Introduction

Constraint programming is a paradigm for solving combinatorial problems that draws on a wide range of techniques from artificial intelligence, computer science, and operations research. MiniZinc is a free and open-source constraint modeling language. Constraint satisfaction and discrete optimization problems can be formulated in a high-level modeling language. Models are compiled into an intermediate representation that is understood by a wide range of solvers. MiniZinc itself provides several solvers, for instance GeCode. The existing packages in R are not powerful enough to solve even mid-sized problems in combinatorial optimization.

Until recently, there were implementations of an Interface to MiniZinc in Python like MiniZinc Python and pymzn and JMiniZinc for Java but none for R.

rminizinc as a GSOC project

rminizinc started as a Google Summer of Code Project in 2020. Initially, the goal was to provide infrastructure/support for creating 15-20 commonly used MiniZinc problems by creating classes and functions for providing the basic syntax/constructs used in those MiniZinc models. However, it was decided that the libminizinc (MiniZinc C++ API) library can be leveraged using Rcpp for parsing various MiniZinc models and a mirror API can be created to construct the MiniZinc models. Using the library helped me in understanding MiniZinc more which in turn also helped to to provide more features and test the package on larger problems. The following objectives were achieved at the end of the GSOC period:

- Parse a MiniZinc model into R.
- Find the model parameters which have not been assigned a value yet.
- Set the values of unassigned parameters. (Scope needs to be extended)
- Solve a model and get parsed solutions as a named list in R.
- Create a MiniZinc model in R using the R6 classes from MiniZinc API mirror.
- Manipulate a model.

The development continued till the end of GSOC but the package was not yet submitted to CRAN.

Post GSOC

The package submission to CRAN was very challenging. Libminizinc and solver binaries were required in order to use the package because the Rcpp functions were using them to parse and solve the models. To tackle this, we leveraged the autotools to create a configure script for letting the users provide custom paths and configure the package during the installation, used #ifdef macros to provide alternative definitions in case libminizinc and/or solvers are not present on the system.

Examples

There are a lot of features provided by the package but let's start with something simple say solving a knapsack problem. Knapsack problem is known by everyone who has interest in constraint programming. The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

The `knapsack()` function can be used to directly solve the knapsack problem. Here, `n` is the number of items, `capacity` is the total capacity of carrying weight, `profit` is the profit corresponding to each item and `weight` is the weight/size of each item. The goal is to maximize the total profit. The solution is returned in the form of a named list with all the solutions along with the optimal solution if found.

Please find the installation instructions in the vignette or the github readme.

```
library(rminizinc)
# knapsack problem
print(knapsack(n = 3, capacity = 9, profit = c(15,10,7), size =
c(4,3,2)))
$SOLUTION0
$SOLUTION0$x
[1] 0 0 0


$SOLUTION1
$SOLUTION1$x
[1] 0 0 1


$SOLUTION2
$SOLUTION2$x
[1] 0 0 2


$SOLUTION3
$SOLUTION3$x
[1] 0 0 3


$SOLUTION4
$SOLUTION4$x
[1] 0 0 4


$SOLUTION5
$SOLUTION5$x
[1] 0 1 3


$OPTIMAL_SOLUTION
$OPTIMAL_SOLUTION$x
[1] 1 1 1
```

A function to solve the assignment problem has also been provided. More common examples will be provided in the next package releases based on the user feedback. This will especially be useful for the users who don’t have any knowledge of MiniZinc.

Basic knowledge of MiniZinc is required in order to understand the next examples. MiniZinc tutorial would be a good place to start.

The users can also create a MiniZinc model using the API. Let's compute the base of a right angled triangle given the height and hypotenuse. The Pythagoras theorem says that In a right-angled triangle, the square of the hypotenuse side is equal to the sum of squares of the other two sides i.e \(a^2 + b^2 = c^2\). The theorem give us three functions, \(c = \sqrt{(a^2 + b^2)}\), \(a = \sqrt(c^2 – b^2)\) and \(b = \sqrt(c^2 – a^2)\).

MiniZinc Representation of the model:

```
int: a = 4;
int: c = 5;
var int: b;

constraint b>0;
constraint a² + b² = c²;

solve satisfy;
```

Let's create and solve the model using rminizinc.

```
a = IntDecl(name = "a", kind = "par", value = 4)
c = IntDecl(name = "c", kind = "par", value = 5)
b = IntDecl(name = "b", kind = "var")

# declaration items
a_item = VarDeclItem$new(decl = a)
b_item = VarDeclItem$new(decl = b)
c_item = VarDeclItem$new(decl = c)

# b > 0 is a binary operation
b_0 = BinOp$new(lhs = b$getId(), binop = ">", rhs = Int$new(0))
constraint1  = ConstraintItem$new(e = b_0)

# a ^ 2 is a binary operation
# a$getId() gives the variable identifier
a_2 = BinOp$new(lhs = a$getId(), binop = "^", Int$new(2))
b_2 = BinOp$new(lhs = b$getId(), binop = "^", Int$new(2))
a2_b2 = BinOp$new(lhs = a_2, binop = "+", rhs = b_2)
c_2 = BinOp$new(lhs = c$getId(), binop = "^", Int$new(2))
a2_b2_c2 = BinOp$new(lhs = a2_b2, binop = "=", rhs = c_2)
constraint2  = ConstraintItem$new(e = a2_b2_c2)

solve  = SolveItem$new(solve_type = "satisfy")

model = Model$new(items = c(a_item, b_item, c_item, constraint1,
constraint2, solve))

cat(model$mzn_string())
int: a;

var int: b;
```

```
int: c;

constraint (b > 0);

constraint (((a ^ 2) + (b ^ 2)) = (c ^ 2));

solve  satisfy;
```

Creating the model required a lot of code which is more cumbersome that writing the model in MiniZinc itself. However, that was to give the users a taste of various classes that can be used to create items and expressions. This will especially be useful in modifying an existing model.

The items can directly be provided as strings.

```
a = VarDeclItem$new(mzn_str = "int: a = 4;")
c = VarDeclItem$new(mzn_str = "int: c = 5;")
b = VarDeclItem$new(mzn_str = "var int: b;")
constraint1 = ConstraintItem$new(mzn_str = "constraint b > 0;")
constraint2 = ConstraintItem$new(mzn_str = "constraint a^2 + b^2 =
c^2;")
solve = SolveItem$new(mzn_str = "solve satisfy;")
model = Model$new(items = c(a, b, c, constraint1, constraint2, solve))
cat(model$mzn_string())
int: a = 4;

var int: b;

int: c = 5;

constraint (b > 0);

constraint (((a ^ 2) + (b ^ 2)) = (c ^ 2));

solve  satisfy;
```

The model can directly be parsed by providing the string representation as the argument to `mzn_parse()`. This method uses libminizinc to parse the model. The included mzn files are appearing because the parsed model is serialized back by libminizinc.

```
pythagoras_string =
  " int: a = 4;
    int: c = 5;
    var int: b;

    constraint b > 0;
    constraint a^2 + b^2 = c^2;

    solve satisfy;
 "
model = mzn_parse(model_string = pythagoras_string)
cat(model$mzn_string())
int: a = 4;
```

```
int: c = 5;

var int: b;

constraint (b > 0);

constraint (((a ^ 2) + (b ^ 2)) = (c ^ 2));

solve  satisfy;

include "solver_redefinitions.mzn";

include "stdlib.mzn";
```

Let’s solve the model now.

```
solution = mzn_eval(r_model = model)
print(solution)
$SOLUTION_STRING
[1] "{\n  \"b\" : 3\n}\n----------\n==========\n"

$SOLUTIONS
$SOLUTIONS$OPTIMAL_SOLUTION
$SOLUTIONS$OPTIMAL_SOLUTION$b
[1] 3
```