Sub-Problem Pre-Solving in MiniZinc

Jip J. Dekker
Abstract

Sub-Problem Pre-Solving in MiniZinc

Jip J. Dekker

Constraint models often describe complicated problems that contain sub-problems that could be solved in sub-models. Although pre-solving these sub-models might improve the performance of the model, manually splitting the model to accommodate the occurrences of these sub-models can be a great inconvenience for the modeler. This thesis introduces an approach to automate the process of solving sub-models and replacing them by constraints that provide more inference, based on the solutions found. An implementation of this approach is provided for the MiniZinc language, and using various examples we show how the strain of the implementation of pre-solving is moved from the modeler to the modeling toolchain.
In combinatorial optimization, problems are faced where exhaustive search is not feasible. Many of these problems, like for instance the traveling salesman problem, have proven to be NP-complete. This means that unless $P = NP$ [Coo71], no algorithm can be constructed to solve these problems in polynomial time in the size of the input. If $P \neq NP$, then these problems can only be solved by algorithms that need at least exponential time in the size of the input. The consequence of algorithms needing at least exponential time is that every algorithm requires too much time to run when the input gets large enough.

Solutions to these problems are interesting to both the academic and the business worlds, as these problems include finding an optimal route to deliver packages, finding a feasible schedule for employees, and the Boolean satisfiability problem. So even though no polynomial-time algorithm has so far been constructed, various techniques have been developed to form solutions that will postpone the input size at which the problem cannot be solved within a reasonable amount of time. These techniques include integer programming, SMT solving, constraint programming, and various derivatives.

For the use of these techniques, the way in which a problem is modeled is of the highest importance. Although two models in various techniques, or even within the same technique, might describe the same problem, the time necessary to solve the same problem instance might vary a lot. Because of the nature of NP-complete problems, a small improvement to the model could be the difference between solving a particular problem instance within a set time frame and wasted computational time.

Finding improvements for the models using these techniques, however, is a complex task. Modelers usually have to try various choices of variables and various ways of expressing the constraints imposed on these variables. As many of the tricks used by constraint modelers result in complicated execution paths, it is hard to predict if using of one of these tricks will be beneficial to the model. Therefore the use is triggered by the intuition of the modeler, and usually many tricks are tried on smaller problem instances in an attempt to predict the performance on larger problem instances.

In more complex models it can occur that part of the model can, to a certain extent, be solved separately. By this act of pre-solving the model, as a whole, can solve a problem instance more rapidly. This trick has been very effectively used in various models, such as the Black Hole Patience model used in the 2009 MiniZinc Challenge [Min09] and the Eltserien model used in the 2014 MiniZinc Challenge [Min14]. However, because no tool is available to do this automatically, it is currently up to the modeler to solve a sub-model separately and reinsert the solutions into the model in which it was originally contained. This can be a particular burden on the modeler.
if parts of the model have to be pre-solved for every problem instance. But since this
trick is not guaranteed to work and requires a lot of work from the modeler, it is
therefore not often used.

This thesis addresses the complications that come with manual pre-solving by
introducing automated pre-solving into a constraint modeling toolchain, MiniZinc.
The goal of adding automated pre-solving to the toolchain is to have the possible
benefits of pre-solving part of the model, but reducing the modeler’s involvement in
the process to choosing the sub-model to be pre-solved and in which way it should
be solved. This will enable modelers to make use of pre-solving even if the outcome
is not guaranteed to have better runtimes.
To understand the concepts of automated pre-solving of sub-models in MiniZinc, a basic understanding of constraint-based modeling in the style of constraint programming (CP), MiniZinc, and their concepts is required. For the reader’s convenience, short introduction into these subjects and their terminology is provided to the extent of their use in this thesis. For more more details on CP and MiniZinc, see [RvW06] and [Mar+14].

2.1 Constraint-Based Modeling

Constraint-based modeling is a style of modeling based on the CP paradigm.

2.1.1 Satisfaction Problems

A constraint satisfaction problem (CSP) consists of a set of decision variables \( X = \{x_1, \ldots, x_k\} \), where each variable \( x_i \in X \) has an associated finite domain \( D(x_i) \subseteq \mathbb{Z} \) of possible values, and a collection of constraints \( C \). A constraint defines a finitary relation over a set of variables (e.g., \( c(x_{i_1}, \ldots, x_{i_j}) \)). In a CP solver, a constraint corresponds to a propagator that aims to eliminate unsupported domain values. A solution to a CSP consists of a value \( d_i \in D(x_i) \) for each variable \( x_i \in X \), for which all constraints are satisfied.

A real-life example of a CSP is a sudoku puzzle [Sim05], an example being shown in Figure 2.1. The goal of a sudoku puzzle is to find a value, between 1 and 9, for each small square in such a way that no value occurs twice in the same row, column, or darkly marked square of nine small squares. This problem is a CSP and every small square translates into a decision variable. Each of the variables is involved in three constraints on sets of variables among which each value has to be unique (e.g., by the use of an all_different constraint, available in many constraint libraries). Human solving a sudoku puzzle is done in a similar fashion to solving a CSP by a CP solver. Instead of trying combinations of values, the puzzler will try to (mentally) eliminate possible values until there is only one possible value for every square, and only guess when it is necessary.

The basic CP-solving technique is similar to the human approach of solving a sudoku; it uses a tree exploration approach with propagation. In each tree node, the propagators are executed until they are at fix-point (i.e., no more values can be eliminated from the domains of the variables). If at this point not all domains are reduced to a single value, which would represent a solution, and there is no domain without at least one possible value, which would represent a failure, a variable \( x_i \) is selected and new branches are created on the tree. Each of these branches corresponds to a mutually exclusive assumption on \( x_i \).
Constraints like the previously mentioned \texttt{all\_different} constraint are classified as global constraints. A global constraint can act on a non-fixed number of variables and is usually a shorthand for a frequently recurring pattern that can ease the modeling of a CSP. Depending on the solver, a global constraint might have to be decomposed into a formula over simpler constraints. Decomposing constraints is, in practice, only done when strictly necessary for the solver as it can have bad space complexity, and in many cases propagation algorithms are known that are low in complexity. A large set of frequently used global constraints can be found in the Global Constraint Catalog \cite{Bel07}.

2.1.2 Optimization Problems

Constraint-based modeling can also be used to model optimization problems. A \textit{constrained optimization problem} (COP) is an extension to a CSP, where an \textit{objective function} is added. This function is used to map a solution of the CSP to a specific value. The goal of a COP is to find the solution that minimizes or maximizes the objective function.

An example of a COP is the Knapsack Problem, where a knapsack has to filled with items, each having a specified size and weight. The knapsack can only carry a certain weight. The goal is then to find a set of items that maximizes the value of the items contained in the knapsack, while the total weight of the items stays within the maximum weight the knapsack is able to carry.

A common way to solve COPs is to use the \textit{branch-and-bound} (BAB) search strategy. This search strategy is similar to the ordinary search strategy used for CSP problems, however, after every solution found a constraint is added that requires the value for subsequent solutions to be better than the currently found solution. When no new solutions can be found, the last found solution was the optimal solution.

2.1.3 Symmetry

The permutations of solutions to a CSP are referred to as symmetries. A (constraint) symmetry of a CSP $P$ is a permutation of a solution to $P$, for which all constraints are satisfied, and which is thus in itself a solution to $P$. The existence of symmetries in a CSP implies that both solutions and non-solutions may be represented more than once in the search space. By identifying and eliminating symmetries within a CSP,
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint 1000 * S + 100 * E + 10 * N + D
    + 1000 * M + 100 * O + 10 * R + E
    = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint all_different([S,E,N,D,M,O,R,Y]);
solve satisfy;

output [" ",show(S),show(E),show(N),show(D),"\n",
    "+ ",show(M),show(O),show(R),show(E),"\n",
    "+= ",show(M),show(O),show(N),show(E),show(Y),"\n"];

Figure 2.2: An example MiniZinc model of the SEND+MORE=MONEY problem, as found in [Mar+11]

the search space and possibly the solving time can be reduced. The exploitation of symmetry within a CSP is referred to as symmetry breaking.

2.1.4 Reification

Reification binds a variable with a binary domain to a constraint in such a way that the variable equals the designated true-value if and only if the constraint is satisfied. The reification of constraint \( c \) by Boolean variable \( B \) is here written \( B \leftrightarrow c \). Reification can be used to make some constraints optional. For example, one could create a model where at least 5 out of 8 constraints would have to hold, but it is undecided which ones. Reification is often used in the decomposition of constraints.

2.2 MiniZinc

MiniZinc is a solver-independent, medium-level constraint modeling language. It aims to have a high enough abstraction level so that the modeler is able to easily express most constraint satisfaction problems, but low-level enough that the models can be run on existing solvers. To allow for the level of abstraction, a MiniZinc model is processed into a more low-level language called FlatZinc, a process also referred to as flattening. The MiniZinc-to-FlatZinc compiler simplifies the model in various ways; it will: unroll loops in the model, decompose expressions, and inline functions and predicate calls. The resulting FlatZinc model will consist of native constraints (i.e., constraints that are not composed of other constraints in the used MiniZinc library), decision variables, and constants. During the decomposition of constraints in the model, auxiliary variables can be introduced to the model to allow for the use of constraints without a native implementation.
Figure 2.2 shows a full MiniZinc model. This particular model is visibly split into the various parts in which a MiniZinc model is described.

- The first statement is an include statement; these statements can be used to import other partial MiniZinc models. MiniZinc offers a full library of MiniZinc files that contain functions and predicates (functions whose results are Boolean variables, and when used in a constraint item are forced to equal true) for the most used constraints that are implemented by the solver.

- Between lines 3 and 10 integer decision variables are declared. MiniZinc allows the notation with two dots to limit an interval domain of the variables. However, using the keywords var int, the whole integer domain could have been used.

- The following two items are constraint declarations. Using the MiniZinc abstractions, constraints can be easily expressed in various ways, as shown in the example by the use of mathematical operators in the first constraint item and a call to a predicate of a global constraint in the second item.

- In MiniZinc there are two types of goals: one can either search for a solution that does not violate the declared constraints (indicated in this model by solve satisfy), or the solver can be instructed to look for an optimum (which can be indicated by the maximize/minimize keywords). The last statement in this example instructs the solver how to format the output.

Here, it is important to note two aspects of the language. First of all, it is often easy to model a problem in MiniZinc. No procedural code is used in MiniZinc, and the modeler does not have to take the internal representation of the constructs in the model into account. The specifics of how the model will be solved are left up to the solver. In addition, the modeler is able to give solving hints through the use of annotations, indicated by ::, should it be necessary.

Secondly, MiniZinc has a focus on extensibility. Modelers can build their own libraries of functions and predicates. Moreover, this extensibility enables the solver-independence of the MiniZinc language. There are only a few basic language structures that a solver has to support to be able to process a FlatZinc model produced from MiniZinc. For other constraints, for which predicates are available, a solver can define its own decomposition for the predicate or simply not support the predicate. Additional constraints supported by the solver can also be added to the library as predicates without a decomposition.
Pre-Solving Sub-Models in Constraint-Based Modeling

One way in which some models can be improved is by pre-solving parts of the model. For instance, in the search process used by a CP solver, there is a branching order. By choosing an efficient branching strategy the performance of a model can be improved. However, there are situations in which no matter which branch is chosen, the search in the chosen branch will be evaluated in the same way because of the underlying nature of the model. An example of a search tree that has such a structure is given in Figure 3.1. Notice that all numbered nodes have the same branching structure. The corresponding part of the model, which is evaluated for every branch taken, can be described as a sub-model. To prevent the solver from having to evaluate this sub-model for every branch, stronger propagators are required. One way to find a stronger propagator is by the use of pre-solving. By pre-solving these sub-models it is possible to (partially) eliminate the need for search in the sub-model by using one or more constraints specific to the solutions found for the sub-model. Furthermore, because of the stronger inference of these posted constraints, the propagation of the whole model might become more effective.

Specifically, a sub-model of a CSP \( \langle X, C \rangle \), consisting of the set of variables \( X \) and set of constraints \( C \), is in itself a CSP \( \langle X_s, C_s \rangle \), where \( X_s \subseteq X \land C_s \subseteq C \). The act of pre-solving consists of first precomputing the solutions to the sub-model \( \langle X_s, C_s \rangle \), and then replacing the sub-model in its containing CSP \( \langle X, C \rangle \) by a CSP \( \langle X_r, C_r \rangle \), based on the precomputed results, whose solutions are equal to those of the sub-model \( \langle X_s, C_s \rangle \). This implies that \( X_r = X_s \), however, in practice we allow \( X_r \subset X_s \) to support the removal of variables that are contained to the sub-model and do not contribute to the solution of the problem (e.g., reifications). \( X_r \) must however contain all variables \( x_i \in X \) for which there is a constraint \( c(..., x_i, ...) \in C \setminus C_s \). The goal of this task is thus to find constraints \( C_r \) that constrain the variables equally, but have stronger inference than the constraints \( C_s \).

The effectiveness of pre-solving, however, highly depends on three factors: the choice of sub-model(s) to be pre-solved, the strategy used to pre-solve the sub-model(s), and the choice of constraint(s) used to implement the pre-solved sub-model(s). The choice of sub-model(s) is of great importance, because choosing a sub-model that is not self-contained can be harmful to the time needed to reach a solution for the original model, or pre-solving of the sub-model might not be possible within reasonable time, or both.
3.1 Pre-Solving Strategies

In practice, a CSP is not modeled for just one problem instance. A model is, instead, made in such a way that it accepts data so the model can be used for all instances of the same problem. In various languages, like MiniZinc, it is also allowed to group sets of constraints and post them for different sets of decision variables. This makes the modeling process more productive and, potentially, provides structures to the occurrence of sub-models, which can be used to pre-solve the sub-models more efficiently.

The notion that the sub-models of a CSP could be more efficiently pre-solved is ascertained from the fact that similar sub-models often occur in the same CSP. This is due to the nature of the problems and the way in which the constraint model is constructed. In particular, it concerns sub-models with the same set of constraints, differing only in variables involved in the constraints and their associated domains. In mathematical terms, this applies for any two sub-models \( (X_1, C_1) \) and \( (X_2, C_2) \) of CSP \( (X, C) \) for which a function \( f : X \rightarrow X \) exists for which

\[
\forall c(x_1, ..., x_j) \in C_1 : \quad c(f(x_1), ..., f(x_j)) \in C_2
\]

Sub-models with this relationship can in some instances be pre-solved more efficiently by solving a combination of the sub-models.

In this thesis three strategies are considered to handle the pre-solving of sub-models containing the same constraints: either each sub-model is solved separately, called the per-call-based strategy; or a combination of all sub-models with the same constraints within the problem instance is solved at once, called the model-based strategy; or the combination of constraints present in the sub-models is solved for all possible domains for the involved variables, called the global strategy.

3.1.1 Model-Based Strategy

If we for example look at the toy MiniZinc model in Figure 3.2, we see that the predicate \( \text{pred} \) is the basis of various sub-models in the model: every call to the predicate is a separate sub-model. However, because of the similarities between the sub-models, there is a more efficient way to model. Using our model-based pre-solving strategy, pre-solving replaces the definition of the predicate \( \text{pred} \) by a constraint that constrains the variables \( v \) and \( w \) to the possible solutions of \( \text{pred} \) in the model. Figure 3.3 shows a possible structure for the pre-solved predicate in this example using

Figure 3.1: An example CP search tree, possibly containing a sub-model.
array[1..150] of int: A;
array[1..2] of var 1..100: x;
array[1..2] of var 50..150: y;

predicate pred(var int: v, var int: w) = A[v] == (A[w] + v);

cost constraint pred(x[1], x[2]);
cost constraint pred(y[1], y[2]);
cost constraint pred(x[1], y[1]);
cost constraint pred(x[2], y[2]);

Figure 3.2: An example predicate and constraints where a model-based pre-solving strategy would be beneficial

predicate pred(var int: v, var int: w) = table([v,w], [
% A list of solutions to the predicate in this model in the form "[v,w]".
]);

Figure 3.3: A possible structure for the model-based pre-solved predicate

a table constraint.

The model-based pre-solving strategy pre-solves all similar sub-models for the problem instance at once. It does this by constructing a CSP with the same constraints as the sub-models, but with new variables whose domains include the domains of the respective variables in each of the sub-models. Constraints are then posted for each sub-model constraining the variables to the solutions of the constructed model. This strategy does not influence the correctness of the model as it does not eliminate any possible solutions to the original constraints. Although more combinations of values might be found as possible solutions to the constructed CSP, these values will be eliminated in the domains of the variables in the individual sub-models.

Using the model-based strategy can reduce the overhead of pre-solving multiple similar sub-models within a CSP. This can be a big advantage when there are multiple similar sub-models, especially since solving a sub-model can be very time intensive. This strategy is particularly effective if the domains of the involved decision variables are closely related. The cost, however, is that this strategy can result in a large number of “possible” solutions being used in the propagator, which are excluded in the domains of the variables. Although it does not result in false results, it can slow down the solver. There is also the possibility that the constructed CSP requires exponentially more time to solve than any of the sub-problems themselves.

3.1.2 Global Strategy

In the previous MiniZinc example, shown in Figure 3.2, the predicate was dependent on the parameter A. In the example shown in Figure 3.4 there are no such dependencies. It is thus possible not only to pre-solve all sub-models for the model using the current parameters, but to solve it for all possible model instances at the same time. This strategy is called the global pre-solving strategy. A possible structure for a pre-solved predicate looks very similar to the previously proposed structure as depicted in Figure 3.3 but differs in the variable ranges for the predicate arguments and includes the solutions to all possible solutions to the predicate, not just the ones used in
1   predicate pred(var 1..100: v, var 1..100: w)
2     = (v div w) in {v mod w, w};

Figure 3.4: An example predicate where a global pre-solving strategy would be beneficial

1   predicate pred(var 1..100: v, var 1..100: w)
2     = table([v,w], [
3       | 1,1,
4       | 3,2,
5       | 4,2,
6       | 5,2,
7       | 4,3,
8       | % etc.,
9       | 1]);

Figure 3.5: A possible structure for the globally pre-solved predicate

the model. These differences result in a structure such as the one shown in Figure 3.5.

Global pre-solving aims to pre-solve a class of sub-models once and for all. Instead of solving the sub-models according to the domains contained in the sub-models in one problem instance, the largest allowed domains in accordance to the constraints are chosen for the decision variables in a constructed CSP. Akin to the model-based strategy, constraints are then posted constraining the variables of matching sub-models to the possible solutions of the constructed CSP. Similar to the model-based strategy, the correctness is not influenced by the use of this pre-solving strategy.

This strategy is mainly used when the number of possible results is expected to be limited and none of the constraints are specifically dependent on the problem instance. If this is the case, by pre-solving once, a constraint can be generated for every sub-model that fits the class of sub-models, for any problem instance. There are a few dangers about using the global strategy. The number of possible solutions might be (almost) endless, making solving incredibly slow, and solving the problem without more specific domain information might be impossible. Furthermore, if the constraints depend on problem-instance-specific data, using the global strategy has no benefit over using the model-based strategy, as pre-solving is necessary for every problem instance and the domains in the constructed CSP using the model-based strategy are smaller than or equal to the domains in the global strategy.

3.1.3 Per-Call-Based Strategy

The more efficient pre-solving strategies cannot always be used. There are constraints that can be imposed on any finite number of decision variables: the number of variables between the similar sub-models can thus be different. There are also situations where combined pre-solving can take exponentially more time. For these situations we formulate a strategy we refer to as the per-call-based pre-solving strategy. Figure 3.6 shows an example where the number of variables differs between the sub-models. In this case pre-solved solutions would have to replace the individual calls to the predicate. An example replacement for the first call to the predicate pred using the table constraint is shown in Figure 3.7.

The per-call-based pre-solving strategy consists of solving each sub-model separ-
array[1..2] of var 1..100: x;
array[1..2] of var 50..150: y;
int: cost = 250;

predicate pred(array[int] of var int: v) = sum(v) in {i*5 | i in 0..(cost div 5)} \ nvalue(2, v);

constraint pred(x ++ y);
constraint pred([x[1], x[2], y[1]]);
constraint pred([x[1], x[2], y[2]]);

Figure 3.6: An example predicate and constraints where a calls-based pre-solving strategy would be required

constraint table(x ++ y, [
  % A list of all solutions to the predicate call in the model
  % in the form "[x[1], x[2], y[1], y[2]]".
  ]);  

Figure 3.7: A possible structure to replace the first call in the example shown in Figure 3.6 when pre-solved using the per-call-based strategy

Constraints are then posted for each sub-model. These constraints limit the possible solutions for the variables to the corresponding solutions found for the sub-model. Because the definition of this strategy constrains the decision variables of sub-models to the exact combination of values that would be possible according to the constraints in the sub-model, the correctness of the CSP is not influenced by the use of this pre-solving strategy.

Although this strategy might be seen as the simplest of the three, it is still useful in cases where the number of variables between sub-models differs, or the use of the other pre-solving strategies would exponentially increase the pre-solve time. Taking a potential slowdown of propagators into account, the per-call-based strategy can also be considered when combining the different sub-models result in an abundant amount of possible solutions irrelevant to the domains of the individual sub-models. In this case, the inefficiency of the propagator might outweigh the added time of pre-solving each sub-model separately.

### 3.2 Choice of Constraint

The success of the pre-solving depends mostly on the way in which the results are introduced back into the model. Only when the constraint(s) that are introduced have more inference than the constraints that are replaced will pre-solving make the model more efficient. There are many options to generate a constraint using the solutions found by pre-solving. Our choice of constraint depends not only on which propagator of the constraint can prune the search space the fastest, but also on how fast the propagator can be run. Since there is no standard on which constraints are supported by which solver, the following paragraphs explore several options of constraints that have the capability of being generated from a set of solutions and are either common to solvers, or have been proven to be useful in similar situations.
3.2.1 The table Constraint

The table constraint, also referred to as the tabular constraint in some solvers, is a natural choice to use for the generated constraints. It takes a sequence of variables and a set of possible assignments to these variables and constrains the variables to these possible assignments. This constraint is not present in the Global Constraint Catalogue [Bel+07], but can be decomposed using element constraints: each variable has a separate element constraint, the possible values are ordered according to the solutions found by the pre-solver, and the same index variable is used for all these constraints.

The generation of a table constraint according to the solutions found during the pre-solving process does not require any processing. The solutions can be used directly for the constraint.

Although the generation of table constraints is ideal for our usage, it does not consider the relations between variables. This would be of particular interest to us when either the number of possible solutions, or the number of variables, or both are large. The use of the relations could limit the number of possibilities that the solver needs to take into considerations, limiting the decrease in performance.

3.2.2 The smart_table Constraint

An extension on the table constraint to accommodate more complex structures is the smart_table constraint [MDL15], not yet available in MiniZinc. The usage of the smart_table constraint is similar to the original table constraint, but instead of assignments to the variables, for each variable either an equality or an inequality can be used (to either a value or another decision variable) for each of the considered possibilities.

The generation of a smart_table constraint can initially use the same approach as the generation of a table constraint, using equalities to values as found during pre-solving. Using analysis the number of possibilities to be considered by the solver could then be reduced by combining possibilities to have equivalent effects replacing one or multiple values by inequalities.

The use of smart_table could improve the performance of the solver. However, the level of the improvement is highly dependent on the effectiveness of the analysis used to reduce the number of possibilities. This analysis can be complex and might require more computational time than is lost compared to using a table constraint.

3.2.3 The regular Constraint

Another constraint that can express basic relations between variables is the regular constraint [Pes04]. It constrains a sequence of decision variables to form a word of a regular language, possibly expressed by a regular expression or a (non-)deterministic automaton [Kle56].

Using the solutions found during the pre-solving process, a deterministic automaton (DFA) can be generated. From a single starting state each possible value is added on an edge to a new unique state, and to each of these states the values of solutions starting with the values on the path to reach the state are added as edges with unique states, and so on. To reduce the number of initial nodes, the edges for the final input can all go to a single accepting state. Using DFA minimization algorithms, like Hopcroft’s algorithm [Hop71], the DFA can even be reduced to an equivalent DFA with a minimum number of states with a complexity of $O(n \log n)$, where $n$ is the
number of states, which will accelerate the solving.

The order in which the variables are used as input can be important though, as it will determine how many states and transitions will be present in the automaton. Although the use of a minimization algorithm can reduce the number of states and transitions to a minimum, the difference in number of states between different orderings of variables can make a big difference.

Although the construction of a regular constraint is more complex, solving a regular constraint will have the same result as solving a table constraint. Depending on the implementation of the constraints in the solvers, the regular constraint might perform better or worse than the table constraint. The regular constraint also leaves the modeler with the choice in which the variables should be ordered.
MiniZinc Implementation

It is possible to implement automated pre-solving into the MiniZinc toolchain. In this thesis we provide an extended version of the MiniZinc to FlatZinc compiler that implements this. Installation instructions for the provided implementation can be found in Appendix A.

For this extension of the MiniZinc compiler, an approach has been chosen that restricts the changes necessary to use automated pre-solving to the MiniZinc compiler. No changes have to be made to the solvers to use the FlatZinc models produced by the extended MiniZinc compiler.

To accommodate the pre-solving process, additions are made to the MiniZinc standard library and an extra compilation step is used.

4.1 Introduced Syntax

To automate the pre-solving process in MiniZinc, the compiler has to know which parts of the model to pre-solve. Although we are automating part of the process in MiniZinc, the decision of which parts still falls onto the modeler. This decision is non-trivial and automating this process is beyond the scope of this thesis.

The MiniZinc grammar is formally specified by the MiniZinc team [G12-I4]. The language contains a natural structure that effectively splits off a set of constraints, their involved decision variables, and their domains from the main model: the predicate item. A fragment of the formal specification of MiniZinc related to the predicate item can be found in Figure 4.1. This figure shows that predicate items offer a natural way of creating sub-models according to our definition in Chapter 3. It includes the set of involved decision variables, including their type instance, declared in the predicate parameters. It also contains the set of constraints that have to be satisfied, expressed in a single MiniZinc expression; this does not mean that a predicate can only contain one constraint, merely that they need to be expressed in a single expression (e.g., by the use of the \ operator). Thus introducing syntax that marks predicates as “to be pre-solved” would be ideal for our purpose.

MiniZinc already provides such a way, through the use of their annotation system. Annotations are meant as a way for the modeler to provide additional information to the model that can be used by the solver to improve performance [Net+07]. Although our annotations are not reconciled by the solver, adding annotations to the predicates provides a native way of marking our predicates and enables us to pre-solve them within the MiniZinc compiler. Resolving annotations within the MiniZinc compiler is not a common practice, but is done for other annotations in similar situations (e.g., is_reverse_map).
(predicate-item) ::= \texttt{predicate} \ (operation-item-tail)

(operation-item-tail) ::= (ident) \ (params) \ (annotations) \ [ \ = \ (expr) \ ]

(params) ::= \ [ \ (ti-expr-and-id), \ ... \ ]

(ti-expr-and-id) ::= \ (ti-expr) \ : \ (ident)

(annotations) ::= \ ( \ (annotation) \ )^*

(abbreviation) ::= \ (expr-atom-head) \ (expr-atom-tail)

Figure 4.1: An excerpt of the MiniZinc specification containing grammar rules most important to the predicate item.

1 \hspace{1em} \texttt{annotation presolve( ann: strategy, string: fzn_solver );}
2 \hspace{1em} \texttt{annotation presolve( ann: strategy );}
3 \hspace{1em} \texttt{annotation presolve;}

Figure 4.2: Definition of the MiniZinc pre-solve annotations for predicates, as provided in the accompanying implementation.

In the implementation provided in this thesis we add three forms of an annotation to signal the MiniZinc compiler that a predicate should be pre-solved as shown in Figure 4.2. In the implementation, these annotations have been added to the standard library and no extra files need to be included to use pre-solving.

The first form of the annotation is the most complete form. This form of the annotation takes two arguments. The first argument is the pre-solving strategy to be used. The strategy is chosen by providing an annotation. Valid options in the provided implementation are, in accordance to the strategies in Section 3.1:

\texttt{global} \hspace{1em} Pre-solve the predicate using a global pre-solving strategy, solving the predicate for all possible models.

\texttt{model} \hspace{1em} Pre-solve the predicate using a model-based pre-solving strategy, solving the predicate for this model instance.

\texttt{calls} \hspace{1em} Pre-solve the predicate using a per-call-based pre-solving strategy, solving the predicate for every MiniZinc call in the model.

The second argument is used to instruct the compiler to use a specific FlatZinc solver to pre-solve the annotated predicate. The FlatZinc solver is provided as a string containing the path to the binary of the solver, as it is available on the system executable path or system root. It can thus be any FlatZinc solver available on the machine on which the model is being compiled. Note that not all FlatZinc solvers may provide a successful result; the solver has to be able to solve the flattened predicate, in accordance with the chosen MiniZinc library, and needs to be able to provide all possible solutions (done by most solver by using the \texttt{-a} flag).

The other two forms of the annotation, as seen in Figure 4.2 on lines 2 and 3, provide abstractions to the first one. They allow the user to only specify the strategy used to pre-solve the predicate or none of these arguments. In cases where arguments are missing, the implementation will use standard values for these arguments. If there is no provided FlatZinc solver, then one could make an educated guess, based on the
predicate content and the FlatZinc solvers available on the machine. However, this is beyond the scope of this thesis, therefore the provided implementation will fall back unto the native FlatZinc solver, which is provided in the current distribution of the MiniZinc toolchain, namely flatzinc. This internal solver is based on constraint programming and should theoretically be able to solve any FlatZinc model that results from the MiniZinc specification of the predicate, compiled using the standard MiniZinc constraint library, given enough time and memory.

When none of the arguments are provided to the annotation, a similar situation occurs with regards to the pre-solving strategy. Again analysis could be applied to guess which strategy could best be used. Instead the provided implementation uses model-based pre-solving when no argument is provided, as it proved to be the best default. Compared to taking a global pre-solving strategy, model-based pre-solving will never be slower; this follows from the fact that, when using the model-based strategy, the domains of all the decision variables are either smaller than or equal to the ones of the global strategy. Per-call-based pre-solving proved, in our test cases, to be beneficial only in specific cases, as the pre-solve time rapidly increases per call, and the tested CP solver rarely profited from the different sets of solutions.

### 4.2 Placement in the Compiler

To automate the pre-solving process within the MiniZinc compiler, changes have to be made in the compilation process that processes the MiniZinc code and produces FlatZinc code. The goal of these alterations is to produce FlatZinc code that does not include the original constraint(s) imposed by the predicate, but FlatZinc code where these constraints are replaced by one or more constraints, based on pre-solving results, that are equivalent to the original constraint(s).

The process of the MiniZinc to FlatZinc compiler can be described by the following steps:

1. Parsing
2. Type checking
3. Flattening
4. Optimizing

These steps are mostly self-explanatory. The parsing step processes the MiniZinc input and generates a parse tree to be used during the rest of the compilation. During the type checking step all expressions are validated to be of the right type and matched against their declarations where applicable. When flattening, the compiler traverses the MiniZinc parse tree and a FlatZinc parse tree is produced. Finally this FlatZinc parse tree is optimized, including some optional optimizations specific to various solvers, and the FlatZinc output model is produced.

To accommodate automated pre-solving we need to ensure that the marked predicates are not flattened using their implementation, but by their pre-solved results. In the provided implementation, an extra compilation step between the type checking and flattening steps was added to integrate this process. This additional compilation step provides all pre-solving functionality; from finding the pre-solve annotations to replacing the use of the predicate in the MiniZinc parse tree by one or more constraints based on the pre-solving results.

There are other possible approaches that would give similar results. The most prominent one would be to postpone pre-solving until the MiniZinc parse tree has
been transformed into a FlatZinc parse tree. This would be possible by marking the predicates to be pre-solved as internal to the target solver, executing the flattening step without any other alterations, and then replacing all calls to the predicates to be substituted by their pre-solved counterparts. This approach has the benefit of dealing with the flattened calls. In FlatZinc, the calls to the predicate will contain the least amount of unknowns and can contain combined information of various statements in the MiniZinc input.

The provided implementation, however, does not take this approach for various reasons. First of all, implementing only one compilation step contributes to the modular approach of the MiniZinc compiler. None of the other steps are influenced by enabling pre-solving in the compiler, and none of the other steps need to be changed to accommodate pre-solving. Additionally, most of the information gained from pre-solving once the flattening step has been executed can also be gained by using various analyzing functions that are used while flattening the MiniZinc parse tree. There is information that is not easily reproduced: the domain information that can be gained from constraints directly limiting the variable domains (i.e., equalities and inequalities to parameters). These constraints are preempted by the compiler and the information is used in the resulting domains of the decision variables in the FlatZinc parse tree. Using full parse tree traversal this information could also be gained before the flattening step in the compiler. The provided implementation will not search for this information, as it would be too time intensive and because it is best practice to contain this information in the domain of a decision variable; this accommodates compact and understandable models.

4.3 Pre-Solving Predicates

We now describe the specific actions taken in this compilation step to successfully replace predicates to be pre-solved by their pre-solved counterparts. The provided implementation will, for each predicate to be pre-solved, complete the following ordered set of actions:

1. Find the predicate item and the calls to the predicate.
2. Construct a MiniZinc model that finds solutions to the predicate according to the pre-solve strategy.
3. Solve the model for all possible solutions.
4. Replace the usage of the predicate by one or multiple constraints based on the solutions.

Using a top-down traversal of the MiniZinc parse tree, the first step is trivial. All predicates can be found at the top level. For the calls to these predicates, we have to traverse the tree to find them. Although this is not strictly necessary when the pre-solve annotation instructs for a global pre-solving strategy (Section 3.1), the provided implementation will always look for the calls of the predicates to be pre-solved. It has been implemented as such because finding the calls is relatively low-cost and if no calls are found, then we can skip the pre-solving.

Constructing MiniZinc Models for Pre-solving

The construction of the MiniZinc model to find all solutions to a predicate depends on the pre-solving strategy chosen to pre-solve the sub-model, as defined by the annotation to the predicate. Among the pre-solving strategies defined in Section 3.1 the global pre-solving strategy can, for our implementation, be seen as the base case. This strategy will pre-solve the predicate according to its definition. Thus a MiniZinc
array[1..2] of var 1..3: a;
array[1..2] of var 5..8: b;
predicate bar(var int: q, var int: w) = (q == w);
predicate foo(var 0..10: x, var 0..10: y) ::presolve(global) = bar(x,y);
constraint foo(a[1], a[2]);
constraint foo(b[1], b[2]);
solve satisfy;

Figure 4.3: Example MiniZinc model

Figure 4.4: Constructed MiniZinc model to pre-solve the foo predicate using a global strategy for the MiniZinc model in Figure 4.3

model that can pre-solve the predicate in such a way needs to contain the following items:

- The predicate and all its dependencies (referenced global parameters, global decision variables, and predicates).
- Decision variables in accordance to the declaration of the arguments of the predicate (matching their type and domain).
- A constraint item containing a call to the copied predicate using the constructed decision variables.
- A solve item declaring a satisfaction problem (solve satisfy).

An example is given in Figure 4.4, which shows the constructed model for the MiniZinc model shown in Figure 4.3. As mentioned in Section 3.4, reusing the pre-solved predicate for different data instances is possible when the predicate and any of its dependencies have no dependencies on data that changes between these instances, otherwise it can result in invalid results. For MiniZinc it would thus be advisable only to use global pre-solving when all predicate dependencies are part of the model, and not of the data files. For this reason, parameters are in the provided implementation not allowed in the arguments of a predicate that is to be pre-solved using the global pre-solve strategy.

When constructing models for the model-based and the per-call-based strategies, the items contained in the models are essentially the same. The changes made in the construction process are in the domains of the decision variables. By changing the domains of these decision variables, the extent to which the predicate is solved is changed. For the per-call-based strategy, a new model is constructed for each call made to the predicate. The domains of the variables will then be determined by the domains of the values used as arguments in the call to the predicate. For example for the MiniZinc model in Figure 4.3 the constructed model for foo, as shown in Figure 4.4, will be changed so that the decision variables x and y are representative of the domains of the call currently being pre-solved. This means that for the first call...
to \texttt{foo}, the domains for \(x\) and \(y\) would be set to \(1..3\), and for the second call they would be set to \(5..8\). In a similar way, the domains for the decision variables for the model-based strategy are also fashioned using the calls to the predicate. However, instead of using the domains of each call separately, for each of the decision variables the union of all the domains of the arguments of the calls is used, thus leaving us with a single model to pre-solve the predicate for the current model instance. Taking our small example, the decision variables, \(x\) and \(y\), in the constructed model would be changed to have a domain of \(1..3 \cup 5..8\).

Using this implementation for the construction of the MiniZinc models to pre-solve predicates, we can be sure that no valid solutions will be excluded. All of the implemented strategies will have a constructed model that either includes all solutions to the predicate, or at least all solutions as used in the instance of the model. This ensures us of the validity of the results of the pre-solved predicates if all other steps are completed correctly.

\textbf{Processing MiniZinc Models for Pre-Solving}

Once a model is constructed, this model needs to be solved to acquire the solutions to the predicates, and these solutions need to be inserted back, as one or more constraints, into the original model to replace calls to the predicate or the predicate itself.

Solving the constructed models happens in a similar fashion as solving a regular MiniZinc model. In the provided implementation, the constructed model is flattened internally and then sent to a FlatZinc solver. As previously mentioned, it is important that this FlatZinc solver supports complete search, and thus finds all solutions. In the standard used by MiniZinc, a solver that supports such a mode must implement the \(-a\) flag to enable it. If the used solver does not support complete search, then no guarantee can be made that all solutions have been found, which could give invalid results. In the provided implementation, the results of the solver are then stored for use in the reinsertion.

In Section 3.2 various options for data-based constraints where given, which could be used for the reinsertion of data into the original model. In MiniZinc only two of these options are natively available; MiniZinc supports the table constraint (\texttt{table}) and a constraint that implements a (non-)deterministic finite automaton (\texttt{regular}), which equals the functionality of regular expressions. Both these constraints are in MiniZinc limited to specific types. The \texttt{regular} constraint only supports integers and the \texttt{table} constraint has two versions, one for just Boolean data and one for just integer data. Using (automatic) coercion \[G12-14\] both these constraints can support a mix of Boolean and integer variables and data. But because of this limitation, the provided implementation only supports variables that are either Booleans or integers.

The provided implementation uses the \texttt{table} constraint, because, as discussed in Section 3.2, there is no clear advantage to the use of the \texttt{regular} constraint, while the construction is more complex. The \texttt{table} constraint also adheres to the requirements that all pre-solving strategies will have the same (correct) result. Although solving using different strategies might not run with the same speed, it will eliminate the same values.

The construction of the \texttt{table} constraint using the solutions found for the constructed model is straightforward. The MiniZinc \texttt{table} constraint takes two arguments. The first argument is an array of the variables involved in the \texttt{table} constraint and the second argument is a two-dimensional array that contains arrays of possible values for said variables. In our case, this means that the array of variables will contain the variables from the original model that were used for the predicate being pre-solved and the array of values will contain all solutions to the predicate as
array[1..2] of var 1..3: a;
array[1..2] of var 5..8: b;
predicate bar(var int: q, var int: w) = (q == w);
constraint table(a[1], a[2]), array2d(1..3, index_set([a[1], a[2]], [1,1,2,2,3,3]));
constraint table(b[1], b[2]), array2d(1..3, index_set([b[1], b[2]], [5,5,6,6,7,7,8,8]));
solve satisfy;

Figure 4.5: Pre-solved version of the MiniZinc model in Figure 4.3 using a per-call-based strategy

array[1..2] of var 1..3: a;
array[1..2] of var 5..8: b;
predicate bar(var int: q, var int: w) = (q == w);
predicate foo(var 1..3 union 5..8: x, var 1..3 union 5..8: y) = table([x, y], array2d(1..7, index_set([x, y]), [1,1,2,2,3,3,5,5,6,6,7,7,8,8]));
constraint foo(a[1], a[2]);
constraint foo(b[1], b[2]);
solve satisfy;

Figure 4.6: Pre-solved version of the MiniZinc model in Figure 4.3 using a model-based strategy

provided by the chosen FlatZinc solver. There are, however, a few exceptions that have to be made. In case of mixed types of variables and data, coercion is used to fit the data into an integer table constraint. Furthermore, if the predicate contained (multidimensional) arrays of variables as arguments, then both the variables and its data will be made to fit in the arguments of the table call. This will result in a table call that can be used in the original model.

The insertion of this constraint depends, as with the construction of the models to be pre-solved, on the pre-solve strategy. When using the per-call-based pre-solving strategy, we replace each of the calls to the predicate with the generated call to the table constraint. This can be shown using our previously used MiniZinc model, shown in Figure 4.3. Pre-solving the foo predicate will result in the MiniZinc model given in Figure 4.5. In the provided implementation, the predicate itself is removed from the model when using this strategy, to indicate that it is not used. Its dependencies, like the predicate bar on line 3 in Figure 4.5, will remain in place as they might be used in other parts of the models.

When using a global or model-based strategy, instead of replacing the calls to the predicate, the content of the predicate is replaced. By replacing the original content of the predicate we can more efficiently insert the generated call to the table constraint, as there is only one site that needs to be replaced. It, furthermore, enables the user to reuse the pre-solved predicates in different models or instances of the same model, provided that there are no dependencies specific to the instance of the model. To support this in the model-based strategy, the domains of the arguments of the predicates are changed to match the domain for which the predicate is solved upon reinsertion. The pre-solved result of a model based strategy on the model of Figure 4.3 is shown in Figure 4.6.
The use of automated pre-solving can best be shown with practical examples. In this chapter, we will show a few cases in which pre-solving is beneficial to the run time of the model and show how the additions to the MiniZinc toolchain make the pre-solving easier for the modeler. We also show that for these instances the use of our pre-solve annotations leaves the model more complete and insightful because the definition of the predicates are left in the model.

All benchmarks in the test cases are performed on a machine with the specifications supplied in Table 5.1. For each of the benchmarks, the provided implementation’s mzn2fzn is used to generate the FlatZinc version of the model, as supplied by the installation instructions in Appendix A. For both solving and pre-solving fzn-gecode is used, which is based on Gecode version 4.4.0. To maximize the number of native propagators used, the Gecode MiniZinc library is used while flattening the MiniZinc models (by providing -G gecode in the flags for mzn2fzn). Statistics are gathered using the native statistics supplied by fzn-gecode’s “emit statistics” functionality and pre-solving times are gathered using the verbose mode of mzn2fzn, which times the steps of the presolving process. Neither of the tools operate using concurrency, and all processes can thus be seen as run on a single-core machine. All run times mentioned in the various tables are averages over 5 runs of the specific problem instance, to allow for variance in system performance as well as for randomisation during search.

5.1 Black Hole Patience

The black hole patience model, as available in the MiniZinc Benchmark Suite [Min12], is based on the likewise named card game. The player is presented with 17 fans of 3 cards. These fans will contain all cards except for the ace of spades, which will be placed in the middle of the table, the black hole. Once all cards are in position, the goal of the game is to move all cards into the black hole. However, the player is only allowed to move the cards on the top of the fans, and a card can only go into the black hole if it is one rank apart from the top card of the black hole. (e.g., after an eight comes either a seven or a nine, and after an ace comes either a king or a two.)

In the model, the cards are emulated by decision variables. There is a decision variable both for which card is in which position in the black hole, and for which position every card holds. As these variables express identical information in a different way, a channeling constraint is used to keep the domains of these variables consistent. The layout of the fans is used as the input for the model. The constraints used to enforce the rules of the game are then easily expressed.
Operating System: OpenSUSE 13.1  
Architecture: x86_64  
CPU Model: Intel® Core™ i7 CPU 950 @ 3.07GHz  
L1d cache: 32KB  
L1i cache: 32KB  
L2 cache: 256KB  
L3 cache: 8192KB  

Table 5.1: Specifications of the machine running the benchmarks

1 % Consecutive cards match
2 predicate rank_apart(var 1..52: a, var 1..52: b)
3 :: presolve(global, "fzn-gecode")
4 = abs( (a - b) mod 13 ) in {1,12};

Figure 5.1: rank_apart predicate as found in the pre-solve version of the black hole patience model.

**Rank-Apart Constraint**

Although all constraints in the model could be easily expressed, the constraint enforcing that the cards in the black hole must be one rank apart, in the original model, is already a *table* constraint. The *table* constraint contains an array with all possible combinations that two cards can have and be one rank apart. This manual pre-solving gives us an ideal opportunity to test our automated pre-solving as provided by the extended compiler.

Using the pre-solve syntax as in the provided implementation, the table used in the original model can be replaced by the predicate shown in Figure 5.1. As this predicate does not contain any use of parameters and its variables are tied to a rather specific domain, it is warranted to use the global pre-solving strategy. The second argument of the annotation expresses our use of the *fzn-gecode* solver.

In accordance with the use of the “neighbor”-table in the original model, the *rank_apart* predicate is applied to all decision variables expressing the cards in the positions on top of each other in the black hole.

**Benchmark Results**

The full version of the pre-solve model can be found in Appendix B.1. The complete results for all instances of the black hole model can be found in Appendix B.2. Benchmarks have been run on the original model as found in the MiniZinc Benchmark Suite [Min12], on the previously mentioned pre-solve model, and on the same model with pre-solving disabled.

Table 5.2 shows the average run times of the various models for the model instances where the reference model can find a solution within 15 minutes, the same timeout that was used for the MiniZinc Challenge 2009 [Min09], where this model was introduced. From this model we can safely conclude that the performance of the model that was pre-solved by the compiler has results equaling the results of the manually pre-solved reference model. As this predicate is pre-solved using the *global* strategy, it only had to be pre-solved once for all problem instances. The time it takes the compiler to pre-solve this constraint averages 11 milliseconds. Compared to the non pre-solved model, the speedups range from 28 times to more than a thousand times. These results are very encouraging for the use of the pre-solve model, as its results are equivalent to
the reference model, but this leaves the modeler open to have insight in the predicate and how it functions. They furthermore show the impact that pre-solving can have on the run time of a model.

**Other Solvers**

An advantage of targeting the MiniZinc toolchain, instead of a specific solver, is that multiple solvers, and solving techniques, might benefit from the effects of the improvements made. For the black hole patience model we tested the benefits of pre-solving on other, non Constraint Programming, solvers; we tested using a mixed-integer programming solver, Gurobi, a local search solver, OscaR-CBLS, an SMT solver, fzn2smt, and a lazy clause generation solver, Chuffed.

In Tables 5.2 and 5.3 the results from our tests with Chuffed and fzn2smt are shown. The results for Chuffed show a significant speedup over the non pre-solved results. The run times for fzn2smt are significantly larger than the run times using Gecode and Chuffed, but still show a significant and consistent speedup. These results indicate that pre-solving can be very beneficial for these solvers.

Gurobi and OscaR-CBLS were unable to complete solving both the pre-solved and the non pre-solved model within a reasonable amount of time. Although we speculate that, with a good implementation of the table constraint, pre-solving might be beneficial for these solvers, or solvers using the same solving technique, it is not shown in our tests. We expect that the black hole patience model does not lend itself well to the techniques that these solvers are based on and that there are other models

<table>
<thead>
<tr>
<th>Data</th>
<th>Speedup</th>
<th>Pre-solved Model (s)</th>
<th>Non Pre-solved Model (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.dzn</td>
<td>30.9</td>
<td>0.30</td>
<td>9.27</td>
</tr>
<tr>
<td>1.dzn</td>
<td>47.1</td>
<td>0.09</td>
<td>4.24</td>
</tr>
<tr>
<td>3.dzn</td>
<td>3.0</td>
<td>0.17</td>
<td>0.51</td>
</tr>
<tr>
<td>5.dzn</td>
<td>11.7</td>
<td>0.29</td>
<td>3.38</td>
</tr>
<tr>
<td>7.dzn</td>
<td>10.0</td>
<td>0.19</td>
<td>1.90</td>
</tr>
<tr>
<td>9.dzn</td>
<td>1.5</td>
<td>3.24</td>
<td>5.02</td>
</tr>
<tr>
<td>12.dzn</td>
<td>11.7</td>
<td>0.59</td>
<td>6.88</td>
</tr>
<tr>
<td>13.dzn</td>
<td>5.4</td>
<td>0.20</td>
<td>1.07</td>
</tr>
<tr>
<td>18.dzn</td>
<td>5.0</td>
<td>2.67</td>
<td>13.46</td>
</tr>
<tr>
<td>20.dzn</td>
<td>5.5</td>
<td>12.80</td>
<td>70.16</td>
</tr>
</tbody>
</table>
Table 5.4: Run times from the benchmark results for the black hole patience model using fzn2smt

<table>
<thead>
<tr>
<th>Data</th>
<th>Speedup</th>
<th>Pre-solved Model (s)</th>
<th>Non Pre-solved Model (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.dzn</td>
<td>46.5</td>
<td>630</td>
<td>29254</td>
</tr>
<tr>
<td>1.dzn</td>
<td>26.6</td>
<td>519</td>
<td>13806</td>
</tr>
<tr>
<td>3.dzn</td>
<td>82.9</td>
<td>140</td>
<td>11571</td>
</tr>
<tr>
<td>5.dzn</td>
<td>12.4</td>
<td>417</td>
<td>5171</td>
</tr>
<tr>
<td>7.dzn</td>
<td>2.7</td>
<td>2072</td>
<td>5551</td>
</tr>
<tr>
<td>9.dzn</td>
<td>12.2</td>
<td>141</td>
<td>1728</td>
</tr>
<tr>
<td>12.dzn</td>
<td>14.8</td>
<td>518</td>
<td>7643</td>
</tr>
<tr>
<td>13.dzn</td>
<td>54.8</td>
<td>354</td>
<td>19401</td>
</tr>
<tr>
<td>18.dzn</td>
<td>81.9</td>
<td>515</td>
<td>42162</td>
</tr>
</tbody>
</table>

Figure 5.2: A picture of the Block Party Game (Taken from: Three Pixel Heart LLC, source: Kickstarter project page)

that would show the benefit of pre-solving for these solvers.

5.2 Block Party

Honoring its name, the block party model is based on the block party game [TPHa]; a picture of the game is shown in Figure 5.2. The game consists of eight cubes, where each corner of a side of a cube has been outfitted with an icon. Icons can differ in the following attributes: their shape, their color, and their pattern. The goal of the game is to place four of the eight cubes into a square in such a way that the icons in the center of the square are for each of the attributes either identical, or all different (e.g., four icons with a heart shape, in four different colors, and a solid pattern; or four icons each having a different shape, colored black, and each with a different pattern). In the original game, this is played with multiple players: they look for these patterns, also referred to as parties, on the sides of the cubes facing upwards. When a pattern is found the player scores points and the cubes involved are rolled so a different side might face upwards.

The model, included with the provided implementation (see Appendix C), solves a puzzle variation of the game where instead of arranging just four of the cubes, and looking for a party on one side, all eight cubes are arranged into a cube and a party
% Linking cube IDs with the three symbols appearing in relevant positions

```solidity
predicate link_cube_and_symbols(array[1..4] of var int: cs) 
::presolve(model, "fzn-gecode")
= let{
  var 1..24: pos;
  var int: cube = cs[1];
} in forall(i in 1..3)(data[cube,pp[pos,i]]=cs[i+1]);
```

Figure 5.3: link_cube_and_symbols predicate as found in the block party patience model.

must be present on every side, this variation is named “The METACUBE” [TPHa]. This puzzle was given as a challenge by the developers of the game and was first solved by Jean-Noël Monette using this model [TPHa]. The goal of the model was to find a viable solution and prove that there was only one solution.

**Linking Cubes and Symbols**

In the model the positions of the icons on each of the cubes are constrained using the link_cube_and_symbols predicate, as shown in Figure 5.3. Note that the array used as an argument, cs, contains both the cube identifier and the icons to be constrained. The pp array is a positional array used to express the positional relation between icons. This choice of variables and constraints makes it easier to model the other constraints. However, the necessary reification is a strain on the runtime.

In order to improve the runtime of this model, pre-solving can be used for the link_cube_and_symbols predicate. The choice of pre-solving strategy is limited to either model- or call-based pre-solving, as the predicate makes use of the data array, which is changed for every instance of the model. As the calls to the predicate have the same domains, pre-solving in a model-based approach is the best solution, as it avoids pre-solving separately for every call. The annotation also shows the use of the fzn-gecode solver instead of the default solver.

**Benchmark Results**

The complete benchmark results for the Block Party model can be found in Appendix C. The data used in the original model was based on the configuration of the blocks as sold by Three Pixel Heart LLC: we considered this data to be proprietary. So for benchmark purposes a data-generating MiniZinc model is provided that produces icon configurations that can be used for the model using a randomized search strategy that have similar run times to the original problem instance. The model is run as:

```
mzn-gecode block-party-generate.mzn -r <n> > <n>.dzn
```

where <n> is the seed used for the random branching in the model. For the benchmarks performed in this thesis random numbers are used as seeds and their resulting data is then used as input for our benchmarks. These benchmarks run a complete search on the models to find the number of possible solutions to the model.

Table 5.5 shows the run times for our generated data instances. The pre-solved results are very positive: with an added solve time of only 18 milliseconds on average, the run time of the total model drops more than 6.5 times on average. That the pre-solving of this predicate is an improvement to this model also shows in the fact that the number of failures is cut in half and the number of variables is reduced by more than 50. These results are once again highly encouraging for our automated
The pre-solving approach: it is easily implemented, the pre-solve times are low, and the speedup gained by pre-solving is high. This demonstrates that automated pre-solving can be a valuable tool for a constraint modeler.

<table>
<thead>
<tr>
<th>Data</th>
<th>Pre-solve time (ms)</th>
<th>Run time (ms)</th>
<th>Run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1176.dzn</td>
<td>18</td>
<td>4024</td>
<td>26632</td>
</tr>
<tr>
<td>1266.dzn</td>
<td>18</td>
<td>3892</td>
<td>25955</td>
</tr>
<tr>
<td>2254.dzn</td>
<td>18</td>
<td>3921</td>
<td>26529</td>
</tr>
<tr>
<td>3237.dzn</td>
<td>18</td>
<td>3861</td>
<td>25716</td>
</tr>
<tr>
<td>3345.dzn</td>
<td>19</td>
<td>4121</td>
<td>26238</td>
</tr>
<tr>
<td>412.dzn</td>
<td>18</td>
<td>4159</td>
<td>27359</td>
</tr>
<tr>
<td>4629.dzn</td>
<td>18</td>
<td>4075</td>
<td>26508</td>
</tr>
<tr>
<td>7059.dzn</td>
<td>18</td>
<td>4012</td>
<td>26944</td>
</tr>
<tr>
<td>7324.dzn</td>
<td>17</td>
<td>4069</td>
<td>27077</td>
</tr>
<tr>
<td>7326.dzn</td>
<td>19</td>
<td>3924</td>
<td>26400</td>
</tr>
<tr>
<td>7444.dzn</td>
<td>18</td>
<td>4007</td>
<td>26379</td>
</tr>
<tr>
<td>7661.dzn</td>
<td>19</td>
<td>3974</td>
<td>25550</td>
</tr>
<tr>
<td>7750.dzn</td>
<td>19</td>
<td>4164</td>
<td>26714</td>
</tr>
<tr>
<td>8848.dzn</td>
<td>18</td>
<td>4016</td>
<td>25834</td>
</tr>
</tbody>
</table>

Table 5.5: Run times from the benchmark results for the Block Party model
Related Work

The work most closely related to this thesis is the strong constraint in the OPL language [BM15], which is used in the CPLEX Optimization Studio. For instances of the strong constraint, the set of constrained variables and their constraints are pre-solved. Each strong constraint is then replaced by a constraint that contains the solution tuples of the model on the variables of the call to the strong constraint. This approach is very similar to the method used in this thesis, but differs on a few key points. The strong constraint only offers a black-box method. It does not allow the modeler to choose how the strong constraints are solved, thus forcing the solver to preprocess each instance of the strong constraint separately even though instances could have more efficiently been solved together. It furthermore limits the model to a choice of variables instead of a combination of variables and constraints. This can affect the time necessary to perform the pre-solving.

Furthermore, research was also conducted into a different area of pre-solving within MiniZinc [LT15]. There, MiniZinc models are pre-solved using bounds propagators and other forms of specialized analysis to tighten the bounds of the decision variables.

The practical use of pre-solving, as in the pre-computation of constraints, in scheduling problems has also been shown in recent research [CJL16]. This research shows that the use of pre-solving can lead to a considerable increase in performance.

Finally, there is the work on propagator generation to prune the search tree faster. In particular, there is research that enables the use of domain-independent propagators [LW92], enabling the use of new inference rules within constraint logic programming. In [AR04], a solving approach is used to generate propagators and simplification rules to achieve more expressive and efficient models. A recent study [MFP15] demonstrates a way to eliminate auxiliary decision variables through the use of propagator generation. The research into propagator generation is fundamentally the same as the research on pre-solving. Both focussed on the use of pre-computation to enable the use of more efficient constraint propagators. Propagator generation is, however, focussed on the use of a specific solver, whereas pre-solving can be beneficial to all (combinations of) solvers.
Conclusions and Future Work

This thesis introduces a technique to automate the process of pre-solving part of a constraint model. Through the explanation of the process, it is shown that the provided extension of the MiniZinc toolchain supplies modelers with a way to automatically pre-solve sub-models in a MiniZinc model without losing the correctness of the results. Although it was already shown previously that pre-solving can drastically improve the performance of a model, it is a big time investment for the modeler. Using anecdotal evidence, this thesis shows the ease with which a sub-model can be automatically pre-solved, while leaving the modeler with options as to how it should be done.

Future work on the implementation of automated pre-solving in the MiniZinc toolchain could include the addition of new features to the pre-solving step. Most notably, the implementation could offer the modeler the choice of which constraint to use when reinserting the pre-solved solutions to the model. This could involve research into other constraints in MiniZinc that might perform better than the table constraint (in at least some situations), but could also enable the support for MiniZinc features that are currently unsupported in our implementation of pre-solving (e.g., float and set variables). Other features might be added to help modelers. One could, for instance, add a timeout feature to the pre-solving feature, where if a sub-model cannot be pre-solved within a set time limit, the original MiniZinc implementation is used. Another possibility is adding more advanced defaults for the choice of pre-solve strategy and choice of solver to match them better to the contents of a predicate.

In general, there has not been a lot of work on the use of pre-solving within constraint modeling environments. More work on the effects of pre-solving within different solving techniques would be beneficial. Future work into this subject could also include research into the formation of sub-models and formal approaches to find them. Such research could be the start of fully automated solver-based sub-model pre-solving, where the solver would decide if pre-solving might be beneficial. Furthermore, research could be done into more solver involvement in pre-solving. The information that could be gathered during pre-solving could be beneficial to the solving of the whole model. It could, for instance, provide indications towards a good branching strategy.
Appendices
The provided implementation of the MiniZinc compiler that has been extended to perform automated predicate pre-solving can be found on github [Dek16]. This extended version of the MiniZinc compiler is based on the beta version of MiniZinc version 2.1.0 (tagged as 2.1.0b3). The extension of the compiler does not influence the performance of the compiler unless pre-solve annotations are used. The version described in this thesis has been tagged thesis; if changes are made the newest version of this extension is available on the feature/presolve branch.

A.1 Compilation using CMake

The compilation of this extended version of the compiler is similar to the compilation of the regular version of the MiniZinc compiler. Once compiled, the binaries can be installed by placing them on the executable path of the system. Compiling the source code requires CMake and a recent version of a C++ compiler. If your system meets these requirements, the following shell statements should be able to compile the extended MiniZinc compiler when executed in the source folder:

```
mkdir build
cd build/
cmake -D PRESOLVE:BOOL=TRUE ..
cmake --build .
```

In this extended version of the compiler, all executables that compile MiniZinc models into FlatZinc will be extended with the ability to pre-solve predicates. When using the pre-solve functionality, make sure that the MiniZinc library is provided in the repository (in share/minizinc) is used, otherwise the compiler will not recognize the pre-solve annotation. Various executables might use different default solvers to pre-solve the predicates (i.e., mzn-gecode uses the Gecode backend by default), however if they do not support complete search, they won’t produce correct results.

A.2 Added Functionality

Other than the actual pre-solving functionality as described in Chapter 4 the extended version of the compiler includes a few extra functions to accommodate the modeler.

By adding a flag on the command line (`--no-presolve`), pre-solving can be disabled. When this flag is set, all pre-solve annotations will be ignored. This can be particularly useful when comparing the performance of models using pre-solving and without pre-solving.
When using verbose mode (command line flag \(-v\)), the extended compiler will output the time spent in each pre-solving step. This is again something that can be taken into account when running benchmarks on a model, but can also be very useful when debugging a model. The output when pre-solving a predicate should look similar to this:

```
Presolving ...
  Presolving 'foo' ...
    Constructing predicate model ... done (7 ms)
    Presolving problem ... done (19 ms)
    Inserting solutions ... done (0 ms)
  done (26 ms)
  done (31 ms)
```

Finally, it is possible to output the pre-solved MiniZinc model. This can be done by adding the flag \(--\text{presolved}\) followed by a filename. Note that using a filename that is already in use will overwrite the file that already existed.
In the Black Hole Patience test case, as found in Section 5.1 two models were used. The first model is the reference model, found in the MiniZinc benchmark suite [Min12] and used in the MiniZinc Challenge of 2009 [Min09]. The second model is a variation on the first model, where the table constraint is replaced by a predicate to be pre-solved. For both models the data that is provided for the first model can be used. Section B.1 describes the changes to be made to the first model to use pre-solving according to the provided implementation. Furthermore, we include in Section B.2 the raw statistics as generated by our benchmarks. More information on this test case can be found in Section 5.1.

B.1 Pre-Solve Model

Using a few changes, the model for black hole patience, as provided in the MiniZinc benchmark suite [Min12], can use automated pre-solving instead of the manual pre-solved table. To recreate the model as used for our benchmarks, replace the constraint statement in Figure B.1 with the predicate and constraint statement in Figure B.2. Optionally the neighbours table can be removed from the model; this will have no impact on the runtime of the produced FlatZinc model. Note that for the benchmarks the Gecode FlatZinc solver was used: to use this solver the pre-solve annotation in Figure B.2 needs to be extended with the solver location, as described in Section 4.1.

97 \% Consecutive cards match
98 constraint
99  forall(i in 1..51) (  
100    table([x[i], x[i+1]], neighbours)  
101      );

Figure B.1: Table constraint in the black hole patience model to enforce the rank-apart constraint

97 \% Consecutive cards match
98 predicate rank_apart(var 1..52: a, var 1..52: b)
99    ::= presolve(global)
100      = abs( (a - b) mod 13 ) in {1,12};
101 constraint forall(i in 1..51) ( rank_apart(x[i], x[i+1]) );

Figure B.2: Replacement rank_apart predicate and constraint to use automated pre-solving under the global strategy
B.2 Benchmark Results

The following three tables contain the raw benchmark results generated by our Black Hole Patience test case. Note that the different number of propagators for instances 18.dzn and 19.dzn in the pre-solved models is no mistake. For these instances the MiniZinc compiler is able to eliminate the necessity of two of the propagators. Further discussion of these results can be found in Section 5.1.

<table>
<thead>
<tr>
<th>Data</th>
<th>Run time (ms)</th>
<th>Variables</th>
<th>Propagators</th>
<th>Propagations</th>
<th>Nodes</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.dzn</td>
<td>292</td>
<td>105</td>
<td>86</td>
<td>355774</td>
<td>10067</td>
<td>5018</td>
</tr>
<tr>
<td>1.dzn</td>
<td>52</td>
<td>105</td>
<td>86</td>
<td>49790</td>
<td>893</td>
<td>431</td>
</tr>
<tr>
<td>2.dzn</td>
<td>time &gt; 900000</td>
<td>105</td>
<td>86</td>
<td>1022221215</td>
<td>19601615</td>
<td>9800796</td>
</tr>
<tr>
<td>3.dzn</td>
<td>104</td>
<td>105</td>
<td>86</td>
<td>114967</td>
<td>2114</td>
<td>1044</td>
</tr>
<tr>
<td>4.dzn</td>
<td>time &gt; 900000</td>
<td>105</td>
<td>86</td>
<td>1037546162</td>
<td>22875488</td>
<td>11437736</td>
</tr>
<tr>
<td>5.dzn</td>
<td>519</td>
<td>105</td>
<td>86</td>
<td>617766</td>
<td>15516</td>
<td>7744</td>
</tr>
<tr>
<td>6.dzn</td>
<td>10</td>
<td></td>
<td></td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.dzn</td>
<td>163</td>
<td>105</td>
<td>86</td>
<td>201791</td>
<td>4938</td>
<td>2453</td>
</tr>
<tr>
<td>8.dzn</td>
<td>10</td>
<td></td>
<td></td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.dzn</td>
<td>16141</td>
<td>105</td>
<td>86</td>
<td>16804376</td>
<td>346973</td>
<td>173470</td>
</tr>
<tr>
<td>10.dzn</td>
<td>10</td>
<td></td>
<td></td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.dzn</td>
<td>time &gt; 900000</td>
<td>105</td>
<td>86</td>
<td>1168261346</td>
<td>27330800</td>
<td>13665391</td>
</tr>
<tr>
<td>12.dzn</td>
<td>573</td>
<td>105</td>
<td>86</td>
<td>686026</td>
<td>14031</td>
<td>7003</td>
</tr>
<tr>
<td>13.dzn</td>
<td>124</td>
<td>105</td>
<td>86</td>
<td>164057</td>
<td>6714</td>
<td>3343</td>
</tr>
<tr>
<td>14.dzn</td>
<td>time &gt; 900000</td>
<td>105</td>
<td>86</td>
<td>1100192892</td>
<td>23243025</td>
<td>11621503</td>
</tr>
<tr>
<td>15.dzn</td>
<td>time &gt; 900000</td>
<td>105</td>
<td>86</td>
<td>1220680445</td>
<td>30326564</td>
<td>15163271</td>
</tr>
<tr>
<td>16.dzn</td>
<td>time &gt; 900000</td>
<td>105</td>
<td>86</td>
<td>1206736103</td>
<td>39162492</td>
<td>19581234</td>
</tr>
<tr>
<td>17.dzn</td>
<td>11</td>
<td></td>
<td></td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.dzn</td>
<td>10965</td>
<td>105</td>
<td>84</td>
<td>14472918</td>
<td>312905</td>
<td>156436</td>
</tr>
<tr>
<td>19.dzn</td>
<td>time &gt; 900000</td>
<td>105</td>
<td>84</td>
<td>1156547188</td>
<td>29188505</td>
<td>14594243</td>
</tr>
<tr>
<td>20.dzn</td>
<td>20412</td>
<td>105</td>
<td>86</td>
<td>28945884</td>
<td>916488</td>
<td>458229</td>
</tr>
</tbody>
</table>

Table B.1: Raw benchmark results for the black hole patience model as found in the MiniZinc Benchmark Suite [Min12].
### Pre-solved Model

<table>
<thead>
<tr>
<th>Data</th>
<th>Pre-solve time (ms)</th>
<th>Run time (ms)</th>
<th>Variables</th>
<th>Propagators</th>
<th>Propagations</th>
<th>Nodes</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. dzn</td>
<td>10</td>
<td>294</td>
<td>104</td>
<td>86</td>
<td>355774</td>
<td>10067</td>
<td>5018</td>
</tr>
<tr>
<td>1. dzn</td>
<td>11</td>
<td>52</td>
<td>104</td>
<td>86</td>
<td>49790</td>
<td>893</td>
<td>431</td>
</tr>
<tr>
<td>2. dzn</td>
<td>11</td>
<td>time &gt; 900000</td>
<td>104</td>
<td>86</td>
<td>1033429090</td>
<td>19824431</td>
<td>9912202</td>
</tr>
<tr>
<td>3. dzn</td>
<td>11</td>
<td>104</td>
<td>86</td>
<td>114967</td>
<td>2114</td>
<td>1044</td>
<td></td>
</tr>
<tr>
<td>4. dzn</td>
<td>10</td>
<td>time &gt; 900000</td>
<td>104</td>
<td>86</td>
<td>1046054779</td>
<td>23056310</td>
<td>11528147</td>
</tr>
<tr>
<td>5. dzn</td>
<td>11</td>
<td>519</td>
<td>104</td>
<td>86</td>
<td>617766</td>
<td>15516</td>
<td>7744</td>
</tr>
<tr>
<td>6. dzn</td>
<td>10</td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. dzn</td>
<td>11</td>
<td>164</td>
<td>104</td>
<td>86</td>
<td>201791</td>
<td>4938</td>
<td>2453</td>
</tr>
<tr>
<td>8. dzn</td>
<td>10</td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. dzn</td>
<td>10</td>
<td>16078</td>
<td>104</td>
<td>86</td>
<td>16804376</td>
<td>346973</td>
<td>173470</td>
</tr>
<tr>
<td>10. dzn</td>
<td>10</td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. dzn</td>
<td>10</td>
<td>time &gt; 900000</td>
<td>104</td>
<td>86</td>
<td>1174819041</td>
<td>27488032</td>
<td>13744003</td>
</tr>
<tr>
<td>12. dzn</td>
<td>10</td>
<td>569</td>
<td>104</td>
<td>86</td>
<td>686026</td>
<td>14031</td>
<td>7003</td>
</tr>
<tr>
<td>13. dzn</td>
<td>10</td>
<td>124</td>
<td>104</td>
<td>86</td>
<td>164057</td>
<td>6714</td>
<td>3343</td>
</tr>
<tr>
<td>14. dzn</td>
<td>11</td>
<td>time &gt; 900000</td>
<td>104</td>
<td>86</td>
<td>1109829880</td>
<td>23440395</td>
<td>11720187</td>
</tr>
<tr>
<td>15. dzn</td>
<td>11</td>
<td>time &gt; 900000</td>
<td>104</td>
<td>86</td>
<td>1223141591</td>
<td>30388926</td>
<td>15194903</td>
</tr>
<tr>
<td>16. dzn</td>
<td>11</td>
<td>time &gt; 900000</td>
<td>104</td>
<td>86</td>
<td>1214975634</td>
<td>39399461</td>
<td>19699719</td>
</tr>
<tr>
<td>17. dzn</td>
<td>11</td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18. dzn</td>
<td>10</td>
<td>10924</td>
<td>104</td>
<td>84</td>
<td>14472918</td>
<td>312905</td>
<td>156436</td>
</tr>
<tr>
<td>19. dzn</td>
<td>10</td>
<td>time &gt; 900000</td>
<td>104</td>
<td>84</td>
<td>1166746225</td>
<td>29463008</td>
<td>14731494</td>
</tr>
<tr>
<td>20. dzn</td>
<td>11</td>
<td>20270</td>
<td>104</td>
<td>86</td>
<td>28965884</td>
<td>916488</td>
<td>458229</td>
</tr>
</tbody>
</table>

### Non Pre-solved Model

<table>
<thead>
<tr>
<th>Data</th>
<th>Run time (ms)</th>
<th>Variables</th>
<th>Propagators</th>
<th>Propagations</th>
<th>Nodes</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. dzn</td>
<td>375395</td>
<td>257</td>
<td>289</td>
<td>11091116217</td>
<td>43485691</td>
<td>21742824</td>
</tr>
<tr>
<td>1. dzn</td>
<td>22560</td>
<td>257</td>
<td>289</td>
<td>71392566</td>
<td>2683799</td>
<td>1341879</td>
</tr>
<tr>
<td>2. dzn</td>
<td>time &gt; 900000</td>
<td>257</td>
<td>289</td>
<td>1817324160</td>
<td>78797413</td>
<td>39398693</td>
</tr>
<tr>
<td>3. dzn</td>
<td>2963</td>
<td>257</td>
<td>289</td>
<td>8209348</td>
<td>330373</td>
<td>165169</td>
</tr>
<tr>
<td>4. dzn</td>
<td>time &gt; 900000</td>
<td>257</td>
<td>289</td>
<td>2217325078</td>
<td>98538776</td>
<td>49269370</td>
</tr>
<tr>
<td>5. dzn</td>
<td>26044</td>
<td>257</td>
<td>289</td>
<td>59232998</td>
<td>2592186</td>
<td>1299073</td>
</tr>
<tr>
<td>6. dzn</td>
<td>10</td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. dzn</td>
<td>30426</td>
<td>257</td>
<td>289</td>
<td>83853929</td>
<td>3406066</td>
<td>1715282</td>
</tr>
<tr>
<td>8. dzn</td>
<td>10</td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. dzn</td>
<td>702732</td>
<td>257</td>
<td>289</td>
<td>1485851438</td>
<td>63151000</td>
<td>31575478</td>
</tr>
<tr>
<td>10. dzn</td>
<td>10</td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. dzn</td>
<td>time &gt; 900000</td>
<td>257</td>
<td>289</td>
<td>2144091036</td>
<td>96571843</td>
<td>48285904</td>
</tr>
<tr>
<td>12. dzn</td>
<td>109030</td>
<td>257</td>
<td>289</td>
<td>260246092</td>
<td>11486003</td>
<td>5742979</td>
</tr>
<tr>
<td>13. dzn</td>
<td>4404</td>
<td>257</td>
<td>289</td>
<td>12723594</td>
<td>506527</td>
<td>252794</td>
</tr>
<tr>
<td>14. dzn</td>
<td>time &gt; 900000</td>
<td>257</td>
<td>289</td>
<td>2170854321</td>
<td>84890449</td>
<td>42445212</td>
</tr>
<tr>
<td>15. dzn</td>
<td>time &gt; 900000</td>
<td>257</td>
<td>289</td>
<td>1910589551</td>
<td>94829208</td>
<td>47414587</td>
</tr>
<tr>
<td>16. dzn</td>
<td>time &gt; 900000</td>
<td>257</td>
<td>289</td>
<td>2500128196</td>
<td>104364191</td>
<td>52182077</td>
</tr>
<tr>
<td>17. dzn</td>
<td>11</td>
<td>=========UNSATISFIABLE========</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18. dzn</td>
<td>1430218</td>
<td>257</td>
<td>289</td>
<td>1916055305</td>
<td>84275704</td>
<td>42362835</td>
</tr>
<tr>
<td>19. dzn</td>
<td>time &gt; 900000</td>
<td>257</td>
<td>289</td>
<td>2279560566</td>
<td>96395827</td>
<td>48197898</td>
</tr>
<tr>
<td>20. dzn</td>
<td>2011730</td>
<td>257</td>
<td>289</td>
<td>2797044387</td>
<td>105506906</td>
<td>52753435</td>
</tr>
</tbody>
</table>

Table B.2: Raw benchmark results for the black hole patience model with pre-solving

Table B.3: Raw benchmark results for the black hole patience model without pre-solving
The Block Party model and corresponding data-generating model, as used for the test case, are distributed together with the extended MiniZinc compiler [Dek16]. (For more information about the repository, see the installation instructions in Appendix A) The models can be found in the tests/examples folder and are named block_party.mzn and block_party_generate.mzn.

The following two tables contain the raw benchmark results generated by our Block Party test case. More information on this test case can be found in Section 5.2.

<table>
<thead>
<tr>
<th>data</th>
<th>Pre-solve time (ms)</th>
<th>Run time (ms)</th>
<th>Solutions</th>
<th>Variables</th>
<th>Propagators</th>
<th>Propagations</th>
<th>Nodes</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1176.dzn</td>
<td>18</td>
<td>4024</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>12988899</td>
<td>149073</td>
<td>74536</td>
</tr>
<tr>
<td>1266.dzn</td>
<td>18</td>
<td>3892</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>12684225</td>
<td>141913</td>
<td>70956</td>
</tr>
<tr>
<td>2254.dzn</td>
<td>18</td>
<td>3921</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>12807897</td>
<td>142659</td>
<td>71329</td>
</tr>
<tr>
<td>3237.dzn</td>
<td>18</td>
<td>3861</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>12564684</td>
<td>141851</td>
<td>70925</td>
</tr>
<tr>
<td>3345.dzn</td>
<td>19</td>
<td>4121</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>13420950</td>
<td>152951</td>
<td>76475</td>
</tr>
<tr>
<td>412.dzn</td>
<td>18</td>
<td>4159</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>13498743</td>
<td>155655</td>
<td>77827</td>
</tr>
<tr>
<td>4629.dzn</td>
<td>18</td>
<td>4075</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>13182654</td>
<td>150185</td>
<td>75092</td>
</tr>
<tr>
<td>7059.dzn</td>
<td>18</td>
<td>4012</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>13079636</td>
<td>148955</td>
<td>74477</td>
</tr>
<tr>
<td>7324.dzn</td>
<td>17</td>
<td>4069</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>13253786</td>
<td>149657</td>
<td>74828</td>
</tr>
<tr>
<td>7326.dzn</td>
<td>19</td>
<td>3924</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>12759491</td>
<td>146829</td>
<td>73414</td>
</tr>
<tr>
<td>7444.dzn</td>
<td>18</td>
<td>4007</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>13063073</td>
<td>147123</td>
<td>73561</td>
</tr>
<tr>
<td>7661.dzn</td>
<td>19</td>
<td>3974</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>12918229</td>
<td>146995</td>
<td>73497</td>
</tr>
<tr>
<td>7750.dzn</td>
<td>19</td>
<td>4164</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>13544511</td>
<td>154065</td>
<td>77032</td>
</tr>
<tr>
<td>8848.dzn</td>
<td>18</td>
<td>4016</td>
<td>1</td>
<td>357</td>
<td>305</td>
<td>13045039</td>
<td>146817</td>
<td>73408</td>
</tr>
</tbody>
</table>

Table C.1: Raw benchmark results for the block party model with pre-solving
<table>
<thead>
<tr>
<th>Data</th>
<th>Run time (ms)</th>
<th>Solutions</th>
<th>Variables</th>
<th>Propagators</th>
<th>Propagations</th>
<th>Nodes</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1176.dzn</td>
<td>26632</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>31076070</td>
<td>322501</td>
<td>161250</td>
</tr>
<tr>
<td>1266.dzn</td>
<td>25955</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>30048497</td>
<td>308181</td>
<td>154090</td>
</tr>
<tr>
<td>2254.dzn</td>
<td>26529</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>31161709</td>
<td>321381</td>
<td>160690</td>
</tr>
<tr>
<td>3237.dzn</td>
<td>25716</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>29964392</td>
<td>304791</td>
<td>152395</td>
</tr>
<tr>
<td>3345.dzn</td>
<td>26238</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>30731902</td>
<td>318213</td>
<td>159106</td>
</tr>
<tr>
<td>412.dzn</td>
<td>27359</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>31589405</td>
<td>333489</td>
<td>166744</td>
</tr>
<tr>
<td>4629.dzn</td>
<td>26508</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>30499160</td>
<td>311885</td>
<td>155942</td>
</tr>
<tr>
<td>7059.dzn</td>
<td>26944</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>31488252</td>
<td>324819</td>
<td>162409</td>
</tr>
<tr>
<td>7324.dzn</td>
<td>27077</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>31267784</td>
<td>320149</td>
<td>160074</td>
</tr>
<tr>
<td>7326.dzn</td>
<td>26400</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>30907946</td>
<td>320147</td>
<td>160073</td>
</tr>
<tr>
<td>7444.dzn</td>
<td>26379</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>30745250</td>
<td>320777</td>
<td>160388</td>
</tr>
<tr>
<td>7661.dzn</td>
<td>25550</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>29696843</td>
<td>304241</td>
<td>152120</td>
</tr>
<tr>
<td>7750.dzn</td>
<td>26714</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>31202254</td>
<td>321685</td>
<td>160842</td>
</tr>
<tr>
<td>8848.dzn</td>
<td>25834</td>
<td>1</td>
<td>409</td>
<td>366</td>
<td>30219053</td>
<td>308499</td>
<td>154249</td>
</tr>
</tbody>
</table>

Table C.2: Raw benchmark results for the block party model without pre-solving


[TPHa] Three Pixel Heart LLC. *Block Party Website*. URL: http://w.3pxh.com/

[TPHb] Three Pixel Heart LLC. *Block Party Website - Puzzles*. URL: http://w.3pxh.com/learn/puzzles