Search Combinators

Tom Schrijvers[1], Guido Tack[2], Pieter Wuille[2], Horst Samulowitz[3], and Peter J. Stuckey[4]

[1] Universiteit Gent, Belgium
[2] Katholieke Universiteit Leuven, Belgium
[3] IBM Research, USA
[4] National ICT Australia (NICTA) and University of Melbourne, Victoria, Australia

This work introduces search combinators, an approach to modeling search in constraint solvers that enables users and system developers to quickly design complex efficient search heuristics.

Search heuristics often make all the difference between effectively solving a combinatorial problem and utter failure. Heuristic enable a search algorithm to become efficient for a variety of reasons, e.g., incorporation of domain knowledge, or randomization to avoid heavy tailed runtimes. Hence, the ability to swiftly design search strategies that are tailored towards a problem domain is essential to performance improvement.

While we focus on systematic tree search in the area of Constraint Programming (CP), we believe that the results can be adapted to other search-driven areas in the field of Artificial Intelligence (AI) and related areas such as Operations Research (OR) (e.g., for Mixed Integer Programming (MIP) solvers). In fact, we also believe that our approach anticipates the challenges that will surface when one needs to control search in the context of *hybrid systems* that are composed of a variety of solvers (e.g., a mix of CP and MIP). In CP, much attention has been devoted to facilitating the modeling of combinatorial problems. A range of high-level modeling languages, such as Zinc [2] and OPL [4], enable quick development and exploration of problem models. However, we see very little support on the side of formulating accompanying search heuristics. Either the design of search is restricted to a small set of predefined heuristics (e.g., MiniZinc [3]), or it is based on a low-level general-purpose programming language (e.g., Comet [5]). The former is clearly too confining, while the latter leaves much to be desired in terms of productivity, since implementing a search strategy quickly becomes a non-negligible effort. This also explains why the set of available heuristics is typically small: it takes a lot of time for CP system developers to implement heuristics, too – time they would much rather spend otherwise improving their system.

In this work we show how to resolve this stand-off between solver developers and users with respect to a high-level search language.

For the user, we provide a compositional approach for expressing complex search heuristics based on an (extensible) set of primitive combinators. Even if the users are only provided with a small set of combinators, they can already express a vast range of combinations. Moreover, programming search in terms of combinators is far more productive than resorting to a low-level language.

The following heuristic briefly demonstrates the conciseness of our search combinators. This search heuristic can be used to solve radiotherapy treatment planning problems [1]. The heuristic minimizes a variable obj using branch-and-bound (bab),

first searching the variables N, and then verifying the solution by partitioning the problem along the row_i variables, one row at a time. Failure on one row must be caused by the search on the variables in N, and consequently search never backtracks into other rows (exh_once). This is a good example of integrating domain knowledge in search:

```
bab(obj, and([int_search(N, input_order, bisect_low),
              exh_once(int_search(row_1, input_order, bisect_low)),
              ...,
              exh_once(int_search(row_n, input_order, bisect_low))]))
```

Here we assume that a basic search construct like the following is available:

$$s = \mathsf{int\_search}(vars, \textit{var-select}, \textit{value-select})$$

which specifies a systematic search over the variables *vars*, applying *var-select* and *value-select* as variable- and value-selection strategies respectively. Another primitive combinator is and, with the obvious meaning. The remainders, bab and exh_once, are themselves defined in terms of other primitive combinators.

For the system developer, we show how to design and implement modular combinators. Developers do not have to cater explicitly for all possible combinator combinations. Small implementation efforts result in providing the user with a lot of expressive power. Moreover, the cost of adding one more combinator is small, yet the return in terms of additional expressiveness can be quite large.

In summary, the tough technical challenge we face is to bridge the gap between conceptually simple specification language (high-level, purely functional and naturally compositional) and efficient implementation (typically low-level, imperative and highly non-modular). We overcome this challenge with a systematic approach that disentangles different primitive concepts into separate modular components, search combinators, that interact through a *message protocol*. This protocol dictates how the combinators should collaborate to process a node in the search tree. Overall search then consists of a queue of unprocessed nodes that are fed one by one to the combinators, which in turn may produce new (child) nodes for the queue. The message-based combinator approach lends itself well to different implementation strategies. We have developed two diametrically opposed approaches for the Gecode C++ library: *dynamic composition* (interpretation) and *static composition* (compilation). Experimental evaluation shows that both implementation approaches have competitive performance and match the performance of the native implementation of the same search heuristics in Gecode.

References

1. Baatar, D., Boland, N., Brand, S., Stuckey, P.: CP and IP approaches to cancer radiotherapy delivery optimization. Constraints (2011)
2. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3), 229–267 (2008)
3. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessire, C. (ed.) CP. LNCS, vol. 4741, pp. 529–543. Springer (2007)
4. Van Hentenryck, P.: The OPL optimization programming language. MIT Press (1999)
5. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)