Priority Search with MiniZinc

Thibaut Feydy[1], Adrian Goldwaser[3], Andreas Schutt[1,2], Peter J. Stuckey[1,2], and Kenneth D. Young[2]

[1] Data61, CSIRO, Australia
[2] University of Melbourne, Australia
[3] University of New South Wales, Australia

Abstract. MiniZinc is a powerful modeling language for constraint programming, which is capable of concisely specifying complex models. But much of the power of constraint programming is not available to users of MiniZinc since it only supports a very limited search language. Extending the MiniZinc search language is important to allow much more complex searches to be specified, but also challenging since extensions must be implemented by solver writers. Indeed only a minority of systems that support MiniZinc support its existing search language. In this paper we add one new search constructor, `priority_search`, to MiniZinc which allows the specification of complex nested searches. Priority searches, allow the use of a variable array to select which of an array of search constructs should be processed next. We explain how this search constructor can be straightforwardly implemented in systems that support MiniZinc's existing search language. We show that the expressiveness of `priority_search` allows us to improve the solving of a number of MiniZinc models.

1 Introduction

Constraint programming (CP) is a highly successful approach to discrete optimization. The success is founded on two important features of CP:

- *heterogeneous global constraints* which allow the user to write a high level model using the global constraint to capture important combinatorial substructures, such that the conjunction can be solved, since the CP formalism handles arbitrary heterogeneous collections of constraints.
- *user defined search* which allows the user to specify how to look for good solutions, using their knowledge of the problem.

MiniZinc [8] has been very successful at allowing the user to make use of global constraints, even when the constraints are not natively supported by the underlying solver they use. Hence the advantages of the CP modeling approach transfer to other solving technology such as MIP and SAT. But MiniZinc has been less successful at providing extensive search facilities. The MiniZinc search language is quite restrictive and even then it is not implemented by all CP solvers that support MiniZinc. Extending the search language for MiniZinc is

thus a two-edged sword: each extension can improve the usability of the system for CP solving, but each new feature has to be seen as beneficial enough that the solver writer will actually implement it.

In this paper we argue that adding one new search combinator, priority search, is worth the tradeoff. The priority search combinator allows us to dynamically choose the order of a number of searches, using the state of some solver variables.

The most common kind of search strategy that is supported by priority search is one where we want to select some object dynamically and then fix all the decisions about that object.

Example 1. In a flexible job shop scheduling we must determine for each task which machine it should run on, and when it should run, respecting the precedences of tasks which make up a job. A good search strategy for this is to choose a task with the smallest possible start time and then fix its start time and the machine it runs on. Its not possible to represent such a strategy in the current MiniZinc search, since effectively we need to use the start time to select the next *two* variables to label. We can represent this using priority search however. □

Similar example arise in packing problems, where we wish to fix all the placement decisions about an object being packed at the same time, or in configuration problems, where once we choose to include a large device in the configuration we wish to fix all the decisions related to configuring that device.

Priority search allows a separation of the variables used to choose what to search on from the actual variables which are searched on. This freedom allows for much more interesting search strategies to be defined directly in MiniZinc without adding significant new overhead to the solver writer. Other uses it can be put to include: user-defined tie-breaking on variable selection, randomization of search, and complex value-selection strategies.

2 Programmed Search in MiniZinc

Programmed search in MiniZinc is currently quite restricted. There are basic search routines, exemplified by `int_search`[4]

```
annotation int_search(array[int] of var int, ann, ann)
```

An `int_search` declaration `int_search`$(x, varsel, valsplit)$ takes an array of integer variables x, and a variable selection strategy $varsel$ and a value splitting strategy $valsplit$. The strategy repeatedly chooses a variable from x using the selection strategy $varsel$ and then chooses a splitting constraint using the $valsplit$ and branches on the splitting constraint and its negation. During this process variables in the array x which are fixed are omitted from consideration.

Variable selection strategy include such things as: `input_order` in the order of the array, `first_fail` ordered by current domain size, `smallest` ordered

[4] Currently `int_search` has a fourth argument, a holdover from the Eclipse definitions it is based on, which is deprecated so we will omit it in this paper.

by current domain minimum, and `largest_smallest` ordered by largest of the current domain minimum.

Value splitting methods include such things as: `indomain_min` equate to minimum value, `indomain_split` constrain to lower half of values, and `indomain_random` equate to a random value in the domain.

Example 2. For example, consider a search annotation `int_search([x,y,z], smallest, indomain_min)` with the current domains $x = 0$, $y \in 5..9$, $z \in 4..12$. Then variable z is selected and the splitting constraint is $z = 4$. □

Similar basic search strategies are available for arrays of `bools`, arrays of `floats` and arrays of set of integers.

The other component of MiniZinc search is a single search combinator

```
annotation seq_search(array[int] of ann)
```

which takes an array of search annotations and applies them in order, exhausting each search strategy, i.e. fixing all the variables mentioned in the search strategy, before proceeding to the next in the list.

Example 3. For example, consider a search annotation

```
seq_search( [bool_search([b1,b2,b3], input_order, indomain_random),
             int_search([x1,x2,x3],  smallest,    indomain_min)   ])
```

which will choose a random value for the Booleans $b1$, $b2$, $b3$, before proceeding to label the x variables in order of smallest value. □

Since search annotations are part of the annotation type `ann` we can define parameters (variables) that take these values, and use any other MiniZinc construct that is applicable to this type, in particular `if-then-else-endif`. Since this computation is managed by MiniZinc itself it adds no overhead to the solver.

Example 4. Given a 2D packing problem, we can calculate the density of packing in the two directions l and w and choose to search first on the most dense dimension. Consider a set of rectangles of lengths `len` and widths `wid` to be packed in a box of length `boxlen` and width `boxwid`. The search annotation

```
if   sum(len) div boxlen > sum(wid) div boxwid
then seq_search([int_search(l, smallest, indomain_min),
                 int_search(w, smallest, indomain_min)])
else seq_search([int_search(w, smallest, indomain_min),
                 int_search(l, smallest, indomain_min)])  endif
```

will encode this. Note that the solver will only receive one of the sequential search annotations. □

3 Priority Search

The priority search strategy uses an array of variables to select out of array of search strategies, which to do next

```
annotation priority_search(array[int] of var int, ann, array[int] of ann)
```

An annotation of the form `priority_search(`*selvars*`,` *varsel*`,` *searches*`)` considers the array of variables *selvars* using the variable selection strategy *varsel* to select a variable, or equivalently an index into the array *selvars*. The index is then used to determine which of the array of search strategies *searches* is executed next.

Example 5. Given a flexible job shop model with decisions

```
array[TASK] of var TIME: start;      % start time of task
array[TASK] of var MACHINE: machine; % machine for task
array[TASK] of set of MACHINE: ms;   % machines possible for task
array[MACHINE] of var int: load;     % load on each machine
```

then a natural search strategy is to select the task with the earliest possible start time and fix its start time and machine. Using priority search we express this as

```
priority_search(start, smallest,
                [ int_search([start[t],machine[t]], input_order, indomain_min)
                | t in TASK])
```

We can use priority search to also implement complex value selection search strategies since we can map these to variable selection strategies.

Example 6. A more involved search for the flexible job shop scheduling problem using nesting priority searches is

```
priority_search(start, smallest,
  [ seq_search([int_search([start[t]], input_order, indomain_min),
                priority_search([ load[m] | m in ms[t] ], smallest,
                                [ bool_search([ machine[t] = m | m in ms[t] ],
                                              input_order, indomain_max)
                                | m in ms[t] ])
               ])
  ])
```

Here we pick the task with the smallest start time, and then choose from its possible machines the machine with the least current load, and assign it to that machine. Note how we make use of priority search here to implement a value selection heuristic. □

Note that is critical that the selection strategy itself does not ignore fixed variables, it may often be the case that variables in *selvars* are already fixed. Instead the criteria for ignoring a variable/index for selection is that all the variables in the corresponding search strategy are already fixed.

Example 7. Consider a routing problem encoded using decision variables

```
array[NODE] of var NODEx: next;     % next NODE after this one or dummy
array[NODE] of var TIME: visit;     % visit time for this NODE
constraint forall(n in NODE)
                 (visit[next[n]] = visit[NODE] + travel_time[n,next[n]]);
```

We wish to search by deciding the next places to visit. But we would like to choose that node based on earliest possible visit time. The search strategy is

```
priority_search(visit, smallest,
                [ int_search([next[n]],input_order,indomain_random)
                | n in NODE ])
```

Note that by the time we pick a node its visit time will be fixed by the constraints defining visit time. □

Another usage of priority search is when we wish to search on Boolean variables. Since the traditional measures on variables, e.g. domain size, are not very meaningful for Booleans its not really possible to select the best Boolean out of an array to branch on. Using priority search we can make this decision based on a related integer expression.

Example 8. Consider the Feedback Arc Set [5] problem of totally ordering the nodes in a directed graph in order such that the least number of edges point from a larger numbered node to a smaller numbered node. A MiniZinc model for the problem is

```
int: n;
set of int: NODE = 1..n;
int: m;
set of int: EDGE = 1..m;
array[int] of NODE: from;  % directed edge from node from[e]
array[int] of NODE: to;    % to node to[e]
array[NODE] of var NODE: order;    % total order of nodes
array[EDGE] of var bool: reversed; % which edges have reversed order
constraint forall(e in EDGE)(reversed[e] = (order[from[e]] >= order[to[e]]));
include "alldifferent.mzn";
constraint alldifferent(order);
solve minimize sum(reversed);
```

A basic search strategy for this is analogous to scheduling, and labels the `order` variables from first to last `int_search(order,smallest,indomain_min)`. This is often a very bad order, assuming the graph is sparse, since all that needs to be decided is the `reversed` variables, which will guarantee that a total order exists. Using existing search facilities we can do little more than `bool_search(reversed, input_order,indomain_min)` since we cant really attach usefully choose from a list of Boolean variables. Using priority search we can define some auxiliary integer expressions to help choose.

```
priority_search([ from[e] - to[e] | e in EDGE], largest_smallest,
                [ bool_search([reversed[e]],input_order,indomain_min)
                | e in E ])
```

The above definition selects the edge with largest of the smallest possible value of the difference between the `from` and `to` nodes, and sets it to be not reversed. This will select the edge which is closest to being satisfied without being reversed and set it as not reversed, thus trying to drive quickly to a good solution. □

Priority search allows us to build tie breaks into decisions about which variable to label.

Example 9. In a carpet cutting problem [11] we may wish to pick the carpet that can be placed closest to the beginning of the roll. But initially all carpets can be placed there, hence is essential to tie break, which we can do using area of carpet. Given data and variable declarations

```
array[CARPET] of int: area;
array[CARPET] of var XPOS: x;
array[CARPET] of var YPOS: y;
```

a priority search which chooses the carpet that goes closest to the beginning of the roll, tie breaking on carpet area is expressed as

```
let { int: maxarea = max(area)+1; } in
    priority_search([ maxarea*x[c] - area[c] | c in CARPET], smallest,
                    [ int_search([x[c],y[c]], input_order, indomain_min)
                    | c in CARPET ])
```

by encoding the lexicographic order in some auxiliary variables. □

A less obvious use of priority search is to select between different searches for the whole problem. Given n possible search strategies for a given problem we can calculate some features of the problem in order to rank them and then choose the choice which is ranked highest. If the ranking is static, then this can be managed with the existing MiniZinc features using `if-then-else-endif`, but if the ranking is dependent on solver internals we need priority search.

Example 10. Consider a complex optimization problem with 4 competing search strategies `search1`, ..., `search4` that each will fix all the decisions of the problem. We can randomly select which strategy to use.

```
priority_search([0|i in 1..4],random_order,[search1, search2, search3, search4])
```

While at face value this seems uninteresting, just selecting a search strategy once on commencement, if we combine this with restarts, since the priority search is rerun from scratch on each restart, we do get to make use of each of the search strategies. □

3.1 Implementing Priority Search

Implementing the priority search combinator is fairly straightforward. The solver already implements the dynamic variable selection strategies used in priority search, so this can be reused from the existing code base. One caveat is that the strategies need to be modified to select fixed variables, and ignore exhausted searches.

The chief change, depending on how the `seq_search` combinator is implemented, is that now search strategies can no longer be seen as simply a list of basic searches (a flattening of `seq_search` annotations). Instead the solver must represent the annotation structure, although it may already do so if `seq_search`s were not flattened. The remainder of the implementation is fairly standard, when running a priority search combinator the solver must be able to notice when a search is exhausted, so as to finish it and try to select the next search to perform. Since search exhaustion is already required for the sequential combinator this is not burdensome.

We argue that the addition of `priority_search` is thus minimally burdensome on solver implementers. Indeed, the implementation required 75 lines of code to added to `Chuffed` (as well as 166 lines required to change the parser which was previously specialized to return a flat list of base searches).

Table 1. Comparing searches using existing MiniZinc search language, versus those enabled by priority search.

Benchmark	Original Search					Priority Search				
	#opt	#best	Obj	Fails	Time	#opt	#best	Obj	Fails	Time
`fjsp` (5)	**2**	0	3828	**250k**	**180s**	**2**	**3**	**1802**	1206k	**180s**
`carpet` (20)	3	3	2042	1511k	257s	**4**	**13**	**1984**	**977k**	**244s**
`fas` (38)	**30**	1	18.3	478k	**77s**	**30**	**4**	**18.1**	**337k**	88s

4 Experiments

To illustrate the benefit we show how the programmed search for a number of benchmarks used in the MiniZinc challenge, which use the existing MiniZinc search facilities, can be improved using priority search. We use `fjsp` (flexible job shop scheduling: Challenge 2013) and `carpet` (carpet cutting: Challenge 2011). For `fjsp` we pick the task with the smallest start time and then it picks the machine with the current smallest load (as in Example 6). For `carpet` we first set the rotation of all carpets (as in the original search) and then choose the carpet with leftmost position tie-breaking on size of carpet, and fix its position (as in Example 9). We also compare the effectiveness on the Feedback Arc Set (`fas`) problem of Example 8 trying the fixed Boolean search, versus the priority Boolean search. For each benchmark we show the aggregated results over a number of instances (shown in parentheses) of the problem in Table 1. We ran the CP solver `Chuffed` [2] extended with priority search for at most five minutes per instance and recorded the number of optimal solutions (#opt), the number of instances the search found a better solution within five minutes (#best), the mean value of the objective, the mean number of failure/conflicts (Fails), and the mean runtime (Time).

Unsurprisingly from a CP perspective, if we increase the expressiveness of the search language we should be able to find better programmed searches. By driving towards better solutions earlier priority search tends to improve the number of instances that can be proved optimal, and improve the best solution found within a time limit.

5 Related Work

Unsurprisingly any search that can be specified with a priority search annotation can be directly programmed in the implementation language of the underlying solver. Specialized searches for particular problems, which are expressible as priority searches have been used in CP from its very beginnings (see e.g. [1]). Constraint logic programming [4] allowed the specification of such searches, as did early search languages like SALSA [6]. The priority search annotation makes these searches possible for a modeler with no idea about the underlying solver or its implementation language.

The search language for OPL [12] is highly expressive, allowing complex variable and value selection strategies as well as event handling, and search limits.

The priority search combinator implements the `ordered by` construct of OPL which allows a selection of an object ordered by a the value of a complex expression which can include run time solver information through functions like `dsize` (domain size), `dmin` (domain min), and `regretdmin` (minimal regret value). Implementing such a complex search language requires tight communication with the solver, and OPL's search is tied to a single solver.

Search in ObjectiveCP [7] is similar in expressiveness to OPL but makes use of closures of Objective-C to allow searches to be expressed in Objective-C using a library of higher order procedures. The search language can express priority search. ObjectiveCP, like MiniZinc, allows the specification of search strategies independent of the underlying solver, using a reversible mapping from model variables to solver variables to obtain runtime information from the solver. The interface between ObjectiveCP model and solver is thus much richer, and bidirectional, whereas MiniZinc sends the search strategy to the underlying solver.

The lack of expressiveness of MiniZinc search has been noted before, and a number of approaches to defined to extend it.

Search combinators [10] are a high level approach to extending search using combinators. It separates the search concerns of: labeling, i.e. what decision to make at each search node; queuing, i.e. which nodes should be selected to expand; and search heuristic which decides which nodes by which labeling, which nodes are processed again, and which are cut from the search. The labeling strategies are assumed to be base search combinators that exist in the solver, and in that sense search combinators are mainly orthogonal to this work, which adds a new kind of base search capability. While search combinators do overlap with priority search in their ability to change labeling strategies dependent on current solver properties, it is not possible to define priority search using the search combinators of [10], since variable selection is restricted to base searches. Hence the extensions are orthogonal.

MiniSearch [9] is an approach to extend MiniZinc search without changing the interface to solvers. It is a drastically cut down version of search combinators, which does not rely on the solving implementing anything except the existing FlatZinc interface. MiniSearch allows the specification of meta-search approaches such as lexicographic branch and bound, large neighborhood search, or diverse solution search, by only interacting with the solver when it finds solutions. The MiniSearch approach to extending search is orthogonal to the priority search extension, which allows a different search tree to be specified to the solver, by changing the way in which it chooses to make decisions. The two approaches are completely compatible.

6 Conclusion

The priority search combinator is a straightforward, yet powerful, addition to MiniZinc's search language. It is simple to implement, but significantly extends the kinds of search that can be specified in MiniZinc. Hence we hope it is attractive for solver writers to implement.

The priority search combinator was originally developed for and used in two applications [3,13] whose papers will appear at CP2017, including the paper [3] recieving the best student paper award. Without priority search these applications are not nearly as efficient as without it. This gives more evidence of the utility of this modest addition to MiniZinc search.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Journal of Mathematical and Computer Modelling 17(7), 57–93 (1993)
2. Chu, G.: Improving Combinatorial Optimization. Ph.D. thesis, Department of Computing and Information Systems, University of Melbourne (2011)
3. Goldwaser, A., Schutt, A.: Optimal torpedo scheduling. In: Beck, C. (ed.) Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming. p. to appear. LNCS, Springer (2017)
4. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program. 19/20, 503–581 (1994)
5. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W., Bohlinger, J.D. (eds.) Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations. pp. 85–103. Springer US, Boston, MA (1972)
6. Laburthe, F., Caseau, Y.: SALSA: A language for search algorithms. Constraints 7(3-4), 255–288 (2002)
7. Michel, L., Van Hentenryck, P.: A microkernel architecture for constraint programming. Constraints 22(2), 107–151 (Apr 2017)
8. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessiere, C. (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 529–543. Springer-Verlag (2007)
9. Rendl, A., Guns, T., Stuckey, P.J., Tack, G.: MiniSearch: a solver-independent meta-search language for MiniZinc. In: Pesant, G. (ed.) Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming. pp. 376–392. No. 9255 in LNCS, Springer (2015)
10. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.: Search combinators. Constraints 18(2), 269–305 (2013)
11. Schutt, A., Stuckey, P., Verden, A.: Optimal carpet cutting. In: Lee, J. (ed.) Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 6876, pp. 69–84. Springer (2011)
12. Van Hentenryck, P., Perron, L., Puget, J.F.: Search and strategies in OPL. ACM TOCL 1(2), 285–315 (2000)
13. Young, K.D., Feydy, T., Schutt, A.: Constraint programming applied to the multi-skill project scheduling problem. In: Beck, C. (ed.) Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming. p. to appear. LNCS, Springer (2017)