of a solving algorithm is becoming an intractable task for most users. Therefore, there is a rising need for an alternative problem-solving framework. According to the above brief historical review, we have observed that these approaches have indeed their own specificities that are induced by their seminal supporting works. In this chapter, we propose to integrate the main

motivations and goals into the more general concept of Autonomous Search (AS) [HSe12, HMS08a, HMS08b].

This chapter is organized as follows. In Sect. 7.2, we describe the general architecture of modern solvers. We present the specificities of autonomous solvers and formalize their solving mechanisms with a set of rules in Sect. 7.3. In Sect. 7.4, we illustrate different solver architectures by providing examples from the literature and we characterize these solvers using our previous rule-based description framework.

7.2 Solver Architecture

In this section, we present the basic concepts related to the notion of solver in the context of general constraint problems solving, which provide an introduction to problem solving. By general problems, we mean optimization or constraint satisfaction problems, whose variables may take their values over various domains (boolean, integer, real number, etc.). In fact, solving such problems is the main interest of different but complementary communities in computer science: operations research, global optimization, mathematical programming, constraint programming, and artificial intelligence. From the different underlying paradigms that are associated to these research areas, we may try to identify common principles, which are shared by the resulting solving algorithms and techniques that can be used for the ultimate solving purpose.

As it has finally been suggested by the notion of metaheuristics [GK03], solvers could be viewed as a general skeleton whose components are selected according to the problem or the class of problems to be solved. Indeed, from our point of view we want to look carefully at the components of the solver that define its structural properties and at its parameters or external features that define its behavior. On one hand, one has to choose the components of the solver, and on the other hand one should configure how these internal components are used during the solving process. We identify the core of the solver which is composed by one or several solving algorithms. Note that here we distinguish between the solver and the solving algorithm, which is a part of the solver but corresponds to the real operational solving process. A basic solving algorithm corresponds to the management of solving techniques, abstracted by the notion of operators, making use of a solving strategy that schedules the use of these operators. A solving algorithm is designed of course according to the internal model, which defines the search space, and uses a function to evaluate the elements of the search space. All these components can be subjected to various parameters that define their behavior. A given parameterization defines thus what we call a configuration of the solver. At this level, a control layer can be introduced, especially in an autonomous solver, to manage the components and modify the configuration of the solver during the solving process. The general description of a solver architecture is illustrated by Fig. 7.1.

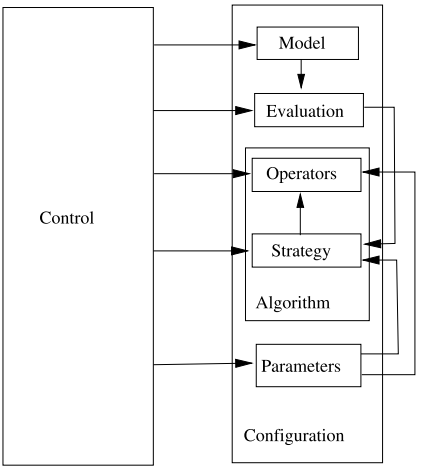


Fig. 7.1 The general architecture of a solver

7.2.1 Problem Modeling/Encoding

The encoding of the problem is considered separately from the solver itself. In fact, most of the time, a solver is designed for a specific encoding framework that induces a specific internal representation that corresponds to the model. While the classic CSP modeling framework [Tsa93] is commonly used as a description tool for all solving methods, the internal encoding of the problem and its possible configurations involve different representations (e.g., complete vs. partial assignments, etc.). One should note that different modeling and encoding paradigms can be used. In constraint programming [Apt03, Dec03, MS98, Hen89] one could encode constraints as tuples of allowed values or use a more declarative first order language with relations and functions. Moreover, other paradigms can be used to encode CSPs, such as SAT [BHvMW09], and various transformation schemes have been investigated [BHZ06, Wal00, Hoo99b]. On the metaheuristics side, the encoding of the possible configurations of the problem has a direct impact on the search space and on the search landscape. For instance, one may include directly some of the constraints of the problem in the encoding as this is the case when using permutations for the Traveling Salesman Problem (TSP [ABCC07]), which corresponds to the constraint *each city is visited once and only once*. In genetic algorithms [Jon06, ES03b, Mic92] or local search [AL03, HM05], encoding may have a significant impact on the performance of the algorithm. The encoding of continuous optimization problems (i.e., over real numbers) also requires providing suitable data structures, for instance, floating point representation for genetic algorithms [JM91] or continuous and interval arithmetic in constraint programming [BG06]. The internal representation of the model can be considered as a component of the solver. This representation has of course a direct computational impact on the evaluation function and also on the solving techniques that are implemented through operators.

7.2.2 The Evaluation Function

The evaluation function is related to the nature of the problem. From a general point of view, a function is needed to evaluate possible configurations of the problem with regard to its constraints and variable values. An evaluation function may evaluate the number of conflicts or check the satisfiability of a given constraint set, or use particular consistency notions (global or local). Such a function can also be used to prune the search space when dealing with optimization problems. Again, this notion is more traditionally used in the context of metaheuristics than in classic complete constraint programming solvers. But it seems rather intuitive to have such a function to assess the current search state in order to be able to check if the solver has reached a solution or not. Moreover, this evaluation function clearly appears when dealing with constraint optimization problems and using branch-and-bound algorithms.

7.2.3 The Solving Algorithm

Our purpose is to distinguish between the basic structure of the algorithm and its configurable components. For instance, in a classic complete constraint solver, the skeleton of the algorithm is the basic backtracking process, whose heuristics and propagation rules can be configured. In an evolutionary algorithm, the core of the solver is constituted by the population management. A solver may include the following components that we have to take into account:

- *A set of operators*: operators are used in the solving process to compute search states. These operators may basically achieve variable instantiation, constraint propagation, local moves, recombination or mutation selection, etc. Most of the time, they are parameterized and use an evaluation function to compute their results (e.g., number of violated constraints or evaluation of the neighborhood in local search algorithms). Note that these operators may be used to achieve a complete search (i.e., find a solution or prove unsatisfiability of the problem) or to perform an incomplete search (i.e., find a solution if possible or a sub-optimal solution).
  - Concerning tree search-based methods, the notion of operator for performing solving steps during the search process corresponds to basic solving techniques. For instance if we consider a classic backtracking-based solver in constraint programming, we need an enumeration operator that is used to assign values to variables and reduction operators that enforce consistencies in order to reduce the domains of the variables. The search process then corresponds to the progressive construction of a search tree whose nodes are subjected to the application of the previously described operators. When considering numerical variables over intervals, we may add splitting operators. Of course these operators may include heuristics concerning the choice of the variables to be enumerated, and the choice of the values, but also other parameters to adjust

their behavior. Indeed, constraint propagation can be formalized by means of rules [Apt03, FA03], which support operator-based description and provide a theoretical framework to assess properties of the solver such as termination.
– On the metaheuristics side, in evolutionary computing [Gol89, Jon06, ES03b] we usually consider variation operators (mutation operators and recombination operators) and selection operators. Considering an evolutionary algorithm, it is possible to established some convergence properties such as the famous schemata theorem [Hol75]. There exist some general purpose operators as, for instance, the uniform crossover [Syw89] or the Gaussian mutation [Kje91]. To get better performance, these operators are often designed with respect to the specificities of the problem to be solved. In local search [AL03], local moves are based on neighborhood functions.

All these operators are most of the time subjected to parameters that may modify their behavior but, more important, that also control their application in the search process.

- *A solving strategy*: the solving strategy schedules how operators are used. In the previous example, in a complete tree-based search process, the strategy will consist in alternating enumeration and constraint propagation. The strategy can be subjected to parameters that will indicate which operators to choose in the general scheduling of the basic solving process.

7.2.4 Configuration of the Solver: The Parameters

The solver usually includes parameters that are used to modify the behavior of its components. A configuration of the solver is then an instance of the parameters together with its components. Parameters are variables that can be used in the general search process to decide how the other components are used. These parameters may correspond to various data that will be involved in the choice of the operator to be applied at a given search state. For instance, we may consider the probability of application of the operators (e.g., genetic operators in evolutionary algorithms, the noise in random walk for local search algorithms [SKC94a]) or of some tuning of the heuristics themselves (e.g., tabu list length in Tabu Search [GL97]).

Parameter setting is an important issue for evolutionary algorithms [LLM07]. Parameter setting for local search algorithms is also handled in [BBM08]. In constraint programming much work has been done to study basic choice heuristics (see [EFW+02] for instance), but also to evaluate the possible difficulties related to the classic use of basic heuristics such as heavy-tailed problems [GSCK00] (these studies particularly demonstrate the benefit of randomization when solving multiple instances of a given family of problems compared to the use of a single predefined heuristic).

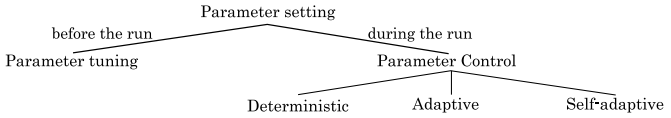


Fig. 7.2 Control taxonomy proposed by Eiben et al. [EHM99]

7.2.5 Control

Modern solvers also include external or internal mechanisms that allow the solver to change its configuration by selecting the suitable operators to apply, or tuning the parameters, or adding specific information to the model. These mechanisms often include machine learning techniques and will be detailed later. Of course, control rules will often focus on the management of the parameters and/or of the operators of the solver.

7.2.6 Existing Classifications and Taxonomies

As mentioned before, we may identify at least three important domains where related work has already been conducted. These lines of work have led to the use of different terminologies and concepts that we try to recall here.

In evolutionary computing, parameters setting [LLM07] constitutes a major issue and we may recall the taxonomy proposed by Eiben et al. [EHM99] (see Fig. 7.2).

Methods are classified depending on whether they attempt to set parameters before the run (tuning) or during the run (control). The goal of parameter tuning is to obtain parameter values that could be useful over a wide range of problems. Such results require a large number of experimental evaluations and are generally based on empirical observations. Parameter control is divided into three branches according to the degree of autonomy of the strategies. Control is deterministic when parameters are changed according to a previously established schedule, adaptive when parameters are modified according to rules that take into account the state of the search, and self-adaptive when parameters are encoded into individuals in order to evolve conjointly with the other variables of the problem.

In [SE09], Eiben and Smit recall the difference between numeric and symbolic parameters. In [NSE08], symbolic parameters are called components, whose elements are operators. In this chapter, we choose to use the notions of parameters for numeric parameters. As defined above, the operators are configurable components of the solver that implement solving techniques.

In [BB09], reactive search is characterized by the integration of machine learning techniques into search heuristics. A classification of the source of information that is used by the algorithm is proposed to distinguish between problem-dependent information, task-dependent information, and local properties.

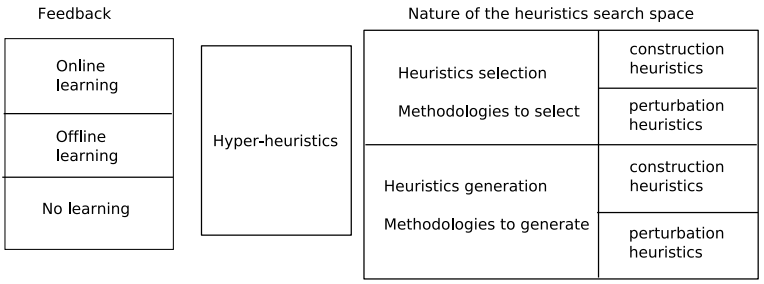


Fig. 7.3 Classification of hyper-heuristics by Burke et al. [BHK+09b]

In their survey [BHK+09b], Burke et al. propose a classification of hyper-heuristics, which are defined as search methods or learning mechanisms for selecting or generating heuristics to solve computational search problems. As mentioned above, this classification also distinguishes between two dimensions: the different sources of feedback information and the nature of the heuristic search space. This classification is summarized in Fig. 7.3.

The feedback, when used, corresponds here to the information that is learned during solving (online) or using a set of training instances (offline). The authors identify two families of low-level heuristics: construction heuristics (used to incrementally build a solution) and perturbation heuristics (used to iteratively improve a starting solution). The hyper-heuristics level can use heuristic selection methodologies, which produce combinations of pre-existing low-level heuristics, or heuristics generation methodologies, that generate new heuristics from basic blocks of low-level heuristics.

Another interesting classification is proposed in [GS08], in which Gagliolo et al. are interested in the algorithm selection problem [Ric75] and describe the different selection techniques according to the following points of view. The problem consists in assigning algorithms from a set of possible alternative solving methods to a set of problem instances in order to improve the performance. Different dimensions are identified with regard to this algorithm selection problem:

- The nature of the problems to be solved: decision vs. optimization problems.
- The generality of the selection process: selection of an algorithm for a set of instances or selection of an algorithm for each instance.
- The reactivity of the selection process: the selection can be static and made before running all the selected algorithms or can be dynamically adapted during execution.
- The feedback used by the selection process: the selection can be made from scratch or using previously acquired knowledge.
- The source of feedback: as in the previous classification, when learning is used in the selection process, one may consider offline (using separated training instances) or online (updating information during solving) learning techniques.

As claimed in the introduction, autonomous search aims at providing a more uniform description and characterization of these different trends, which have close relationships.

7.3 Architecture of Autonomous Solvers

We may define autonomous solvers as solvers that contain control in their search process (i.e., the solvers described in Sect. 7.4.2). We want to study such autonomous systems w.r.t. their specific control methods.

A general control process includes a strategy that manages the modification of some of the solver's components and behavioral features after the application of some solving functions. The overall strategy to combine and use components and parameters can be based on learning that uses information from the current solving process or from previous solved instances (see remarks in Sect. 7.2.6). Therefore, modifications are often based on a subset of search states. Given a solver, we have to consider the interactions between the heuristics and the strategy which selects the heuristics at a meta-level (notion of hyper-heuristics).

On the one hand, one can consider the solver and its history and current environment (i.e., the previously computed search states and eventually other external information related to previous computations) as an experimental system, which is observed from an external point of view. Such a supervised approach then consists in correctly controlling the solver by adjusting its components according to criteria and decision rules (these rules may be automatically generated by means of statistics and machine learning tools or even by human experts). On the other hand, one may consider that the solver changes the environment at each step of the solving process and that this environment returns feedback information to the solver in order to manage its adaptation to this changing context (different types of feedback may be taken into account, as mentioned in Sect. 7.2.6). In this case, we will use self-adaptation. To illustrate these ideas, we propose a high-level picture of an autonomous search system (see Fig. 7.4).

7.3.1 Control by Self-adaptation

In self-adaptation, the adaptive mechanism is coupled with the search components, directly changing them in response to the consequences of their actions. Self-adaptive techniques are tightly integrated with the search process and should usually require little overhead. The algorithm is observing its own behavior in an online fashion, modifying its parameters accordingly. This information can be either directly collected from the problem or indirectly computed through the perceived efficiency of individual components. Because the adaptation is done online, there is an important trade-off between the time spent computing heuristic information and the gains that are to be expected from this information.

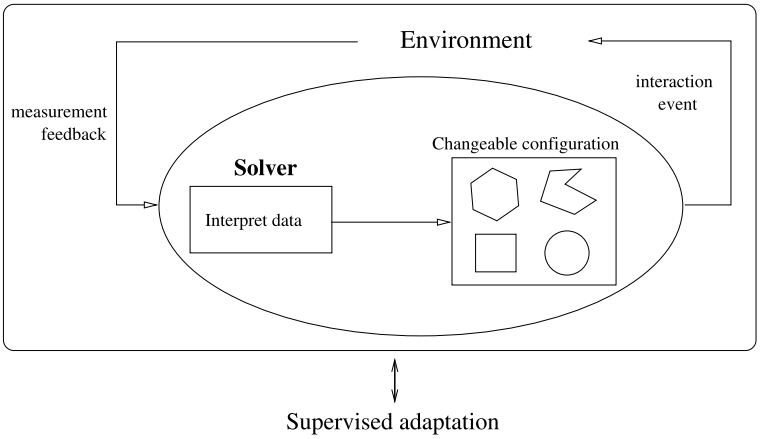


Fig. 7.4 The global architecture of an Autonomous Search System

7.3.2 Control by Supervised Adaptation

Supervised adaptation works at a higher level. It is usually external and its mechanisms are not coupled with the search process. It can be seen as a monitor that observes the search and analyzes it. It can modify the components of the solver (or require the solver to modify its components) in order to adapt it. Supervised adaptation can use more information, e.g., learning-based knowledge, etc. In some cases, we can imagine that typical supervised actions can be *compiled* into self-adaptive mechanisms.

7.3.3 Searching for a Solution vs. Solutions for Searching

It appears now that the problem of building a good Autonomous Search solver is more ambitious than that of finding a solution to a given instance of a problem. Indeed, inspired by the seminal consideration of John Rice [Ric75] when he was abstracting the problem of finding the best algorithm for solving a given problem, we need to take into account at least three important spaces in which an autonomous search process takes place.

- The search space: the search space is induced by the encoding of the problem and corresponds to the set of all potential configurations of the problem that one has to consider in order to find a solution (or to find all solutions, or to find an optimal solution, etc.). This search space can also be partitioned, for optimization problems, into the set of feasible solutions and infeasible solutions with respect to the constraints of the problem.
- The search landscape: the search landscape is related to the evaluation function that assigns a quality value to the elements of the search space. If indeed this

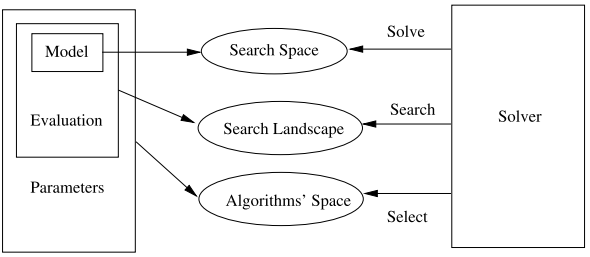


Fig. 7.5 The solver and its action with respect to different spaces

notion is rather of limited use in the area of complete solvers, it is a crucial notion when using heuristics or metaheuristics, search algorithms whose purpose is to explore and exploit this landscape in order to find solutions. Most of the metaheuristics, designed according to the management of this exploration-exploitation balance and the characteristics of the search landscapes, often use geographical metaphors: How to travel across plateaus? How to escape from a local optimum by climbing hills?, etc.

- The algorithms space: according to the previous description of solver architecture, we have highlighted that a solver consists of components that define its structural properties together with a set of behavioral features (parameters and control rules). As mentioned before, given a basic algorithmic skeleton we may consider a set of possible solvers that correspond to the possible component choices and configurations. This algorithms space can also be composed of different solvers when dealing with portfolio-based algorithm selection.

The relationships between these spaces are illustrated in Fig. 7.5. Indeed, the ultimate autonomous search purpose can be formulated as: finding a suitable algorithm that is able to efficiently explore and exploit the search landscape in order to suitably manage the search space and find solutions to the initial problem.

7.3.4 A Rule-Based Characterization of Solvers

As already mentioned, the solving techniques used for solving such problems may include very different features, from complete tree-based solvers to local search or evolutionary algorithms. In this presentation, we will attempt to abstract these solving features in order to be able to address general solving algorithms, focusing on their autonomous aspects as described above. Indeed, such rule-based formalizations have already been proposed for modeling some constraint programming solving processes [Apt03, FA03] and also for hybrid solvers including local search [MSL04]. Here, our purpose is not really to prove some properties of the solvers but rather to highlight their basic operational mechanisms in order to classify them with regard to their behavioral and structural characteristics.

When using a solver, one may distinguish two main tasks that correspond indeed to different but closely related levels of technical accuracy that can be achieved by more or less specialized users:

- The component design: this phase consists in choosing the suitable components described in Sect. 7.2.3 that should be included in the solver with regard to the problem characteristics for instance. As mentioned above, these components constitute the architecture of the solver.
- The configuration of the solver through parameters settings and controls: this second phase consists in defining through control features how the components can be used during the solving process.

Based on this consideration and on the general solver architecture depicted in Fig. 7.1, we propose a formal description in the next section.

Formal Description We define here some basic notions in order to characterize the behavior of solvers with a computationally oriented taxonomy. This approach will allow us to characterize the solvers. We first recall some basic concepts related to constraint satisfaction and optimization problems.

Definition 7.1 (CSP) A CSP is a triple $(X, D, \mathcal{C})$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables whose values are restricted to given domains $D = \{D_1, \ldots, D_n\}$. There exists a bijective mapping that assigns each variable x_i to its corresponding domain, D_{x_i}. We consider a set of constraints $\mathcal{C}$ as a set of relations over the variables X.

Definition 7.2 (Search Space) The search space $\mathcal{S}$ is a subset of the possible configurations of the problem and can be the Cartesian product of domains, $\prod_{x \in X} D_x$. The choice of the internal representation (i.e., the model) defines the search space. An element s of the search space will be called a *candidate solution*.

Definition 7.3 (Solution) A feasible solution is an assignment of values to variables, which can be seen as an element of $\mathcal{S}$ (i.e., given an assignment $\theta : X \rightarrow \prod_{i=1}^{n} D_i$, $\theta(x_i) \in D_{x_i}$), and which satisfies all the constraints of $\mathcal{C}$. In the context of optimization problems, we also consider an objective function $f : \mathcal{S} \rightarrow \mathbb{R}$. An optimal solution is a feasible solution maximizing or minimizing, as appropriate, the function f.

We have now to define, according to Sect. 7.2, the different elements that are included in the solver.

Definition 7.4 (Evaluation Functions) We denote by E the set of evaluation functions $e : \mathcal{S} \rightarrow \mathbb{R}$.

Definition 7.5 (Parameters) We denote by P the set of parameters, and a parameterization π is a mapping that assigns a value to each parameter. We denote by Π the set of parameterizations.

Definition 7.6 (Solving Operators) We denote by Ω a set of solving operators (operators for short) that are functions $o : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$.

Definition 7.7 (Solving Strategy) We denote by H the set of solving strategies that are functions $h : 2^{\mathcal{S}} \times \Pi \times E \rightarrow \Omega$.

For sake of simplicity, in the following we will refer to solving strategies as strategies. Solving strategies and solving operators are the key points of the solving algorithm (see Fig. 7.1): a strategy manages some operators to compute the solutions. We obtain:

$$\textbf{Solving algorithm} = \textbf{solving strategy} + \textbf{solving operators}$$

We now formalize the solving processes as transitions using rules over computation states.

Definition 7.8 (Computation State) Given a CSP (X, D, C), a search space $\mathcal{S}$, a set of operators Ω, a set of evaluation functions E, a set of parameters P and a set of solving strategies H, a computation state is a tuple $\langle O, \mathcal{S}, e, \pi, h|S\rangle$ where:

- $O \subseteq \Omega$, where O is the set of operators currently used in the solver
- $S \subseteq \mathcal{S}$ is the current subset of candidate solutions
- $e \in E$ is an evaluation function
- $\pi \in \Pi$ is the current parameterization
- $h \in H$ is the current solving strategy

Remarks

- It is important to note that Ω, E, and H are sets that may not be yet computable. For example, H represents the set of all possible strategies, either already existing or that will be discovered by the solver (as defined in Definition 7.11). Similarly, all the operators of Ω are not known since they can be designed later by the solver. However, O is known, and all its operators as well.
- S corresponds to the internal basic search structure: the search state. For instance, if we consider a genetic algorithm the search state will be a population. In the case of a complete backtracking solver, it will consist in an incomplete assignment, etc.
- O is the current set of operators available in the solver at a given stage extracted from a set Ω of potential operators that could be used in this solver. Indeed, some solvers may use new solving operators that are produced online or offline according to a general specification or according to design rules. Note that an operator allows the solver to perform a transition from one search state to another. This is therefore the key concept of the solving process and we want to keep it as general as possible to handle various solving paradigms (as mentioned above).
- The evaluation function e must evaluate the candidate solutions. This evaluation is used by the strategy in order to drive the basic solving task and by the control in order to drive the solver behavior.
- The solving strategy h will be used to select the suitable operator to apply on the current candidate solutions with respect to the current parameterization π and the evaluation function e.

Note that, for the sake of simplicity, we restrict ourselves to solvers that have only one evaluation function and one search space at a time. This is typically the case but this framework could be easily generalized to capture more exotic situations.

We denote by CS the set of computation states. Note that a computation state corresponds in fact to a search state together with the current configuration of the solver.

Definition 7.9 (Computation Rules) A computation rule is a rule $\frac{\sigma'}{\sigma}$ where σ and σ' are computation states from CS.

Identification of Computation Rules We identify here specific families of computation rules with respect to the way they modify the computation states.

- *Solving*: The fundamental solving task of a classic solver consists in computing a new state from the current one according to a solving strategy that chooses the suitable operator to apply with respect to the current candidate solutions, the parameterization, and the evaluation function. This corresponds to the following rule:

  [Solv] Solving

  $$\frac{\langle O, \mathcal{S}, e, \pi, h|S\rangle}{\langle O, \mathcal{S}, e, \pi, h|S'\rangle}$$

  where $S' = o(S)$ and $o = h(S, \pi, e) \in O$.

- *Parameterization*: The modification of the solver's parameters changes its configuration and can be used either to tune the solver before running it or to adjust its behavior during the run. A parameterization rule can be abstracted as:

  [Par] Parameterization

  $$\frac{\langle O, \mathcal{S}, e, \pi, h|S\rangle}{\langle O, \mathcal{S}, e, \pi', h|S\rangle}$$

- *Evaluation function modification*: Since we address here autonomous systems that are able to modify not only their configuration through their parameters but also their internal components, we have to consider more intricate rules. A first way to adapt the solver to changes is to modify its evaluation function, which directly induces changes on the search landscape. This is the case when changing weights or penalties in the evaluation function (there are many examples, for instance [KP98, PH06]).

  [EvalCh] Evaluation modification

  $$\frac{\langle O, \mathcal{S}, e, \pi, h|S\rangle}{\langle O, \mathcal{S}, e', \pi, h|S\rangle}$$

- *Operator modification*: Another way to modify the internal configuration of the solver is to change its set of operators. Note that operators can be added or discarded from the set O.

[OpCh] Operator modification

$$\frac{\langle O, \mathcal{S}, e, \pi, h|S\rangle}{\langle O', \mathcal{S}, e, \pi, h|S\rangle}$$

- *Strategy modification*: Similarly, solving strategies can be changed to manage differently the operators and achieve a different solving algorithm. As mentioned above, a backtracking algorithm can apply different strategy for enforcing local consistency at each node, or in hybrid solving one may switch from complete to approximate methods.

  [StratCh] Strategy modification

$$\frac{\langle O, \mathcal{S}, e, \pi, h|S\rangle}{\langle O, \mathcal{S}, e, \pi, h'|S\rangle}$$

- *Encoding modification*: We also have to take into account solvers that will be able to change their encoding during execution. As this is the case for the evaluation modification, such changes will affect the search landscape.

  [EncCh] Encoding modification

$$\frac{\langle O, \mathcal{S}, e, P, h|S\rangle}{\langle O, \mathcal{S}', e, P, h|S\rangle}$$

Note that applying one of these rules (except [Res]) will generally require applying other computation rules. For example, a change of encoding ([EncCh]) will certainly require a change of operators ([OpCh]), of evaluation function ([EvalCh]), of strategy ([StratCh]), and of parameterization ([Par]). However, a change of strategy does not always imply a change of operators.

Control of the Computation Rules and Solvers The most important part of our characterization concerns the control of the algorithm to finally build the solver. The control is used to act on the configuration of the solver through its parameters, but also to modify the internal components of the solver (parameters, operators, strategies, etc.).

Definition 7.10 (Control) Let $\mathcal{S}_{CS}$ be the set of all the finite sequences of elements of CS. A control function $K : \mathcal{S}_{CS} \rightarrow R$ is a function that selects a computation rule from the set R according to a sequence of computation states.

A solver state can be defined by a set of computation rules, and a sequence of computation states that have been previously computed.

Definition 7.11 (Solver) A solver is a pair (K, R) composed of a control function K and a set of computation rules R that will define a sequence of solver states.

A way of describing a solver is to use regular expressions which schedule computation rules to describe its control. Let's come back to the rules defined in Sect. 7.3.4. We consider the set of rules $R = Par \cup Res \cup EvalCh \cup EncCh \cup OpCh \cup StratCh$ where *Par* represents some parameterization rules [*Par*], *EvalCh* some evaluation modification rules [*EvalCh*], etc. Given two subsets R_1 and R_2 of R, R_1^* means that zero or more rules of R_1 are sequentially applied and $R_1 R_2$ means the sequential application of one rule of the subset R_1 is followed by the application of one rule of R_2. $R_1|R_2$ corresponds to use of one rule from R_1 or one from R_2. These notations will be used in the following section to highlight the characteristics of the solvers by means of the sequences of rules that they apply in their solving processes.

Definition 7.12 (Solver State) A solver state is a pair (R, Σ) where:

- R is a set of computation rules as defined above
- Σ is a sequence of computation states that are recorded during the solving process.

Starting from a solver state (R, Σ), with $\Sigma = (\sigma_0, \ldots, \sigma_n)$, the next state is obtained as (R, Σ') where $\exists r \in R$, such that $K(\Sigma) = r$ and $\Sigma' = (\sigma_0, \ldots, \sigma_n, \sigma_{n+1} = r(\sigma_n))$.

Note that in practice, a solver state does not contain the complete history. Thus, the sequence of computation states is either limited to a given length, or only the most relevant computation states are kept.

We now have:

$$\textbf{Solver} = \textbf{control} + \textbf{configured solving algorithms}$$

We recall that we stated before that Solving algorithm = solving strategy + solving operators. Coming back to Fig. 7.3 that shows a classification of hyper-heuristics, we notice that we obtain a similar distinction here: solvers correspond to the hyper-heuristics of Fig. 7.3, solving algorithms to heuristic search spaces, strategies to heuristic selection or generation, and operators to heuristic construction or perturbation. We can finally identify an autonomous solver:

Definition 7.13 (Autonomous Solver) Consider a solver given by a regular expression ex of computation rules from $R = Par \cup Solv \cup EvalCh \cup EncCh \cup OpCh \cup StratCh$. A solver is autonomous if ex contains at least a rule from $Par \cup EvalCh \cup EncCh \cup OpCh \cup StratCh$ (i.e., ex is not only composed of rules from *Solv*).

An autonomous solver is a solver that modifies its configuration during solving, using a control rule. Of course, there are various degrees in this autonomy scale. We can now come back to the previous taxonomy of offline/tuning and online/control (e.g., for parameters). Consider a solver given by a regular expression ex of computation rules from $R = Par \cup Solv \cup EvalCh \cup EncCh \cup OpCh \cup StratCh$, and the word w given by flattening this expression ex. The offline/tuning of a solver consists of the rules that appear in ex before the first *Solv* rule of ex. The online/control

is composed of all the rules that appear after the first rule *Solv* and that are not of the *Solv* family of rules.

In the next section we will illustrate how these rules are used in real solvers and how they can be used to characterize families of solvers within our autonomous search scope.

7.4 Case Studies

In this section, we will not attempt to present an exhaustive view of existing solvers but we will rather choose some representative solvers or algorithms in order to illustrate different solving approaches and how the previous computation rules can be used to characterize these approaches. As mentioned in the introduction, autonomous search has been indeed investigated for many years, across many different areas and under different names. Therefore, we could not imagine providing an exhaustive discussion of all approaches.

7.4.1 Tuning Before Solving

As in [EHM99, LLM07], we use the word tuning for the adjustment of the different components of the algorithm before trying to solve an instance (see end of Sect. 7.3.4).

Preprocessing Techniques

Even if preprocessing is not directly linked to the core of the solving mechanism but relies on external processes, we have to consider it as an important component in the design of modern solvers. Nowadays, efficient solvers (e.g. DPLL) use simplification preprocessing before trying to solve an instance (see for instance the SAT solver SatElite [EMS07]). Note that the model transformation can maintain equisatisfiability or a stronger equivalence property (the set of solutions is preserved).

Parameter Tuning on Preliminary Experiments

Such a tuning phase may consist in setting correct parameters in order to adjust the configuration of the solver. Here, these settings are performed according to a given set of preliminary experiments. Tuning before solving will correspond to the configuration of the solver and then its use for properly solving the problem. Therefore, the general profile of the solvers will be mainly described as:

$$[Config]Solv^*$$

where $[Config]$ is of the form $(Par|EvalCh|OpCh|EncCh)^*$.

Empirical Manual Tuning We include in this family the classic tuning task involved when using single metaheuristic based solvers where experiments are required to tune the various parameters [SE09, NSE08]. Of course there exist similar studies in constraint programming to choose the suitable variable and value heuristics, and this task is often not formalized. Most of the time, parameters are tuned independently since it appears difficult to control their mutual interaction without a sophisticated model. Here, the parameterization is not really part of the solver but rather a preliminary experimental process.

$$Solver: Solv^*$$

Determining the Size of a Tabu List Experiments or other previous analysis can be used to extract general parameters or heuristic settings. In the context of Tabu Search for SAT, [MSG97b] have used an extensive offline experimental step to determine the optimal length of a tabu list. They used simple regression to derive the length of the list according to the number of variables n. Remarkably, the length is independent of the size of the constraints, and their formula applies to any hard random k-SAT instance. Therefore the parameterization can be included as a first step of the solving process.

$$Solver: Par\_Solv^*$$

Automatic Parameter Tuning by an External Algorithm Recently, [HHLBS09] proposed an algorithm to search for the best parameters in the parameter space and therefore to automatically tune a solver. Now, if we consider that this automated process is included in the solver, we have then the following description.

$$Solver: (Solv^* Par)^* Solv^*$$

Note that $(Solv^* Par)^*$ corresponds to a series of runs and parameter tuning, which is achieved automatically.

Component Setting Before Solving

We consider here methods that consist in choosing the correct components of the solver by using experiments and/or external knowledge that has been acquired separately from the current solving task. This knowledge can be formulated as general rules, can use more or less sophisticated learning techniques, or may also use an external computation process.

A. Learning Solver's Components External mechanisms can be used before tuning to discover or learn efficient components for the solver.

Discovering Heuristics In [Fuk08], genetic programming is used to discover new, efficient variable selection heuristics for SAT solving with local search algorithms. Candidate variable selection heuristics are evaluated on a set of test instances. This automatic process can be inserted before solving (the variable selection heuristics can induce a change of parameters or operators depending on the description granularity). Note that here the first *Solv** is not applied to the problem at hand.

$$Solver\text{: } \big(Solv^*(OpCh|Par)\big)^* Solv^*$$

The choice of heuristics can be parameters of the operators in our formalism; heuristic discovering can be considered as the selection of suitable operators and their parameterization.

Learning Evaluation Functions In [BMK00], a new method is proposed in order to learn evaluation functions in local search algorithms and improve search efficiency based on previous runs.

$$Solver\text{: } (Solv^* EvalCh)^* Solv^*$$

B. Empirical Prediction of Instance Hardness The following techniques are based on a learning component (e.g., clustering tools), which can be used to detect automatically the suitable heuristics and strategies to apply.

Portfolio-Based In SATzilla [XHHLB08], offline linear basis function regression and classifiers are used on top of instance-based features to obtain models of SAT solvers runtime. During the exploitation phase, instance features are used to select the best algorithm from a portfolio of tree- and local search-based SAT solvers. We may also cite the works of Gebruers et al. [GGHM04] and Guerri et al. [GM04] that use case-based reasoning and learning technique from to choose the appropriate solving technique among constraint programming and integer linear programming. In these solver schemes, the first *Solv** corresponds again to preliminary experiments.

$$Solver\text{: } Solv^*(OpCh|StratCh|Par|EvalCh)^* Solv^*$$

Parameter-Based In [HH05, HHHLB06], the authors use an approach similar to SATzilla. They show that it is possible to predict the runtime of two stochastic local searches (SLSs). In this work, the selection of the best method to apply on a given instance is changed into the selection of the best parameters of a given SLS algorithm.

$$Solver\text{: } ParSolv^*$$

7.4.2 Control During Solving

The control of the solver's behavior during the run can be achieved by modifying its components and/or its parameters. This corresponds, for instance, to an online

adjustment of the parameters or heuristics. Such control can be achieved by means of supervised control schemes or self-adaptive rules. Of course, such approaches often rely on a learning process that tries to benefit from previously encountered problems during the search or even during the solving of other problems. Therefore, the profile of the solvers will generally be:

$$\left([Config]Solv^*\right)^*$$

where [*Config*] is of the form $(Par|EvalCh|OpCh|EncCh)^*$. Note that the outer * loop represents the control loop.

Controlling Encoding

[Han08] proposes an adaptive encoding in an evolutionary algorithm in order to solve continuous function optimization problems. The representation of the solutions are changed along the search to reach an optimal representation that could simplify the solving of the initial problem.

$$Solver: (EncChSolv^*)^*$$

Controlling Variable Orderings and Value Selection in Search Heuristics

We consider here approaches where the heuristic functions change during the search w.r.t. the current state and parameters.

Hybrid Approaches to Discover Efficient Variable Ordering To illustrate this kind of approach, we may mention the SAT solving technique of [MSG98] where a Tabu Search is used at each node of a DPLL to find the next variable to branch on.

$$Solver: \left((OpChStratCh)Solv^*ParSolv^*\right)^*$$

Continuous Search In [AHS10], the authors propose to exploit the result of an offline learning stage to select the best variable and value heuristics. They use a restart-based tree search algorithm and tune the previous heuristics at each new restart point. Moreover, this approach perpetually refines its learning stage by re-assessing its past choices in between successive calls to the search procedure. This approach is presented in Chap. 6.

$$Solver: (ParSolv)^*$$

Conflict Driven Heuristic In [BHLS04], important variables are deemed to be the ones linked to constraints that have frequently participated in dead ends. During

the search, this information is collected and used to order variables. Eventually, the system has enough knowledge to branch on important variables and quickly solve the problem. The system learns weights from conflicts that are used in the computation of the variable selection heuristics; this corresponds to an update of the parameters each time a conflict is met.

$$Solver: (ParSolv^*)^*$$

Variable Dependency-Based Heuristic In [AH09] and in Chap. 5, the constraint propagation engine is exploited to detect so called weak dependencies between variables. These correspond to situations when the instantiation of a given variable leads to the instantiation of others. These events are perceived as positive, and are used to rank the variables, favoring the ones whose branching on results in the largest number of instantiations. This heuristic is shown to outperform [BHLS04] on many domains.

$$Solver: (ParSolv^*)^*$$

Implicit Feedback Loops in Modern DPLL Solvers In modern SAT solvers like the one presented in [ES03a], many implicit feedback loops are used. For instance, the collection of conflicts feeds the variable selection heuristic, and the quality of unit propagation is sometimes used to control the restart strategy. Similarly, the deletion of learned clauses, which is necessary to preserve performances uses activity-based heuristics that can point to the clauses that were the least useful for the unit propagation engine. Therefore, it induces changes in the model itself and in the heuristic parameters.

$$Solver: \left((EncCh|Par)Solv^*\right)^*$$

Adapting Neighborhood During the Search Variable neighborhood search [MH97, HR06, PR08] consists in managing simultaneously several neighborhood functions and/or parameters (according to the description granularity) in order to benefit from various exploration/exploitation facilities.

$$Solver: \left((OpCh|Par)Solv\right)^*$$

Evolving Heuristics

Hyper-heuristics Hyper-heuristics [BKN+03] is a general approach that consists in managing several metaheuristic search methods from a higher strategy point of view. Therefore, it is closely related to autonomous search and has already been applied for many problems (e.g., SAT solving [BEDP08]). Since they switch from one solving technique to another, hyper-heuristics could be characterized by:

$$Solver: \left((OpCh|StratCh|Par|EvalCh)^*Solv^*\right)^*$$

Learning Combinations of Well-known Heuristics In the ACE project [EFW05], learning is used to define new domain-based weighted combinations of

branching heuristics (for variable and value selection). ACE learns the weights to apply through a voting mechanism. Each low-level heuristic votes for a particular element of the problem (variable, value). Weights are updated according to the nature of the run (successful or not). The learning is applied to a given class of problems. The combination is learned on a set of representative instances and used during the exploitation step. A similar approach has been used in [GJ08] in order to learn efficient reduction operators when solving numerical CSPs.

$$Solver: (ParSolv^*)^*$$

Controlling Evaluation Function

This aspect may concern local search algorithms that use, for instance, adaptive weighting of the constraints in their evaluation function [Mor93, Tho00]. Constraint weighting schemes solve the problem of local minima by adding weights to the cost of violated constraints. These weights increase the cost of violating a constraint and so change the shape of the cost surface w.r.t. the evaluation function. Note that these techniques are also widely used in SAT solvers [BHvMW09].

$$Solver: (EvalChSolv^*)^*$$

Parameter Control in Metaheuristic Algorithms

We consider here approaches that change the parameters during the search w.r.t. the current state and other parameters. Of course, these parameters have a direct influence on the heuristic functions, but these latter functions stay the same during the solving process.

Reactive Search In [BBM08] (formerly presented in [BBM07]), Battiti et al. propose a survey of so-called reactive search techniques, highlighting the relationship between machine learning and optimization processes. In reactive search, feedback mechanisms are able to modify the search parameters according to the efficiency of the search process. For instance, the balance between intensification and diversification can be automated by exploiting the recent past of the search process through dedicated learning techniques.

$$Solver: (ParSolv^*)^*$$

Adaptive Genetic Algorithms Adaptability is well known in evolutionary algorithm design. For instance, there are classical strategies to dynamically compute the usage probability of GA search operators [Thi05, WPS06a, WLLH03]. Given a set of search operators, an adaptive method has the task of setting the usage probability of each operator. When an operator is used, a reward is returned. Since the environment is non-stationary during evolution, an estimate of the expected reward for each operator is only reliable over a short period of time [WPS06b]. This is addressed by introducing a quality function, defined such that past rewards influence opera-

tor quality to an extent that decays exponentially with time. We may also mention other works that use more sophisticated evaluation functions, reward computation and operator probability adjustments in order to manage dynamically the application parameters of the EA [MFS+09, MS08, FDSS08].

$$Solver: (ParSolv^*)^*$$

7.4.3 Control During Solving in Parallel Search

The solvers described in this section also belong to the previous family of solvers that include control within their proper solving process. But here, due to the parallel/distributed architecture of solver, the sequence of computation rules is more difficult to schedule. Thus, the profile could be described as $([Config]|Solv^*)^*$.

Value Ordering in Portfolio-Based Distributed Search In [RH05] and in Chap. 2, the authors present portfolio-based distributed searches. The system allows the parallel execution of several agent-based distributed search. Each search requires the cooperation of a set of agents which coordinate their local decisions through message passing. An agent is a part of multiple distributed searches, and maintains the context of each one. Each agent can aggregate its context to dynamically rank the values of its local variables. The authors define several efficient portfolio-based value-ordering heuristics. For instance, one agent can pick up the value which is used most frequently in competing searches, or the one which is most supported in other searches, etc.

$$Solver: (Par|Solv^*)^*$$

Adaptive Load-Balancing Policies in Parallel Tree-Based Search Disolver is an advanced Constraint Programming library which particularly targets parallel search [Ham03]. This search engine is able to dynamically adapt its inter-processes knowledge sharing activities (load-balancing, bound sharing). In Disolver, the end user can define constraint-based knowledge sharing policies by adding new constraints. This second modeling can be linked to the constraint-based formulation of the problem to control the knowledge sharing according to the evolution of some problem components. For instance, the current value of the objective function can be used to draft answers to incoming load-balancing requests when the quality of the current subtree is perceived as good, etc. Interestingly, since the control of the knowledge sharing policies is made through classical constraints, it is automatically performed by the constraint propagation engine. We can see this as a dynamic adjustment of knowledge sharing activities, and customize it to model (learned clauses) and parameter (selection heuristics) changes.

$$Solver: \left((EncCh|Par)|Solv^*\right)^*$$

Control-Based Clause Sharing in Parallel SAT Solving Conflict driven clause learning, one of the most important components of modern DPLL, is crucial to the

performance of parallel SAT solvers. Indeed, this mechanism allows clause sharing between multiple processing units working on related (sub-)problems. However, without limitations, sharing clauses might lead to an exponential blow-up in communication or to the sharing of irrelevant clauses. In [HJS09a], the authors propose new innovative policies to dynamically select shared clauses in parallel solvers. The first policy controls the overall number of exchanged clauses whereas the second one additionally exploits the relevance or quality of the clauses. This dynamic adaptation mechanism allows us to reinforce/reduce the cooperation between different solvers which are working on the same SAT instance. This approach is fully described in Chap. 3.

$$Solver: (Par|Solv^*)^*$$

7.5 Summary

In this chapter, we have proposed a taxonomy of search processes w.r.t. their computation characteristics. To this end, we have presented the general basic concepts of a solver architecture: the basic components of a solver, and its configurations. We have then identified autonomous solvers as solvers that can control their solving process, either by self-adaptation (internal process) or by supervised adaptation (external process).

We have proposed a rule-based characterization of autonomous solvers: the idea is to formalize solver adaptations and modifications with some computation rules that describe solver transformation. Using our formalism, we could then classify, characterize, and identify the scope of autonomous search representative solvers by outlining their global mechanism.

Our description framework allows us to handle solving techniques:

- of various and different types: complete, incomplete, or hybrid
- based on different computation paradigms: sequential, distributed, or parallel
- dedicated to different problem families: CSP, SAT, optimization, etc.

This work was also an attempt to highlight the links and similarities between different communities that aim at building such autonomous solvers and that may benefit from more exchanges and more collaborative approaches (including constraint programming, SAT, machine learning, numerical optimization, clustering, etc.).

We have identified the notion of control in autonomous constraint solvers and two main techniques for achieving it: control by supervised adaptation and control by self-adaptation, depending on the level of interaction between the solver, its environment, and the control itself. These two control management approaches are indeed complementary. Moreover, they open new challenges for the design of more autonomous search systems that would run continuously, alternating (or combining, or executing in parallel) solving and self-improving phases. A first attempt in this direction has been presented in Chap. 6.

Chapter 8
Conclusion and Perspectives

Writing this book gave me the occasion to put my work into perspective and to reassess its homogeneity and consistency. Clearly, my work on distributed constraint satisfaction put me on the distributed system side very early. In that world, algorithms are more than monolithic sets of instructions and have value in their well-timed and controlled interactions.

I decided to exploit the richness of this setting to mitigate the risk of being wrong in a constructive search process, initially by adding parallelism to distributed search [Ham99, Ham02b], then as presented in Chap. 2 by organizing competition and cooperation between multiple distributed search strategies. Competition is rather straightforward to organize. On the other hand, cooperation opens a new space where the benefit of the knowledge exchanged has to be balanced against the cost of sharing knowledge. When information is shared, we have to consider the ramp-up time to prepare information, and the time it takes to effectively exchange the information. When information is not shared we have to consider that costly redundant work can occur, and that in divide-and-conquer systems task starvation can happen.

Therefore, controlling the way knowledge is shared and what knowledge is exchanged is crucial to the performance. In DisCSP settings, we managed to exploit agents' locality to share information between strategies. This allowed exchange at virtually no cost. Concerning the knowledge to share, we tried to be systematic by exploring policies based on diversification and emulation principles. In future work we think that it would be interesting to investigate how portfolios are best composed. Dynamic adaptation of portfolios looks also promising in the DisCSP context. Adaptation could provide more resources to the most promising efforts. Knowledge aggregation could be easily improved at no cost by adding extra information to existing message passing operations. This would give a better view of the distributed system, and could benefit from new aggregation methods

In the parallel SAT settings, complete solvers allow the exchange of conflict clauses. However, since they can generate millions of clauses during their effort, the exchange has to be well controlled. Technically, we decided to exploit lock-less data structures to maximize performance. Content-wise, we managed to develop new techniques to assess the quality of conflict clauses in an attempt to exchange

Y. Hamadi, *Combinatorial Search: From Algorithms to Systems*,
DOI 10.1007/978-3-642-41482-4_8, © Springer-Verlag Berlin Heidelberg 2013

meaningful information. We got inspired by control theory techniques to finely tune the exchanges.

Parallel portfolios exploit the stochasticity of modern SAT solvers, which are worth differentiating for better performance. More importantly, they benefit from a crucial property of these solvers: they do not need to exhaust a search space to definitely qualify an input as satisfiable or not. Portfolios have completely deprecated divide-and-conquer approaches, and nowadays the whole SAT community has adopted this architecture [Bie10, Kot10, SLB10].

We came up with the ManySAT architecture thanks to our early experience with distributed portfolios, and thanks to our experience with parallel divide-and-conquer in constraint programming [Ham03]. As future work, the question of the scalability of parallel SAT portfolios able to exchange conflict clauses has to be asked. Many attempts have been made to mix portfolios and divide-and-conquer approaches [MML10]; however, the results so far are not convincing.

To improve parallel local search techniques for SAT, we could only rely on heuristic hints. One good piece of information to exchange in this setting is the best configuration found so far. We used that as a basis to explore diversification and intensification strategies to find out that the latter was giving the best performance improvement.

Further work should investigate the use of additional information to exchange, for instance, tabu list, the age and score of a variable, information on local minima, etc. It should also consider the best way to integrate this extra knowledge in the course of a given algorithm. State-of-the-art local searches perform better when they do not restart. Incorporating extra information without forcing the algorithm to restart is likely to be important.

Dynamic variable ordering heuristics are key to the performance of constraint solvers. We showed how to heuristically discover a simplified form of functional dependency between variables called weak dependency. Once discovered, these relations are used to rank branching decisions. Our method shows that these relations can be detected with some acceptable overhead during constraint propagation. Experiments on a large set of problems show that, on the average, the search trees are reduced by a factor of 3 while runtime is decreased by one third.

Our heuristic learns from successes, allowing a quick exploitation of the solver's work. In a way, this is complementary to dom-wdeg which learns from failures. Moreover, both techniques rely on the computation of Mindom. Combining their respective strengths seems interesting.

When one cannot multiply search strategies to avoid being wrong, the selection of the right strategy is crucial. One way to avoid mistakes is to learn offline a predictive model which accurately matches instance features to good solver's parameters [HH05, HHHLB06]. This approach requires a good understanding of the application domain and a large set of representative instances. This last requirement can be dropped by streamlining the learning process across executions of the search procedure. Since the learning is performed on *real* instances, the model is more accurate. As a downside, such a system cannot give top performance with the first instances but can only gradually improve over time. Such a Continuous Search system was presented in Chap. 6.

Continuous computation addresses the issue of finding not the best (boundedly optimal) use of time in solving a given problem, but the best use of idle computational resources between bouts of problem solving. This approach broadens the definition of a problem to include not just individual instances, but the class of challenges that a given computational system is expected to face in its lifetime. Eventually, the end of the current search is just another event for the search system. As an effect, the priority of its long-lasting self-improving task is raised and the task comes to the foreground. That search is used to enrich the knowledge of the system and is eventually exploited during this new task.

We can envision a wide range of actions that can be taken over by the search algorithm while it is idle: analyzing the strategies that have succeeded and failed during the last runs; applying costly machine learning techniques in order to improve a supervised tuning method; using knowledge compilation techniques in order to compile new deduction rules, or new patterns that were detected in the recently solved problems and that can prove useful for future problems of the same application area; exchanging gained knowledge with similar AS systems, e.g., features-based prediction function.

In fact, such a continuous system would include a self-adaptive strategy during the solving process while it could switch to a supervised controller while waiting for another problem instance. This architecture would allow it to react dynamically to incoming events during solving and to exploit the knowledge acquired through its successive experiences.

The performance evaluation of a search system able to work in continuous search mode is also an important problem which is highly related to the arrival rate and to the quality of new problem instances. Here quality corresponds to how good the instances are for the system for gaining important knowledge on the whole problem class.

Finally, to capture our contributions in a unifying framework which will also embed related work as much as possible, we moved to the notion of Autonomous Search. We defined autonomous solvers as solvers that contain control in their search process, and studied such autonomous systems w.r.t. their specific control methods. A control process includes a strategy that manages the modification of some of the solver's components and behavioral features after the application of some solving functions. We gave a formalization of solver adaptation and modification with computation rules that describe solvers' component transformation.

An important issue is evaluating performance of Autonomous Search systems with respect to classical criteria, used in solver competitions, for instance. We think that the performance evaluation of an autonomous search may actually focus on three points: show that an autonomous search can (re)discover the best known or approximate a very good strategy for a specific problem; show the ability of an autonomous search to adapt itself to a changing environment, e.g., more or less computational resources; show that an autonomous search can adapt itself and converge to an efficient strategy for a class of problems.

There exists an optimal search strategy for a particular problem. However, determining such a strategy could require much more computational power than solving

the problem at hand. One possible way to assess the performance of AS systems is to run them on artificial problems where the optimal strategy is well known and to see if their adaptive mechanisms are able to build a strategy close to the optimal.

The efficiency of an AS system can also be measured as its ability to maintain the competitiveness of its search strategy in a changing environment. Here, the goal is more to assess the reaction time of the system under changing settings than the ultimate quality of the produced strategies.

A major challenge associated to AS is that classical tools for algorithm analysis typically provide weak support for understanding the performance of autonomous algorithms. This is because autonomous algorithms exhibit a complex behavior that is not often amenable to a worst-/average-case analysis. Instead, autonomous algorithms should be considered as full-fledged complex systems, and studied as such.

References

[ABCC07] D. Applegate, R. Bixby, V. Chvatal, W. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics (Princeton University Press, Princeton, 2007)

[ABH+08] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, L. Sais, A generalized framework for conflict analysis, in *Theory and Applications of Satisfiability Testing, SAT 2008*, ed. by H.K. Büning, X. Zhao. Lecture Notes in Computer Science, vol. 4996 (Springer, Berlin, 2008), pp. 21–27

[AD97] A. Armstrong, E. Durfee, Dynamic prioritization of complex agents in distributed constraint satisfaction problems, in *Proc. of the 15th Int. Joint Conf. on AI (IJCAI-97)* (1997), pp. 620–625

[AGKS00] D. Achlioptas, C.P. Gomes, H.A. Kautz, B. Selman, Generating satisfiable problem instances, in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, Austin, Texas, USA (AAAI Press/MIT Press, Menlo Park/Cambridge, 2000), pp. 256–261

[AH09] A. Arbelaez, Y. Hamadi, Exploiting weak dependencies in tree-based search, in *ACM Symposium on Applied Computing (SAC)*, Honolulu, Hawaii, USA (ACM, New York, 2009), pp. 1385–1391

[AH11] A. Arbelaez, Y. Hamadi, Improving parallel local search for SAT, in *Learning and Intelligent Optimization, 5th International Conference (LION'11)*, Rome, Italy, ed. by C.A.C. Coello. Lecture Notes in Computer Science, vol. 6683 (Springer, Berlin, 2011), pp. 46–60

[AHS09] A. Arbelaez, Y. Hamadi, M. Sebag, Online heuristic selection in constraint programming, in *International Symposium on Combinatorial Search*, Lake Arrowhead, USA, 2009

[AHS10] A. Arbelaez, Y. Hamadi, M. Sebag, Continuous search in constraint programming, in *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010* (IEEE Comput. Soc., Los Alamitos, 2010), pp. 53–60

[AKJ04] R. Akbani, S. Kwek, N. Japkowicz, Applying support vector machines to imbalanced datasets, in *ECML*, ed. by J.F. Boulicaut, F. Esposito, F. Giannotti, D. Pedreschi. Lecture Notes in Computer Science, vol. 3201 (Springer, Berlin, 2004), pp. 39–50

[AL03] E. Aarts, J.K. Lenstra (eds.), *Local Search in Combinatorial Optimization* (Princeton University Press, Princeton, 2003)

[ALMS09] G. Audemard, J.M. Lagniez, B. Mazure, L. Sais, Learning in local search, in *21st International Conference on Tools with Artificial Intelligence (ICTAI'09)*, Newark, New Jersey, USA (IEEE Comput. Soc., Los Alamitos, 2009), pp. 417–424

DOI 10.1007/978-3-642-41482-4,

[Apt03] K. Apt, *Principles of Constraint Programming* (Cambridge University Press, Cambridge, 2003)

[ARR02] R.M. Aiex, M.G.C. Resende, C.C. Ribeiro, Probability distribution of solution time in GRASP: an experimental investigation. J. Heuristics **8**(3), 343–373 (2002)

[AST09] C. Ansótegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, in *15th International Conference on Principles and Practice of Constraint Programming*, Lisbon, Portugal, ed. by I.P. Gent. LNCS, vol. 5732 (Springer, Berlin, 2009), pp. 142–157

[BB09] R. Battiti, M. Brunato, Reactive search optimization: learning while optimizing, in *Handbook of Metaheuristics*, 2nd edn., ed. by M. Gendreau, J.Y. Potvin (Springer, Berlin, 2009)

[BBM07] R. Battiti, M. Brunato, F. Mascia, Reactive search and intelligent optimization. Technical Report, Dipartimento di Informatica e Telecomunicazioni, Universita di Trento, Italy (2007)

[BBM08] R. Battiti, M. Brunato, F. Mascia, *Reactive Search and Intelligent Optimization*. Operations Research/Computer Science Interfaces, vol. 45 (Springer, Berlin, 2008)

[BBMM05] C. Bessiere, I. Brito, A. Maestre, P. Meseguer, Asynchronous backtracking without adding links: a new member in the ABT family. Artif. Intell. **161**, 7–24 (2005)

[BEDP08] M. Bader-El-Den, R. Poli, Generating SAT local-search heuristics using a GP hyper-heuristic framework, artificial evolution, in *8th International Conference, Evolution Artificielle, EA 2007*. Revised Selected Papers. Lecture Notes in Computer Science, vol. 4926 (Springer, Berlin, 2008), pp. 37–49

[BG06] F. Benhamou, L. Granvilliers, Continuous and interval constraints, in *Handbook of Constraint Programming*, ed. by F. Rossi, P. van Beek, T. Walsh (Elsevier, Amsterdam, 2006). Chapter 16

[BGS99] L. Brisoux, E. Grégoire, L. Sais, Improving backtrack search for SAT by means of redundancy, in *Foundations of Intelligent Systems, 11th International Symposium, ISMIS'99*. Lecture Notes in Computer Science, vol. 1609 (Springer, Berlin, 1999), pp. 301–309

[BHK+09a] E.K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, R. Qu, A survey of hyper-heuristics. Technical Report NOTTCS-TR-SUB-0906241418-2747, School of Computer Science and Information Technology, University of Nottingham (2009)

[BHK+09b] E.K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, J. Woodward, A classification of hyper-heuristics approaches, in *Handbook of Metaheuristics*, 2nd edn., ed. by M. Gendreau, J.Y. Potvin (Springer, Berlin, 2009)

[BHLS04] F. Boussemart, F. Hemery, C. Lecoutre, L. Sais, Boosting systematic search by weighting constraints, in *Proceedings of the 16th European Conference on Artificial Intelligence*, Valencia, Spain, ed. by R.L. de Mántaras, L. Saitta (IOS Press, Amsterdam, 2004), pp. 146–150

[BHS09] L. Bordeaux, Y. Hamadi, H. Samulowitz, Experiments with massively parallel constraint solving, in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, ed. by C. Boutilier (2009), pp. 443–448

[BHvMW09] A. Biere, M.J.H. Heule, H. van Maaren, T. Walsh (eds.), *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185 (IOS Press, Amsterdam, 2009)

[BHZ06] L. Bordeaux, Y. Hamadi, L. Zhang, Propositional satisfiability and constraint programming: a comparative survey. ACM Comput. Surv. **9**(2), 135–196 (2006)

[Bie08] A. Biere, Adaptive restart strategies for conflict driven SAT solvers, in *Theory and Applications of Satisfiability Testing—Proceedings of the 11th International Conference, SAT 2008*, ed. by H.K. Büning, X. Zhao. Lecture Notes in Computer Science, vol. 4996 (Springer, Berlin, 2008), pp. 28–33

[Bie10] A. Biere, Lingeling, plingeling, PicoSAT and PrecoSAT at SAT race 2010. Technical Report 10/1, FMV Reports Series (2010)

[BKN+03] E.K. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, S. Schulenburg, Hyper-heuristics: an emerging direction in modern search technology, in *Handbook of Metaheuristics*, ed. by F. Glover, G. Kochenberger (Kluwer Academic, Dordrecht, 2003), pp. 457–474

[BMK00] J. Boyan, A. Moore, P. Kaelbling, Learning evaluation functions to improve optimization by local search. J. Mach. Learn. Res. **1**, 77–112 (2000)

[Bou09] C. Boutilier (ed.), *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, Pasadena, California, USA, July 11–17 (2009)

[BR96] C. Bessiere, J.C. Regin, MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems, in *CP* (1996), pp. 61–75

[Bre79] D. Brelaz, New methods to color the vertices of a graph. Commun. ACM **22**, 251–256 (1979)

[BS96] M. Böhm, E. Speckenmeyer, A fast parallel SAT-solver—efficient workload balancing. Ann. Math. Artif. Intell. **17**(3–4), 381–400 (1996)

[BSK03] W. Blochinger, C. Sinz, W. Küchlin, Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Comput. **29**(7), 969–994 (2003)

[BSPV02] M. Birattari, T. Stützle, L. Paquete, K. Varrentrapp, A racing algorithm for configuring metaheuristics, in *GECCO'02: Proceedings of the Genetic and Evolutionary Computation Conference* (Morgan Kaufmann, San Mateo, 2002), pp. 11–18

[BT94] R. Battiti, G. Tecchiolli, The reactive tabu search. INFORMS J. Comput. **6**(2), 126–140 (1994)

[Buc06] B.G. Buchanan, What do we know about knowledge? AI Mag. **27**(4), 35–46 (2006)

[BZ08] H.K. Büning, X. Zhao (eds.), *Theory and Applications of Satisfiability Testing—Proceedings of the 11th International Conference, SAT 2008*, Guangzhou, China, May 12–15, 2008. Lecture Notes in Computer Science, vol. 4996 (Springer, Berlin, 2008)

[CB01] J. Crispim, J. Brandão, Reactive tabu search and variable neighbourhood descent applied to the vehicle routing problem with backhauls, in *Proceedings of the 4th Metaheuristics International Conference, Porto, MIC 2001* (2001), pp. 631–636

[CB04] T. Carchrae, J.C. Beck, Low-knowledge algorithm control, in *AAAI*, ed. by D.L. McGuinness, G. Ferguson (AAAI Press/MIT Press, Menlo Park/Cambridge, 2004), pp. 49–54

[CB05] T. Carchrae, J.C. Beck, Applying machine learning to low-knowledge control of optimization algorithms. Comput. Intell. **21**(4), 372–387 (2005)

[CB08] M. Correira, P. Barahona, On the efficiency of impact based heuristics, in *14th International Conference on Principles and Practice of Constraint Programming*, Sydney, Australia, ed. by P.J. Stuckey. LNCS, vol. 5202 (Springer, Berlin, 2008), pp. 608–612

[CBH+07] F. Corblin, L. Bordeaux, Y. Hamadi, E. Fanchon, L. Trilling, A SAT-based approach to decipher gene regulatory networks, in *Integrative Post-Genomics, RIAMS, Lyon* (2007)

[CGTT63] W. Crowston, F. Glover, G. Thompson, J. Trawick, Probabilistic and parametric learning combinations of local job shop scheduling rules. Technical Report, ONR research memorandum no. 117, GSIA, Carnegie-Mellon University, Pittsburgh (1963)

[CI96] B. Cha, K. Iwama, Adding new clauses for faster local search, in *13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference (AAAI'06/IAAI'06)*, Portland, Oregon, USA, ed. by W.J. Clancey, D.S. Weld, vol. 1 (AAAI Press/MIT Press, Menlo Park/Cambridge, 1996), pp. 332–337

[CIR12] R. Celso, R. Isabel, V. Reinaldo, Exploiting run time distributions to compare sequential and parallel stochastic local search algorithms. J. Glob. Optim. **54**, 405–429 (2012)

[CKS02] P. Cowling, G. Kendall, E. Soubeiga, Hyperheuristics: a tool for rapid prototyping in scheduling and optimisation, in *Applications of Evolutionary Computing, EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN*. Lecture Notes in Computer Science, vol. 2279 (Springer, Berlin, 2002), pp. 1–10

[CL85] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)

[CS00] P. Cowling, E. Soubeiga, Neighborhood structures for personnel scheduling: a summit meeting scheduling problem (abstract), in *Proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling*, Constance, Germany, ed. by E.K. Burke, W. Erben (2000)

[CS08] G. Chu, P.J. Stuckey, Pminisat: a parallelization of Minisat 2.0. Technical Report, Sat-race 2008, solver description (2008)

[CST00] N. Christianini, J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods* (Cambridge University Press, Cambridge, 2000)

[CT10] T.G. Crainic, M. Toulouse, Parallel meta-heuristics, in *Handbook of Metaheuristics*, ed. by M. Gendreau, J.-Y. Potvin. International Series in Operations Research & Management Science, vol. 146 (Springer, Berlin, 2010), pp. 497–541

[CW06] W. Chrabakh, R. Wolski, GridSAT: a system for solving satisfiability problems using a computational grid. Parallel Comput. **32**(9), 660–687 (2006)

[DBL09] *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2009*, Trondheim, Norway, 18–21 May 2009 (IEEE Press, New York, 2009)

[DBL10] *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010*, Arras, France, 27–29 October 2010, vol. 1, (IEEE Comput. Soc., Los Alamitos, 2010)

[DD01] O. Dubois, G. Dequen, A backbone-search heuristic for efficient solving of hard 3-SAT formulae, in *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'01* (2001), pp. 248–253

[Dec03] R. Dechter, *Constraint Processing* (Elsevier/Morgan Kaufmann, Amsterdam/San Mateo, 2003)

[DGS07] B.N. Dilkina, C.P. Gomes, A. Sabharwal, Tradeoffs in the complexity of backdoor detection, in *CP'07*, ed. by C. Bessiere. LNCS, vol. 4741 (Springer, Berlin, 2007), pp. 256–270

[DLL62] M. Davis, G. Logemann, D.W. Loveland, A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)

[dMB08] L.M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in *TACAS*, ed. by C.R. Ramakrishnan, J. Rehof. Lecture Notes in Computer Science, vol. 4963 (Springer, Berlin, 2008), pp. 337–340

[EB05] N. Eén, A. Biere, Effective preprocessing in SAT through variable and clause elimination, in *Theory and Applications of Satisfiability Testing, SAT 2005*, ed. by F. Bacchus, T. Walsh. Lecture Notes in Computer Science, vol. 3569 (Springer, Berlin, 2005), pp. 61–75

[EFW+02] S. Epstein, E. Freuder, R. Wallace, A. Morozov, B. Samuels, The adaptive constraint engine, in *Principles and Practice of Constraint Programming—CP 2002, 8th International Conference*. Lecture Notes in Computer Science, vol. 2470 (Springer, Berlin, 2002), pp. 525–542

[EFW05] S. Epstein, E. Freuder, R. Wallace, Learning to support constraint programmers. Comput. Intell. **21**(4), 336–371 (2005)

[EHM99] A.E. Eiben, R. Hinterding, Z. Michalewicz, Parameter control in evolutionary algorithms. IEEE Trans. Evol. Comput. **3**(2), 124–141 (1999)

[EMS07] N. Eén, A. Mishchenko, N. Sörensson, Applying logic synthesis for speeding up SAT, in *Theory and Applications of Satisfiability Testing—SAT 2007*. Lecture Notes in Computer Science, vol. 4501 (Springer, Berlin, 2007), pp. 272–286

[ES03a] N. Eén, N. Sörensson, An extensible SAT-solver, in *6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, Santa Margherita Ligure, Italy, ed. by E. Giunchiglia, A. Tacchella. Lecture Notes in Computer Science, vol. 2919 (Springer, Berlin, 2003), pp. 502–518

[ES03b] A. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*. Natural Computing Series (Springer, Berlin, 2003)

[FA03] T. Frühwirth, S. Abdennadher, *Essentials of Constraint Programming* (Springer, Berlin, 2003)

[FD94] D. Frost, R. Dechter, In search of the best constraint satisfaction search, in *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94* (1994), pp. 301–306

[FDSS08] A. Fialho, L. Da Costa, M. Schoenauer, M. Sebag, Extreme value based adaptive operator selection, in *Parallel Problem Solving from Nature—PPSN X, 10th International Conference*, ed. by G. Rudolph et al. Lecture Notes in Computer Science, vol. 5199 (Springer, Berlin, 2008), pp. 175–184

[FM02] S. Fitzpatrick, L. Meertens, Scalable, anytime constraint optimization through iterated, peer-to-peer interaction in sparsely-connected networks, in *Proc. IDPT'02* (2002)

[FR04] H. Fang, W. Ruml, Complete local search for propositional satisfiability, in *9th National Conference on Artificial Intelligence (AAAI'04)*, San Jose, California, USA, ed. by D.L. McGuinness, G. Ferguson (AAAI Press/MIT Press, Menlo Park/Cambridge, 2004), pp. 161–166

[FS02] S.L. Forman, A.M. Segre, Nagsat: a randomized, complete, parallel solver for 3-SAT, in *Proceedings of Theory and Applications of Satisfiability Testing, SAT'02* (2002), pp. 236–243

[FT63] H. Fisher, L. Thompson, Probabilistic learning combinations of local job-shop scheduling rules, in *Industrial Scheduling*, (Prentice Hall, New York, 1963)

[Fuk08] A. Fukunaga, Automated discovery of local search heuristics for satisfiability testing. Evol. Comput. **16**(1), 31–61 (2008)

[Gec06] Gecode Team, *Gecode: GenE. Constraint development environment* (2006). Available from http://www.gecode.org

[GFS08] L. Gil, P. Flores, L.M. Silveira, PMSat: a parallel version of minisat. J. Satisf. Boolean Model. Comput. **6**, 71–98 (2008)

[GGHM04] C. Gebruers, A. Guerri, B. Hnich, M. Milano, Making choices using structure at the instance level within a case based reasoning framework, in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR*. Lecture Notes in Computer Science, vol. 3011 (Springer, Berlin, 2004), pp. 380–386

[GGS07] R. Greenstadt, B.J. Grosz, M.D. Smith, SSDPOP: improving the privacy of DCOP with secret sharing, in *AAMAS* (2007), p. 171

[GHJS10] L. Guo, Y. Hamadi, S. Jabbour, L. Sais, Diversification and intensification in parallel SAT solving, in *CP*, ed. by D. Cohen. Lecture Notes in Computer Science, vol. 6308 (Springer, Berlin, 2010), pp. 252–265

[GJ08] F. Goualard, C. Jermann, A reinforcement learning approach to interval constraint propagation. Constraints **13**(1–2), 206–226 (2008)

[GK03] F. Glover, G. Kochenberger, *Handbook of Metaheuristics*. International Series in Operations Research & Management Science (Springer, Berlin, 2003)

[GL97] F. Glover, M. Laguna, *Tabu Search* (Kluwer Academic, Dordrecht, 1997)

[GM04] A. Guerri, M. Milano, Learning techniques for automatic algorithm portfolio selection, in *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004* (IOS Press, Amsterdam, 2004), pp. 475–479

[Gol89] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning* (Addison-Wesley, Reading, 1989)

[Gom03] C. Gomes, Randomized backtrack search, in *Constraint and Integer Programming: Toward a Unified Methodology*, ed. by M. Milano (Kluwer Academic, Dordrecht, 2003), pp. 233–283

[GS97] C.P. Gomes, B. Selman, Algorithm portfolio design: theory vs. practice, in *Proc. UAI'97* (1997), pp. 190–197

[GS01] C.P. Gomes, B. Selman, Algorithm portfolios. Artif. Intell. **126**, 43–62 (2001)

[GS07] S. Gelly, D. Silver, Combining online and offline knowledge in UCT, in *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, Corvallis, Oregon, USA, ed. by Z. Ghahramani. ACM International Conference Proceeding Series, vol. 227 (ACM, New York, 2007), pp. 273–280

[GS08] M. Gagliolo, J. Schmidhuber, Algorithm selection as a bandit problem with unbounded losses. Technical Report, IDSIA-07-08 (2008)

[GSCK00] C. Gomes, B. Selman, N. Crato, H. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reason. **24**(1/2), 67–100 (2000)

[GSK98] C. Gomes, B. Selman, H. Kautz, Boosting combinatorial search through randomization, in *AAAI/IAAI* (1998), pp. 431–437

[GW] I. Gent, T. Walsh, CSPLib, a problem library for constraints

[GZG+08] A. Gershman, R. Zivan, T. Grinshpoun, A. Grubstein, A. Meisels, Measuring distributed constraint optimization algorithms, in *AAMAS DCR* (2008)

[Ham99] Y. Hamadi, Traitement des problèmes de satisfaction de contraintes distribués. PhD thesis, Université Montpellier II (1999) (in French)

[Ham02a] Y. Hamadi, Distributed, interleaved, parallel and cooperative search in constraint satisfaction networks. Technical Report HPL-2002-21, HP laboratories (2002)

[Ham02b] Y. Hamadi, Interleaved backtracking in distributed constraint networks. Int. J. Artif. Intell. Tools **11**(2), 167–188 (2002)

[Ham03] Y. Hamadi, Disolver: a distributed constraint solver. Technical Report MSR-TR-2003-91, Microsoft Research (2003)

[Han08] N. Hansen, Adaptive encoding: how to render search coordinate system invariant, in *Parallel Problem Solving from Nature—PPSN X, 10th International Conference*. Lecture Notes in Computer Science, vol. 5199 (Springer, Berlin, 2008), pp. 204–214

[HBQ98] Y. Hamadi, C. Bessière, J. Quinqueton, Distributed intelligent backtracking, in *ECAI* (1998), pp. 219–223

[HE79] R.M. Haralick, G.L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, in *IJCAI*, San Francisco, CA, USA (1979), pp. 356–364

[Hen89] P.V. Hentenryck, *Constraint Satisfaction in Logic Programming* (MIT Press, Cambridge, 1989)

[HH93] T. Hogg, B.A. Huberman, Better than the best: the power of cooperation, in *1992 Lectures in Complex Systems*. SFI Studies in the Sciences of Complexity, vol. V (Addison-Wesley, Reading, 1993), pp. 165–184

[HH05] F. Hutter, Y. Hamadi, Parameter adjustment based on performance prediction: towards an instance-aware problem solver. Technical Report MSR-TR-2005-125, Microsoft Research, Cambridge, UK (2005)

[HHHLB06] F. Hutter, Y. Hamadi, H. Hoos, K. Leyton-Brown, Performance prediction and automated tuning of randomized and parametric algorithms, in *CP*, ed. by F. Benhamou. Lecture Notes in Computer Science, vol. 4204 (Springer, Berlin, 2006), pp. 213–228

[HHLB10] F. Hutter, H. Hoos, K. Leyton-Brown, Tradeoffs in the empirical evaluation of competing algorithm designs. Ann. Math. Artif. Intell. **60**(1–2), 65–89 (2010). Special Issue on Learning and Intelligent Optimization

[HHLBS09] F. Hutter, H. Hoos, K. Leyton-Brown, T. Stützle, ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**, 267–306 (2009)

[Hil75] B. Hill, A simple general approach to inference about the tail of a distribution. Ann. Stat., 1163–1174 (1975)

[HJPS11] Y. Hamadi, S. Jabbour, C. Piette, L. Sais, Deterministic parallel DPLL. J. Satisf. Boolean Model. Comput. **7**(4), 127–132 (2011)

[HJS08] Y. Hamadi, S. Jabbour, L. Sais, ManySAT: solver description. Technical Report MSR-TR-2008-83, Microsoft Research (2008)

[HJS09a] Y. Hamadi, S. Jabbour, L. Sais, Control-based clause sharing in parallel SAT solving, in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, ed. by C. Boutilier (2009), pp. 499–504

[HJS09b] Y. Hamadi, S. Jabbour, L. Sais, ManySAT: a parallel SAT solver. J. Satisf. Boolean Model. Comput. **6**(4), 245–262 (2009)

[HM05] P.V. Hentenryck, L. Michel, *Constraint-Based Local Search* (MIT Press, Cambridge, 2005)

[HMS08a] Y. Hamadi, E. Monfroy, F. Saubion, Special Issue on Autonomous Search. Constr. Program. Lett. **4** (2008)

[HMS08b] Y. Hamadi, E. Monfroy, F. Saubion, What is autonomous search? Technical Report MSR-TR-2008-80, Microsoft Research (2008)

[HMSW11] Y. Hamadi, J. Marques-Silva, C.M. Wintersteiger, Lazy decomposition for distributed decision procedures, in *PDMC*, ed. by J. Barnat, K. Heljanko. EPTCS, vol. 72 (2011), pp. 43–54

[Hol75] J. Holland, *Adaptation in Natural and Artificial Systems* (University of Michigan Press, Ann Arbor, 1975)

[Hoo98] H. Hoos, Stochastic local search—methods, models, applications. PhD thesis, Darmstadt University of Technology, Germany (1998)

[Hoo99a] H. Hoos, On the run-time behaviour of stochastic local search algorithms for SAT, in *16th National Conference on Artificial Intelligence and 8th Conference on Innovative Applications of Artificial Intelligence (AAAI'99/IAAI'99)*, Orlando, Florida, USA, ed. by J. Hendler, D. Subramanian (AAAI Press/MIT Press, Menlo Park/Cambridge, 1999), pp. 661–666

[Hoo99b] H. Hoos, SAT-encodings, search space structure, and local search performance, in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99* (Morgan Kaufmann, San Mateo, 1999), pp. 296–303

[Hoo02a] H. Hoos, An adaptive noise mechanism for WalkSAT, in *AAAI/IAAI* (2002), pp. 655–660

[Hoo02b] H. Hoos, An adaptive noise mechanism for WalkSAT, in *8th National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence (AAAI'02/IAAI'02)*, Edmonton, Alberta, Canada, ed. by R. Dechter, R.S. Sutton (AAAI Press/MIT Press, Menlo Park/Cambridge, 2002), pp. 655–660

[HR06] B. Hu, G. Raidl, Variable neighborhood descent with self-adaptive neighborhood-ordering, in *Proc. of the 7th EU Meeting on Adaptive, Self-Adaptive and Multilevel Metaheuristics* (2006)

[HR11] Y. Hamadi, G. Ringwelski, Boosting distributed constraint satisfaction. J. Heuristics **17**(3), 251–279 (2011)

[HSe12] Y. Hamadi, F. Saubion, E. Monfroy (eds.), *Autonomous Search* (Springer, Berlin, 2012)

[HT07] H. Hoos, D.A.D. Tompkins, Adaptive Novelty+, in *Solver Description, SAT Competition* (2007)

[HTH02] F. Hutter, D.A.D. Tompkins, H. Hoos, Scaling and probabilistic smoothing: efficient dynamic local search for SAT, in *8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, Ithaca, NY, USA, ed. by P. Van Hentenryck. Lecture Notes in Computer Science, vol. 2470 (Springer, Berlin, 2002), pp. 233–248

[Hua07] J. Huang, The effect of restarts on the efficiency of clause learning, in *IJCAI*, ed. by M.M. Veloso (2007), pp. 2318–2323

[Hut09] F. Hutter, Automating the configuration of algorithms for solving hard computational problems. PhD thesis, Department of Computer Science, University of British Columbia (2009)

[HvM06] M.J.H. Heule, H. van Maaren, March dl: adding adaptive heuristics and a new branching strategy. J. Satisf. Boolean Model. Comput. **2**, 47–59 (2006)

[HW93] T. Hogg, C.P. Williams, Solving the really hard problems with cooperative search, in *11th National Conference on Artificial Intelligence (AAAI'93)*, Washington, DC, USA, ed. by W.G. Lehnert, R. Fikes (AAAI Press/MIT Press, Menlo Park/Cambridge, 1993), pp. 231–236

[HW09] S. Haim, T. Walsh, Restart strategy selection using machine learning techniques, in *12th International Conference on Theory and Applications of Satisfiability Testing*, Swansea, UK, ed. by O. Kullmann. LNCS, vol. 5584 (Springer, Berlin, 2009), pp. 312–325

[HW12] Y. Hamadi, C.M. Wintersteiger, Seven challenges in parallel SAT solving, in *AAAI*, ed. by J. Hoffmann, B. Selman (AAAI Press, Menlo Park, 2012)

[HW13] Y. Hamadi, C.M. Wintersteiger, Seven challenges in parallel SAT solving. AI Mag. **34**(2) (2013)

[Ing89] L. Ingber, Very fast simulated re-annealing. Math. Comput. Model. **12**(8), 967–973 (1989)

[Jac88] V. Jacobson, Congestion avoidance and control, in *SIGCOMM* (1988), pp. 314–329

[JLU05] B. Jurkowiak, C.M. Li, G. Utard, A parallelization scheme based on work stealing for a class of SAT solvers. J. Autom. Reason. **34**(1), 73–101 (2005)

[JM91] C. Janikow, Z. Michalewicz, An experimental comparison of binary and floating point representations in genetic algorithms, in *Fourth International Conference on Genetic Algorithms* (1991), pp. 31–36

[Jon06] K.D. Jong, *Evolutionary Computation: A Unified Approach* (MIT Press, Cambridge, 2006)

[KHR+02] H.A. Kautz, E. Horvitz, Y. Ruan, C.P. Gomes, B. Selman, Dynamic restart policies, in *AAAI/IAAI* (2002), pp. 674–681

[Kje91] G. Kjellstroem, On the efficiency of Gaussian adaptation. J. Optim. Theory Appl. **71**(3) (1991)

[Kot10] S. Kottler, SArTagnan: solver description. Technical Report, SAT race 2010 (2010)

[KP98] S. Kazarlis, V. Petridis, Varying fitness functions in genetic algorithms: studying the rate of increase of the dynamic penalty terms, in *Parallel Problem Solving from Nature—PPSN V, 5th International Conference*. Lecture Notes in Computer Science, vol. 1498 (1998), pp. 211–220

[Kra08] O. Kramer, *Self-adaptive Heuristics for Evolutionary Computation* (Springer, Berlin, 2008)

[KSGS09] L. Kroc, A. Sabharwal, C.P. Gomes, B. Selman, Integrating systematic and local search paradigms: a new strategy for MaxSAT, in *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, Pasadena, California, ed. by C. Boutilier (2009), pp. 544–551

[LA97] C.M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'97* (1997), pp. 366–371

[Lam78] L. Lamport, Time, clocks and the ordering of events in distributed systems. Commun. ACM **2**, 95–104 (1978)

[LB08] H. Larochelle, Y. Bengio, Classification using discriminative restricted Boltzmann machines, in *Proceedings of the Twenty-Fifth International Conference on Machine Learning*, Helsinki, Finland, ed. by W.W. Cohen, A. McCallum, S.T. Roweis. ACM International Conference Proceeding Series, vol. 307 (ACM, New York, 2008), pp. 536–543

[LBNA+03] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, Y. Shoham, A portfolio approach to algorithm selection, in *Proc. IJCAI'03* (2003), p. 1542

[LH05] C.M. Li, W.Q. Huang, Diversification and determinism in local search for satisfiability, in *8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, St Andrews, UK, ed. by F. Bacchus, T. Walsh. Lecture Notes in Computer Science, vol. 3569 (Springer, Berlin, 2005), pp. 158–172

[LLM07] F. Lobo, C. Lima, Z. Michalewicz (eds.), *Parameter Setting in Evolutionary Algorithms*. Studies in Computational Intelligence, vol. 54 (Springer, Berlin, 2007)

[LS07] M.Z. Lagerkvist, C. Schulte, Advisors for incremental propagation, in *13th International Conference on Principles and Practice of Constraint Programming*, Providence, RI, USA, ed. by C. Bessiere. LNCS (Springer, Berlin, 2007), pp. 409–422

[LSB07] M. Lewis, T. Schubert, B. Becker, Multithreaded SAT solving, in *12th Asia and South Pacific Design Automation Conference* (2007)

[LSZ93] M. Luby, A. Sinclair, D. Zuckerman, Optimal speedup of Las Vegas algorithms. Inf. Process. Lett. **47**, 173–180 (1993)

[LWZ07] C.M. Li, W. Wei, H. Zhang, Combining adaptive noise and look-ahead in local search, in *10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, Lisbon, Portugal, ed. by J. Marques-Silva, K.A. Sakallah. Lecture Notes in Computer Science, vol. 4501 (Springer, Berlin, 2007), pp. 121–133

[MFS+09] J. Maturana, A. Fialho, F. Saubion, M. Schoenauer, M. Sebag, Extreme compass and dynamic multi-armed bandits for adaptive operator selection, in *IEEE Congress on Evolutionary Computation* (IEEE Press, New York, 2009), pp. 365–372

[MH97] N. Mladenovic, P. Hansen, Variable neighborhood search. Comput. Oper. Res. **24**(11), 1097–1100 (1997)

[Mic92] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Artificial Intelligence (Springer, Berlin, 1992)

[MML10] R. Martins, V.M. Manquinho, I. Lynce, Improving search space splitting for parallel SAT solving, in *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010* (IEEE Comput. Soc., Los Alamitos, 2010), pp. 336–343

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in *38th Design Automation Conference (DAC'01)*, Las Vegas, NV, USA (ACM, New York, 2001), pp. 530–535

[Mor93] P. Morris, The breakout method for escaping from local minima, in *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI)* (AAAI Press, Menlo Park, 1993), pp. 40–45

[MS98] K. Marriott, P. Stuckey, *Programming with Constraints: An Introduction* (MIT Press, Cambridge, 1998)

[MS08] J. Maturana, F. Saubion, A compass to guide genetic algorithms, in *Parallel Problem Solving from Nature—PPSN X, 10th International Conference*, ed. by G. Rudolph et al. Lecture Notes in Computer Science, vol. 5199 (Springer, Berlin, 2008), pp. 256–265

[MSG97a] B. Mazure, L. Sais, E. Grégoire, Tabu search for SAT, in *14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI'97/IAAI'97)*, ed. by B. Kuipers, B.L. Webber (AAAI Press/MIT Press, Menlo Park/Cambridge, 1997), pp. 281–285

[MSG97b] B. Mazure, L. Sais, E. Grégoire, Tabu search for SAT, in *AAAI/IAAI* (1997), pp. 281–285

[MSG98] B. Mazure, L. Sais, E. Grégoire, Boosting complete techniques thanks to local search methods. Ann. Math. Artif. Intell. **22**(3–4), 319–331 (1998)

[MSK97] D.A. McAllester, B. Selman, H.A. Kautz, Evidence for invariants in local search, in *14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI'97/IAAI'97)*, Providence, Rhode Island, USA, ed. by B. Kuipers, B.L. Webber (AAAI Press/MIT Press, Menlo Park/Cambridge, 1997), pp. 321–326

[MSL04] E. Monfroy, F. Saubion, T. Lambert, On hybridization of local search and constraint propagation, in *Logic Programming, 20th International Conference, ICLP 2004*. Lecture Notes in Computer Science, vol. 3132 (Springer, Berlin, 2004), pp. 299–313

[MSS96] J. Marques-Silva, K.A. Sakallah, GRASP—a new search algorithm for satisfiability, in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design* (1996), pp. 220–227

[MSTY05] P.J. Modi, W.M. Shen, M. Tambe, M. Yokoo, ADOPT: asynchronous distributed constraint optimization with quality guarantees. Artif. Intell. **161** (2005)

[NE06] V. Nannen, A.E. Eiben, A method for parameter calibration and relevance estimation in evolutionary algorithms, in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2006* (ACM, New York, 2006), pp. 183–190

[NE07] V. Nannen, A.E. Eiben, Relevance estimation and value calibration of evolutionary algorithm parameters, in *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence* (2007), pp. 975–980

[NOT06] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(t). J. ACM **53**(6), 937–977 (2006)

[NSE08] V. Nannen, S. Smit, A. Eiben, Costs and benefits of tuning parameters of evolutionary algorithms, in *Parallel Problem Solving from Nature—PPSN X, 10th International Conference*. Lecture Notes in Computer Science, vol. 5199 (Springer, Berlin, 2008), pp. 528–538

[OGD06] L. Otten, M. Grönkvist, D.P. Dubhashi, Randomization in constraint programming for airline planning, in *CP* (2006), pp. 406–420

[OGMS02] R. Ostrowski, E. Grégoire, B. Mazure, L. Sais, Recovering and exploiting structural knowledge from CNF formulas, in *CP*, ed. by P. Van Hentenryck. Lecture Notes in Computer Science, vol. 2470 (Springer, Berlin, 2002), pp. 185–199

[OHH+08] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, B. O'Sullivan, Using case-based reasoning in an algorithm portfolio for constraint solving, in *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science* (2008)

[PD07] K. Pipatsrisawat, A. Darwiche, A lightweight component caching scheme for satisfiability solvers, in *Theory and Applications of Satisfiability Testing SAT 2007*, ed. by J. Marques-Silva, K.A. Sakallah. Lecture Notes in Computer Science, vol. 4501 (Springer, Berlin, 2007), pp. 294–299

[PG09] D.N. Pham, C. Gretton, gNovelty+ (v.2), in *Solver Description, SAT Competition* (2009)

[PH06] W.J. Pullan, H. Hoos, Dynamic local search for the maximum clique problem. J. Artif. Intell. Res. **25**, 159–185 (2006)

[PK01] D. Patterson, H. Kautz, Auto-Walksat: a self-tuning implementation of Walksat. Electron. Notes Discrete Math. **9**, 360–368 (2001)

[PL06] S.D. Prestwich, I. Lynce, Local search for unsatisfiability, in *9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, Seattle, WA, USA, ed. by A. Biere, C.P. Gomes. Lecture Notes in Computer Science, vol. 4121 (Springer, Berlin, 2006), pp. 283–296

[PPR96] P.M. Pardalos, L.S. Pitsoulis, M.G.C. Resende, A parallel GRASP for MAX-SAT problems, in *Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, Lyngby, Denmark, ed. by J. Wasniewski, J. Dongarra, K. Madsen, D. Olesen. Lecture Notes in Computer Science, vol. 1184 (Springer, Berlin, 1996), pp. 575–585

[PR08] J. Puchinger, G. Raidl, Bringing order into the neighborhoods: relaxation guided variable neighborhood search. J. Heuristics **14**(5), 457–472 (2008)

[PTGS08] D.N. Pham, J. Thornton, C. Gretton, A. Sattar, Combining adaptive and dynamic local search for satisfiability. J. Satisf. Boolean Model. Comput. **4**(2–4), 149–172 (2008)

[RBB05] A. Roli, M.J. Blesa, C. Blum, Random walk and parallelism in local search, in *Metaheuristic International Conference (MIC'05)*, Vienna, Austria (2005)

[RBea05] I. Rish, M. Brodie, S. Ma et al., Adaptive diagnosis in distributed systems. IEEE Trans. Neural Netw. **16**, 1088–1109 (2005)

[Ref04] P. Refalo, Impact-based search strategies for constraint programming, in *10th International Conference on Principles and Practice of Constraint Programming*, Toronto, Canada, ed. by M. Wallace. LNCS, vol. 2004 (Springer, Berlin, 2004), pp. 557–571

[RH05] G. Ringwelski, Y. Hamadi, Boosting distributed constraint satisfaction, in *CP*, ed. by P. van Beek. Lecture Notes in Computer Science, vol. 3709 (Springer, Berlin, 2005), pp. 549–562

[Ric75] J.R. Rice, The algorithm selection problem. Technical Report CSD-TR 152, Computer Science Department, Purdue University (1975)

[Ric76] J.R. Rice, The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976)

[Rin11] J. Rintanen, Heuristics for planning with SAT and expressive action definitions, in *ICAPS*, ed. by F. Bacchus, C. Domshlak, S. Edelkamp, M. Helmert (AAAI Press, Menlo Park, 2011)

[Rol02] A. Roli, Criticality and parallelism in structured SAT instances, in *8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, Ithaca, NY, USA, ed. by P. Van Hentenryck. Lecture Notes in Computer Science, vol. 2470 (Springer, Berlin, 2002), pp. 714–719

[RS08] V. Ryvchin, O. Strichman, Local restarts, in *Theory and Applications of Satisfiability Testing—Proceedings of the 11th International Conference, SAT 2008*, ed. by H.K. Büning, X. Zhao. Lecture Notes in Computer Science, vol. 4996 (Springer, Berlin, 2008), pp. 271–276

[SC06] C. Schulte, M. Carlsson, Finite domain constraint programming systems, in *Handbook of Constraint Programming*, ed. by F. Rossi, P. van Beek, T. Walsh. Foundations of Artificial Intelligence (Elsevier, Amsterdam, 2006), pp. 495–526. Chapter 14

[SE09] S.K. Smit, A.E. Eiben, Comparing parameter tuning methods for evolutionary algorithms, in *IEEE Congress on Evolutionary Computation* (IEEE Press, New York, 2009), pp. 399–406

[SF94] D. Sabin, E.C. Freuder, Contradicting conventional wisdom in constraint satisfaction, in *ECAI* (1994), pp. 125–129

[SF05] M.C. Silaghi, B. Faltings, Asynchronous aggregation and consistency in distributed constraint satisfaction. Artif. Intell. **161** (2005)

[SGS07] M. Streeter, D. Golovin, S.F. Smith, Combining multiple heuristics online, in *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, Vancouver, British Columbia, Canada (AAAI Press, Menlo Park 2007), pp. 1197–1203

[SK93] B. Selman, H.A. Kautz, Domain-independent extensions to GSAT: solving large structured satisfiability problems, in *13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, Chambéry, France, ed. by R. Bajcsy (Morgan Kaufmann, San Mateo, 1993), pp. 290–295

[SKC94a] B. Selman, H. Kautz, B. Cohen, Noise strategies for improving local search, in *AAAI* (1994), pp. 337–343

[SKC94b] B. Selman, H.A. Kautz, B. Cohen, Noise strategies for improving local search, in *12th National Conference on Artificial Intelligence (AAAI'94)*, Seattle, WA, USA, vol. 1, ed. by B. Hayes-Roth, R.E. Korf (AAAI Press/MIT Press, Menlo Park/Cambridge, 1994), pp. 337–343

[SLB10] T. Schubert, M. Lewis, B. Becker, Antom: solver description. Technical Report, SAT race (2010)

[SLM92] B. Selman, H.J. Levesque, D.G. Mitchell, A new method for solving hard satisfiability problems, in *10th National Conference on Artificial Intelligence (AAAI'92)*,

San Jose, CA, USA, ed. by W.R. Swartout (AAAI Press/MIT Press, Menlo Park/Cambridge, 1992), pp. 440–446

[SLS+08] M. Silaghi, R. Lass, E. Sultanik, W. Regli, T. Matsui, M. Yokoo, The operation point units of distributed constraint solvers, in *AAMAS DCR* (2008)

[SM07] H. Samulowitz, R. Memisevic, Learning to solve QBF, in *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, Vancouver, British Columbia (AAAI Press, Menlo Park 2007)

[SM08] K. Smith-Miles, Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput. Surv. **41**(1), 1–25 (2008)

[SSHF00] M.C. Silaghi, D. Sam-Haroud, B. Faltings, Asynchronous search with aggregations, in *Proc. AAAI/IAAI 2000* (2000), pp. 917–922

[Syw89] G. Sywerda, Uniform crossover in genetic algorithms, in *Proceedings of the Third International Conference on Genetic Algorithms*, San Francisco, CA, USA (Morgan Kaufmann, San Mateo, 1989), pp. 2–9

[TH04] D.A.D. Tompkins, H. Hoos, UBCSAT: an implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT, in *7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, ed. by H. Hoos, D.G. Mitchell. Lecture Notes in Computer Science, vol. 3542 (Springer, Berlin, 2004), pp. 306–320

[Thi05] D. Thierens, An adaptive pursuit strategy for allocating operator probabilities, in *Proc. GECCO'05*, ed. by H.-G. Beyer (ACM, New York, 2005), pp. 1539–1546

[Thi07] D. Thierens, Adaptive strategies for operator allocation, in *Parameter Setting in Evolutionary Algorithms*, ed. by F.G. Lobo, C.F. Lima, Z. Michalewicz (Springer, Berlin, 2007), pp. 77–90

[Tho00] J. Thornton, Constraint weighting for constraint satisfaction. PhD thesis, School of Computing and Information Technology, Griffith University, Brisbane, Australia (2000)

[TPBF04] J. Thornton, D.N. Pham, S. Bain, V. Ferreira Jr., Additive versus multiplicative clause weighting for SAT, in *9th National Conference on Artificial Intelligence and 16th Conference on Innovative Applications of Artificial Intelligence (AAAI'04/IAAI'04)*, San Jose, California, USA, ed. by D.L. McGuinness, G. Ferguson (AAAI Press/MIT Press, Menlo Park/Cambridge, 2004), pp. 191–196

[Tsa93] E. Tsang, *Foundations of Constraint Satisfaction*, 1st edn. (Academic Press, San Diego, 1993)

[Vap95] V. Vapnik, *The Nature of Statistical Learning* (Springer, New York, 1995)

[Wal00] T. Walsh, SAT v CSP, in *Proc. of CP 2000*. Lecture Notes in Computer Science, vol. 1894 (Springer, Berlin, 2000), pp. 441–456

[WF05] I.H. Witten, E. Frank, *Data Mining—Practical Machine Learning Tools and Techniques* (Elsevier, Amsterdam, 2005)

[WGS03] R. Williams, C.P. Gomes, B. Selman, Backdoors to typical case complexity, in *IJCAI* (2003), pp. 1173–1178

[WHdM09] C.M. Wintersteiger, Y. Hamadi, L.M. de Moura, A concurrent portfolio approach to SMT solving, in *CAV*, ed. by A. Bouajjani, O. Maler. Lecture Notes in Computer Science, vol. 5643 (Springer, Berlin, 2009), pp. 715–720

[WL09] W. Wei, C.M. Li, Switching between two adaptive noise mechanisms in local search for SAT, in *Solver Description, SAT Competition* (2009)

[WLLH03] Y.Y. Wong, K.H. Lee, K.S. Leung, C.W. Ho, A novel approach in parameter adaptation and diversity maintenance for GAs. Soft Comput. **7**(8), 506–515 (2003)

[WLZ08] W. Wei, C.M. Li, H. Zhang, A switching criterion for intensification and diversification in local search for SAT. J. Satisf. Boolean Model. Comput. **4**(2–4), 219–237 (2008)

[WPS06a] J. Whitacre, Q.T. Pham, R. Sarker, Credit assignment in adaptive evolutionary algorithms, in *Genetic and Evolutionary Computation Conference, GECCO 2006* (ACM, New York, 2006), pp. 1353–1360

[WPS06b] J. Whitacre, T. Pham, R. Sarker, Use of statistical outlier detection method in adaptive evolutionary algorithms, in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* (ACM, New York, 2006), pp. 1345–1352

[WvB08] H. Wu, P. van Beek, Portfolios with deadlines for backtracking search. Int. J. Artif. Intell. Tools **17**(5), 835–856 (2008)

[XHHLB07] L. Xu, F. Hutter, H. Hoos, K. Leyton-Brown, SATzilla-07: the design and analysis of an algorithm portfolio for SAT, in *Principles and Practice of Constraint Programming—CP 2007* (2007)

[XHHLB08] L. Xu, F. Hutter, H. Hoos, K. Leyton-Brown, SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008)

[YD98] M. Yokoo, E.H. Durfee, The distributed constraint satisfaction problem: formalization and algorithms. IEEE Trans. Knowl. Data Eng. **10**(5) (1998)

[YDIK92] M. Yokoo, E.H. Durfee, T. Ishida, K. Kuwabara, Distributed constraint satisfaction for formalizing distributed problem solving, in *Proc. ICDCS'92* (1992), pp. 614–621

[YG04] B. Yuan, M. Gallagher, Statistical racing techniques for improved empirical evaluation of evolutionary algorithms, in *Parallel Problem Solving from Nature—PPSN VIII, 8th International Conference*, ed. by X. Yao et al. Lecture Notes in Computer Science, vol. 3242 (Springer, Berlin, 2004), pp. 172–181

[YG05] F.Y.-H. Yeh, M. Gallagher, An empirical study of Hoeffding racing for model selection in k-nearest neighbor classification, in *IDEAL*, ed. by M. Gallagher, J. Hogan, F. Maire. Lecture Notes in Computer Science, vol. 3578 (Springer, Berlin, 2005), pp. 220–227

[YG07] B. Yuan, M. Gallagher, Combining meta-EAs and racing for difficult EA parameter tuning tasks, in *Parameter Setting in Evolutionary Algorithms*, ed. by F. Lobo, C. Lima, Z. Michalewicz. Studies in Computational Intelligence, vol. 54 (Springer, Berlin, 2007), pp. 121–142

[ZBH96] H. Zhang, M.P. Bonacina, J. Hsiang, PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. **21**(4), 543–560 (1996)

[ZHZ02] W. Zhang, Z. Huang, J. Zhang, Parallel execution of stochastic search procedures on reduced SAT instances, in *Pacific Rim International Conferences on Artificial Intelligence (PRICAI'02)*, Tokyo, Japan, ed. by M. Ishizuka, A. Sattar. Lecture Notes in Computer Science, vol. 2417 (Springer, Berlin, 2002), pp. 108–117

[ZM03] R. Zivan, A. Meisels, Synchronous vs asynchronous search on DisCSPs, in *Proc. EUMAS'03* (2003)

[ZM05] R. Zivan, A. Meisels, Concurrent search for distributed CSPs. Artif. Intell. **161** (2005)

[ZMMM01] L. Zhang, C.F. Madigan, M.W. Moskewicz, S. Malik, Efficient conflict driven learning in boolean satisfiability solver, in *ICCAD* (2001), pp. 279–285

[ZS94] H. Zhang, M.E. Stickel, Implementing the Davis-Putnam algorithm by tries. Technical Report, Artificial Intelligence Center, SRI International, Menlo (1994)