Search in Constraint Programming

Gilles Pesant (Polytechnique Montréal),
Meinolf Sellmann (Brown University)

November 29, 2009 – December 4, 2009

1 Overview of the Field

Constraint Programming (CP) is a powerful technique to solve combinatorial problems. It applies sophisticated inference to reduce the search space and a combination of variable- and value-selection heuristics to guide the exploration of that search space. Like Integer Programming, one states a model of the problem at hand in mathematical language and also builds a search tree through problem decomposition. But there are mostly important differences, among them: CP works directly on discrete variables instead of relying mostly on a continuous relaxation of the model; the modeling language offers many high-level primitives representing common combinatorial substructures of a problem — often a few constraints are sufficient to express complex problems; each of these primitives, called *constraints*, may have its own specific algorithm to help solve the problem; one does not branch on fractional variables but instead on indeterminate variables, which currently can take several possible values (variables are not necessarily fixed to a particular value at a node of the search tree); even though CP can solve optimization problems, it is primarily designed to handle feasibility problems.

We believe that a more principled and autonomous approach for search efficiency has to be started in Constraint Programming. Our main global objective is to advance in a significant way research on search automatization in CP so that combinatorial problems can be solved in a much more robust way. By furthering the automation of search for combinatorial problem solving, this workshop should have a direct impact on the range and size of industrial problems that we will be able to solve with this approach.

2 Open Problems

Here are some of the open problems identified by the participants, some of which were investigated in smaller groups (see Section 4):

1. Can reinforcement learning applied to complete search lead to superior search performance?
   (a) What is a good reward function for CSP/COP?
   (b) Branching dependent on expected satisfiability? More generally, how can a statistical prediction of an instance's objective function and/or feasibility value be useful in a solver?
   (c) How can we measure progress in a complete search trajectory?
   (d) How should we evaluate the performance of (new) algorithms?
2. Can we make no-good learning work for CP?

3. Is there a correlation between trajectory-based intensification and diversification measures and the performance of Local Search solvers?
4. What well-defined, macroscopic properties of an algorithm's behaviour can effectively explain performance differences between (existing) solvers / on different types of landscapes?
5. How can we formulate the notions of intensification and diversification to achieve 3 and 4?
6. What is the conceptual difference/unification between local and "non-local" (systematic) search?
7. Can we achieve an improvement in the state of the art by using automated algorithm configuration in a large design space of CP solvers?
8. What are useful/practical design spaces for modelling in CP, MIP?

3 Recent Developments

In order to familiarize participants with the range of techniques currently considered to improve search, several hour-long background talks were given followed by a discussion period, taking place during the first two days of the workshop.

3.1 Statistical Measures for Local Search—Summary of Talk (M. Sellmann)

In the realm of local search, there exist a number of interesting techniques to boost expected search performance by learning. We argue that any heuristic search bias that is introduced for the solution of a particular problem must be justified by a rigorous statistical analysis of characteristic problem instances.

We reviewed two statistical measures that have been considered in the literature, fitness-distance correlation and so-called global valley structure. Fitness-distance correlation is defined by the correlation of solutions' objective function value and their "distance" to the nearest global optimum. Fitness-distance correlation is low when on average the better the quality of a solution, the lower the distance to the nearest optimum. Distance can be measured in the number of local search steps (and therefore depends on the neighborhood used) or, more practically, as Hamming distance. Jones and Forrest found that, in general, low fitness-distance correlation correlates with good performance of genetic algorithms. We argued that low fitness-distance correlation gives a statistical justification for intensifying the search in the neighborhood of previously visited high-quality solutions.

The second measure that we reviewed is the so-called "global valley structure." The name conjures a questionable mental image. What it formally denotes is an anti-correlation of the quality of local minima and the average distance to other local minima. That is, we speak of a "global valley structure" when local minima have the better objective values the closer they are, on average, to other local minima. Boese found such a "global valley structure" on TSP instances by analyzing 2500 random local minima. We argued that "global valley structure" can justify a bias towards searching for improving solutions between previously found local minima. This is, e.g., the case when performing path relinking.

Now, each search bias that is introduced needs to be balanced with a counter-force that ensures effective search space exploration. This in combination with the idea to intensify the search around high quality solutions and to search between local minima leads to a local search method which we introduced in 2009, Dialectic Search. In dialectic search, we employ a greedy improvement heuristic to find a local optimum (the "thesis"). Then, we randomly perturbate the local minimum and locally improve the perturbation (the "anti-thesis"). We then search the space between those two local minima (for example by performing path relinking). If the best solution found during this search (the "synthesis") is better than the thesis, we locally optimize it and continue from this solution. Otherwise, we perform a new random perturbation of the thesis to obtain a new anti-thesis. Dialectic search conservatively continues its search with the best found solution (the current thesis) until no improvement is achieved for some time. Only then does the method move to the anti-thesis and continues the search from there.

Experimental results on set covering problems, magic squares problems, and continuous optimization problems show that this meta-heuristic, which is by design conservative and combines ideas from other methods, such as iterated local search, path relinking, variable neighborhood search, and genetic algorithms, can effectively lead the design to highly efficient optimization algorithms that beat the existing state-of-the-art by one or more orders of magnitude in performance. What is currently missing is an evaluation of the performance of dialectic search and its components and the statistical measures mentioned in the beginning.

3.2 Lessons from MIP Search—Summary of Talk (J. N. Hooker)

Mixed integer programming (MIP) offers several ideas that can benefit search in general. Chief among these is the use of a strong relaxation, normally a continuous linear programming (LP) relaxation, to guide the search.

For more than 50 years, the solution method of choice for general-purpose MIP solvers has been *branch-and-bound* search . At any point in the search there is a partial search tree, in which some of the leaf nodes are open and some are closed. The LP relaxation is solved at a selected open node, whereupon the search closes the node or branches to creates two more open nodes—depending on whether the LP solution is integral or better than the incumbent integer solution. The search branches on a selected variable with a fractional value in the LP solution. *Branch-and-cut* methods add cutting planes at various nodes to strengthen the LP relaxation.

The two main search decisions are *variable selection* (which variable on which to branch next) and *node selection* (which open node to explore next). Two methods for variable selection use pseudo-costs and strong branching, both of which rely on the LP relaxation. *Pseudo-costs* are estimates of the effect on the objective function value of rounding a fractional variable. *Strong branching* uses the dual simplex method to calculate a bound on this effect. The two most common node selection heuristics are depth-first and best bound. Depth first uses minimal memory and allows fast LP updates, but the tree can explode. Best bound opens the node with the best LP relaxation value.

Feasibility heuristics find integer solutions. *Local branching* uses a strategic branch to create a large-neighborhood local search at the left branch. It uses MIP to search the large neighborhood of a known integer solution. The process repeats at the right branch. The *feasibility pump* alternately rounds the current LP solution and then finds the LP solution that is closest to the rounded solution. This is an LP problem because the L1 norm is used.

At least two lessons can be learned from MIP for search in general. One is the use of duality. The LP dual can be generalized as an *inference dual*. Lagrangean, surrogate, subadditive, and many other duals are inference duals. The solution of an inference dual is a proof. The premises that are active in the proof can be used to construct a *nogood* or Benders cut that directs the search away from poor solutions. This method appears in OR as Benders decomposition, in SAT solvers as clause learning, and in AI as (partial-order) dynamic backtracking. *Logic-based Benders decomposition*, a generalization, has been used in a number of problems areas, often with order of magnitude improvement in solution time. Recently, at least one MIP solver has incorporated nogoods, returning the idea full circle to OR.

A second lesson from MIP is that a richer and stronger relaxation makes it worthwhile to invest more processing time at each node of the search tree. By contrast, constraint programming (CP) solvers use a weak relaxation (the domain store) and consequently prefers to search very large trees with little processing at the nodes. *Binary* (or *multivalued*) *decision diagrams* (BDDs) are an alternative to the constraint store that propagates more information from one constraint to the next. They provide a relaxation whose strength depends on the specified maximum width (and becomes equivalent to the domain store for a max width of one). Constraint-specific BDD refinement algorithms subsume the filtering algorithms currently used in CP. Preliminary computational testing shows that BDDs can reduce the search tree for multiple all-different constraints from a million nodes to one node, with more than a hundredfold reduction in computation time. In testing with multiple among constraints in nurse scheduling problems, they reduce the search tree by a factor of 10,000 and the computation time by a factor of fifty.

MIP search methods are surveyed in [9] and cutting planes in [10]. Local branching is described in [3] and the feasibility pump in [3]. Inference duality and applications are described in [7, 8], and Benders cuts for MIP in [11]. An introduction to the use of BDDSs in constraint programming can be found in [1], with further applications in [4, 5, 6].

3.3 Impact-Based and Counting-Based Search—Summary of Talk (G. Pesant)

To solve difficult problems, successful constraint programming requires both effective inference and search. After many years of advances on inference, there has been a more recent focus on search heuristics. This talk presented two proposals from the literature that take a more global view than individual variable domains.

Impact-Based Search Inspired by *pseudo-costs* in Integer Programming, Impact-Based Search (IBS) [21] aims to provide an efficient general-purpose search strategy for Constraint Programming. It learns from the observation of search space reduction during search, approximated as the change in the domains of the variables. The impact of assigning value d to variable x is computed roughly as the ratio of the size of the search space after that assignment to its size beforehand. The impact of variable x sums the individual impacts of the values in its domain. IBS typically branches on the variable with the greatest impact and then on its value with the smallest impact.

Computing such impacts for every variable at each search tree node requires probing and is time consuming. Typically one computes them at the root of the search tree and then occasionally at other nodes on a small subset of the variables and to break ties among heuristic choices. Justification for this is based on the belief that impacts do not change much from one search tree node to another but supporting evidence is not well documented in the scientific literature.

Nevertheless IBS generally performs very well and forms the basis of the default search heuristic in commercial CP solvers (e.g. IBM ILOG CP Optimizer).

Counting-Based Search Constraints have played a central role in CP because they capture key substructures of a problem and efficiently exploit them to boost inference. Counting-Based Search (CBS) proposes to do the same thing for search [22]. Up to now, the only visible effect of the consistency algorithms had been on the domains, projecting the set of tuples on each of the variables. Additional information about the number of solutions of a constraint (its *solution count*) can help a search heuristic to focus on critical parts of a problem or on promising solution fragments. At a more fine-grained level, the proportion of solutions to a given constraint in which variable x is assigned value d, termed the *solution density* of that variable-value pair, turns out to be a powerful indicator of the likelyhood that this assignment appears in a global solution.

For each family of constraints, providing a counting algorithm is sometimes trivial (e.g. `element` constraint), sometimes a simple extension of the filtering algorithm (e.g. `regular` constraint), and sometimes much harder to do exactly (e.g. `alldifferent` constraint). In the latter case, counting is only estimated through sampling, bounds, or relaxation.

Given the counting information, many different search heuristics can be considered. Despite trying several sophisticated ways to combine that information from each constraint, one of the conceptually simplest heuristics — branching on the highest solution density among every constraint, variable, and value (termed *maxSD*) — works just as well and often better.

Such a heuristic compares very well with the state of the art. Based on a varied set of benchmark problems, it has proved more consistent and often more efficient to solve satisfiable instances. More surprising, there is some empirical evidence that it can solve unsatisfiable instances very fast as well (i.e. provide a very small failed search tree).
This fairly recent line of research raises some questions. Among them:

- Why aren't more sophisticated ways of aggregating the counting information much better than the simpler *maxSD*?
- Why is a heuristic guiding the search toward satisfiable parts of the search tree also good to prove infeasibility?
- How can this approach be adapted to solve optimization problems?

3.4 Probabilistic Inference Techniques—Summary of Talk (E. Hsu)

In the last few years, the constraint satisfaction research community has begun to adapt a number of techniques that are normally associated with probabilistic reasoning over graphical models [25, 23, 24, 26, 28].

Interestingly, though, many aspects of these techniques can be be viewed in terms of soft analogues of existing constraint satisfaction concepts, for instance conditioning and inference. This is unsurprising given a long history of constraint satisfaction research dealing with graph structure–probabilistic distributions modeled as Bayes Networks [29] and Markov Random Fields [30] are continuous cousins to the discrete sets of solutions represented as constraint problems.

In particular, probabilistic models and constraint problems each express an otherwise intractable function over configurations of variables as a product of simpler functions. Using the factor graph formalism, and the algebraic notion of a semi-ring, then, we can make this comparison explicit. A *factor graph* [27] is a bipartite graph containing two types of nodes: *variables* and *factors* (i.e., *functions*). Associated with each such node is a variable or a function over variables, respectively; usually we will not need to distinguish between variables/factors and their associated nodes in the graph. The edges in a graph are defined unambiguously by the scopes of the functions. Thus, we can typically denote a factor graph $G = (X, F)$ solely in terms of its variables X and factors F. A *configuration* is an assignment to all the variables in X; the exponentially-sized set of all such configurations is denoted $\textsc{configs}(X)$. The semantics of a factor graph are such as to map each configuration of its variables to a real value; the graph's structure shows how this value factorizes into a product of individual instantiations. In particular, a factor graph G defined as above realizes a function W_G over configurations $\vec{x} \in \textsc{configs}(X)$:

$$W_G(\vec{x}) = \prod_{f_a \in F} f_a(\vec{x}_{|\sigma_a}) \tag{1}$$

Here "$|\sigma_a$" represents the projection of a configuration onto the domain of function a. In other words, to calculate the value of a configuration, we can just perform several function evaluations and multiply the results together. In its role as a probabilistic graphical model, a factor graph represents the joint probability distribution over X, whose contents are considered random variables. Thus W_G should be interpreted as issuing weights over configurations; such weights must then be normalized to form a proper probability distribution:

$$p(\vec{x}) = \frac{1}{\mathcal{N}} W_G(\vec{x}), \quad \text{where} \quad \mathcal{N} = \sum_{\vec{x} \in \text{CONFIGS}(X)} W_G(\vec{x}) \tag{2}$$

$\mathcal{N}$ is known as the normalizing constant or partition function (of G); its calculation is equivalent in structure and complexity to the marginal computation problem stated in the next section. Alternatively, the most basic insight of the student's research is that a factor graph can also be used to represent a constraint satisfaction problem:

A constraint satisfaction problem can be represented as a factor graph whose factors are all $0/1$ functions. That is, all factors evaluate to either 0, expressing the violation of a constraint, or to 1, expressing the satisfaction of a constraint. The weight function W_G for such a factor graph thus equals one if and only if a configuration meets all of the constraints, indicating the satisfaction of the problem as a whole. The normalizing constant $\mathcal{N}$ represents the number of solutions for such problems–accordingly the completed Equation 2 represents a uniform distribution over the set of solutions. (For unsatisfiable problems both the equation and the concept of a distribution over solutions are undefined.)

The term "algebraic semi-ring" is used to indicate the essential difference between probabilistic and combinatorial reasoning: in the former, sum and product are realized by standard arithmetic addition and multiplication, while in the latter they are realized by disjunction and conjunction. But by the simple definition given directly above, any constraint problem can be directly encoded in terms of addition/multiplication, as a uniform distribution over its set of solutions. Thus, we can ask standard queries defined over such distributions, and for a starting point we can consider standard probabilistic algorithms defined over graphical representations of such distributions.

3.5 Computer-Aided Algorithm Design: Principled Procedures for Building Better Solvers —Summary of Talk (H. Hoos)

High-performance algorithms can be found at the heart of many software systems; they often provide the key to effectively solving the computationally difficult problems encountered in the application areas in which these systems are deployed. Examples of such problems include scheduling, time-tabling, resource allocation, production planning and optimisation, computer-aided design and software verification.

Many of these problems are $\mathcal{NP}$-hard and considered computationally intractable. Nevertheless, these 'intractable' problems arise in practice, and finding good solutions to them in many cases tends to become more difficult as economic constraints tighten.

In most (if not all) cases, the key to solving computationally challenging problems effectively lies in the use of heuristic algorithms, that is, algorithms that make use of heuristic mechanisms, whose effectiveness can be demonstrated empirically, yet remains inaccessible to the analytical techniques used for proving theoretical complexity results. (We note that our use of the term 'heuristic algorithm' includes methods without provable performance guarantees as well as methods that have provable performance guarantees, but nevertheless make use of heuristic mechanisms; in the latter case, the use of heuristic mechanisms often results in empirical performance far better than the bounds guaranteed by rigorous theoretical analysis.) The design of such effective heuristic algorithms is a difficult task that requires considerable expertise and experience.

High-performance heuristic algorithms are typically constructed in an iterative, manual process in which the designer gradually introduces or modifies components or mechanisms whose performance is then tested by empirical evaluation on one or more sets of benchmark problems. During this iterative design process, the algorithm designer has to make many decisions. These concern choices of the heuristic mechanisms being integrated and the details of these mechanisms, as well as implementation details, such as data structures. Some of these choices take the form of parameters, whose values are guessed or determined based on limited experimentation.

This traditional approach for designing high-performance algorithms has various shortcomings. While it can and often does lead to satisfactory results, it tends to be tedious and labour-intensive; furthermore, the resulting algorithms are often unnecessarily complicated while failing to realise the full performance potential present in the space of designs that can be built using the same underlying set of components and mechanisms.

As an alternative to the traditional, manual algorithm design process, we advocate an approach that uses fully formalised procedures, implemented in software, to permit a human designer to explore large design spaces more effectively, with the aim of realising algorithms with desirable performance characteristics. This approach automates both, the construction of target algorithms as well as the assessment of their performance. Computer-aided algorithm design allows human designers to focus on the creative task of specifying a design space in terms of potentially useful components. This design space is then explored using powerful heuristic search and optimisation procedures in combination with significant amounts of computing power, with the aim of finding algorithms that perform well on given sets or distributions of input instances.

Our approach shares much of its motivation with existing work on automated parameter tuning, algorithm configuration, algorithm portfolios and algorithm selection, all of which can be seen as special cases of computer-aided algorithm design. Using this approach, we have achieved substantial improvements in the state of the art in solving a broad range of challenging combinatorial problems, ranging from SAT and mixed integer programming to course timetabling and protein structure prediction problems.

As a design approach, computer aided algorithm design is also more principled than the *ad-hoc* methods currently used, which makes it easier to disseminate and support, in the form of software systems for computer-aided algorithm design. Because of their more formalised nature, computer-aided algorithm design methods are also easier to evaluate and to improve. Their development and application therefore constitutes a key step in transforming the design of high-performance algorithm from a craft that is based primarily on experience and intuition to an engineering effort involving formalised procedures and best practices.

Drawing from methology and insights from a number of areas, including artificial intelligence, empirical algorithmics, algorithm engineering, operations research, numerical optimisation, machine learning, statistics, databases, parallel computing and software engineering, we believe that computer-aided algorithm design will fundamentally change the way in which we design high-performance algorithms. As a result, human experts will be able to more easily design effective solvers for computational problems encountered in application domains ranging from bioinformatics to industrial scheduling, from compiler optimisation to robotics, from databases to production planning.

4 Scientific Progress Made

The following two days were spent in subgroups exploring some topics identified at the end of the second day.

4.1 Learning during Search

Topic During the working session we concentrated our efforts on describing what would be the contributions of machine learning techniques, such as reinforcement learning, to the field of search in constraint programming. The focus of the discussion was put towards what could be achieved using learning *during* search, that is not using any offline training. We believe this is important to make such technique readily available as part of black box CP packages without requiring lengthy model training phases.

State Learning techniques are based on the principle that given that one can recognize that a process is in a given state, it is possible to learn the best suited action that should be performed. We consider the search state to be represented by the position in the search tree. Such position can be described by:

- the set of all branching decisions
- the set of all variable-value assignments
- the domains of all variables

Features Since learning mechanisms are subject to the curse of dimensionality and furthermore since the above state description has non constant dimensionality, it is desirable to succinctly describe the states as a short vector. To do so we defined features that could possibly abstract states by mapping them to real numbers. It is obviously crucial for performance that the features allow us to discriminate well between states. We have identified the following features:

- number of fixed variables
- size of the search space defined by a metric on domain sizes (like the Cartesian product)
- structure of the constraint graph
- solution space defined by information based on solution counting
- lower bound on the objective value
- backdoor information (probing, information based on the path from root to failure)

Actions Given a state identified by a set of features, the possible actions that can be taken during search are typically:

- branching on a variable/value assignment
- branching on a constraint (like precedence or domain splitting)
- restarting
- changing inference levels (from bound to arc consistency, or *vice versa*)
- changing the amount of propagation
- changing the search heuristics (variable and value selection heurisitics)
- changing the tree traversal strategy (DFS, LDS, etc.)

Reward function We discussed the possible reward functions that could be used during learning. When a reward is given after following an action, this reward would be back propagated up in the tree to previously taken decisions and visited states. If a failure has clearly a negative reward, it is not obvious how to define positive rewards. Given that we are solving a CSP, finding a feasible solution is not only a success but also the goal of the search... It was suggested to use as positive rewards something similar to pseudo costs in MIP. In CP this could be the difference in feature value for two successive states like, for instance, what is done during impact based search.

Notwithstanding the focus of the discussion, we agreed that training the reward function offline, perhaps using sophisticated parameter tuning packages such as paramILS[14] would probably be a good idea.

Algorithm We agreed that temporal difference learning will not be sufficient as propagation is expensive and it would be impractical to evaluate all possible branching decisions. Therefore it was suggested that a variant of Q learning would thus be more appropriate in such a case. There are a few papers on learning for Job Shop Scheduling by Zhang and Dietterich which seems apply to such a framework.

4.2 Empirical Models for Local Search

Topic The group addressed the issue of developing measurements of local search behaviour that could make a contribution to the principled understanding of local search performance. For example, there are a considerable number of papers that make appeals to two ill-defined notions (intensification and diversification) as a basis for motivating new local search algorithms and variations or as a basis for an intuitive explanation of search performance. There appears to be a notion that it is useful to balance intensification and diversification. To some extent such work is problematic as there is no formal definition (or even agreement) on what these measurements really are or how to measure them. Therefore, controlled experiments that seek to test if the superior performance of a given algorithm is related to its ability to, say, intensify, better than another algorithm cannot be done and the "empirical science" of heuristic search is not well developed.

Group The working group consisted of: Chris Beck, University of Toronto; Holger Hoos, University of British Columbia; Eric Hsu, University of Toronto; Serdar Kadioglu, Brown University; and Steve Smith, Carnegie Mellon University.

Summary of Important Points A detailed discussion over approximately three hours, developed the following main points:

- **Search Trajectory Features** We are primarily interested in characteristics of the states visited by a search method and the order in which the states were visited. Focusing on such measures, it was felt, would abstract away from algorithm details as well as from landscape features (e.g,. "the big valley"). While both algorithm details and landscape features are important it was generally agreed that they only impact search performance via the search trajectory and so focusing on the trajectory may help in developing and testing more precise ideas.

- **Quality and Distance Measures** It is important to distinguish quality and distance measures. A distance measure has something to do with the number of steps (or an estimate or bound on the number of steps) that it takes to get from one state to another. For example, one measure of stagnation might be that the maximum Hamming distance between the starting solution and any solution visited is not growing very quickly. In contrast, we have measures that include the cost function such as statistics about the quality of local optima (e.g., mean or variance or coefficient of variation of the costs of the local optima encountered). We also have potential measures that include them both such as statistics about the distance between consecutive local optima. Quality and distance metrics may reflect the same underlying search behavior in different ways. For example, if a local search is lost of a large plateau, a distance-based measure would show progress (e.g., getting increasingly further from the starting solution) while a cost-based measure would show stagnation (e.g., narrow variance in the cost of local optima). The group discussed intensification and diversification from the perspective of both quality and distance. It seems reasonable that for distance-based measures, intensification and diversification should be inverses: the more search visit states close (in e.g. Hamming distance) to a starting point (intensification), the less it explores different solutions (diversification). Quality-based intensification would seem to have the flavour of aggressively following a cost gradient whereas quality-based diversification might be measured by the variance of the costs of states (or local optima) that have been visited. It was felt that these are not directly inverses. No clear formal definition of these notions was developed–when such a definition is, different names should be used.

- **Measures of Local Optima Trajectory** Search trajectory features should be broad enough to include sub-trajectories. It was suggested that rather than looking at the state-by-state trajectory, it might be valuable to abstract to a local optimal-by-local optima trajectory: look at the characteristics of consecutive local optima. The local search features used in SATzilla [13] were given as examples. Over one local search run, measures include: change in cost from starting solution to minimum local optima found, number of search steps to first local optima, number of steps to minimum local optima, ratio of the difference in cost from starting solution to first local optima to the difference in cost from the starting solution to the minimum local optima, coefficient of variation of the cost of local optima. Distance and quality measures that can be defined on the overall search trajectory could be defined for the local optima trajectory and may be more meaningful/indicative.

- **Desired Characteristics of Measures** There are a variety of desirable qualities for any measures that we suggest. In particular, a measure over time ism ore useful than a single summary number. To take one example, it would seem that some sort of moving coefficient of variation of local optima qualities, visualized in a graph, is more meaningful than a single number for the entire trajectory. Measures should also “make sense” for particular, simple local search algorithms. For example, random walk intuitively should have some medium-level of diversity (if measured as a distance metric) while random sampling should have a relatively high measure (though perhaps not as high as a more sophisticated sampling method specifically designed for coverage of a space). Any suggested measure should be tested to confirm such “intuitive” behaviour.

- **Specific Measure Suggestions** One starting point for specific measures is the work of Schuurmans & Southey [12] who suggest: mobility (average Hamming distance between states that are k-steps apart in the trajectory), coverage (a measure of the maximal distance between visited and unvisited states), and depth (average objective value over the trajectory). One suggestion, based on mobility, is the expected number of steps to encounter a state with a Hamming distance of more than k from the starting point.

Further Information It is intended that a more in-depth description of the above points will be posted to the website of the Constraint Programming Society of North America.

4.3 Algorithm (CP solver) configuration

Topic The group discussed the application of automatic algorithm design to Constraint Programming. Automatic Algorithm design opens a new paradigm of thinking about and writing solvers. Usually the few parameters in solvers that are left open to the user for configuration have some “semantics”. When writing a highly configurable solver to be tuned by an automatic algorithm design, the designer can expand the exposed tunable parameters to aspects that do not have clear “semantics”. In complex configurable algorithms, one needs a way to express conditional dependencies between parameters. This can be handled currently by methods such as ParamILS [14]. In a complex design space, it might help to be able to express preferences over the design space such as “I expect this parameter with this value to have negative interactions with this other parameter-value pair.” or “If you change this parameter, you might need to change this other parameter.” One can also consider specifying a prior over the parameter space.

The challenge in applying automatic algorithm design to CP, as opposed to SAT, is that in CP there is a tight connection between the formulation of the model and the solving method applied (e.g. constraints (model) and propagators (algorithmic)).

Design Space of CP Solvers The group discussed the possible design space for a CP solver and identified relevant paper references:

- encoding/formulation/modeling alternatives (ref. [15] for automatic reformulation)
- ordering of propagation of constraints (ref. [18] for configurable priority queue of constraint groups in Gecode)
- restarts
- branching heuristics (e.g. automatic selection between the different variants of impact-based heuristics)
- filtering levels of constraints (ref. [20] for automatically choosing propagators)
- overall search approach (i.e. local search versus tree search)

Relevant Work We identified some existing work and systems that include partial automated algorithm design: AEON [19], Essence [16], Minion [17] (highly parametrized, reformulation SAT/LP, different propagations for alldiff), and Tailor [15] (direct compilation to Gecode and Minion from model language).

Challenge Problem To make this study more concrete, it was suggested that we pick a particular problem and have different people create different reconfigurable solution methods with exposed parameters. Then once these models are submitted, we could put everything as the components of a global design space that can be then explored through the automatic algorithm design framework. It was suggested that the challenge problem could be quasi-group with holes (QWH).

Instance-based Algorithm Configuration We also had a brief discussion on instance-based automatic algorithm selection and configuration and in particular the instance features of CP problems important for algorithm selection. We identified relevant work by Pascal van Hentenryck on the use of syntactic features of scheduling problems in COMET. One could also consider algorithmic features that are more informative but maybe more expensive to compute than the ones used in COMET, such as some of the features used in Satzilla for SAT, e.g. running a local search on the instance and recording runtime until first feasible solution found.

5 Outcome of the Meeting

We expect that some of the discussions started during the workshop will be continued and eventually lead to scientific publications. Many informal discussions in smaller groups also took place and the general feeling was that this had been a great opportunity to network within the north-american constraint programming community and to spark new collaborations. Some of these could also lead to publications in the next few years. We agreed to repeat the experience.

References

[1] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann, A constraint store based on multivalued decision diagrams, in C. Bessiere, ed., *Principles and Practice of Constraint Programming* (CP 2007), LNCS 4741, 118–132.

[2] M. Fischetti, F. Glover, and A. Lodi, The feasibility pump, *Mathematical Programming* **104** (2005), 91–104.

[3] M. Fischetti and A. Lodi, Repairing MIP infeasibility through local branching, *Computers and Operations Research* **35** (2008), 1436–1445.

[4] T. Hadzic and J. N. Hooker, Cost-bounded binary decision diagrams for 0-1 programming, in E. Loute and L. Wolsey, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* (CPAIOR 2007), LNCS 4510, 84–98.

[5] T. Hadzic, J. N. Hooker, and P. Tiedemann, Approximate compilation of constraints into multivalued decision diagrams, in P. J. Stuckey, ed., *Principles and Practice of Constraint Programming* (CP 2007), LNCS 5202, 448–462.

[6] W. van Hoeve, S. Hoda, and J. N. Hooker, MDD-based propagation of *among* constraints, presentation at INFORMS 2009, October.

[7] J. N. Hooker, *Integrated Methods for Optimization*, Springer, 2007.

[8] J. N. Hooker, Planning and scheduling by logic-based Benders decomposition, *Operations Research* **55** (2007), 588–602.

[9] J. T. Linderoth and M. W. P. Savelsbergh, A computational study of search strategies for mixed integer programming, *INFORMS Journal on Computing* **11** (1999), 173–187.

[10] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey, Cutting planes in integer and mixed integer programming, *Discrete Applied Mathematics* **123** (2002), 397-446.

[11] T. Sandholm and R. Shields, Nogood learning for mixed integer programming, CMU, November 2006.

[12] D. Schuurmans, D. and F Southey, Local search characteristics of incomplete SAT procedures, *Artificial Intelligence* **132**(2) (2001), 121–150.

[13] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown, SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT, In *Proceedings of the Thirteenth International Conference on the Principles and Practice of Constraint Programming*, 712–727, 2007.

[14] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proc. of the Twenty-Second Conference on Artifical Intelligence (AAAI '07)*, pages 1152–1157, 2007.

[15] Andrea Rendl, Ian Miguel, Ian P. Gent and Chris Jefferson. Enhancing Constraint Model Instances during Tailoring. In *SARA*. AAAI Press, 2009.

[16] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[17] Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102. IOS Press, 2006.

[18] Mikael Z. Lagerkvist and Christian Schulte. Propagator groups. In *CP*, pages 524–538, 2009.

[19] J.-N. Monette, Y. Deville, and P. van Hentenryck. *Aeon: Synthesizing Scheduling Algorithms from High-Level Models*, pages 43–59. 2009.

[20] Efstathios Stamatatos and Kostas Stergiou. Learning how to propagate using random probing. In *CPAIOR*, pages 263–278, 2009.

[21] Philippe Refalo. Impact-Based Search Strategies for Constraint Programming. In *CP*, pages 557–571, 2004.

[22] Alessandro Zanarini and Gilles Pesant. Solution Counting Algorithms for Constraint-Centered Search Heuristics. *Constraints*, 14(3):392–413, 2009.

[23] A. Braunstein, M. Mezard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27:201–226, 2005.

[24] Eric Hsu and Sheila McIlraith. Characterizing propagation methods for boolean satisfiability. In *Proc. of 9th International Conference on Theory and Applications of Satisfiability Testing (SAT '06), Seattle, WA*, 2006.

[25] Kalev Kask, Rina Dechter, and Vibhav Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Proc. of 10th International Conference on Constraint Programming (CP '04), Toronto, Canada*, 2004.

[26] Lukas Kroc, Ashish Sabharwal, and Bart Selman. Leveraging belief propagation, backtrack search, and statistics for model counting. In *Proc. of Fifth International Conference on Integration of AI and OR Techniques (CP-AI-OR '08), Paris, France*, 2008.

[27] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2), 2001.

[28] Ronan Le Bras, Alessandro Zanarini, and Gilles Pesant. Efficient generic search heuristics within the EMBP framework. In *Proc. of 15th International Conference on Constraint Programming (CP '09), Lisbon, Portugal*, 2009.

[29] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[30] Christopher J. Preston. *Gibbs States on Countable Sets*. Cambridge University Press, Cambridge, U.K., 1974.