MiniSearch: a solver-independent meta-search language for MiniZinc

Andrea Rendl[1], Tias Guns[2], Peter J. Stuckey[3], and Guido Tack[1]

[1] National ICT Australia (NICTA) and Faculty of IT, Monash University, Australia
andrea.rendl@nicta.com.au, guido.tack@monash.edu
[2] KU Leuven, Belgium
tias.guns@cs.kuleuven.be
[3] National ICT Australia (NICTA) and University of Melbourne, Victoria, Australia
pstuckey@unimelb.edu.au

Abstract. Much of the power of CP comes from the ability to create complex hybrid search algorithms specific to an application. Unfortunately there is no widely accepted standard for specifying search, and each solver typically requires detailed knowledge in order to build complex searches. This makes the barrier to entry for exploring different search methods quite high.
Standardizing search is a difficult task. Search is a core part of the solver, and usually highly optimized. Any imposition on the solver writer to change this part of their system is significant, hence it is difficult to see a path to a standard search language for CP systems.
In this paper we investigate how powerful we can make a uniform language for meta-search *without placing any burden on the solver writer*. The key to this is to only interact with the solvers when a solution is found. We present MiniSearch, a search language that can directly use any FlatZinc solver. We illustrate the expressiveness of the language and performance using different solvers on a number of examples.

1 Introduction

When using constraint programming (CP) technology, one often needs to exert some control over the search mechanism. This aids finding good solutions to highly complex problems, by encoding meta-information a modeller has about the problem.

Unfortunately, there is no widely accepted standard for controlling search. A wide range of high-level languages have been proposed that are quite similar in how constraints are specified. However, they differ significantly in the way search can be specified. This ranges from built-in minimization and maximization only to fully programmable search. The main trade-off in developing search specification languages is the expressivity of the language versus the required integration with the underlying solver. Fully programmable search is most expressive but requires deep knowledge of and tight integration with a specific solver. Languages like OPL [23] and COMET [13] provide convenient abstractions for programmable search for the solvers bundled with these languages.

On the other hand, solver-independent languages such as Zinc [12], Esra [3], Essence [5] and Essence′ [6] do not support search specifications. MiniZinc [15] has support for suggesting variable and value ordering heuristics to CP solvers, but also no real control over the search.

Search combinators [18] was recently proposed as a generic search language for CP solvers. It interacts with the solver *at every node* in the search tree. While very expressive, it requires significant engineering effort to support in existing solvers as typically the search engine is highly optimized and tightly tied to other components, such as the propagation engine and state space maintenance, which must not be for these combinators.

In this paper, we introduce MiniSearch, a new combinator-like search language that incorporates three objectives: a *minimal solver interface* to facilitate solver support, *expressiveness*, and, most importantly, *solver-independence*.

The objective to obtain a minimal solver interface stems from lessons learnt in the design of search combinators that interact with the solver at every *node*. In contrast, MiniSearch interacts with the underlying solving system only *at every solution*, which is a minimal interface. At every solution, constraints can be added or constraints in a well-defined *scope* can be removed, before asking for the next solution. If the underlying solver does not support dynamic adding and removing of constraints, MiniSearch can emulate this behaviour, for little overhead.

Despite the lightweight solver interface, MiniSearch is surprisingly expressive and supports many meta-search strategies such as Branch-and-Bound search (BaB), lexicographic BaB, Large Neighborhood Search variants, AND/OR search, diverse solution search, and more. Furthermore, MiniSearch can be used to create interactive optimisation applications as well. Moreover, since MiniSearch builds upon MiniZinc, all MiniZinc language features and built-ins can be used in MiniSearch, for instance to formulate custom neighbourhoods.

Solver-independence is the most important contribution of MiniSearch. All solvers that can read and solve FlatZinc, which the majority of CP solvers do [14,21], can be used with MiniSearch. Moreover, solvers that provide native meta-search variants, such as branch-and-bound, can choose to apply their native version instead of executing the MiniSearch decomposition. At the language level, this is similar to the handling of global constraints in MiniZinc. Thus, solvers can apply their strengths during meta-search, despite the minimal interface.

2 The MiniSearch language

MiniSearch is a high level meta-search language based on MiniZinc 2.0 [22]. The same language is used for formulating the model and the constraints posted during search, with language extentions for specifying the search. A MiniSearch specification is provided as an argument to MiniZinc's `solve` item, and executed by the MiniSearch kernel (see Sec. 4). With the built-in language exten-

Table 1. MiniSearch built-ins

MiniSearch built-ins	Description
`next()`	find the next solution
`post(`c`)`	post the MiniZinc constraint c in the current scope
`scope(`s`)`	open a local scope containing search s
`commit()`	commit to last found solution in function scope
`time_limit(`ms,s`)`	run s until timelimit ms is reached
s_1 `/\` s_2	run s_1 and iff successful, run s_2
s_1 `\/` s_2	run s_1 and iff it fails, run s_2
`repeat(`s`)`	repeat search s until `break` is executed
`repeat (i in 1..`N`)(`s`)`	repeat search s N times or until `break` is executed
`if` s `then` s_1 `else` s_2	if s is successful, run s_1, otherwise s_2
`break()`	break within a `repeat`
`fail()`	return ‘failure’
`skip()`	return ‘success’
`print(`S`)`	print MiniZinc output string S
`print()`	print solution according to model output specification
c `:=` v	assign parameter c the value v
`sol(`v`)`	return solution value of variable v
`hasSol()`	returns true if a solution has been found

tions summarized in Tab. 1, users can define MiniSearch functions such as the following branch-and-bound (BaB) minimization:

```
1 include "minisearch.mzn";
2 var int: obj; % other variables and constraints not shown
3 solve search minimize_bab(obj);
4 output ["Objective: "++show(obj)];
5
6 function ann: minimize_bab(var int: obj) =
7   repeat( if next() then commit() /\ print() /\ post(obj < sol(obj))
8           else break endif );
```

The `include` item on line 1 includes the built-in MiniSearch function declarations. This is necessary for any MiniZinc model that uses MiniSearch. Line 3 contains the MiniZinc `solve` item followed by the new `search` keyword and a user-defined MiniSearch function that takes a variable representing the objective as argument. Line 4 is the MiniZinc `output` item, specifying how solutions should be printed. Lines 7–8 contain the actual MiniSearch specification. We will explain the different built-ins in more detail below, but the specification can be read as follows: repeatedly try to find the next solution; and if that is successful, commit to the solution, print it and add the constraint that the objective must have a lower value than the current solution. If unsuccessful, break out of the repeat.

All MiniSearch built-ins are typed as *annotations* which are MiniZinc functions that return annotations. Semantically, every MiniSearch built-in returns a value that represents either ‘success’ or ‘failure’, with respect to finding a solution. The handling of these implicit return values is done by the MiniSearch interpreter (Sec. 4.1).

2.1 MiniSearch built-ins involving the solver

Communication with the solver is restricted to three forms: invoking the solver, adding constraints/variables to the model, and removing constraints/variables.

Invoking the solver. The MiniSearch instruction for finding the next solution is `next()`. It is successful if a solution has been found, and fails otherwise. The variable/value labelling strategy (such as first-fail on the smallest domain value) can be set in two ways: either by annotating the `solve` item (as in standard MiniZinc), which sets the labelling globally, for every call to `next()`. Otherwise, by annotating any MiniSearch function call, such as `minimize_bab`, with a labelling strategy. Note, however, that as in MiniZinc, solvers may ignore these annotations, for example if the labelling is not supported. Solvers may also support more complex search strategies natively, such as `minimize_bab`, in which case the corresponding MiniSearch functions are treated as built-ins.

Adding constraints and variables. A constraint is added by calling the `post()` built-in with a constraint as argument. Constraints can be formulated using the same MiniZinc constructs as in the model, including global constraints, user-defined functions and predicates. Variables can be dynamically added during search too, using the MiniZinc `let` construct (see policy-based search).

Search Scopes. Search scopes define the lifespan of constraints and variables in the model. MiniSearch has an implicit *global* search scope that contains all variables and constraints of the model. A new search scope can be created by using the `scope(s)` built-in that takes a MiniSearch specification `s` as an argument. When entering a scope, search starts from the root again. Whenever execution leaves a scope, all constraints and variables that were added in the scope are removed from the model and the solver. Execution in the enclosing scope resumes from the point where it left off.

2.2 MiniSearch control built-ins

All MiniSearch built-ins have an implicit return value that represents either 'success' (*true*) or 'failure' (*false*). Using this concept, we introduce MiniSearch control built-ins. All built-ins execute their arguments in order.

And, Or, Repeat. The `/\`-built-in runs its arguments in order and stops and returns *false* as soon as one of its arguments fails. Similarly, the `\/`-built-in stops and returns *success* as soon as one of its arguments succeeds. The `repeat(s)` built-in takes a MiniSearch specification `s` and repeats it until the built-in `break` is executed. `break` stops *s* and `repeat` returns *s*' current status. There also exists a delimited variant `repeat(i in 1..N)(s)` that will executes `s` for `N` times (or until `break` is executed). Existing control mechanisms of MiniZinc such as `if then else endif` expressions can be used as well.

Time-Limits. The built-in `time_limit(ms,s)` imposes a time limit `ms` (in milliseconds) on any MiniSearch specification `s`. This way, `s` stops whenever the time limit is reached, returning its current status. Time-limits are handled transparently by the MiniSearch kernel as an exception.

Assigning values to constants. Standard MiniZinc does not allow a constant such as `int: N=10;` to be assigned a value twice. However, in MiniSearch we often want to change constants across different iterations. For this purpose, we added the assignment operator `:=` which may only be used inside a MiniSearch specification. It overwrites that constants' current value by the value supplied.

2.3 Solution management

The strength of any meta-search language lies in using intermediate solutions to guide the remaining search. For instance, branch-and-bound needs to access the objective to post further constraints, and a Large Neighbourhood Search thaws some of the variables in a solution to continue in that neighbourhood.

To facilitate working with solutions, the most recently found solution is always accessible in MiniSearch using the `sol` built-in, where `sol(x)` returns the value of `x` in the last solution. MiniSearch also provides a `hasSol()` built-in to test whether a solution exists.

User-defined functions. When a MiniSearch strategy is defined as a MiniZinc function, a local solution scope is created. This means that any solution found by a call to `next` inside the function is visible for the code in the function body, but not for the caller of the function when the function returns. This architecture allows for calls to `next` to be *encapsulated*, i.e., a function can make "tentative" calls to next in a nested search scope and only commit if these succeed. Sect. 3.3 shows how AND/OR search can be implemented based on this principle. In order to make the current solution accessible to the caller, the function must call the `commit` built-in. A function returns 'success' if it called `commit` at least once, and 'failure' otherwise, and the last solution committed by the function will then become the current solution of the caller.

Printing solutions. The `print()` function without any arguments will print the last found solution in the format specified in the model's `output` item. Alternatively, `print(s)` provides more fine-grained control over the output. It prints the string `s`, which can be constructed dynamically from values in the solution using calls to `sol`.

2.4 A library of search strategies

Using the MiniSearch built-ins, we have defined and collected the most common meta-search approaches in the standard library `minisearch.mzn`.[4] These meta-search approaches can be used within any MiniZinc model that includes the library. In the next section we present some of these meta-searches in detail.

[4] To enable anonymous access we put the library into the Easychair appendix

3 MiniSearch examples

Despite MiniSearch's limited communication with the solver, it provides enough power to implement many useful complex searches that we illustrate here.

3.1 Lexicographic BaB

In multi-objective optimisation, lexicographic optimisation can be used if the objectives can be ranked according to their importance. The idea is to minimize (or maximize) an array of objectives lexicographically by posting a lexicographic constraint on the array of objective variables. Lexicographic optimisation can be more efficient than the commonly used approach of obtaining a single objective term by multiplying the components of the lexicographic objective with different constants. The reason for this is that a single objective term can leads to large objective values, and potentially overflow.

Analogous to the implementation of branch-and-bound we post the global constraint lex_less so that the next solution is lexicographically smaller than the previous one. Below we show the respective MiniSearch specification.

```
1 function ann: minimize_lex(array[int] of var int: objs) =
2   next() /\ commit() /\ print() /\
3   repeat( scope(
4                post(lex_less(objs, [sol(objs[i]) | i in index_set(objs)])) /\
5                if next() then commit() /\ print() else break endif ) );
```

In line 2 we search for an initial solution and, if successful, repeatedly open a new scope (line 3). Then, we post the lexicographic (lex) constraint (line 4) and search for another solution in line 5. This way, in each iteration of repeat, we add one lex constraint, and all previously added lex constraints are removed due to the scope. This is not required but beneficial, since posting several lex-constraints can cause overhead if many intermediate solutions are found.

3.2 Large Neighbourhood Search (LNS)

Large area neighbourhood search (LNS) [19] is an essential method in the toolkit of CP practitioners. It allows CP solvers to find very good solutions to very large problems by iteratively searching large neighbourhoods (close) to optimality.

Randomized LNS explores a random neighbourhood of a given size, which can be surprisingly effective in practice as long as the neighbourhood size is chosen correctly. Cumulative scheduling is an example of a successful application [7].

The following MiniSearch specification of randomized LNS takes the objective variable, an array of decision variables that will be searched on, the number of iterations, the destruction rate (the size of the neighbourhood) and a time limit for exploring each neighbourhood. We have two scopes: in the global scope, we post BaB style constraints (line 10); in the sub-scope (line 5), we search the neighbourhoods. The predicate uniformNeighbourhood defines the neighbourhood: for each search variable we decide randomly whether to set it to its solution value of the previous solution (line 15).

```
function ann: lns(var int: obj, array[int] of var int: vars,
                  int: iterations, float: destrRate, int: exploreTime) =
  repeat (i in 1..iterations) (
    print("Iteration "++show(i)++"\n") /\
    scope(
      post(uniformNeighbourhood(vars,destrRate)) /\
      time_limit(exploreTime, minimize_bab(obj)) /\
      commit() /\ print()
    ) /\
    post(obj < sol(obj))
  );
predicate uniformNeighbourhood(array[int] of var int: x, float: destrRate) =
  if hasSol() then
    forall(i in index_set(x)) (
      if uniform(0.0,1.0) > destrRate then x[i] = sol(x[i]) else true endif )
  else true endif;
```

Adaptive LNS modifies the neighbourhood size over the course of the iterations, depending on the success of previous iterations. Below is a simple variant, where the neighbourhood size parameter `nSize` (line 3) is step-wise enlarged each time no solution is found (line 9). The `fail` command fails the current conjunction and will hence avoid that the post command on line 10 is executed.

```
 1 function ann: adaptive_lns(var int: obj, array[int] of var int: vars,
 2                            int: iterations, int: initRate, int: exploreTime) =
 3   let { int: nSize = initRate, int: step = 1; } in
 4   repeat (i in 1..iterations) (
 5     print("Iteration "++show(i)++", rate="++show(nSize)++"\n") /\
 6     scope( ( post(uniformNeighbourhood(vars,nSize/100.0)) /\
 7              time_limit(exploreTime, minimize_bab(obj)) /\
 8              commit() /\ print()
 9            ) \/ (nSize := nSize + step /\ fail) )
10     /\ post(obj < sol(obj))      );
```

Custom Neighbourhoods can sometimes be effective if they capture some insight into the problem structure. For instance, in a Vehicle Routing Problem (VRP), we might want to keep certain vehicle tours or vehicle-customer assignments. Below we show such a custom neighbourhood that is easily specified in MiniSearch. The predicate `keepTour` (line 1) posts the tour constraints of a given vehicle number `vNum`. If a solution exists, the neighbourhood predicate (line 4) determines the number of customers of each vehicle (line 7), and then randomly chooses to keep the vehicle's tour (line 8) where a high customer usage results in a higher chance of keeping the vehicle. This predicate can be used instead of the uniform neighbourhood in the LNS specifications above.

```
 1 predicate keepTour(int: vNum) =
 2   forall (i in 1..nbCustomers where sol(vehicle[i]) == vNum)
 3          ( successor[i] = sol(successor[i]) );
 4 predicate vehicleNeighbourhood() =
 5   if hasSol() then
 6     forall (v in 1..nbVehicles) (
 7       let {int: usage = sum(c in 1..nbCustomers) (sol(vehicle[c]) == v) } in
 8       if usage < uniform(0,nbCustomers) then
 9         keepTour(v) % higher usage -> higher chance of keeping the vehicle
10       else true endif             )
11   else true endif;
```

3.3 AND/OR Search

Search in CP instantiates variables according to a systematic variable labelling, corresponding to an OR tree. AND/OR search decomposes problems into a master and several conjunctive slave sub-problems. An AND/OR search tree consists of an OR tree (master problem), an AND node with one branch for each sub-problem, and OR trees underneath for the sub-problems. A prominent example is stochastic two-stage optimisation [17], where the objective is to find optimal first-stage variable assignments (master problem) such that all second-stage variable assignments (sub-problems) are optimal for each scenario. AND/OR search for stochastic optimisation is called *policy based search* [26]. AND/OR search is also applied in other applications, such as graphical models [11].

Below is a MiniSearch example of AND/OR search for stochastic two-stage optimisation. The variables and constraints of each scenario (sub-problem) are added incrementally during search.

```
 1 function ann: policy_based_search_min(int:sc) =
 2    let {
 3      array[1..sc] of int: sc_obj = [ 0 | i in 1..sc ];
 4      int: expectedCosts = infinity;
 5    } in (
 6      repeat (
 7        if next() then  % solution for master
 8          repeat (s in 1..sc) (
 9            scope( % a local scope for each subproblem
10             let {
11                 array[int] of var int: recourse; % subproblem variables
12                 var 0..maxCosts: scenarioCosts;
13             } in (
14               post(setFirstStageVariables() /\ % assign master variables
15                    secondStageCts(s,recourse)) /\ % subproblem constraints
16               if minimize_bab(scenarioCosts) then
17                 sc_obj[s]  := sol(scenarioCosts)
18               else print("No solution for scenario "++show(s)++"\n") /\
19                 break endif
20             ))
21          ) % end repeat
22         /\ if expectedCosts > expectedValue(sc_obj) then
23              expectedCosts := expectedValue(sc_obj)
24            /\ commit()  % we found a better AND/OR solution
25            else  skip() endif
26       else break endif % no master solution
27     ));
28 % the following predicates are defined in the model according to the problem class
29 predicate setMasterVariables();
30 predicate secondStageCts(int: scenario, array[int] of var: y);
31 function int: expectedValue(var int: sc_obj);
```

Lines 3-4 initializes parameters that represent the costs for each scenario/subproblem and the combined, expected cost of the master and subproblems. Line 7 searches for a solution to the master problem (OR tree), and if this succeeds, we continue with the AND search by finding the optimal solution for each subproblem, based on the master solution (line 9): we create each subproblem in a local scope, and add the respective variables (line 11- 12) and constraints (line 15). Furthermore, we set the master variables to the values in the master solution (line 14). Then we search for the minimal solution for the scenario (line 16), and, if successful, store it in `sc_obj` (line 17). If we find a solution for each scenario, then we compute the combined objective (expectedValue) and compare it to the incumbent solution (line 22). If we found a better solution, we store its value (line 23) and commit to it (line 24) and continue in the master scope (line 7). Otherwise, if we find no solution for one of the scenarios (line 18), the master solution is invalid. We therefore break (line 19) and continue in the master problem scope, searching for the next master solution (line 7).

3.4 Diverse solutions

Sometimes we don't just require a satisfying or optimal solution, but a diverse set of solutions [9]. The MiniSearch specification below implements the greedy approximation method that iteratively constructs a set of K diverse solutions:

```
 1 function ann: greedy_maxDiverseKset(array[int] of var int: Vars, int: K) =
 2   let { array[int,int] of int: Store = array2d(1..K, index_set(Vars),
 3                                           [0 | x in 1..K*length(Vars)]),
 4         int: L = 0 % current length
 5   } in
 6   next() /\ commit() /\ print() /\
 7   repeat(
 8     L := L+1 /\ repeat(i in 1..length(Vars)) (Store[L,i] := sol(Vars[i])) /\
 9     if L <= K then
10       scope(
11         let {var int: obj;} in
12         post(obj = sum(j in 1..L,i in index_set(Vars)) (Store[j,i] != Vars[i]))/\
13         maximize_bab(obj) /\ commit() /\ print() )
14     else print(show(Store)++"\n") /\ break endif
15   );
```

The first few lines initialize the `Store`, which will contain the K diverse solutions, as well as the current *length* of the store up to which it already contains solutions. On line 8 the length is increased and the previously found solution is saved. If the length does not exceed K, we construct a new objective on line 12. This objective expresses how diverse a solution is from the previously found solutions using the Hamming distance. This objective is then maximized, resulting in the most diverse solution.

3.5 Interactive Optimisation

Interactive optimisation lets users participate in the solving process by inspecting solutions, adding constraints, and then re-solving. This has been shown to improve the trust of end-users into decision support systems, and a number of successful application have been implemented [2].

MiniSearch supports interactive optimisation by calling MiniZinc built-ins that ask for user input. The MiniZinc library contains functions such as `read_int()` which accept keyboard input. More advanced input facilities can be added through user-defined MiniZinc built-ins that execute arbitrary C++ code, such as consulting a user through a graphical user interface or other means.

The following is an example of an interactive Vehicle Routing Problem solver implemented using MiniSearch. We use a Large Neighbourhood search where in every N-th iteration, the user can chose a vehicle route that should be kept. This code can be used with the LNS implementations described earlier.

```
predicate interactiveNeighbourhood(int: iteration, int: N) =
  if iteration mod N = 0 then
  let { string: msg = "Enter a vehicle tour to keep (0 for none):\n";
        int: n = read_int(msg)
  } in if n > 0 then keepTour(n) else true endif
  else true endif;
```

4 The MiniSearch kernel

This section describes the architecture of the MiniSearch *kernel*, the engine that executes MiniSearch specifications, interacting with both the MiniZinc compiler and the backend solver.

First, let us briefly review how MiniZinc models are solved. The MiniZinc compiler (usually invoked as `mzn2fzn`) takes text files containing the model and instance data and compiles them into FlatZinc, a low-level language supported by a wide range of solvers. Solvers read FlatZinc text files and produce text-based output for the solutions. The compiler generates FlatZinc that is specialised for the capabilities of the particular target solver using a solver-specific *library* of predicate declarations. Version 2.0 of MiniZinc is based on the `libminizinc` C++ library, which provides programmatic APIs, eliminating the need to communicate through text files. The library also defines an API for invoking solvers directly in C++. MiniSearch is built on top of these APIs.

4.1 The MiniSearch Interpreter

The MiniSearch kernel implements an *interpreter* that processes MiniSearch specifications and handles the communication between the MiniZinc compiler and the solver. The interpreter assumes that the solver interface is *incremental*, i.e. variables and constraints can be added dynamically during search. Note that we provide an *emulation layer* (see Sec. 4.2) for solvers that cannot support incremental operations or cannot be accessed through the C++ interface.

The MiniSearch interpreter is a simple stack-based interpreter that maintains the following state: A stack of *solutions*, one for each function scope; a stack of *time-outs* and *breaks*; and a stack of *search scopes*, containing the solver state of each scope. The interpreter starts by compiling the MiniZinc model into FlatZinc, and then interprets each MiniSearch built-in as follows:

- `next` invokes the solver for a new solution; if successful, it replaces the top-most solution on solution stack. If a time-out has been set, the call to the solver is only allowed to run up to the time-out. Returns `true` iff a new solution was found.
- `commit` replaces the parent solution in the stack with the current solution. This commits the current solution into the parent function scope. Returns `true`. Aborts if not in a function call.
- *function calls* duplicate the top of the solution stack before executing the function body. Return `true` if the function committed a new solution.

- `time_limit(l,s)` adds a new time-out `now+l` to the stack, executes `s`, and then pops the time-out. During the execution of `s`, calls to `next` and `repeat` check whether any time-outs `t` have expired (`t` > `now`), and if so they immediately break. Returns whatever `s` returned.
- `repeat(s)` pushes a break scope on the stack, then repeats the execution of `s` until a break has occurred. The `break` construct sets the break condition in the current break scope (similar to a time limit).
- `post(c)` compiles the MiniZinc expression `c` into FlatZinc. The compilation is incremental, i.e., only the result of compiling `c` is added to the existing FlatZinc. The interpreter then adds the newly generated variables and constraints to the current solver instance.
- `scope(s)` creates a new local scope. The current implementation copies the flat model and creates a new instance of the solver in its initial state.
- Other operations (`/\`,`\/`,`print`) are interpreted wrt. to their semantics.

4.2 Emulating advanced solver behaviour

A key goal of this work is to make MiniSearch available for any solver that supports FlatZinc. Current FlatZinc solvers, however, neither allow incremental addition of constraints and variables, nor do they implement the `libminizinc` C++ API. We therefore need to *emulate* the incremental API.

In order to emulate the incrementality, we can implement the dynamic addition of variables and constraints using a restart-based approach, re-running the solver on the entire updated FlatZinc. To avoid re-visiting solutions, after each call to `next` the emulation layer adds a no-good to the model that excludes the solution that was just found. This emulation reduces the requirements on the solver to be simply able to solve a given FlatZinc model.

Furthermore, our emulation links the C++ API to any text-based FlatZinc solver by creating a textual representation of the FlatZinc, calling an external solver process, and converting its textual output back into `libminizinc` data structures. Using this emulation of the incremental C++ solver API, any current FlatZinc solver can be used with the MiniSearch kernel.

4.3 Built-in primitives

Solvers can declare native support for a MiniSearch search specification. For every MiniSearch call such as `f(x,y,z)`, the kernel will check whether the declaration of `f` has a function body. If it does not have a function body, the function is considered to be a solver built-in, and it is executed by passing the call to a generic `solve` function in the C++ solver API. This is similar to the handling of global constraints in MiniZinc, where solvers can either use their own primitives for a given global constraint, or use the respective MiniZinc decomposition of the global constraint. This way solvers can easily apply their own primitives, but can use alternatives if they do not support a certain feature.

5 Experiments

In our experimental evaluation, we analyse MiniSearch on practical examples to study the efficiency and compare the overhead of MiniSearch to native implementations. Furthermore, we study the benefits of specialised, heuristic MiniSearch approaches to standard branch-and-bound optimisation; the only available option in solver-independent CP modelling languages.

5.1 Experimental Setup

The problems and instances are taken from the MiniZinc benchmarks repository[5] and will be part of the example collection in the next MiniZinc release.

Experiments are run on Ubuntu 14.04 machines with eight i7 cores and 16GB of RAM. The MiniSearch kernel is based on the MiniZinc 2.0.1 toolchain and will become part of the MiniZinc 2 distribution. The native Gecode interface and incremental C++ interface were implemented using the latest Gecode source code (version 4.3.3+, 20 April 2015). The FZN solvers used are: Gecode (20 April 2015), Choco 3.3.0, Or-tools source (17 February), and Opturion CPX 1.0.2. All solvers use the variable/value ordering heuristics that is part of the MiniZinc model.

5.2 Overhead of different MiniSearch interfaces

First, we study the comparative overhead of MiniSearch with respect to the different interface levels. We do this by analysing the performance of a meta-search approach on the same solver, Gecode. We compare the following four approaches: the solver's *native* approach without MiniSearch either through the C++ API (Nat) or FlatZinc (Nat-F), the *incremental API* interface to MiniSearch (Inc), the *non-incremental text-based* (MS-F) interface to MiniSearch. We use Rectangle Packing [20] as problem benchmark, and standard BaB as meta-search, since it is available natively in Gecode.

The results are summarized in Table 2, columns 2–5. It show the runtimes (in seconds) taken to find the optimal solution for the different approaches using Gecode. As one can expect, the native (API) approach is fastest, though MiniSearch through the incremental API interface performs quite similary. Moreover, MiniSearch through the FZN interface is, as expected, the slowest, but only by a small factor. These results are very promising, since they show that the overhead of the MiniSearch interfaces is low.

In addition, we analyse the overhead of MiniSearch for other FlatZinc solvers in Table 2, columns 6–11. We ran the Rectangle Packing problem for the FlatZinc solvers or-tools, Choco, and Opturion CPX on a MiniZinc model using both native branch-and-bound (Nat-F), and MiniSearch branch-and-bound (MS-F), both through the FlatZinc interface. Overall the MiniSearch FlatZinc interface, while slower, still gives acceptable performance.

[5] https://github.com/MiniZinc/minizinc-benchmarks

Table 2. Rectangle Packing: Solving times (sec), using BaB. Comparing FLATZINC-solvers with MINISEARCH BaB through FLATZINC (MS-F) MINISEARCH BaB through incremental API (Inc), native BaB with FLATZINC (Nat-F) and native BaB (Nat).

Rectangle	Gecode				or-tools		choco		Opturion CPX	
Size (N)	MS-F	Inc	Nat	Nat-F	MS-F	Nat-F	MS-F	Nat-F	MS-F	Nat-F
14	0.35	0.13	0.10	0.16	0.84	0.15	6.19	0.94	0.35	0.45
15	0.52	0.28	0.30	0.35	0.66	0.32	5.77	1.05	0.46	0.19
16	0.86	0.48	0.47	0.54	1.01	0.46	8.38	1.41	0.79	0.23
17	4.25	3.01	2.86	4.67	5.18	4.60	13.83	5.56	0.91	0.37
18	37.60	43.50	36.70	38.10	40.47	40.97	31.82	23.79	6.68	5.69
19	81.50	45.20	42.77	44.90	58.29	33.63	51.27	21.88	16.11	7.06
20	105.00	102.00	97.27	103.00	97.00	103.65	82.98	63.09	6.81	5.20
21	493.00	491.00	435.00	470.00	497.00	403.74	201.64	189.79	100.85	69.01

5.3 Heuristic search

MINISEARCH strategies are not restricted to complete search. For example, different variants of Large Neighborhood Search can be tried. To demonstrate the benefits of having heuristic search at hand, we compare the Randomized LNS approach from Section 3.2 with the standard out-of-the-box branch-and-bound approach for a selection of FLATZINC solvers on the Capacitated Vehicle Routing problem (CVRP).

We run each solver on (1) the standard MINIZINC model that uses standard branch-and-bound, and (2) the MINIZINC model with the MINISEARCH specification of Randomized LNS. We use the Augerat et al [1] CVRP instance sets A,B and P, where the 27 Set-A instances contain random customer locations and demands, the 22 Set-B instances have clustered locations and the 23 Set-P instances are modified versions of instances from the literature. The LNS approach uses a destroy rate of 0.3 (the neighbourhood size is 30%) and an exploration timeout (for each neighbourhood) of 5 seconds. Both the LNS and BaB approach have an overall 2 minute timeout, and we report the quality of the best solution found on time-out.[6]

We can see that LNS provides better solutions than BaB, and Table 3 reports the *average* improvement of the objective of LNS over BaB on the three instance sets. We observe an improvement between 8-15% which is considerable within the given timelimit. Furthermore, we see that the improvements are similar for all solvers across the instances sets, in particular for the Set-B instances. This demonstrates the added value of MINISEARCH search strategies to the existing out-of-the-box optimisation approaches of standard modelling languages.

6 Related work

The starting point for MINISEARCH is search combinators [18], and MINISEARCH is indeed a combinator language. The MINISEARCH language is more restricted

[6] We report detailed results in the Easychair appendix for our reviewers

Table 3. Average relative improvement of MiniSearch LNS over standard BaB for the achieved objective within a timelimit for different solvers. The figures are averages over the Augerat et al [1] CVRP instance sets A,B and P.

CVRP Instance-Set	Gecode	or-tools	choco	Opt. CPX
Set-A	11.60%	11.76%	11.17%	12.11%
Set-B	13.38%	11.82%	12.62%	14.92%
Set-P	9.78%	10.53%	7.98%	11.35%

than search combinators. In particular, search combinators can interact with the search at every node of the tree, essentially *replacing* the solver's built-in search, while MiniSearch only interacts with the solver's search at the granularity of solutions. Examples of the additional expressivity of search combinators are strategies based on average depth of failure, or custom limits based on how many search nodes with a particular property have been encountered, or heuristics such as limited discrepancy search [8] and iterative deepening. All of these strategies are beyond the reach of MiniSearch. Many of the examples for search combinators are however expressible with MiniSearch, as witnessed by the library of strategies presented in this paper, and indeed MiniSearch has more natural handling of solutions than search combinators.

Neither search combinators nor MiniSearch allow interaction with the internal data structures of the solver, in particular complex problem specific branch selection procedures that make use of the current domain. These kinds of search strategies are important, but cannot be implemented without deep hooks into the solver.

Possibly the closest system to MiniSearch is OPL Script [24], a scripting language for combining models. It essentially also communicates to solvers via models and solutions. Distinct from MiniSearch is that it is object-oriented and hence allows the construction and modification of multiple models, as opposed to our simpler mechanisms. The language is tightly linked to its CP/MIP solver ILOG CPLEX CP Optimizer, and can communicate more information than just variables and constraints including dual values, bases, and various statistics. This allows some efficient meta-searches that are not possible with MiniSearch.

AMPL Script [4] is similar in many respects to OPL script, though a more basic scripting language, not supporting function definitions for example. It does not support scopes, instead allowing addition of variables and constraints, and dropping and restoring constraints from the model. It also has specific commands to: modify a parameter value, fix a variable to its current value or unfix a fixed variable, or just reset the entire model. It interacts with the solver through requests to solve, which continue until some termination condition arises, like optimality proven or a timeout. Hence its natural interaction level is more like `minimize_bab()` than `next()`.

Objective CP [25] allows complex meta-search to be specified at the model level, and then executed by any of the Objective CP solvers, for example a CP search can be implemented using a MIP solver. So it also provides solver-independent search, but through a much more fine grained interface, allowing interaction at each search node. Because of this more complex meta-searches are possible than with MiniSearch. But the much finer grained interface means adding a new solver to Objective CP is complex, in particular supporting the reverse mapping of information from solver objects to model objects.

Any search that can be implemented in MiniSearch could also be implemented directly in a solver such as Gecode, Choco, Comet, or CP Optimizer, although the implementation burden for doing so could be significant. Similarly, APIs for CP embedded in scripting languages such as Numberjack [10] (through Python) could re-implement MiniSearch style search straightforwardly. The benefit of MiniSearch is providing a concise language for many searches, a library of common (meta-)search combinators, and a uniform language for specifying them. Meta-search languages are easier to use when they extend the language for expressing constraints, since this is often an important component of meta-search.

7 Conclusion and Future Work

In this paper we present MiniSearch, a solver-independent lightweight language to specify meta-search in MiniZinc models. The first contribution of MiniSearch is its *expressiveness*: we showed how to formulate different complex search approaches in MiniSearch, such as Large Neighbourhood Search or AND/OR search. Since MiniSearch is based on the MiniZinc modelling language, it is possible to use predicates and functions to specify custom neighbourhoods for a given problem, which even enables the implementation of interactive optimisation approaches where the user guides the search process.

The second contribution is the *minimal interface* of MiniSearch. Since the communication between the solver and MiniSearch is only necessary at every *solution* (instead of at every search *node* as in search combinators), it is easy for solver developers to implement a native interface. Furthermore, in our experiments we observed that both the incremental and the text-based interface provide a satisfactory performance.

The third and most important contribution is *solver-independence*. In contrast to existing search languages, MiniSearch is already supported by any solver that can process FlatZinc, which is the majority of CP solvers. This allows the users to test different solvers with complex meta-searches without having to commit to one single solver.

For future work, we plan to extend MiniSearch with parallelism, so that independent searches can be executed in parallel. This would enable us to express advanced parallel meta-searches such as Embarrassingly Parallel Search [16] in MiniSearch. Furthermore, we plan to integrate portfolio search, so that different search functions can be executed by different solvers.

References

1. Augerat, P., Belenguer, J., Benavent, E., Corbern, A.and Naddef, D., Rinaldi, G.: Computational results with a branch and cut code for the capacitated vehicle routing problem. Technical Report 949-M, Universite Joseph Fourier, Grenoble (1995)
2. Belin, B., Christie, M., Truchet, C.: Interactive design of sustainable cities with a distributed local search solver. In: Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings. pp. 104–119 (2014)
3. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings. p. 971 (2003)
4. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Cengage Learning (2002)
5. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence : A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
6. Gent, I.P., Miguel, I., Rendl, A.: Tailoring solver-independent constraint models: A case study with Essence' and Minion. In: Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21, 2007, Proceedings. pp. 184–199 (2007)
7. Godard, D., Laborie, P., Nuijten, W.: Randomized large neighborhood search for cumulative scheduling. In: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA. pp. 81–89 (2005)
8. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the 14th IJCAI. pp. 607–613 (1995)
9. Hebrard, E., Hnich, B., O'Sullivan, B., Walsh, T.: Finding diverse and similar solutions in constraint programming. In: Veloso, M.M., Kambhampati, S. (eds.) AAAI. pp. 372–377. AAAI Press / The MIT Press (2005)
10. Hebrard, E., O'Mahony, E., O'Sullivan, B.: Constraint programming and combinatorial optimisation in Numberjack. In: Lodi, A., Milano, M., Toth, P. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Lecture Notes in Computer Science, vol. 6140, pp. 181–185. Springer Berlin Heidelberg (2010)
11. Marinescu, R., Dechter, R.: AND/OR branch-and-bound search for combinatorial optimization in graphical models. Artif. Intell. 173(16-17), 1457–1491 (2009), http://dx.doi.org/10.1016/j.artint.2009.07.003
12. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., de la Banda, M.G., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3), 229–267 (2008)
13. Michel, L., Hentenryck, P.V.: The Comet programming language and system. In: Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings. pp. 881–881 (2005)
14. MiniZinc challenge. http://www.minizinc.org/challenge.html
15. 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)

16. Régin, J., Rezgui, M., Malapert, A.: Embarrassingly parallel search. In: Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings. pp. 596–610 (2013)
17. Ruszczyński, A., Shapiro, A.: Stochastic Programming. Handbooks in operations research and management science, Elsevier (2003)
18. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.J.: Search combinators. Constraints 18(2), 269–305 (2013), http://dx.doi.org/10.1007/s10601-012-9137-8
19. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Principles and Practice of Constraint Programming-CP98, pp. 417–431. Springer (1998)
20. Simonis, H., O'Sullivan, B.: Search strategies for rectangle packing. In: Stuckey, P. (ed.) Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, vol. 5202, pp. 52–66. Springer Berlin Heidelberg (2008)
21. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008-2013. AI Magazine 35(2), 55–60 (2014)
22. Stuckey, P.J., Tack, G.: MiniZinc with functions. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings. pp. 268–283 (2013)
23. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge, MA, USA (1999)
24. Van Hentenryck, P., Michel, L.: OPL Script: Composing and controlling models. In: Apt, K., Monfroy, E., Kakas, A., Rossi, F. (eds.) New Trends in Constraints, Lecture Notes in Computer Science, vol. 1865, pp. 75–90. Springer Berlin Heidelberg (2000)
25. Van Hentenryck, P., Michel, L.: The objective-cp optimization system. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, vol. 8124, pp. 8–29. Springer Berlin Heidelberg (2013)
26. Walsh, T.: Stochastic Constraint Programming. In: van Harmelen, F. (ed.) ECAI. pp. 111–115. IOS Press (2002)