Spring 2019

Constraint Programming Using Multi-Valued Decision Diagrams

Nitesh Gupta
nkgupta@iastate.edu

Follow this and additional works at: https://lib.dr.iastate.edu/creativecomponents

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation

https://lib.dr.iastate.edu/creativecomponents/184

This Creative Component is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Creative Components by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Constraint Programming Using Multi-Valued Decision Diagrams

by

Nitesh Gupta

A Creative Component submitted to the graduate faculty in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Dr. Gianfranco Ciardo, Major Professor
Dr. Andrew Miner
Dr. David Fernández-Baca

Department of Computer Science
Iowa State University
Ames, Iowa

Copyright © Nitesh Kumar Gupta, 2019. All rights reserved.
Abstract

We mainly focuses on the implementation of a solver for Constraint Satisfaction Problems (CSP) using Multi-Valued Decision Diagrams (MDDs). The input to the solver is a constraint problem, modeled using the MiniZinc language. MEDDLY (Multi-terminal and Edge-valued Decision Diagram Library) is used to find the possible solution space for a given constraint problem. The implemented solver also includes support for various global constraints (e.g., all different, increasing, among). Capability to solve maximization and minimization problems is also added to the solver. The size of the intermediate MDD and the running time for any constraint problem is affected by the order in which various constraints are propagated. Work is done towards implementation of various strategies for constraint ordering. Eight different strategies are implemented and experiments are run to compare the performance output of each strategy.

1 INTRODUCTION

A (finite domain) constraint satisfaction problem (CSP) can be expressed in the following form. Given a set of variables, together with a finite set of possible values that can be assigned to each variable, and a list of constraints, find values of the variables that satisfy every constraint. Constraint satisfaction problems are combinatorial in nature. For many categories of CSPs, an efficient algorithm is unlikely to exist (these problems are NP-complete). Thus, an algorithm that guarantees to find a solution that satisfies all constraints, assuming that such a solution exists, is enumerative and therefore has an exponential time requirement in the worst case. In practice, it may be sufficient to find a solution at reasonable computational expense, that satisfies most of the constraints. If all or as many constraints as possible are satisfied, we refer to the solution as exact; otherwise, it is approximate. (When referring to an optimization problem, an approximate solution is one in which all constraints are satisfied, but the optimal value of the objective function is not necessarily attained.)

A variety of approaches can be used to tackle CSPs. Integer programming techniques (cutting plane methods and branch and bound) can be applied to find an exact solution. On the other hand, there are various approaches that provide an approximate solution, including local search
methods (simulated annealing, threshold accepting, tabu search and genetic algorithms) and neural networks.

An efficient representations of the solution space is also important when we want to record all solutions to a CSP or as efficient representations of \( n \)-ary constraints which can be used with consistency algorithms. An efficient representation of all solutions is useful for some applications. For example, a configuration system may be interactive where real-time response is necessary. So finding the solutions of the CSP in advance and filtering that against the user’s requirements is more efficient than solving from scratch. In this work, the efficient and compact representations of CSP is achieved by the use of Multi-Valued Decision Diagrams (MDDs) to represent solution spaces during search. In this project we implement a Constraint Solver for the CSPs using MEDDLY (Multi-terminal and Edge-valued Decision Diagram LibrarY) [1].

The remaining sections of this report are organized as follows. In Section 2, a formal definition of CSPs and MDDs is given. Sections 3 contains the description of the implementation, syntax for the MiniZinc language and the approach followed to solve CSP using MEDDLY. Section 4 includes the description of the global constraints implemented in this work. Corresponding examples are also provided. Section 5 describes various strategies devised for constraint ordering. Section 6 contains the experimental results for the strategies and Section 7 includes the future work towards scheduling of various constraints and variable ordering.

## 2 CONSTRAINT PROGRAMMING PRELIMINARIES

### 2.1 Constraint Satisfaction Problem

Given a variable \( x \), the domain of \( x \) is the set of values that can be assigned to \( x \), and is denoted by \( D(x) \). In this work we only consider integer variables with finite domains. Generalizing to finite sequences of variables \( X = (x_1, x_2, ..., x_k) \), the declared domain of solutions is given by the Cartesian product of the domains of the variables in \( X \), that is, \( D(x) = D(x_1) \times ... \times D(x_k) \).

A constraint \( C \) on \( X \) is defined as a subset of \( D(X) \). A tuple \((d_1, ..., d_k) \in C\) is a solution to \( C \) and we also say that \((d_1, ..., d_k)\) satisfies \( C \). A value \( d \in D(x_i) \) has support in \( C \) (or is consistent with respect to \( C \)) if it belongs to some tuple in \( C \); otherwise \( d \) is unsupported in \( C \).
A constraint satisfaction problem, or CSP, is defined by a finite sequence of variables \( X = (x_1, x_2, \ldots, x_n) \), together with a finite set of constraints \( C \), where each constraint \( C \in C \) is defined over a subsequence of variables \( \text{scope}(C) \subseteq X \). The goal is to find an assignment \( x_i = d_i \) with \( d_i \in D(x_i) \) for \( i = 1, \ldots, n \), such that all that constraints are satisfied. The assignment is called a feasible solution to the CSP. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes or minimizes an objective function.

The solution process of constraint programming interleaves constraint propagation (or propagation in short), and search. The search process effectively enumerates all possible variable-value combinations. The search process continues until a feasible solution is found or it proves that no feasible solution exists. We say that this process constructs a search tree. Each node in the tree has a declared domain which is a subset of its parent’s domain. To reduce the exponential number of combinations, constraint propagation is applied to each node of the search tree: given the current domains and a constraint \( C \), remove domain values that are inconsistent with \( C \). This is repeated for all constraints until no more domain values can be removed. The removal of inconsistent domain values is called filtering.

Treating a problem as a CSP confers several important benefits. Because the representation of states in a CSP conforms to a standard pattern; that is, a set of variables with assigned values; the successor function and goal test can written in a generic way that applies to all CSPs. Furthermore, we can develop effective, generic heuristics that require no additional, domain-specific expertise. Finally, the structure of the constraint graph can be used to simplify the solution process, in some cases giving an exponential reduction in complexity. CSP can be given an incremental formulation as a standard search problem as follows:

- **Initial state**: the empty assignment \( {} \), in which all variables are unassigned.
- **Successor function**: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
• **Goal test:** the current assignment is complete.

• **Path cost:** a constant cost (e.g., 1) for every step.

### 2.2 Multi-Valued Decision Diagrams

Multivalued decision diagrams (MDDs) [3] generalize binary decision diagrams (BDDs) [2]. The MDD for a constraint set is essentially a more compact representation of a branching tree, obtained by superimposing isomorphic subtrees. The shape and size of the resulting MDD depends on the order in which one branches on the variables.

Formally, an ordered MDD is a directed acyclic graph whose nodes are partitioned into \( n \) (possibly empty) subsets or layers \( L_0, ..., L_n \), where the layers \( L_1, ..., L_n \) correspondingly to variables \( x_1, ..., x_n \). \( L_n \) contains a single top node \( T \), and \( L_0 \) contains two bottom nodes 0 and 1. The width of the MDD is the maximum number of nodes in a layer, or \( \max_{i=1}^{n} |L_i| \).

All edges of the MDD are directed from an upper to a lower layer; that is, from a node in some \( L_i \) to a node in some \( L_j \) with \( i > j \). Let \( L(v) \) denote the layer of the node \( v \). Each edge out of layer \( i \) is labeled with an element of the domain \( D(x_i) \) of \( x_i \), and no label occurs more than once on the edges leaving any given node. The set \( E(p, q) \) of edges from node \( p \) to node \( q \) may contain multiple edges, and we denote each with its label.

An edge with label \( v \) leaving a node in layer \( i \) represents an assignment \( x_i = v \). Each path in the MDD from \( T \) to 0 or 1 can be denoted by the edge labels \( v_1, ..., v_n \) on the path and is identified with the assignment \( (x_1, ..., x_n) = (v_1, ..., v_n) \). The MDD as a whole therefore represents a pseudoboollean function \( f \) for which \( f(v_1, ..., v_n) \) has the value 1 when \( v_1, ..., v_n \) is a path from \( T \) to 1, and 0 when it is a path from \( T \) to 0.

It is clear that any pseudoboollean function of finite-domain variables \( x_1, ..., x_n \) can be represented by an MDD. Any constraint set with finite-domain variables can likewise be represented by an MDD, because it defines a pseudoboollean function that maps every assignment to its variables \( x_1, ..., x_n \) to true or false. For our purposes, it is convenient to generate only the portion of an MDD that contains paths from \( T \) to 1. The resulting MDD represents assignments to \( x_1, ..., x_n \) for which \( f(x_1, ..., x_n) = 1 \). A path \( v_1, ..., v_n \) is feasible for a given constraint \( C \) if setting \( (x_1, ..., x_n) = (v_1, ..., v_n) \) satisfies \( C \). Constraint \( C \) is feasible on an MDD if the MDD
contains a feasible path for $C$.

3 CONSTRAINTS SOLVER DESCRIPTION

3.1 Constraints Modeling Language: MiniZinc

Constraints are modeled using the MiniZinc [4] constraint modeling language.

3.1.1 Introduction with Example

MiniZinc is a language designed for specifying constrained optimization and decision problems over integers. A MiniZinc model does not dictate how to solve the problem although the model can contain annotations which are used to guide the underlying solver. MiniZinc is designed to interface easily to different backend solvers. It does this by transforming an input MiniZinc model and data file into a FlatZinc model. FlatZinc models consist of variable declaration and constraint definitions as well as a definition of the objective function if the problem is an optimization problem. The translation from MiniZinc to FlatZinc is specializable to individual backend solvers, so they can control the form of the constraints. In particular, MiniZinc allows the specification of global constraints by decomposition. A MEDDLY-based constraint solver is used for solving the constraints in this implementation.

Example:

![Figure 3.1: Australian states.](image)

In this example we wish to color a map of Australia as shown in Figure 4.1. It is made up of
seven different states and territories each of which must be given a color so that adjacent regions have different colors. We can model this problem very easily in MiniZinc. The model is shown below.

% Colouring Australia using nc colours
int: nc = 3;

var 1..nc: wa;
var 1..nc: nt;
var 1..nc: sa;
var 1..nc: q;
var 1..nc: nsw;
var 1..nc: v;
var 1..nc: t;

constraint wa ! = nt;
constraint wa ! = sa;
constraint nt ! = sa;
constraint nt ! = q;
constraint sa ! = q;
constraint sa ! = nsw;
constraint sa ! = v;
constraint q ! = nsw;
constraint nsw ! = v;

solve satisfy;

The first line in the model is a comment. A comment starts with a ’% ’ which indicates that the rest of the line is a comment. The next part of the model declares the variables in the model.
The line

    int: nc = 3;

specifies a parameter in the problem which is the number of colors to be used. Parameters are similar to variables in most programming languages. They must be declared and given a type. In this case the type is `int`. They are given a value by an assignment. MiniZinc allows the assignment to be included as part of the declaration (as in the line above) or to be a separate assignment statement. Thus, the following is equivalent to the single line above:

    int: nc;
    nc = 3;

A parameter can only be given a single value. It is an error for a parameter to occur in more than one assignment. The basic parameter types are integers (`int`), floating point numbers (`float`), Booleans (`bool`) and strings (`string`). Arrays and sets are also supported. MiniZinc models can also contain another kind of variable called a decision variable. Decision variables are variables in the sense of mathematical or logical variables. The value of a decision variable is unknown and it is only when the MiniZinc model is executed that the solving system determines if the decision variable can be assigned a value that satisfies the constraints in the model and if so what this is. In the above example model we associate a decision variable with each region, `wa, nt, sa, q, nsw, v` and `t`, which stands for the (unknown) color to be used to fill the region. For each decision variable we need to give the set of possible values the variable can take. This is called the variable’s domain. This can be given as part of the variable declaration and the type of the decision variable is inferred from the type of the values in the domain. In MiniZinc decision variables can be Booleans, integers, floating point numbers, or sets. Also supported are arrays whose elements are decision variables. In our MiniZinc model we use integers to model the different colors. Thus, each of our decision variables is declared to have the domain `1..nc` which is an integer range expression indicating the set `{1, 2, ..., nc}` using the var declaration. The type of the values is integer so all of the variables in the model are integer decision variables.

The next component of the model are the constraints. These specify the Boolean expressions that the decision variables must satisfy to be a valid solution to the model. In this case we have
a number of not equal constraints between the decision variables enforcing that if two states are
adjacent then they must have different colors.

MiniZinc provides the relational operators: equal (= or ==), not equal (! =), strictly less
than, (<) strictly greater than (>), less than or equal to (<=), and greater than or equal to
(>=). The next line in the model

    solve satisfy;

indicates the kind of problem it is. In this case it is a satisfaction problem: we wish to find a
value for the decision variables that satisfies the constraints but we do not care which one.

3.1.2 MiniZinc Syntax

The basics of the EBNF used are as follows.

- Non-terminals are written between angle brackets, e.g. ⟨item⟩.
- Terminals are written in bold font. e.g. constraint.
- Optional items are written in square brackets, e.g. [var].
- Sequences of zero or more items are written with parentheses and a star, e.g. ⟨ident⟩*.
- Sequences of one or more items are written with parentheses and a plus, e.g. ⟨msg⟩+.
- Zero or one occurrence of an item is written with parentheses and a question mark, e.g.
  ⟨msg⟩?.
- Non-empty lists are written with an item, a separator/terminator terminal, and “…”.
  For example, this:
  ⟨expr⟩,...
  is short for this:
  ⟨expr⟩(⟨expr⟩)*[,
  The final terminal is always optional in non-empty lists.
- Regular expressions, written in fixed-width font, are used in some productions, e.g. [-+]?(1-9)(0-9)*.
The MiniZinc grammar is described as follows.

- **Items**
  A MiniZinc model consists of multiple items:

  \[
  \langle \text{model} \rangle ::= \langle \text{item}, \ldots \rangle
  \]

  \[
  \langle \text{item} \rangle ::= \langle \text{include-item} \rangle
  \]

  \[
  \quad | \langle \text{var-decl-item} \rangle
  \]

  \[
  \quad | \langle \text{assign-item} \rangle
  \]

  \[
  \quad | \langle \text{constraint-item} \rangle
  \]

  \[
  \quad | \langle \text{solve-item} \rangle
  \]

  \[
  \quad | \langle \text{predicate-item} \rangle
  \]

  \[
  \quad | \langle \text{function-item} \rangle
  \]

  \[
  \langle \text{type-inst-syn-item} \rangle ::= \text{type}\langle \text{ident} \rangle\langle \text{annotations} \rangle\langle \text{ti-expr} \rangle
  \]

  \[
  \langle \text{ti-expr-and-id} \rangle ::= \langle \text{ti-expr} \rangle:\langle \text{ident} \rangle
  \]

  \[
  \langle \text{include-item} \rangle ::= \text{include}\langle \text{string-literal} \rangle
  \]

  \[
  \langle \text{var-decl-item} \rangle ::= \langle \text{ti-expr-and-id} \rangle\langle \text{annotations} \rangle\langle \text{ti-expr} \rangle
  \]

  \[
  \langle \text{assign-item} \rangle ::= \langle \text{ident} \rangle\langle \text{ti-expr} \rangle
  \]

  \[
  \langle \text{constraint-item} \rangle ::= \text{constraint}\langle \text{ti-expr} \rangle
  \]

  \[
  \langle \text{solve-item} \rangle ::= \text{solve} \text{ satisfy}
  \]

  \[
  \langle \text{predicate-item} \rangle ::= \text{predicate}\langle \text{operation-item-tail} \rangle
  \]
Include items provide a way of combining multiple files into a single instance. This allows a model to be split into multiple files. Variable declaration items introduce new global variables and possibly bind them to a value. Assignment items bind values to global variables. Constraint items describe model constraints. Solve items are the “starting point” of a model, and specify exactly what kind of solution is being looked for. Each model must have exactly one solve item. Predicate items and function items introduce new user-defined predicates and functions which can be called in expressions. Predicates, functions, and built-in operators are described collectively as operations. Annotation items can specify non-declarative and/or solver-specific information in a model.

• Type-Inst Expressions

\(<\text{ti-expr}\> ::= \langle\text{base-}\text{ti-expr}\rangle

\langle\text{base-}\text{ti-expr}\rangle ::= \langle\text{var-par}\rangle\langle\text{base-}\text{ti-expr-tail}\rangle

\langle\text{var-par}\rangle ::= \text{var} \mid \text{par} \mid \epsilon

\langle\text{base-}\text{ti-expr-tail}\rangle ::= \langle\text{ident}\rangle

\mid \text{bool}

\mid \text{int}
The instantiation of a variable or value indicates if it is fixed to a known value or not. A pairing of a type and instantiation is called a type-inst. A type-inst expression specifies a type-inst. Type-inst expressions may include type-inst constraints. Type-inst expressions appear in variable declarations, user-defined operation items. Type-inst expressions have this syntax:

- **Expressions**

Expressions represent values. They occur in various kinds of items. They have the following syntax:

\[
\begin{align*}
\langle \text{expr} \rangle & ::= \langle \text{expr-atom} \rangle \langle \text{expr-binop-tail} \rangle \\
\langle \text{expr-atom} \rangle & ::= \langle \text{expr-atom-head} \rangle \langle \text{expr-atom-tail} \rangle \\
\langle \text{expr-binop-tail} \rangle & ::= [\langle \text{bin-op} \rangle \langle \text{expr} \rangle ] \\
\langle \text{expr-atom-head} \rangle & ::= \langle \text{builtin-un-op} \rangle \langle \text{expr-atom} \rangle
\end{align*}
\]
| (expr) |
| (bool-literal) |
| (int-literal) |
| (string-literal) |
| (set-literal) |
| (array-literal) |
| (array-literal-2d) |
| (if-then-else-expr) |

\( \langle \text{expr-atom-tail}\rangle ::= \epsilon \)
| \( \langle \text{array-access-tail}\rangle \langle \text{expr-atom-tail}\rangle \)

\( \langle \text{num-expr}\rangle ::= \langle \text{num-expr-atom}\rangle \langle \text{num-expr-binop-tail}\rangle \)

\( \langle \text{num-expr-atom}\rangle ::= \langle \text{num-expr-atom-head}\rangle \langle \text{expr-atom-tail}\rangle \)

\( \langle \text{num-expr-binop-tail}\rangle ::= [\langle \text{num-bin-op}\rangle \langle \text{num-expr}\rangle] \)

\( \langle \text{num-expr-atom-head}\rangle ::= \langle \text{builtin-num-un-op}\rangle \langle \text{num-expr-atom}\rangle \)
| (\( \langle \text{num-expr}\rangle \))
| \( \langle \text{int-literal}\rangle \)
| \( \langle \text{if-then-else-expr}\rangle \)

\( \langle \text{builtin-op}\rangle ::= \langle \text{builtin-bin-op}\rangle \)
| \( \langle \text{builtin-un-op}\rangle \)

\( \langle \text{bin-op}\rangle ::= \langle \text{builtin-bin-op}\rangle \)
\[
\langle \text{builtin-bin-op} \rangle ::= \langle \text{builtin-un-op} \rangle
\]

\[
\langle \text{builtin-un-op} \rangle ::= \text{not} \mid \langle \text{builtin-num-un-op} \rangle
\]

\[
\langle \text{num-bin-op} \rangle ::= \langle \text{builtin-num-bin-op} \rangle
\]

\[
\langle \text{builtin-num-bin-op} \rangle ::= + \mid - \mid * \mid / \mid \text{div} \mid \text{mod}
\]

\[
\langle \text{builtin-num-bin-op} \rangle ::= + \mid -
\]

\[
\langle \text{bool-literal} \rangle ::= \text{false} \mid \text{true}
\]

\[
\langle \text{int-literal} \rangle ::= [0-9]+
| 0x[0-9A-Fa-f]+
| 0o[0-7]+
\]

\[
\langle \text{string-contents} \rangle ::= ([^"\n\] \mid [^\n(])*
\]

\[
\langle \text{string-literal} \rangle ::= "\langle \text{string-contents} \rangle"
| "\langle \text{string-contents} \rangle\"\langle \text{string-interpolate-tail} \rangle
\]

\[
\langle \text{string-interpolate-tail} \rangle ::= \langle \text{expr} \rangle\langle \text{string-contents} \rangle\"\langle \text{string-interpolate-tail} \rangle
\]

\[
\langle \text{set-literal} \rangle ::= \{\langle \text{expr} \rangle, \ldots \}\}
\]

\[
\langle \text{array-literal} \rangle ::= [\langle \text{expr} \rangle, \ldots ][
\]

\[
\langle \text{array-literal-2d} \rangle ::= [||\langle \text{expr} \rangle, \ldots || \ldots ||]
\]
\[
\langle \text{array-access-tail} \rangle ::= [\langle \text{expr} \rangle, \ldots]
\]
\[
\langle \text{ann-literal} \rangle ::= \langle \text{ident} \rangle [\langle \text{expr} \rangle, \ldots]
\]
\[
\langle \text{if-then-else-expr} \rangle ::= \text{if} (\langle \text{expr} \rangle \text{then} (\langle \text{expr} \rangle)
\]
\[
\quad (\text{elseif} (\langle \text{expr} \rangle \text{then} (\langle \text{expr} \rangle))^* \\
\quad \text{else} (\langle \text{expr} \rangle) \text{endif}
\]

- **Miscellaneous Elements**
  \[
  \langle \text{ident} \rangle ::= [A-Za-z][A-Za-z0-9_]*
  \]
  \[
  \langle \text{annotations} \rangle ::= (:: (\langle \text{annotation} \rangle))^*
  \]
  \[
  \langle \text{annotation} \rangle ::= \langle \text{expr-atom-head} \rangle \langle \text{expr-atom-tail} \rangle
  \]

### 3.1.3 Program Execution

The class **FlatZincModel** contains the methods to create variables and to post constraints. The class **PrinterModel** is responsible for printing the values for the required variable to the standard output. These classes are implemented in the file **flatzinc.cpp**. The actual constraint posting is done in the file **registry.cpp**. A posting function for each implemented constraint is added here, and registered with the corresponding flatzinc name. Functions for constraint solving using MEDDLY are implemented in **solver.cpp**.

The program can be executed using the following command in the terminal:

\[
\$\text{/solver.sh filename}
\]

where, **filename** is the name of the the MiniZinc model file. The above example can be executed using the command

\[
\$\text{/solver.sh aust.mzn}
\]

in the terminal, where **aust.mzn** is the name of the file containing our MiniZinc model. One of the output for this example is displayed in the following form on the standard output.
nsw = 2;
nt = 2;
q = 1;
sa = 3;
t = 1;
v = 1;
wa = 1;

Option -a can be used with the above command to print all the solutions to the standard output.

$$. ./solver.sh -a aust.mzn

To run a particular constraint ordering strategy we use the corresponding strategy number as an option. For example

$$ ./solver.sh aust.mzn 1

can be used to run the Most Frequent strategy for a given input example. The number of a strategy corresponds to the column number in Table 6.1 starting with column Most Frequent.

The functionality to set the value of parameters declared in the original model using the data file is also implemented. This allows the same model to be easily used with different data by running it with different data files. Data files must have the file extension "dzn" to indicate a MiniZinc data file and a model can be run with any number of data files (though a variable/parameter can only be assigned a value in one file). The command to execute a program using the data file is:

$$ ./solver.sh sudoku.mzn sudoku.dzn

Option -a can still be used in the same manner as mentioned above.

4 GLOBAL CONSTRAINTS DEFINITION AND EXAMPLES

The categories of global constraints implemented by the constraint solver in the present work are listed below.
4.1 All-Different and related constraints

1. predicate all_different(array [int] of var int: x)
   Constrain the array of integers \( x \) to be all different.
   Example
   
   • Working Case:
     \[ x = [x_1, x_2, x_3] \text{ where, } x_1 \in \{1, 2, 3\}, x_2 \in \{1, 2, 3\}, x_3 \in \{1, 2, 3\} \]
     \[ \text{Output: } \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\} \]
   
   • Non-Working Case:
     \[ x = [x_1, x_2, x_3] \text{ where, } x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{1, 2\} \]
     \[ \text{Output: } \{\} \]

2. predicate all_equal(array [int] of var int: x)
   Constrain the array of integers \( x \) to be all equal.
   Example
   
   • Working Case:
     \[ x = [x_1, x_2, x_3] \text{ where, } x_1 \in \{1, 2, 3\}, x_2 \in \{1, 2, 3\}, x_3 \in \{1, 2, 3\} \]
     \[ \text{Output: } \{(1, 1, 1), (2, 2, 2), (3, 3, 3)\} \]
   
   • Non-Working Case:
     \[ x = [x_1, x_2, x_3] \text{ where, } x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{3\} \]
     \[ \text{Output: } \{\} \]

3. predicate alldifferent_except_zero(array [int] of var int: vs)
   Constrain the array of integers \( vs \) to be all different except those elements that are assigned the value 0.
   Example
   
   • Working Case:
     \[ vs = [vs_1, vs_2, vs_3] \text{ where, } x_1 \in \{0, 1\}, x_2 \in \{0, 1\}, x_3 \in \{0, 1\} \]
     \[ \text{Output: } \{(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 0, 0)\} \]
• Non-Working Case:
  - \( x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{1, 2\} \)
  - Output: \{\}

4. predicate \texttt{nvalue}(array [int] of var int: \( x \), var int: \( n \))

Requires that the number of distinct values in \( x \) is \( n \).

Example

• Working Case:
  - \( x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{1, 2\}, n = 2 \)
  - Output: \{(1, 1, 2), (1, 2, 1), (1, 2, 2), (2, 1, 1), (2, 1, 2), (2, 2, 1)\}

• Non-Working Case:
  - \( x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1\}, x_2 \in \{1\}, x_3 \in \{1\}, n = 2 \)
  - Output: \{\}

4.2 Sorting constraints

1. predicate \texttt{decreasing}(array [int] of var int: \( x \))

Requires that the array \( x \) is in decreasing order (duplicates are allowed).

Example

• Working Case:
  - \( x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1, 2, 3\}, x_2 \in \{1, 2, 3\}, x_3 \in \{1, 2, 3\} \)
  - Output:
    \{(3, 3, 3), (3, 3, 2), (3, 3, 1), (3, 2, 2), (3, 2, 1), (3, 1, 1), (2, 2, 2), (2, 2, 1), (2, 1, 1), (1, 1, 1)\}

• Non-Working Case:
  - \( x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1\}, x_2 \in \{2\}, x_3 \in \{3\} \)
  - Output: \{\}
2. predicate \textit{increasing}(\text{array [int] of var int: } x)\\

Requires that the array \textit{x} is in increasing order (duplicates are allowed).

\textbf{Example}

- Working Case:
  \begin{itemize}
  \item \textit{x} = [x_1, x_2, x_3] where, \textit{x}_1 \in \{1, 2, 3\}, \textit{x}_2 \in \{1, 2, 3\}, \textit{x}_3 \in \{1, 2, 3\}
  \end{itemize}

- Output:
  \{ (1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 2, 2), (1, 2, 3), (1, 3, 3), (2, 3, 3), (2, 2, 3), (2, 2, 2), (3, 3, 3) \}

- Non-Working Case:
  \begin{itemize}
  \item \textit{x} = [x_1, x_2, x_3] where, \textit{x}_1 \in \{3\}, \textit{x}_2 \in \{2\}, \textit{x}_3 \in \{1\}
  \end{itemize}

  \textit{Output: \{\} }

4.3 Counting constraints

1. predicate \textit{among}(\text{var int: } n, \text{array [int] of var int: } x, \text{set of int: } v)\\

Requires exactly \textit{n} variables in \textit{x} to take one of the values in \textit{v}.

\textbf{Example}

- Working Case:
  \begin{itemize}
  \item \textit{n} = 2, \textit{x} = [x_1, x_2, x_3] where, \textit{x}_1 \in \{1, 2\}, \textit{x}_2 \in \{1, 2\}, \textit{x}_3 \in \{1, 2\}, \textit{v} = \{1\}
  \end{itemize}

  \textit{Output: \{(1, 1, 2), (1, 2, 1), (2, 1, 1)\} }

- Non-Working Case:
  \begin{itemize}
  \item \textit{n} = 2, \textit{x} = [x_1, x_2, x_3] where, \textit{x}_1 \in \{1, 2\}, \textit{x}_2 \in \{1, 2\}, \textit{x}_3 \in \{1, 2\}, \textit{v} = \{4, 5\}
  \end{itemize}

  \textit{Output: \{\} }

2. predicate \textit{at_least}(\text{int: } n, \text{array [int] of var int: } x, \text{int: } v)\\

Requires at least \textit{n} variables in \textit{x} to take the value \textit{v}.

\textbf{Example}

- Working Case:
- \( n = 2, x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{1, 2\}, v = 1 \)
- Output: \( \{(1, 1, 1), (1, 1, 2), (1, 2, 1), (2, 1, 1)\} \)

• Non-Working Case:

- \( n = 2, x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{1, 2, 3\}, v = \{3\} \)
- Output: \( \{\} \)

3. predicate \textit{at\_most}(int: n, array [int] of var int: x, int: v)

Requires at most \( n \) variables in \( x \) to take the value \( v \).

Example

• Working Case:

- \( n = 2, x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{1, 2\}, v = 1 \)
- Output: \( \{(1, 1, 2), (1, 2, 1), (2, 1, 1), (1, 2, 2), (2, 1, 2), (2, 2, 1), (2, 2, 2)\} \)

• Non-Working Case:

- \( n = 2, x = [x_1, x_2, x_3] \) where, \( x_1 \in \{1\}, x_2 \in \{1\}, x_3 \in \{1\}, v = \{1\} \)
- Output: \( \{\} \)

5 CONSTRAINT ORDERING STRATEGIES

In this section, we propose strategies defining the ordering of the constraints i.e. the order in which the constraints are applied or propagated to form the resultant MDD, for a given constraint optimization problem. The ordering for constraints essentially defines the schedule of intersection for them. The intersections can be represented in the form of a binary tree. Both the size of the intermediate MDD and running time of the algorithm are affected by the constraint ordering. The strategies for the constraint ordering proposed in this project are:
Example

\[ C = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\} \]

where,

\[ C_1 : x_1 + x_2 = 9 \]
\[ C_2 : x_1 + x_3 = 8 \]
\[ C_3 : x_2 + x_3 = 14 \]
\[ C_4 : x_1 + x_4 = 16 \]
\[ C_5 : x_2 + x_4 = 7 \]
\[ C_6 : x_2 + x_5 = 10 \]
\[ C_7 : x_3 + x_6 = 17 \]
\[ C_8 : x_2 + x_6 = 26 \]

1. Ordering the constraints on the basis of frequency of different variables, where frequency is defined as how many times a variable appears in the set of constraints. The intersection structure for this strategy can be represented in the form of a full binary tree where intersection of a constraint is performed with the result of the previous intersection.

\[ \text{freq}(x_1) = 3 \]
\[ \text{freq}(x_2) = 5 \]
\[ \text{freq}(x_3) = 3 \]
\[ \text{freq}(x_4) = 2 \]
\[ \text{freq}(x_5) = 1 \]
\[ \text{freq}(x_6) = 1 \]

a) **Most Frequent**: We sum the frequency of variables appearing in each constraint.

The constraint with the highest sum is propagated first. The ordering is partial and
ties are broken randomly.

**Ordering:** \{C_1 > C_3 > C_5 > C_2 > C_6 > C_8 > C_4 > C_7\}

**Schedule of Intersection:** Firstly, intersection is performed between constraint \(C_1\) and \(C_3\), the result of which is then intersected with \(C_5\) and so on.

b) **Least Frequent:** Constraint with the least sum is propagated first. The ordering is partial and ties are broken randomly.

**Ordering:** \{C_7 > C_4 > C_8 > C_6 > C_2 > C_5 > C_3 > C_1\}

2. Ordering the constraints on the basis of variable index. Constraints are compared variable by variable. The second variable is compared only when the first variable is identical in both the constraints. The same follows for subsequent variables. The order is partial and ties are broken randomly in case two constraints contain the same set of variables. The intersection structure for this strategy can be represented in the form of a full binary tree where every node other than the leaves has two children. The initial intersection is performed between top two constraints. Each subsequent intersection for a constraint is performed with the result of the previous intersection. This strategy also leverages the variable ordering of input MiniZinc model.

a) **Highest Index:** Constraint with the highest indexed variable is propagated first.

**Ordering:** \{C_1 > C_2 > C_4 > C_3 > C_5 > C_6 > C_8 > C_7\}

**Explanation:** when comparing \(C_1\) and \(C_2\), the first variable \(x_1\) is the same in both constraints. The second variable \(x_2\) in \(C_1\) has index 2 which is lower than the index 3 of variable \(x_3\) in \(C_2\).

b) **Lowest Index:** Constraints with the lowest indexed variable are propagated first.

**Ordering:** \{C_7 > C_8 > C_6 > C_5 > C_3 > C_4 > C_2 > C_1\}

3. Ordering the constraint pairs on the basis of the value of the intersection between them. The value of the intersection is defined as the total number of distinct variables for a constraint pair. The constraints are propagated in the form of a full binary tree. This heuristic provides the basis for the formation of constraint pairs. Ordering is partial and ties are broken randomly.
a) **Maximum Intersection**: Constraint pairs are formed on the basis of maximum intersection value.

For the above example, $C_1$ is picked up first. Intersection Values are:

$$(C_1, C_2) = 2$$

$$(C_1, C_3) = 2$$

$$(C_1, C_4) = 2$$

$$(C_1, C_5) = 2$$

$$(C_1, C_6) = 2$$

$$(C_1, C_7) = 4$$

$$(C_1, C_8) = 2$$

Here maximum intersection value is 4. Hence $(C_1, C_7)$ are propagated first. They will form a new constraint $C_9$ with variables $\{x_1, x_2, x_3, x_6\}$. The newly formed constraint will only be considered for comparison in next iteration. Now $C_1$ and $C_7$ are removed from the set of constraints and $C_2$ is compared with the remaining constraints.

Intersection values for $C_2$ are:

$$(C_2, C_3) = 2$$

$$(C_2, C_4) = 2$$

$$(C_2, C_5) = 4$$

$$(C_2, C_6) = 4$$

$$(C_2, C_8) = 2$$

Maximum intersection value is 4. The newly formed constraint $C_{10}$ has variables $\{x_1, x_2, x_3, x_4\}$. Now $(C_2, C_5)$ are propagated and removed from constraint set.

Intersection values for $C_3$ are:

$$(C_3, C_4) = 4$$

$$(C_3, C_6) = 2$$

$$(C_3, C_8) = 4$$

Maximum intersection value is 4. The newly formed constraint $C_{11}$ has variables
\{x_1, x_2, x_3, x_4\}. Now \((C3, C4)\) are propagated and removed from constraint set.

Intersection values for \(C6\) are:

\((C6, C8) = 2\)

Maximum intersection value is 2. The newly formed constraint \(C12\) has variables \(\{x_2, x_5, x_6\}\). Now \((C6, C8)\) are propagated and removed from constraint set.

The next iteration will be performed with constraint set \(\{C9, C10, C11, C12\}\).

b) **Minimum Intersection:** Constraint pairs are formed on the basis of minimum intersection value. Same procedure as above is followed using minimum value of intersection instead of maximum.

4. Dynamic ordering of the constraint pairs on the basis of the value of the intersection between them. The procedure followed in this strategy is similar to the Strategy 3 except that the newly formed constrained is also included in the set of constraints for the current iteration.

For the example in Strategy 3, while performing comparison for \(C2\), we also include \(C9\) in the constraint set. The other steps remain the same. Ordering is partial and ties are broken randomly.

a) **Maximum Dynamic Intersection:** Constraint pairs are formed dynamically on the basis of maximum intersection value.

b) **Minimum Dynamic Intersection:** Constraint pairs are formed dynamically on the basis of minimum intersection value.

6 **EXPERIMENTAL RESULTS**

The experiments reported below display the application of strategies to decide propagation order for various constraints for an optimization problem. The criteria for comparison among various strategies are:

1. Number of peak nodes of the intermediate MDD, and

2. Running time of the algorithm.
Performance results for various example problems are shown in Table 6.1 and Table 6.2. Best results for each model are highlighted in the tables.

Examples 4-queen and 8-queen produces the same number of peak nodes for strategies Most Frequent and Least Frequent. This is due to the fact that all variables appear equal number of times in constraint set. There is a significant difference in the number of peak nodes for various strategies for 8-queen problem. Max Intersection strategy enforces intersection of the constraint pair with most distinct variables which results in a MDD with more number of intermediate nodes as compare to Min Intersection strategy. The same applies for Max Intersection Dynamic and Max Intersection Dynamic strategies. The large difference in number of peak nodes between Min Intersection Dynamic and Lowest Index strategy is due to the fact that while doing intersection for constraint pairs using Min Intersection Dynamic strategy, such situation will arise when all the variables for a pair would be distinct.

Example send-more-money has same number of peak nodes for all the strategies. This is due to the existence of single linear equality constraint. The other alldifferent constraint has same impact on the peak number of nodes for all strategies due to its symmetry. The same holds true for 18-hole-golf problem. Another version of this problem has an increasing constraint which affects the number of peaks nodes for different strategies.

The same number of peak nodes for strategies Most Frequent, Least Frequent, Lowest Index and Highest Index for the example aust is due to the nature of not-equal constraints which result in the same ordering for all of these strategies. The same behavior can be observed for linear equality constraints for the examples aaa-bbbccc and ages.

Examples a-puzzle and a-puzzle2 contains set of linear equality constraints. The ordering for various strategies does not differ much which results in similar number of peak nodes for the optimal constraint ordering strategies. Examples a-puzzle2 has higher number of peak nodes due to more unrestricted variables. Examples added-corner, bank-cards and, arch-friends
contains a mixture of *alldifferent*, linear equality, linear non-equality constraints. The higher value and high variability in number of peak nodes in *added-corner* example is due to more variables with larger domain size.

For a higher number of linear equality constraints, strategy **Min Intersection** and **Min Intersection Dynamic** gives superior performance as can be seen with the help of the *Domino* example. These two strategies result in fewer peaks nodes as compare to others. On the other hand strategies **Max Intersection** and **Max Intersection Dynamic** perform poorly for linear equality constraints.

The following conclusions can also be derived from Table 6.1 and Table 6.2:

1. Strategies **Most Frequent** and **Least Frequent** show similar performance for various examples.

2. Strategy **Lowest Index** performs only slightly better than **Highest Index** as the number of peak nodes does not differ much.

3. Strategy **Minimum Intersection** shows a significant performance improvement compared to **Maximum Intersection** for most of the examples. The improvement is shown in terms of both number of peak nodes and running time.

4. Strategy **Minimum Intersection Dynamic** performs significantly better than strategy **Maximum Intersection Dynamic** for most of the examples. Performing the intersection dynamically improves the performance in case of **Minimum Intersection** as can be seen from the above table.

5. Strategies **Most Frequent, Least Frequent, Lowest Index, Highest Index, Min Intersection, Min Intersection Dynamic** can be considered as good strategies due to their superior performance.

6. Running time for any strategy is proportional to the number of peak nodes.
7 FUTURE WORK

Efforts can be put towards deriving more heuristics for constraint ordering in the future. It would provide more insights into the various factors causing one strategy to perform better over the another. Future work includes running all the strategies for more examples in order to generate more confidence in the obtained performance result. Also work should be done in deriving the heuristics for well known variable ordering problem and experiments should be performed.

REFERENCES


## Table 6.1: Number of peak nodes of the intermediate MDD

<table>
<thead>
<tr>
<th></th>
<th>Most Frequent</th>
<th>Least Frequent</th>
<th>Lowest Index</th>
<th>Highest Index</th>
<th>Max Intrset</th>
<th>Min Intrset</th>
<th>Max Intrset Dynamic</th>
<th>Min Intrset Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-queen</td>
<td>132</td>
<td>132</td>
<td>116</td>
<td>127</td>
<td>203</td>
<td>128</td>
<td>198</td>
<td>110</td>
</tr>
<tr>
<td>8-queen</td>
<td>2046</td>
<td>2046</td>
<td>1515</td>
<td>2049</td>
<td>80554</td>
<td>9366</td>
<td>154981</td>
<td>4504</td>
</tr>
<tr>
<td>send-more-money</td>
<td>264481</td>
<td>264481</td>
<td>264481</td>
<td>264481</td>
<td>264481</td>
<td>264481</td>
<td>264481</td>
<td>264481</td>
</tr>
<tr>
<td>18-hole-golf</td>
<td>908</td>
<td>908</td>
<td>908</td>
<td>908</td>
<td>908</td>
<td>908</td>
<td>908</td>
<td>908</td>
</tr>
<tr>
<td>18-hole-golf2</td>
<td>1119</td>
<td>1119</td>
<td>1119</td>
<td>1119</td>
<td>1733</td>
<td>1624</td>
<td>2438</td>
<td>1517</td>
</tr>
<tr>
<td>aust</td>
<td>113</td>
<td>113</td>
<td>113</td>
<td>113</td>
<td>158</td>
<td>122</td>
<td>143</td>
<td>122</td>
</tr>
<tr>
<td>aaa-bbb-ccc</td>
<td>164</td>
<td>154</td>
<td>154</td>
<td>154</td>
<td>164</td>
<td>164</td>
<td>164</td>
<td>154</td>
</tr>
<tr>
<td>ages</td>
<td>244</td>
<td>244</td>
<td>244</td>
<td>244</td>
<td>255</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>a-puzzle</td>
<td>479</td>
<td>479</td>
<td>513</td>
<td>473</td>
<td>824</td>
<td>565</td>
<td>765</td>
<td>533</td>
</tr>
<tr>
<td>a-puzzle2</td>
<td>484</td>
<td>525</td>
<td>499</td>
<td>466</td>
<td>927</td>
<td>608</td>
<td>969</td>
<td>565</td>
</tr>
<tr>
<td>arch-friends</td>
<td>127</td>
<td>128</td>
<td>126</td>
<td>127</td>
<td>220</td>
<td>148</td>
<td>215</td>
<td>136</td>
</tr>
<tr>
<td>bank-card</td>
<td>672</td>
<td>686</td>
<td>654</td>
<td>662</td>
<td>718</td>
<td>752</td>
<td>718</td>
<td>697</td>
</tr>
<tr>
<td>added-corner</td>
<td>809</td>
<td>878</td>
<td>730</td>
<td>953</td>
<td>32389</td>
<td>4829</td>
<td>6707</td>
<td>1312</td>
</tr>
<tr>
<td>domino</td>
<td>56284</td>
<td>56152</td>
<td>56088</td>
<td>56346</td>
<td>5396023</td>
<td>51159</td>
<td>20671595</td>
<td>48517</td>
</tr>
</tbody>
</table>
Table 6.2: Running time of Algorithm (in seconds)

<table>
<thead>
<tr>
<th></th>
<th>Most Frequent</th>
<th>Least Frequent</th>
<th>Lowest Index</th>
<th>Highest Index</th>
<th>Max Intrset</th>
<th>Min Intrset</th>
<th>Max Intrset Dynamic</th>
<th>Min Intrset Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-queen</td>
<td>0.000475</td>
<td>0.000455</td>
<td>0.000453</td>
<td>0.000415</td>
<td>0.00027</td>
<td>0.0000401</td>
<td>0.000395</td>
<td>0.000334</td>
</tr>
<tr>
<td>8-queen</td>
<td>0.005016</td>
<td>0.008506</td>
<td><strong>0.003408</strong></td>
<td>0.005787</td>
<td>0.216071</td>
<td>0.023290</td>
<td>0.344156</td>
<td>0.005473</td>
</tr>
<tr>
<td>send-more-money</td>
<td>2.252723</td>
<td>2.218555</td>
<td>2.293252</td>
<td>2.211306</td>
<td><strong>2.189114</strong></td>
<td>2.331603</td>
<td>2.262247</td>
<td>2.226390</td>
</tr>
<tr>
<td>18-hole-golf</td>
<td>0.001929</td>
<td>0.001442</td>
<td>0.001271</td>
<td>0.001405</td>
<td><strong>0.001269</strong></td>
<td>0.002266</td>
<td>0.001292</td>
<td>0.002147</td>
</tr>
<tr>
<td>18-hole-golf2</td>
<td>0.002080</td>
<td>0.001821</td>
<td><strong>0.001697</strong></td>
<td>0.002124</td>
<td>0.002633</td>
<td>0.003802</td>
<td>0.002699</td>
<td>0.001947</td>
</tr>
<tr>
<td>aust</td>
<td>0.000375</td>
<td>0.000398</td>
<td><strong>0.000344</strong></td>
<td>0.000409</td>
<td>0.000438</td>
<td>0.000455</td>
<td>0.000601</td>
<td>0.000629</td>
</tr>
<tr>
<td>aaa-bbb-ccc</td>
<td>0.000420</td>
<td>0.000415</td>
<td><strong>0.000378</strong></td>
<td>0.000427</td>
<td>0.000491</td>
<td>0.000502</td>
<td>0.000647</td>
<td>0.000942</td>
</tr>
<tr>
<td>ages</td>
<td>0.004097</td>
<td>0.004339</td>
<td><strong>0.003973</strong></td>
<td>0.004027</td>
<td>0.006529</td>
<td>0.004150</td>
<td>0.005465</td>
<td>0.004179</td>
</tr>
<tr>
<td>a-puzzle</td>
<td>0.001911</td>
<td>0.001873</td>
<td><strong>0.001812</strong></td>
<td>0.002027</td>
<td>0.002423</td>
<td>0.002978</td>
<td>0.002287</td>
<td>0.003404</td>
</tr>
<tr>
<td>a-puzzle2</td>
<td>0.001886</td>
<td>0.002064</td>
<td><strong>0.001661</strong></td>
<td>0.002172</td>
<td>0.003259</td>
<td>0.003513</td>
<td>0.002334</td>
<td>0.001873</td>
</tr>
<tr>
<td>arch-friends</td>
<td>0.000395</td>
<td>0.000508</td>
<td>0.000480</td>
<td>0.00441</td>
<td>0.000495</td>
<td>0.000418</td>
<td>0.000660</td>
<td>0.000577</td>
</tr>
<tr>
<td>bank-card</td>
<td>0.003450</td>
<td>0.002587</td>
<td><strong>0.002268</strong></td>
<td>0.004316</td>
<td>0.002507</td>
<td>0.004391</td>
<td>0.002337</td>
<td>0.003772</td>
</tr>
<tr>
<td>added-corner</td>
<td>0.002173</td>
<td>0.002161</td>
<td><strong>0.001904</strong></td>
<td>0.002520</td>
<td>0.08631</td>
<td>0.007225</td>
<td>0.021660</td>
<td>0.002548</td>
</tr>
<tr>
<td>domino</td>
<td>0.098882</td>
<td>0.077805</td>
<td>0.079403</td>
<td>0.091052</td>
<td>25.523</td>
<td>0.287813</td>
<td>58.788783</td>
<td>0.073635</td>
</tr>
</tbody>
</table>