Solution Checking with MiniZinc

Carleton Coffrin[1], Siqi Liu[2], Peter J. Stuckey[2], and Guido Tack[3]

[1] Los Alamos National Laboratory, Los Alamos, New Mexico, USA
[2] Data61, CSIRO and the University of Melbourne, Victoria, Australia
[3] Data61, CSIRO and Monash University, Victoria, Australia

Abstract. A critical need in teaching discrete optimization is to give feedback to learners regarding what is wrong about the "solutions" they provide. MiniZinc provides a solver independent modelling language for expressing discrete optimization problems. In this paper we develop a methodology for using MiniZinc to easily create *solution checkers* that give meaningful feedback to learners. The checkers can be used independently of the technology the learners use to actually solve the problem. The checkers can be included (perhaps in an obfuscated state) in a MiniZinc project file, to allow standalone projects that learners can complete locally through the MiniZinc IDE with meaningful feedback. The checkers can also used as an online service enabling learners to submit solutions for checking and feedback via the MiniZinc IDE or a Python script. We have made use of this checking methodology for automatic feedback and grading of over 65,000 student submissions running in the Coursera Massive Open Online Courseware platform. Automated solution checking provides a pathway to self-directed learning for discrete optimization which we believe is vital to ensure that the technology is used widely.

1 Introduction

Building a correct model for a complex discrete optimization problem is a challenging task. As the complexity and size of discrete optimization problems that we have to solve grows, so does the need to equip the next generation of optimization professionals with the skills and tools to tackle these complex problems.

In terms of tools, high level modelling languages such as IBM OPL [13], MiniZinc [10] or Essence [6], as well as modelling/solver hybrids like Objective-CP [14] or SCIP [1] have been developed that lower the barrier to entry into discrete optimization. Independent of the modelling method used, to support learning how to build effective optimization solutions we still need to make these tools easier to use and *crucially* easier to debug.

Debugging an optimization model is a challenging task. There are three frequent issues that arise when modelling a problem:

- No answer returned by the solver which runs "forever"
- The solver returns that the problem is unsatisfiable when, in fact, it is not
- The solver returns an incorrect answer

The first issue is the most difficult to handle and usually relies on looking at small instances, or examining runtime traces to understand propagation and search behaviour. There is considerable work in runtime visualization, tracing, and logging in order to tackle this challenge (see e.g. [4,11]).

The second issue is also challenging. Essentially we need to help explain why there are no solutions. This can be helped by having known correct solutions, and also by returning minimal unsatisfiable subsets to narrow down the incorrect part of the model (see e.g. [7]). Essentially the user needs to track down which constraints are eliminating correct solutions. Efficient ways to track down which constraints of a complex model are problematic is an interesting topic of research (see e.g. [9]).

The third issue perhaps appears to be the easiest to deal with since we have a *witness* of the incorrect behaviour, but even then for complex optimization problems, it may not be clear why a purported "solution" is not actually a solution. This is the problem we focus on in this paper:

A *learner* (a user who is studying combinatorial optimization) is working on a task set by an *instructor*, where the goal is to model and solve a certain combinatorial optimization problem. The learner generates *candidate solutions* to the problem, and we want to *check* whether those candidates are in fact solutions, and provide *meaningful feedback* to the learner if they are not.

Note that the problem we address is distinct from the checking of solutions returned by LP and MIP solvers (see e.g. [2,5]), which may be wrong because of floating point computation. Rather than just checking or repairing a "solution", we wish to explain why it is *not* a solution in a way that helps learners understand their mistakes, and therefore, learn from them.

We introduce a methodology for creating *solution checkers*, using MiniZinc infrastructure, so that we can build automatic tools for

- Giving detailed feedback about erroneous candidate solutions
- Grading an optimization project candidate solution

We describe how we can make use of and extend existing MiniZinc tools to provide self contained modelling projects with built-in checkers and feedback, and how we can use this as an online resource for checking candidate solutions of projects.

2 Building Solution Checkers

In this section we develop a MiniZinc-based methodology for building solution checkers which give meaningful feedback to learners when they check an erroneous candidate solution.

It is important to emphasise that this checking system does not impose any restrictions on the modelling or solving technology used by the learner. In this work we will use MiniZinc for building example solutions, but this is not required for the use of the solution checkers considered here, although some things will be easier if we are checking MiniZinc models.

2.1 Solution Checking by Correct Model

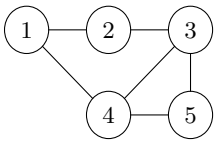
The simplest way to use a MiniZinc model as a solution checker is to encode values of the candidate solution's decision variables in a data file (`.dzn`) format. The resulting data file can then be read in with a correct model of the original problem definition.

Example 1. Consider a simple graph coloring problem, given n nodes, and a list of m edges, color each node with at most k colors such that no two adjacent nodes are colored the same. The aim is to minimize the number of colors used. A correct MiniZinc model `color.mzn` for this problem is

```
1  int: n;
2  set of int: NODE = 1..n;
3  int: m;
4  set of int: EDGE = 1..m;
5  array[EDGE] of NODE: from;
6  array[EDGE] of NODE: to;
7  int: k;
8  set of int: COLOR = 1..k;
9  array[NODE] of var COLOR: x; % color of each node
10 var NODE: nc;                % number of colors used
11 constraint forall(e in EDGE)(x[from[e]] != x[to[e]]);
12 constraint nc = max(x);
13 solve minimize nc;
```

For this example, we will consider a small data file `small.dzn` representing the graph shown on the right:

```
1 n = 5;
2 m = 6;
3 k = 4;
4 from = [1,1,2,3,3,4];
5 to   = [2,4,3,4,5,5];
```



We can check a candidate solution `x = [1,2,3,3,2]; nc = 3;` (written in `.dzn` format) to the model by just adding it to the data used in solving (here using the inline data file flag `-D`)

```
minizinc color.mzn small.dzn -D"x = [1,2,3,3,2]; nc = 3;"
```

The solver will return `=====UNSATISFIABLE=====` indicating that the candidate solution is not correct. We may get a line number that indicates where the model is unsatisfiable, as detected by the MiniZinc system (before it reaches the solver). In this case, the MiniZinc system points at line 11, when `e = 4`, declaring an inconsistency in the `!=` operator expression. □

Note that simply taking an existing model as a solution checker is not always correct. In particular a model may include symmetry or dominance breaking constraints (e.g. [8,3]) which are not *required* of a solution, and are simply there to improve solving behaviour. If a learner's model does not include the same constraints, their correct candidate solutions may be rejected, as shown in the following example.

Example 2. A better model for the graph coloring problem includes a constraint to remove value symmetries, which in MiniZinc could be expressed as

```
include "value_precede_chain.mzn";
constraint value_precede_chain([i | i in COLOR],x);
```

But if we use this model for checking, the system will incorrectly return `=====UNSATISFIABLE====` for correct candidate solutions such as `x = [2,1,2,1,3]; nc = 3;`. □

So we must be careful when using an existing model as a checker, making sure that it only encodes the constraints of the problem, and nothing further.

2.2 Collecting Solutions to be Checked

In order to use solution checking we require that the learner's model outputs solutions in the correct format. Since we want to use MiniZinc for checking, candidate solutions need to be represented as a MiniZinc data file. If the learner is not required to use MiniZinc, then there needs to be clear instructions for the format of the output, so that it matches the definitions of decision variables in the MiniZinc model that is used for solution checking.

If the project is in MiniZinc we can make use of MiniZinc infrastructure to produce standardized output automatically. Frequently when constructing a project for students, the instructor will create a starting point for the model file which includes the declarations required to read the data. In order to ensure that output from the model is of the correct format, they can simply add the decision variables required for solution checking to this starting model, with an annotation `check_output`.

Example 3. A starting model file `mycolor.mzn` for the graph coloring problem of Example 1 could be,

```
int: n;                                      % data declarations: n no of nodes
set of int: NODE = 1..n;                     % NODE set of nodes
int: m;                                      % m: no of edges
set of int: EDGE = 1..m;                     % EDGE: set of edges
array[EDGE] of NODE: from;                   % source of edge
array[EDGE] of NODE: to;                     % dest of edge
int: k;                                      % k: no of colors
set of int: COLOR = 1..k;                    % COLOR: set of colors

array[NODE] of var COLOR: x :: check_output; % color of each node
var NODE: nc;               :: check_output; % number of colors used
```

As long as the learner does not modify the decision variable declarations, the output from this model will always have the correct form for checking. □

We can run MiniZinc with the `--output-mode dzn` to override any `output` item in the user's model, and output those variables annotated as `check_output` in a correct MiniZinc data format. For example

```
minizinc --output-mode dzn mycolor.mzn small.dzn
```

will output solutions of `mycolor.mzn` in the correct format for checking.

In any optimization problem we will almost always want to check that the objective value computed by the learner's model is consistent with its definition

in the problem statement. We have observed that incorrect objective evaluation is a good early warning signal that the learner has significant misconceptions about the problem specification. We can use MiniZinc to extract the objective explicitly, since if the model includes an optimization objective this will be captured in a default variable `_objective`. We can use this for evaluating the candidate solutions without having to specify a name for a specific decision variable. We can ensure that the default variable appears in the output by adding the `--output-objective` option to MiniZinc.

Example 4. We could remove the line declaring `nc` from the `mycolor.mzn` file and build a checking model using `_objective`. When running the learner's model for checking we run

```
minizinc --output-mode dzn --output-objective mycolor.mzn small.dzn
```

which prints out the candidate solutions in the correct format. □

Note that requiring MiniZinc to be run in a particular way in order to create the right output for checking may seem difficult for the student, but we can hide all of this from the learner if they are using a checking-aware framework, such as the MiniZinc IDE or a grading submission script.

2.3 Better Error Messages

Giving feedback to learning modellers is vital for them to improve their modelling skills, and we can generate much better feedback to explain why a candidate solution is incorrect than the simple checking described above. So we advocate that building a *checking model* that can give easily interpretable error messages is essential for improving the learning experience.

We now demonstrate how we can build a checking MiniZinc model to explain errors in a candidate solution for a given discrete optimization problem. The assumption is that the candidate solution is available in MiniZinc data format. We will build a checking model that will consider the solution as fixed parameters, and use the assertion checking facilities of MiniZinc to output suitable error messages.

The first thing we must gracefully handle is errors in the format.

Example 5. Consider the erroneous solution `x = [1,2,4]; nc = 4;` for the coloring problem and `small.dzn`. Running,

```
minizinc color.mzn small.dzn -D"x = [1,2,4]; nc = 4;"
```

gives an error message about an "index set mismatch in variable declaration for x", since the array should have five elements instead of the given three. Similarly, running

```
minizinc color.mzn small.dzn -D"x = [2,1,2,1,3]; nc = 9;"
```

gives an error message about a model inconsistency in the second to last line of the model (because `nc` is not equal to the maximum of the `x` array), which is difficult to communicate to the learner without exposing the entire model `color.mzn`. □

In order to create better error messages than the default messages from MiniZinc we must be able to read in their values from the candidate solution without causing a MiniZinc error, which may be hard to decipher. To do so we relax restrictions on the type of values they can take.

Example 6. We allow `x` to take any array of integer values, and `nc` to take any integer value. Thus the declarations for `x` and `nc` in the checking model `checkc.mzn` are

```
⟨ data declarations ⟩
array[int] of int: x;
int: nc;
```

We can then explicitly check that the index set of `x` and its contents are correct. And we can check that `nc` is given an appropriate value.

```
constraint assert(index_set(x) = NODE,
                  "x array must be indexed from 1..\(n)\n");
constraint forall(i in index_set(x))
                 (assert(x[i] in COLOR,
                         "x[\(i)] is not a correct COLOR (1..\(k))\n"));
constraint assert(nc in COLOR,"nc does not take a value in 1..\(k)\n");
```

Running

```
minizinc checkc.mzn small.dzn -D"x = [1,2,4]; nc = 4;"
```

we now get an explicit error message "x array must be indexed from 1..5."

```
minizinc checkc.mzn small.dzn -D"x = [2,1,2,1,3]; nc = 9;"
```

gives an explicit error message "nc does not take a value in 1..4." □

Since checking that the "solution" fits the basic type definition is the first thing we need to do for almost any solution checkers it is worth building a library of tests for doing so. We introduce a MiniZinc function `check` which plays the same role as assert, but rather than aborting returns a Boolean value. We introduce a library function `check_int`(x, S, n) which checks that integer $x \in S$ and if not outputs an error message, where n is the string name of variable x. Similarly `check_array`(x, l, S, n) checks that a fixed array of integers is of length l and that each element of the array x has a value in S, and otherwise outputs an error message using n as the name of the x variable.

```
test check(bool: b, string: s) =
     if b then true else trace("ERROR: " ++ s, false) endif;

test check_int(int: x, set of int: vals, string: name) =
     check(x in vals, "integer " ++ name ++ " is not in values \(vals)\n");

test check_array(array[int] of int: x, int: length,
                 set of int: vals, string: name) =
     check(length(x) = length,  name ++ " is not of length \(length)\n") /\
```

```
    forall(i in index_set(x))
          (check(x[i] in vals, name ++ "[\(i)] = \(x[i]) " ++
                               "is not in values \(vals)\n"));
```

Example 7. Using the library, we can rewrite `checkc.mzn` as

```
constraint check_int(nc, COLOR, "nc");
constraint check_array(x, n, COLOR, "x");
```

Note that the checking model does not very gracefully handle candidate solutions that are not correct MiniZinc data format since it never actually gets to run without correct input. For example

```
minizinc checkc.mzn small.dzn -D"x = [2,1,2,1,3; nc = 9;"
```

gives a syntax error at the first semicolon in the inlined data due to the missing bracket. Similarly if a parameter is omitted, e.g.

```
minizinc checkc.mzn small.dzn -D"x = [2,1,2,1,3];"
```

gives a symbol error saying `nc` is not defined and should be given in a data file. In these cases we must rely on the syntax error messages from MiniZinc. Still these format errors are easy to understand and fix for a learner.

Once we have determined that the solution fits its type declarations, next we must determine that the constraints are satisfied. We can rewrite the constraints as assertions with an attached error message, in a format very similar to the correct model.

Example 8. For the coloring problem we can write assertions to check the constraints as follows:

```
constraint forall(e in EDGE)
                 (check(x[from[e]] != x[to[e]],
           "The nodes \(from[e]) and \(to[e]) (endpoints of edge \(e))" ++
           " have the same color \(x[from[e]])\n");
constraint forall(i in NODE)
                 (check(nc >= x[i],
                  "nc = \(nc) is less than color \(x[i]) of node \(i)\n"));
constraint check(nc <= max(x),"nc = \(nc) is greater than max(\(x))\n");
```

Then for example running

```
minizinc checkc.mzn small.dzn -D"x = [1,2,3,3,2]; nc = 3;"
```

we get an error message "The nodes 3 and 4 (endpoints of edge 4) have the same color 3." And for example running

```
minizinc checkc.mzn small.dzn -D"x = [2,1,2,1,3]; nc = 2;"
```

we get an error message "nc = 2 is less than color 3 of node 5." Note how we can have a better error message by "decomposing" the max constraint rather than just stating that `nc` is not the max of `x`. □

2.4 Hidden Variable Consistency

For many models, the decision variables which describe the solution may not be the variables that are natural to describe the constraints. There may be other internal variables which are functionally defined by the decision variables, and which are likely to be used in the model for building constraints and/or are much more natural for describing the correctness of a candidate solution.

Example 9. Consider the problem of lining up n people numbered from 1 to n for a photo in a single line. We want to make sure that there are no more than 2 people of the same gender (male, female or other) in sequence in the line. We want to minimize the total distance between the pairs of people who are numbered consecutively. The decisions are an array *pos* which for each person gives their position in the line. A correct model for this is

```
int: n;                          % data declarations: n number of people
set of int: PERSON = 1..n;       % PERSON set of people
enum GENDER = { M, F, O };       % GENDER set of genders
array[PERSON] of GENDER: g;      % the gender of each person
set of int: POSN = 1..n;         % POSN set of positions

array[PERSON] of var POSN: pos;  % decisions: a position for each person

array[POSN] of var PERSON: who;  % view: a person for each position
include "inverse.mzn";
constraint inverse(pos,who);     % channel from decisions to view
constraint forall(i in 1..n-2)
                (g[who[i]] != g[who[i+1]] \/ g[who[i+1]] != g[who[i+2]]);
solve minimize sum(i in 1..n-1)(abs(pos[i] - pos[i+1]));
```

Note that a critical part of learning how to model this problem is to realise that it is worth having the inverse view of the variables `who` in order to easily specify that no more than two people of the same gender can be adjacent. Hence the output specification *should not* include these variables, otherwise the students are effectively given the key to the solution. □

The use of hidden variables makes checking harder in two ways. First we may need to invoke a solver to generate the values of these hidden variables. Second, given an incorrect candidate solution, there may be no possible value for these hidden variables, hence we have to guard the checking to ensure we do not assume that there is some value for the hidden variables.

Example 10. Consider a data file `p9.dzn` for the photo problem of Example 9

```
n = 9;
g = [M,M,M,M,F,F,F,M,M];
```

and the erroneous candidate solution `pos = [1,2,3,4,5,6,7,2,9]; _objective = 25;`. If we simply assert `inverse(pos,who)` then this constraint will fail since there is no inverse of this position array, because persons 2 and 8 are both at position 2. Hence the checker must guard the inverse constraint, and only when that succeeds use the computed values of the `who` variables for checking the solution. In order to ensure we have fixed values of the `who` variables (which are now computed by a solver), we must postpone the checking of them until the output

item of the model, and indeed for simplicity we will move all checking to the output statement.

A checking model for the photo problem `checkp.mzn` is as follows:

```
⟨ data declarations ⟩
array[int] of int: pos;
int: _objective;
array[POSN] of var int: who;
include "inverse.mzn";
include "alldifferent.mzn";
constraint if forall(i in index_set(pos)(pos[i] in POSN) /\
              alldifferent(pos) then inverse(pos,who)
           else forall(i in 1..n)(who[i] = 1) endif;
solve satisfy;

output [if check_array(pos, n, POSN, "pos") /\
           check_alldifferent(pos,"pos") /\
           check_array(fix(who), n, PERSON, "who") /\
           forall(i in 1..n-2)
                 (check(g[fix(who[i])] != g[fix(who[i+1])] \/
                        g[fix(who[i+1])] != g[fix(who[i+2])],
                        "three people of the same gender \(g[fix(who[i])])"
                        ++ " in positions \(i)..\(i+2)\n")) /\
           let { int: obj = sum(i in 1..n-1)(abs(pos[i] - pos[i+1])); } in
           check(obj = _objective, "calculated objective \(obj) " ++
                 "does not agree with computed value \(_objective)\n")
        then "CORRECT: All constraints hold"
        else "INCORRECT" endif];
```

The hidden variables `who` are declared with a relaxed type. We assert the inverse constraint only if all the `pos` variables have the right kind of value and are alldifferent, otherwise we simply give default values to the `who` variables. All the checking is moved to the output statement. Here we rely on the lazy (shortcut) evaluation of conjunction, so that if the `check_alldifferent` call fails then the checking of the gender constraint is not executed. Note how we use the `fix` function, which allows us to coerce the variable type `who` results into their fixed value. This succeeds since within the output statement we can guarantee that all variables have been assigned a fixed value from the solver.

Running

```
minizinc checkp.mzn p9.dzn \
         -D"pos = [1,2,3,4,5,6,7,8,9]; _objective = 8;"
```

gives the error message "three people of the same gender M in positions 1..3."

□

The above example makes use of the `check_alldifferent` test not defined in the code. We can build a library of checkers for global constraints analogous to the global constraint definitions, for reuse in any checking model.

Example 11. The check for alldifferent is defined below. Note that it tries to give the most detailed description of the error as possible.

```
test check_alldifferent(array[int] of int: x, string: name) =
     forall(i, j in index_set(x) where i < j)
           (check(x[i] != x[j], name ++ "[\(i)] = \(x[i]) = " ++
                  name ++ "[\(j)] when they should be different\n"));
```

For example running

```
minizinc checkp.mzn p9.dzn \
         -D"pos = [1,2,3,4,5,6,7,2,9]; _objective = 25;"
```

gives the response "pos[2] = 2 = pos[8] when they should be different." □

Another important part of solution checking is illustrated in the example above. We certainly have to check that the objective value determined by the learner is actually correct given their solution to the decisions. So we recalculate the value and check it with their computed objective value (in this case gathered automatically from the model using `_objective`). In earlier projects we accepted as a solution just the decision variables from the user, and simply calculated the objective value locally, to be used for grading, but often calculating the correct objective is a crucial part of building the model. Also when grading solutions we need to be able to explain to a student why they received the grade they did dependent on objective value, hence they need to expose their view of what the objective is to explain the grading. For optimization problems we therefore always collect the user's computed objective value as part of the candidate solution.

Because we need to check hidden variables whose values will only be available in the output statement, we move all the checks to the output statement. This has the ancillary benefit that we can now also determine in a single place if any checks have failed, as shown in the example above. This will become more important when we wish to also grade a solution.

2.5 Summary

In summary we can create an effective and informative solution checker for a discrete optimization problem by following the steps below:

- Build a MiniZinc model that takes the decision variables *and objective value* as fixed arguments.
- Weaken the type/domain of the decision variables (and objective) to the broadest possible type/domain, in order that we do not get a default MiniZinc error about assigning a wrong value to a type.
- Add variable declarations for hidden variables required to effectively express some constraint of the model
- Constrain the hidden variables, in such a way that regardless of the input decision variables there is always a solution to the hidden variables
- Build an output statement which contains the following checks:
  1. Explicit checks that the type/domain of decisions (and objective) are correct with respect to the specification.
  2. Explicit checks for constraints on the decision variables, which point out the exact point where the constraint is violated as much as possible.
  3. Explicit checks for constraints expressed on the hidden variables (using `fix`) so they can be treated as fixed.
  4. Recalculate the true objective value from the decisions, and explicitly check that this agrees with the input objective value.
- Output "correct" if all the checks succeed or "incorrect" otherwise.

3 Using Solution Checkers

In this section we discuss various ways we can make use of solution checkers to enhance the learning of discrete optimization modelling. Again while we concentrate on MiniZinc, only Section 3.1 and Section 3.4 below rely on the learner using MiniZinc for their project.

3.1 Self Contained Modelling Projects

The MiniZinc IDE supports the concept of a *project*, which is simply a set of related model and data files. It is easy to extend this to include checking models as well. We have built an extension of the IDE that reserves the filename extension `.mzc` for MiniZinc checking model files.

To build a self-contained modelling project we simply bundle a description of the problem (including a description of the input format as well as the required output format for the solution) together with some data files, and perhaps a starting model file (this is particularly useful for beginners to avoid confusion from trivial errors). Indeed we can include in the model files declarations for the data and the output decision variables if we desire.

To include a solution checker we simply bundle a checking model *project*`.mzc` in the project, where *project*`.mzn` is the name of the model the learners need to build.

We have extended the IDE such that the existence of the checking model *project*`.mzc` makes available a new button labelled "CHECK" in the IDE whenever we are visiting the file *project*`.mzn`. Hitting the "CHECK" button as opposed to the "RUN" button will run the model with the `--output-mode dzn` and `--output-objective` flags, and pass each output candidate solution to the checking model in order to generate feedback output, which is appended to the output of the learner's project model.

The idea of self contained projects can still be used if learners are not required to use MiniZinc, but can solve a discrete optimization problem using an arbitrary approach. The instructor will then need to create scripts to take the learner's output and run it through the MiniZinc check model appropriatedly, and of course the learner will need to have MiniZinc installed.

Note that it is important to process every candidate solution found by the model, since there can be bugs in models which are hidden by the search strategy. It may well be that the first candidate solution found by a model is always valid, but later ones are not, or that the optimal candidate solution found happens to satisfy the model while earlier ones do not.

A checking model will in many cases be quite similar to a model for solving the problem, as shown in the next example.

Example 12. Consider a modelling project for n queens. A starting model file `queens.mzn` could be

```
1  int: n; % number of queens
2  set of int: ROW = 1..n;
```

```
  set of int: COL = 1..n;
  array[ROW] of var COL: q :: check_output; % col of queen in each row
  output [ if fix(q[r]) == c then "Q" else "." endif ++
           if c == n the "\n" else "" endif
         | r in ROW, c in COL ];

```

which gives the data declarations (just n here) as well as the decision variable declarations and a pretty printing output item.

A possible checking file is

```
  int: n; % number of queens
  set of int: ROW = 1..n;
  set of int: COL = 1..n;
  array[int] of int: q; % col of queen in each row
  solve satisfy;
  include "check.mzn";
  output [ if check_array(q,n,COL,"q") /\
              forall(r1, r2 in ROW where r1 < r2)
                 (check(q[r1] != q[r2],
                  "queens in rows \(r1) and \(r2) are on same column\n")
                  /\
                  check(q[r1]+r1 != q[r2]+r2,
                  "queens in rows \(r1) and \(r2) are on same up diagonal\n")
                  /\
                  check(q[r1]-r1 != q[r2]-r2,
                  "queens in rows \(r1) and \(r2) are on same down diagonal\n")
                 )
           then "CORRECT: All constraints hold"
           else "INCORRECT" endif ];

```

Note how it is easy to map the checking model to a solution model, though it would not be the preferred solution model (which uses three alldifferent constraints). □

Given this similarity, the question is in which form the checking model should be distributed to learners. The possible options are to

- Store the checking model as plain text, thus not hiding it from the learner. This is perfectly valid for projects designed for self-directed learning, which do not take part in assessment.
- Obfuscate the checking model in a simple way (e.g. `rot13`, or using a binary representation of the internal MiniZinc syntax tree). Again this is valid for projects designed for self-directed learning, and adds a barrier such that only a learner who "really wants to" read the checking model will be able to do so.
- Encrypt the checking model. This may be appropriate for assessed projects, but note that it cannot really be made secure, since the decryption keys must be available to MiniZinc (and hence either hidden inside it, or obfuscated in the rest of the project). Thus there is no real guarantee that a determined attacker could not decrypt the checking model, the bar has just become higher.
- Finally, we may keep the checking models separate from the project, which we discuss in the next section.

3.2 Online Checking of Discrete Optimization Projects

The provision of standalone projects with built-in checking models is valuable but suffers from two disadvantages.

- Inclusion of a checking model into a project even if it is encrypted is inherently insecure, since MiniZinc must have the ability to decrypt the checking model in order to run it, and hence a determined attacker may well be able to extract the decrypted checking model. Checking models may well give important direction for building effective solving models, although not always, so the possibility of their availability to learners being assessed is detrimental.
- Assessors of discrete optimization projects may well wish to have a certification that the model reached an acceptable level of performance, over a set of benchmarks, and this is harder to ensure/guarantee if all checking is performed on the learner's local machine.

For these reasons it is attractive to have checking done remotely in a secure, consistent, and controlled environment.

We can easily set up an online service for checking models, by providing a web service, and a number of data files. We have extended the MiniZinc project files to allow the inclusion of `_coursera` file which defines data required to specify how an online checking service used for Coursera assignments is to be run. We have extended the IDE to make use of this to allow students to use the online checking by adding a Coursera submission button to the IDE, when they view a project with a `_coursera` file.

The same mechanism could easily be generalized. We plan to allow the addition of an `_online` file to MiniZinc projects that specifies, similar to the currently used `_coursera` files an address of an online service and how and what to check. The online site can then be set up with the checking model together with a collection of data files. The MiniZinc IDE can be extended so that a learner who is building a model for a project that has online checking sees a "SUBMIT" button when visiting the model file, analogous to the Coursera submission button.

Running the model with the "SUBMIT" button will cause the last candidate solution to be sent to the online checking server, together with details of which data file is used. The online service can then be run with the candidate solution and the appropriate data file, and the feedback returned to the submitter.

The same facility could be used for projects that are not specified in MiniZinc, by using a generic python submission program `grader.py` which reads the `_online` file to determine how to submit. The learner simply needs to create the output in appropriate MiniZinc data format in order to be checked.

3.3 Automatic Grading of Discrete Optimization Projects

It is simple to extend the checking infrastructure to also grade projects, as long as the grades only rely on correctness of solutions and the objective value obtained. We only need to modify the output for correct solutions to output the grade.

To add grading we typically need to add marking scheme data, either to each data file, or a uniform one (if all the objective values are the same). The marking scheme data can be part of the regular data files distributed to the students (in the case of self-directed learning), or it can be given in the form of additional data files that are only stored on the online server.

A straightforward way to do this is to have a list of objective values, and marks attached with reaching that value. For a minimization problem it can be written as follows:

```
array[int] of int: objvalue;
array[int] of int: marks;
...
     then let { int: i = min(j in index_set(objvalue)
                             where objvalue[j] >= _objective)(j);
                int: mark = marks[i]; } in
          "CORRECT: All constraints hold: MARK=\(mark)\n"
     else "INCORRECT: MARK=0\n" endif ];
```

We compute the index `i` of the minimum objective value which the computed solution betters, and return the corresponding mark. Note that we should have a final objective value greater than all possible objectives so that the computation of `i` cannot fail.[4]

Example 13. Consider adding a grader for the photo problem of Example 9. We decide that a correct solution is worth 2 out of 5, and an optimal 5, give 3 for an objective values that is within 5 of optimal. The data file from Example 10 can be extended with

```
objvalue = [11, 16, 56];
marks    = [5,  3,  2 ];
```

A student submitting the solution `pos = [5, 8, 9, 6, 7, 4, 1, 2, 3]; _objective = 15;` will get the feedback "CORRECT: All constraints hold: MARK=3." □

3.4 Hidden Data

Finally, grading models where all data is made available to the learners may result in the learners fine-tuning their models to work on the given data, and potentially nothing else. Hence it is much more robust to also test a learner's model on hidden data. It is straightforward to use a solution checker on data that is not provided to the user, but in this case the learner submits the *model* to an online server, rather than candidate solutions. The model is then first run with the hidden data (and some time limits), and then has its solutions checked, with the report from the solution checker sent to the learner.

We have used automatic grading for the all Coursera assignments we have developed, with some strict time limits on the time available for computing solutions. It is worthwhile to note that because MiniZinc is an open-source toolset that does not require a license, it is fairly easy to assemble a MiniZinc-based

[4] Alternatively we can add more complicated grade calculation code that accounts for this possibility.

solution-checker in a containerized environment (e.g. docker, kubernetes). A key advantage of containerization is that it combines with cloud computing environments (e.g. Amazon Web Services, Google App Engine) to provide on-demand autoscaling, which is critically valuable just before an assignment deadline when each learner will often submit an assignment for grading 20-100 times over a few days. This kind of scalable infrastructure was essential in our deployment that has graded over 65,000 submissions.

4 Experience Report and Conclusion

The methology of using MiniZinc for checking solutions was first developed for the Lightning Model and Solve competition held at CP2013 in Uppsala. The approach was then used to build the automatic graders for the Coursera course "Modeling Discrete Optimization", which launched in 2015. It has now been used to check over 65,000 submissions, and provide automatic feedback and grading.

Initially the problem of hidden variables was solved using a two-model checking process, where the first model detected whether the hidden variables could be given a value, and computed and printed their values. The second model checked the remaining constraints that required the value of hidden variables. The single model approach using output was developed more recently and used for the automatic graders for the Coursera courses "Basic Modeling for Discrete Optimization" and "Advanced Modeling for Discrete Optimization", which launched in 2017. It has been used for the checking of over 24,000 submissions.

We make use of the `--output-mode` and `--output-objective` features of MiniZinc which were added in order to support checking of solutions in the MiniZinc Challenge [12]. For the MiniZinc Challenge we are only interested in correctness of the submitted solvers, so there checking simply uses the correct model approach of Section 2.1, and indeed the model is given, so feedback about it is irrelevant.

Once we release the MiniZinc IDE version which directly supports checking files, we plan to upgrade all the workshops on the Coursera courses (which do not count for assessment) to include a solution checker in the project. We hope to see an immediate benefit for students undertaking the workshop problems.

Building solution checkers that give meaningful feedback is important for improving how we learn to create discrete optimization solutions. We are excited about the ability to distribute stand-alone projects with automatic checking included, since this makes self-directed learning far easier to support. We plan to use this facilities to add badges to our Coursera courses to encourage good students to tackle more complex or esoteric modelling questions.

There is more work we can do to improve the use of automatic solution checking. The tasks we are working on include:

- Releasing the current checking framework within the IDE.
- Publishing a check library of checkers for global constraints.
- Extending the checking framework to support an `_online` file, with documentation about how to build the required online checking service.

– Building an online service at `minizinc.org` where instructors can register checking models and data files.
– Extending the online service to submit and check models, although this relies on us finding some computing resources to run the models.
– Improving feedback for unsatisfiable results.

References

1. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: a new approach to integrate CP and MIP. In: Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. LNCS, vol. 5015, pp. 6–20. Springer (2008)
2. Applegate, D.L., Cook, W., Dash, S., Espinoza, D.G.: Exact solutions to linear programming problems. Operations Research Letters 35(6), 693 – 699 (2007), http://www.sciencedirect.com/science/article/pii/S0167637707000211
3. Chu, G., Stuckey, P.J.: Dominance breaking constraints. Constraints 20(2), 155–182 (2015)
4. Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.): Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project), Lecture Notes in Computer Science, vol. 1870. Springer (2000)
5. Dhiflaoui, M., Funke, S., Kwappik, C., Mehlhorn, K., Seel, M., Schömer, E., Schulte, R., Weber, D.: Certifying and repairing solutions to large LP's how good are LP-solvers? In: Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA2003). pp. 255–256. ACM Press (2003)
6. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
7. Junker, U.: QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In: McGuinness, D.L., Ferguson, G. (eds.) Proceedings of the Nineteenth National Conference on Artificial Intelligence. pp. 167–172. AAAI Press / The MIT Press (2004)
8. Law, Y.C., Lee, J.H.M., Walsh, T., Yip, J.Y.K.: Breaking symmetry of interchangeable variables and values. In: Bessiere, C. (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 4741. Springer (2007)
9. Leo, K., Tack, G.: Debugging unsatisfiable constraint models. In: Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. p. to appear. LNCS, Springer (2017), http://cp2016.a4cp.org/program/workshops/ws-modref-papers/Leo.pdf
10. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Procs. of CP2007. LNCS, vol. 4741, pp. 529–543. Springer-Verlag (2007)
11. Simonis, H., Davern, P., Feldman, J., Mehta, D., Quesada, L., Carlsson, M.: A generic visualization platform for CP. In: Cohen, D. (ed.) Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, Proceedings. Lecture Notes in Computer Science, vol. 6308, pp. 460–474. Springer (2010)

12. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008-2013. AI Magazine 35(2), 55–60 (2014), http://www.aaai.org/ojs/index.php/aimagazine/article/view/2539
13. Van Hentenryck, P.: The OPL optimization programming language. MIT Press (1999)
14. Van Hentenryck, P., Michel, L.: The objective-cp optimization system. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings. pp. 8–29. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)