Automated Design of Search with Composability

Ashish Sabharwal **Horst Samulowitz**
IBM Watson Research Center
Yorktown Heights, NY 10598, USA
ashish.sabharwal@us.ibm.com
samulowitz@us.ibm.com

Tom Schrijvers
Universiteit Gent, Belgium
tom.schrijvers@ugent.be

Peter J. Stuckey
National ICT Australia
University of Melbourne
Victoria, Australia
pjs@cs.mu.oz.au

Guido Tack
National ICT Australia
Monash University
Victoria, Australia
guido.tack@monash.edu

Introduction

Automated algorithm configuration aims at automatically parameterizing an algorithm or meta-algorithm when given a problem instance (or class) so that it performs most effectively on that instance. The need for algorithm configuration arises with computationally hard problems (e.g., problems in NP) where different solving strategies tend to work best on different classes of problem instances.

The approaches underlying algorithm configuration for combinatorial searh and optimization can be broadly grouped into *Portfolios*, *Parameter Tuning*, or a *combination* thereof. Portfolio based approaches automatically select an algorithm given a problem instance. Most importantly, the choice is amongst a set of pre-defined algorithms or paramterizations of a single algorithm. Such approaches are mostly based on common Machine Learning techniques such as regression, random forests, collaborative filtering, or k-nearest-neighbor, and exploit correlation in the corresponding data (e.g., (Xu et al. 2008; Xu et al. 2012; Stern et al. 2010; Kadioglu et al. 2011; Samulowitz et al. 2013)). Parameter tuning aims at determining novel, potentially previously unseen parameter settings that optimize performance (cf. (Hutter et al. 2009; Ansótegui, Sellmann, and Tierney 2009)). Most algorithms have some numerical or categorical parameters (e.g., number of iterations; deploy preprocessing or not) which can have a significant impact on performance. Due to lack of structure (e.g., convexity), commonly employed tuning methods use heuristic black-box approaches based on local search or genetic algorithms. Methods such as SATenstein (KhudaBukhsh et al. 2009) use a combination of portfolio and parameter tuning by automatically selecting predefined components of search. They are still, however, limited to a particular "structure" of the search itself, in this case a fixed local search "template". Algorithm design thus boils down to deciding which of various available heuristics to use in each line of the template algorithm.

Higher Level Search Design

Here we introduce the idea of automatically designing search algorithms at a much higher level by resorting to a search modeling language that provides one crucial property: a mechanism to design novel heuristic search strategies at will by *composing* existing ones. In essence, the aim of the approach is it to automatically compose a complete search built out of heuristic components such as various restart-, branching-, and value-selection strategies, by exploring the space of a search modeling language that is both human readable and allows automatic translation into very competitive algorithmic implementations. Unlike existing methods such as SATenstein, the recursive structure of the language does not limit the algorithm being designed to fit a particular, fixed template.

To that end we propose to employ *search combinators* (Schrijvers et al. 2013), a recently introduced modeling language for heuristic search. Search combinators provide a lightweight and solver-independent method[1] that bridges the gap between a conceptually simple modeling language for search (high-level, functional, modular, and naturally compositional) and an efficient implementation (low-level, imperative, and highly non-modular). By allowing to define application-tailored search strategies from a small set of primitives, search combinators effectively provide a rich domain-specific language for modeling search. Many commonly used systematic search strategies in Constraint Programming (CP) are already available or easily programmed in a few lines. Most importantly, they can be freely composed to easily design novel strategies.

Our goal is to develop an automated search design approach that can efficiently explore the space of search strategies defined by (a subset of) search combinators. This is in a similar vain as, but different from, the recent push on *probabilistic programming*[2] in AI where inference is performed automatically over joint probability distributions defined not by a structurally restricted graphical model but by a method in a higher level programming language.

Background: Search Combinators

Briefly, search combinators in their current form (see Schrijvers et al. (2013) for details) are tailored towards *systematic* CP search. It uses a base search parameterized by a set of variables and variable/value selection heuristics: base_search(`Vars`, `VarSel`, `ValSel`), with predefined methods like firstfail or impact for variable selection. Ad-

[1] Obviously search combinators must be facilitated by the underlying algorithm, but no fundamental changes are required.

[2] www.probabilistic-programming.org

ditional primitives are available to limit search and to access or manipulate search state. The expressive power of the language comes from *combinators*, which combine search heuristics (which can be basic or themselves constructed using combinators) into more complex heuristics. Supported combinators include and, or, ifthenelse, and portfolio. E.g., the following composite heuristic uses the and combinator to first label all `xs` variables using first-fail strategy, followed by the `ys` variables:

(1) and([base_search(xs, firstfail, min), base_search(ys, smallest, max)])

Similarly, the following snippet implements the commonly used geometric restart strategy:

(2) geo_restart(*fails*, s) ≡ let(*maxfails*, *fails*, restart(true, portfolio([limit(failures < *maxfails*, s), and([assign(*maxfails*, *maxfails* ∗ 1.5), prune])])))

The search initializes the search variable $maxfails$ to *fails*, and then calls search s with $maxfails$ as the limit. If the search is exhaustive, both the portfolio and the restart combinators are finished. If the search is not exhaustive, the limit is multiplied by 1.5, and the search starts over. More sophisticated strategies such as limited discrepancy search (LDS) with an upper limit of l discrepancies for an underlying search s can be expressed as follows, with for and limit themselves being composites:

(3) lds(l, s) ≡ for(n, 0, l, limit(discrepancies ≤ n, s))

Brief Overview of the Approach

To illustrate the approach, we make some simplifying assumptions: restrict configuration to a set of predefined search templates (e.g., lds(s)), allow only a small combinator depth,[3] and provide the design tool with information such as variable groupings $X, Y, Z, ...$ (common in CP). Now we could start with a basic search over the variable groups as shown in (1). As search strategies are fully compositional, we could also extend the initial base search by applying strategies (2) and (3) on top of it:

and([geo_restart(100,base_search(xs, firstfail, min)), lds(5, base_search(ys, smallest, max))])

Note that *composability* makes even predefined templates like lds here vastly richer than SATenstein's fixed template.

Composability, of course, also comes at the price of a vast design space. By exploiting the semantic structure of the search modeling language, we can however reduce this space in an effective and meaningful way. For instance, the composed search strategies considered must be valid both syntactically as well as semantically in terms of meeting the requirements of the search task (e.g., guaranteeting a complete search or an optimal solution). As a starting point, we choose a few predefined templates whose properties (e.g., whether they provide partial assignments, incomplete search, or complete search) are known. When applying a *compositional* operator such as and or ifthenelse to combine these templates in various ways, we must ensure the desired properties continue to hold. To automate the algorithm design process when using search composition, we need to address three main aspects:

1. define a design space of syntactically valid searches,
2. algorithmically define correctness (e.g., completeness) of generated searches, and
3. develop an efficient method to search in this design space.

The first aspect, validity, can be addressed by an appropriately defined context free grammar (CFG) explicitly capturing what is and isn't syntactically valid in the language of search combinators. Terminals of this CFG correspond to the arguments of the base heuristics such as *firstfail* or *min*, while non-terminals capture recursive compositions such as base_search(`Vars`, `VarSel`, `ValSel`) or and($[s_1, s_2, \ldots, s_n]$) where `Vars`, `VarSel`, `ValSel` and s_i are non-terminals. Note that the CFG needs to be extended by additional arguments to take properties like variable scoping and desired restrictions like expansion depth (e.g., 10) into account.

The second aspect can be handled by associating with every combinator specific properties such as completeness, and propagating, using suitably defined conditions on each combinator, the desired properties (e.g., completeness at the root node) down the parse tree of the CFG. For instance, any occurrence of the prune search operator must be enclosed inside a combinator like portfolio in order to be still able to guarantee completeness.

To address the third aspect, we propose to use a search strategy that explicitly takes the search combinator language structure into account, namely, Monte Carlo Tree Search, in particular UCT (Kocsis and Szepesvári 2006). This allows us to perform a top-down exploration of the CFG from Item 1 above. The children of each internal node correspond to selecting a non-terminal in the CFG and expanding it with one substitution. The branching factor is reduced using the additional requirements imposed by Item 2 above. The search of UCT is guided by the performance of the composed search at the "leaves" of the design search tree.

Concluding Remarks

We proposed a new perspective on the automated design of combinatorial search algorithms through an approach that operates at a much higher semantic level than previous algorithm configurators do. Instead of blindly tuning numerical or categorical parameters based on black-box optimization or resorting to a handful of predefined strategies, we propose to automatically search over compositions of search strategies using a light-weight language, while exploiting the semantic knowledge of the modeling language itself to guide the configuration process. Although somewhat reminiscent of the old AI vision that machines will be able to program themselves to solve novel tasks, we believe that the idea restricted to this simple but powerful search language has a chance of success in practice and are in the process of flushing out details of a basic search configurator on top of Gecode (Schulte and others 2009) which already fully supports search combinators.

[3] Schrijvers et al. (2013) showed that even highly domain specific searches can be modeled with depth under 10.

References

[Ansótegui, Sellmann, and Tierney 2009] Ansótegui, C.; Sellmann, M.; and Tierney, K. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In *International Conference on Principles and Practice of Constraint Programming*, LNCS, 142–157.

[Hutter et al. 2009] Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stutzle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

[Kadioglu et al. 2011] Kadioglu, S.; Malitsky, Y.; Sabharwal, A.; Samulowitz, H.; and Sellmann, M. 2011. Algorithm selection and scheduling. In *17th CP*, volume 6876 of *LNCS*, 454–469.

[KhudaBukhsh et al. 2009] KhudaBukhsh, A. R.; Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2009. Satenstein: Automatically building local search sat solvers from components. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, 2009*, 517–524.

[Kocsis and Szepesvári 2006] Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *17th ECML*, volume 4212 of *LNCS*, 282–293.

[Samulowitz et al. 2013] Samulowitz, H.; Reddy, C.; Sabharwal, A.; and Sellmann, M. 2013. Snappy: A simple algorithm portfolio - (tool paper). In *International Conference on Theory and Applications of Satisfiability Testing (SAT'13)*, LNCS.

[Schrijvers et al. 2013] Schrijvers, T.; Tack, G.; Wuille, P.; Samulowitz, H.; and Stuckey, P. J. 2013. Search combinators. *Constraints Journal,* to appear.

[Schulte and others 2009] Schulte, C., et al. 2009. Gecode, the generic constraint development environment. http://www.gecode.org/.

[Stern et al. 2010] Stern, D.; Samulowitz, H.; Herbrich, R.; Graepel, T.; Pulina, L.; and Tacchella, A. 2010. Collaborative expert portfolio management. In *AAAI 2010, Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010*, 210–216.

[Xu et al. 2008] Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: Portfolio-based algorithm selection for SAT. *JAIR* 32(1):565–606.

[Xu et al. 2012] Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2012. Evaluating component solver contributions to portfolio-based algorithm selectors. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*.