MiniZinc 指南

金姆·马里奥特和彼得·斯塔基
以及来自莱斯利·科尼克和霍斯特·塞姆路易斯的贡献

目录

1 引言 **2**

2 MiniZinc 基本模型 **3**

2.1 第一个实例 3
2.2 算术优化实例 7
2.3 数据文件和谓词 8
2.4 实数求解 11
2.5 模型的基本结构 14

3 更多复杂模型 17
3.1 数组和集合 . . . 17
3.2 全局约束 . . . 28
3.3 条件表达式 . . . 29
3.4 枚举类型 . . . 32
3.5 复杂约束 . . . 34
3.6 集合约束 . . . 40
3.7 汇总 . . . 41

4 谓词和函数 47
4.1 全局约束 . . . 47
4.1.1 Alldifferent . . . 47
4.1.2 Cumulative . . . 47
4.1.3 Table . . . 49
4.1.4 Regular . . . 49
4.2 定义谓词 . . . 54
4.3 定义函数 . . . 56
4.4 反射函数 . . . 57
4.5 局部变量 . . . 59
4.6 语境 . . . 60
4.7 局部约束 . . . 63
4.8 定义域反射函数 . . . 63
4.9 作用域 . . . 66

5 选项类型 66
5.1 声明和使用选项类型 . . . 67
5.2 隐藏选项类型 . . . 67

6 搜索 70
6.1 有限域搜索 . . . 70
6.2 搜索注解 . . . 71
6.3 注解 . . . 73

7 MiniZinc 中的有效建模实践 76
7.1 变量界限 76
7.2 无约束变量 77
7.3 有效的生成元 79
7.4 冗余约束 81
7.5 模型选择 82
7.6 多重建模和连通 83

8 在 MiniZinc 中对布尔可满足性问题建模 85
8.1 整型建模 86
8.2 非等式建模 87
8.3 势约束建模 87

A MiniZinc 关键字 96

B MiniZinc 操作符 96

C MiniZinc 函数 96

1 引言

MiniZinc 是一个用来描述关于整型数与实型数的约束优化和决策问题的语言。尽管一个 MiniZinc 模型可能会有一些注解去指导下层的求解器求解，但此模型不会描述如何解决此问题。

MiniZinc 是为了易于与后端的求解器交接而设计的。此交接是通过转化一个输入的 MiniZinc 模型以及数据文件为一个 FlatZinc 模型来达到的。FlatZinc 模型包含了变量声明，约束定义和（若此问题是优化问题）一个目标函数的定义。每一个后端求解器有自己专有的 MiniZinc 到 FlatZinc 的转化，这样他们就可以自己控制得到什么格式的约束。特别地，MiniZinc 允许把全局约束写为其分解形式。

2 MiniZinc 基本模型

在此节中，我们利用两个简单的例子来介绍一个 MiniZinc 模型的基本结构。

2.1 第一个实例

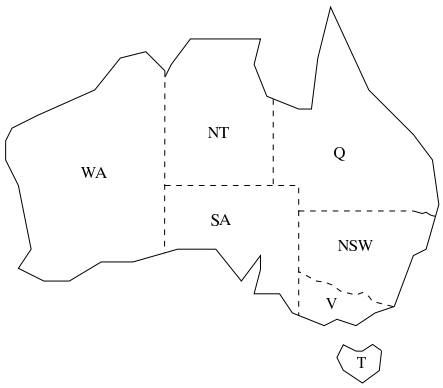


图 1: 澳大利亚各州。

作为我们的第一个例子，假设我们要去给图 1中的澳大利亚地图涂色。它包含了七个不同的州和地区，而每一块都要被涂一个颜色来保证相邻的区域有不同的颜色。

我们可以很容易的用 MiniZinc 给此问题建模。此模型在图 2中给出。模型中的第一行是注释。注释开始于 '%' 来表明此行剩下的部分是注释。MiniZinc 同时也含有 C 语言风格的由 '/*' 开始和 '*/' 结束的块注释。

模型中的下一部分声明了模型中的变量。此行

```
int: nc = 3;
```

定义了一个问题的参数来代表可用的颜色个数。在很多编程语言中，参数和变量是类似的。它们必须被声明并且指定一个类型。在此例中，类型是 **int**。通过赋值，它们被设了值。MiniZinc 允许变量声明时被赋值（就像上面那一行）或者单独给出一个赋值语句。因此下面的表示跟上面的只一行表示是相等的

```
int: nc;
nc = 3;
```

和很多编程语言中的变量不一样的是，这里的参数只可以被赋唯一的值。如果一个参数出现在了多于一个的赋值中，就会出现错误。

aust ≡ [download]

```
% 用 nc 个颜色来涂澳大利亚
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;

output ["wa=\(wa)\t nt=\(nt)\t sa=\(sa)\n",
        "q=\(q)\t nsw=\(nsw)\t v=\(v)\n",
         "t=", show(t),  "\n"];
```

图 2: 一个用来给澳大利亚的州和地区涂色的 MiniZinc 模型 `aust.mzn`。

基本的参数类型包括整型（`int`），浮点型（`float`），布尔型（`bool`）以及字符串型（`string`）。同时也支持数组和集合。

MiniZinc 模型同时也可能包含另一种类型的变量：*决策变量*。决策变量是数学的或者逻辑的变量。和一般的编程语言中的参数和变量不同，建模者不需要给决策变量一个值。而是在开始时，一个决策变量的值是不知道的。只有当 MiniZinc 模型被执行的时候，求解系统才来决定决策变量是否可以被赋值从而满足模型中的约束。若满足，则被赋值。

在我们的模型例子中，我们给每一个区域一个决策变量wa，nt，sa，q，nsw，v 和 t。它们代表了会被用来填充区域的（未知）颜色。

对于每一个决策变量，我们需要给出变量可能的取值集合。这个被称为变量的*定义域*。定义域部分可以在变量声明的时候同时给出，这时决策变量的类型就会从定义域中的数值的类型推断出。

MiniZinc 中的决策变量的类型可以为布尔型，整型，浮点型或者集合。同时也可以

是元素为决策变量的数组。在我们的 MiniZinc 模型例子中，我们使用整型去给不用的颜色建模。通过使用 `var`声明，我们的每一个决策变量被声明为定义域为一个整数类型的范围表示`1..nc`，来表明集合 $\{1,2,...,nc\}$。所有数值的类型为整型，所以模型中的所有的变量是整型决策变量。

标识符

用来命名参数和变量的标识符是一列由大小写字母，数字以及下划线‘_’字符组成的字符串。它们必须开始于一个字母字符。因此 `myName_2` 是一个有效的标识符。MiniZinc（和 Zinc）的关键字不允许被用为标识符名字。它们在 中被列出。所有的 MiniZinc 操作符都不能被用做标识符名字。它们在 中被列出。

MiniZinc 仔细地区别了以下两种模型变量：参数和决策变量。利用决策变量创建的表达式类型比利用参数可以创建的表达式类型更局限。但是，在任何可以用决策变量的地方，同类型的参数变量也可以被应用。

整型变量声明

一个整型参数变量可以被声明为以下两种方式：

int : ⟨ 变量名⟩

⟨l⟩ .. ⟨u⟩ : ⟨ 变量名⟩

l和 u是固定的整型表达式。

一个整型决策变量被声明为以下两种方式：

var int : ⟨ 变量名⟩

var ⟨l⟩ .. ⟨u⟩ : ⟨ 变量名⟩

l和 u是固定的整型表达式。

参数和决策变量形式上的区别在于对变量的实例化。变量的实例化和类型的结合被叫为类型 -实例化。既然你已经开始使用 MiniZinc，毫无疑问的你会看到很多类型 -实例化的错误例子。

模型的下一部分是约束。它们详细说明了决策变量想要组成一个模型的有效解必须要满足的布尔型表达式。在这个例子中我们有一些决策变量之间的不等式。它们规定如果两个区域是相邻的，则它们必须有不同的颜色。

关系操作符

MiniZinc 提供了以下关系操作符：相等（=或者 ==），不等（!=），小于（<），大于（>），小于等于（<=）和大于等于（>=）。

模型中的下一行：

```
solve satisfy;
```

表明了它是什么类型的问题。在这个例子中，它是一个满足问题：我们希望给决策变量找到一个值使得约束被满足，但具体是哪一个值却没有所谓。

模型的最后一个部分是输出语句。它告诉 MiniZinc 当模型被运行并且找到一个解的时候，要输出什么。

输出和字符串

一个输出语句跟着一串字符。它们通常或者是写在双引号之间的字符串常量并且对特殊字符用类似 C 语言的标记法，或者是 *show(e)*格式的表达式，其中 *e*是 MiniZinc 表达式。例子中的 `\n`代表换行符，`\t`代表制表符。

数字的 *show*有各种不同方式的表示：*show_int(n,X)* 在至少 $|n|$ 个字符里输出整型 *X*的值，若 $n > 0$ 则右对齐，否则则左对齐；*show_float(n,d,X)*在至少 $|n|$ 个字符里输出浮点型 *X*的值，若 $n > 0$ 则右对齐，否则则左对齐，并且小数点后有 d 个字符。

字符串常量必须在同一行中。长的字符串常量可以利用字符串连接符 `++` 来分成几行。例如，字符串常量`"Invalid datafile: Amount of flour is non-negative"`和字符串常量表达式 `"Invalid datafile: " ++ "Amount of flour is non-negative"`是相等的。

MiniZinc 支持内插字符串。表达式可以直接插入字符串常量中。`\(e)`形式的子字符串会被替代为 `show(e)`。例如，`"t=\(t)\n"` 产生和`"t=" ++ show(t) ++ "\n"`一样的字符串。

一个模型只可以包含最多一个输出语句。

MiniZinc 中 G12 的实现使得我们可以通过输入

```
$ mzn-g12fd aust.mzn
```

评估我们的模型。其中 `aust.mzn`是包含我们的 MiniZinc 模型的文件名字。我们必须使用文件扩展名 ".mzn" 来表明一个 MiniZinc 模型。`mzn-g12fd`命令使用 G12 有限域求解器去评估我们的模型。

当我们运行上面的命令后，我们得到如下的结果：

cakes ≡ [download]

```
% 为校园游乐会做蛋糕

var 0..100: b; % 香蕉蛋糕的个数
var 0..100: c; % 巧克力蛋糕的个数

% 面粉
constraint 250*b + 200*c <= 4000;
% 香蕉
constraint 2*b  <= 6;
% 糖
constraint 75*b + 150*c <= 2000;
% 黄油
constraint 100*b + 150*c <= 500;
% 可可粉
constraint 75*c <= 500;

% 最大化我们的利润
solve maximize 400*b + 450*c;

output ["no. of banana cakes = \(b)\n",
        "no. of chocolate cakes = \(c)\n"];
```

图 3: 决定为了校园游乐会要烤多少香蕉和巧克力蛋糕的模型。

```
wa=2  nt=3  sa=1
q=2  nsw=3  v=2
t=1
----------
```

10 个破折号 ---------- 这行是自动被 MiniZinc 输出的，用来表明一个解已经被找到。

2.2 算术优化实例

我们的第二个例子来自于要为了本地的校园游乐会烤一些蛋糕的需求。我们知道如何制作两种蛋糕。[1] 一个香蕉蛋糕的制作需要 250 克自发酵的面粉，2 个捣碎的香蕉，75 克糖和 100 克黄油。一个巧克力蛋糕的制作需要 200 克自发酵的面粉，75 克可可粉，150 克糖和 150 克黄油。一个巧克力蛋糕可以卖 $4.50，一个香蕉蛋糕可以卖 $4.00。我

[1] 警告: 请不要在家里使用这些配方制作

们一共有 4 千克的自发酵面粉，6 个香蕉，2 千克的糖，500 克的黄油和 500 克的可可粉。问题是对每一种类型的蛋糕，我们需要烤多少给游乐会来得到最大的利润。一个可能的 MiniZinc 模型在图 3中给出。

第一个新特征是算术表达式的使用。

整数算术操作符

MiniZinc 提供了标准的整数算术操作符。加（+），减（-）, 乘（*），整数除（div）和整数模（mod）。同时也提供了一元操作符 +和 -。

整数模被定义为输出和被除数 a 一样正负的 $(a \texttt{ mod } b)$ 值。整数除被定义为使得 $a = b*(a \texttt{ div } b) + (a \texttt{ mod } b)$ 的 $(a \texttt{ div } b)$ 值。

MiniZinc 提供了标准的整数函数用来取绝对值（abs）和幂函数（pow）。例如 abs(-4)和 pow(2,5)分别求得数值 4 和 32。

算术常量的语法是相当标准的。整数常量可以是十进制，十六进制或者八进制。例如 0，005，123，0x1b7，0o777。

例子中的第二个新特征是优化。这行

```
solve maximize 400 * b + 450 * c;
```

指出我们想找一个可以使 solve 语句中的表达式（我们叫做目标）最大化的解。这个目标可以为任何类型的算术表达式。我们可以把关键字 maximize 换为 minimize来表明一个最小化问题。

当我们运行上面这个模型时，我们得到以下的结果：

```
no. of banana cakes = 2
no. of chocolate cakes = 2
----------
==========
```

一旦系统证明了一个解是最优解，这一行 ========== 在最优化问题中会自动被输出。

2.3 数据文件和谓词

此模型的一个缺点是如果下一次我们希望解决一个相似的问题，即我们需要为学校烤蛋糕（这是经常发生的），我们需要改变模型中的约束来表明食品间拥有的原料数量。如果我们想重新利用此模型，我们最好使得每个原料的数量作为模型的参数，然后在模型的最上层设置它们的值。

更好的办法是在一个单独的数据文件中设置这些参数的值。MiniZinc（就像很多其他的建模语言一样）允许使用数据文件来设置在原始模型中声明的参数的值。通过运行不同的数据文件，使得同样的模型可以很容易地和不同的数据一起使用。

数据文件的文件扩展名必须是 ".dzn"，来表明它是一个 MiniZinc 数据文件。一个模型可以被任何多个数据文件运行（但是每个变量/参数在每个文件中只能被赋一个值）

我们的新模型在图 4中给出。我们可以用下面的命令来运行

```
$ mzn-g12fd cakes2.mzn pantry.dzn
```

数据文件 `pantry.dzn`在 图 5中给出。我们得到和 `cakes.mzn`同样的结果。运行下面的命令

```
$ mzn-g12fd  cakes2.mzn pantry2.dzn
```

利用另外一个图 5中定义的数据集，我们得到如下结果

```
no. of banana cakes = 3
no. of chocolate cakes = 8
----------
==========
```

如果我们从 `cakes.mzn`中去掉输出语句，MiniZinc 会使用默认的输出。这种情况下得到的输出是

```
b = 3;
c = 8;
----------
==========
```

默认输出

一个没有输出语句的 MiniZinc 模型会给每一个决策变量以及它的值一个输出行，除非决策变量已经在声明的时候被赋了一个表达式。注意观察此输出是如何呈现一个正确的数据文件格式的。

通过使用命令行 `-D` *string*，小的数据文件可以被直接输入而不是必须要创建一个`.dzn` 文件，其中 *string*是数据文件里面的内容。例如，命令

```
$ mzn-g12fd cakes2.mzn -D \
     "flour=4000;banana=6;sugar=2000;butter=500;cocoa=500;"
```

cakes2 ≡ [download]

```
% 为校园游乐会做蛋糕（和数据文件一起）

int: flour;  % 拥有的面粉克数
int: banana; % 拥有的香蕉个数
int: sugar;  % 拥有的糖克数
int: butter; % 拥有的黄油克数
int: cocoa;  % 拥有的可可粉克数

constraint assert(flour >= 0,"Invalid datafile: " ++
                  "Amount of flour is non-negative");
constraint assert(banana >= 0,"Invalid datafile: " ++
                  "Amount of banana is non-negative");
constraint assert(sugar >= 0,"Invalid datafile: " ++
                  "Amount of sugar is non-negative");
constraint assert(butter >= 0,"Invalid datafile: " ++
                  "Amount of butter is non-negative");
constraint assert(cocoa >= 0,"Invalid datafile: " ++
                  "Amount of cocoa is non-negative");

var 0..100: b; % 香蕉蛋糕的个数
var 0..100: c; % 巧克力蛋糕的个数

% 面粉
constraint 250*b + 200*c <= flour;
% 香蕉
constraint 2*b  <= banana;
% 糖
constraint 75*b + 150*c <= sugar;
% 黄油
constraint 100*b + 150*c <= butter;
% 可可粉
constraint 75*c <= cocoa;

% 最大化我们的利润
solve maximize 400*b + 450*c;

output ["no. of banana cakes = \(b)\n",
        "no. of chocolate cakes = \(c)\n"];
```

图 4: 独立于数据的用来决定为了校园游乐会要烤多少香蕉和巧克力蛋糕的模型。

pantry ≡ [download]

```
flour = 4000;
banana = 6;
sugar = 2000;
butter = 500;
cocoa = 500;
```

pantry2 ≡ [download]

```
flour = 8000;
banana = 11;
sugar = 3000;
butter = 1500;
cocoa = 800;
```

图 5: cakes2.mzn的数据文件例子

会给出和

```
$ mzn-g12fd cakes2.mzn pantry.dzn
```

一模一样的结果。

数据文件只能包含给模型中的决策变量和参数赋值的语句。

防御性编程建议我们应该检查数据文件中的数值是否合理。在我们的例子中，检查所有原料的份量是否是非负的并且若不正确则产生一个运行错误，这是明智的。MiniZinc 提供了一个内置的布尔型操作符断言用来检查参数值。格式是 *assert(B,S)*。布尔型表达式 *B* 被检测。若它是假的，运行中断。此时字符串表达式 *S* 作为错误信息被输出。如果我们想当面粉的份量是负值的时候去检测出并且产生合适的错误信息，我们可以直接加入下面的一行

```
constraint assert(flour >= 0,"Amount of flour is non-negative");
```

给我们的模型。注意断言表达式是一个布尔型表达式，所以它被看做是一种类型的约束。我们可以加入类似的行来检测其他原料的份量是否是非负值。

2.4 实数求解

通过使用浮点数求解，MiniZinc 也支持“实数”约束求解。考虑一个要在 4 季度分期偿还的一年短期贷款问题。此问题的一个模型在图 6中给出。它使用了一个简单的计算每季度结束后所欠款的利息计算方式。

注意我们声明了一个浮点型变量 *f*，它和整型变量很类似。只是这里我们使用关键字 `float` 而不是 `int`。

loan ≡ [download]

```
% 变量
var float: R;        %  季度还款
var float: P;        %  初始借贷本金
var 0.0 .. 10.0: I;  %  利率

% 中间变量
var float: B1; % 一个季度后的欠款
var float: B2; % 两个季度后的欠款
var float: B3; % 三个季度后的欠款
var float: B4; % 最后欠款

constraint B1 = P * (1.0 + I) - R;
constraint B2 = B1 * (1.0 + I) - R;
constraint B3 = B2 * (1.0 + I) - R;
constraint B4 = B3 * (1.0 + I) - R;

solve satisfy;

output [
 "Borrowing ", show_float(0, 2, P), " at ", show(I*100.0),
 "% interest, and repaying ", show_float(0, 2, R),
  "\nper quarter for 1 year leaves ", show_float(0, 2, B4), " owing\n"
];
```

图 6: 确定一年借款每季度还款关系的模型

浮点型变量声明

一个浮点型参数变量被声明为以下两种方式：

float : ⟨ 变量名⟩

⟨l⟩ .. ⟨u⟩ : ⟨ 变量名⟩

其中 l和 u是固定浮点型表达式.

一个浮点型决策变量被声明为以下两种方式：

var float : ⟨ 变量名⟩

var ⟨l⟩ .. ⟨u⟩ : ⟨ 变量名⟩

其中 l和 u是固定浮点型表达式.

我们可以用同样的模型来回答一系列不同的问题。第一个问题是：如果我以利

息 4% 借款 $1000 并且每季度还款 $260，我最终还欠款多少？这个问题在数据文件 loan1.dzn中被编码。

由于我们希望用实数求解，我们需要使用一个和 mzn-g12fd使用的有限域求解器不同的求解器。一个合适的求解器必须是一个支持混合整数线性规划的求解器。发行 MiniZinc 包含了这样的一个求解器。我们可以使用命令 mzn-g12mip来触发。

```
$ mzn-g12mip loan.mzn loan1.dzn
```

输出是

```
Borrowing 1000.00 at 4.0\% interest, and repaying 260.00
per quarter for 1 year leaves 65.78 owing
----------
```

第二个问题是如果我希望用 4% 的利息来借款 $1000 并且在最后的时候一点都不欠款，我需要在每季度还款多少？这个问题在数据文件 loan2.dzn中被编码。运行命令

```
$ mzn-g12mip loan.mzn loan2.dzn
```

后的输出是

```
Borrowing 1000.00 at 4.0\% interest, and repaying 275.49
per quarter for 1 year leaves 0.00 owing
----------
```

第三个问题是如果我每季度还款 $250，我可以在利息是 4% 的情况下借款多少使得最后一点都不欠款？这个问题在数据文件 loan3.dzn中被编码。运行命令

```
$ mzn-g12mip loan.mzn loan3.dzn
```

后的输出是

```
Borrowing 907.47 at 4.0\% interest, and repaying 250.00
per quarter for 1 year leaves 0.00 owing
----------
```

loan1 ≡ [download]
```
I = 0.04;
P = 1000.0;
R = 260.0;
```

loan2 ≡ [download]
```
I = 0.04;
P = 1000.0;
B4 = 0.0;
```

loan3 ≡ [download]
```
I = 0.04;
R = 250.0;
B4 = 0.0;
```

图 7: loan.mzn的数据文件例子

浮点算术操作符

MiniZinc 提供了标准的浮点算术操作符。加（+），减（-），乘（*）和浮点除（/）。同时也提供了一元操作符 +和 -。

MiniZinc 不会自动地强制转换整数为浮点数。内建函数 int2float被用来达到此目的。

MiniZinc 同时也包含浮点型函数来计算绝对值（abs），平方根（sqrt），自然对数（ln），底数为 2 的对数（log2），底数为 10 的对数（log10），e 的幂（exp），正弦（sin），余弦（cos），正切（tan），反正弦（asin），反余弦（acos），反正切（atan），双曲正弦（sinh），双曲余弦（cosh），双曲正切（tanh），双曲反正弦（asinh），双曲反余弦（acosh），双曲反正切（atanh），和唯一的二元函数次方（pow），其余的都是一元函数。

算术常量的语法是相当标准的。浮点数常量的例子有 1.05，1.3e-5和 1.3+E5。

2.5 模型的基本结构

我们现在可以去总结 MiniZinc 模型的基本结构了。它由多个项组成，每一个在其最后都有一个分号‘;’。项可以按照任何顺序出现。例如，标识符在被使用之前不需要被声明。

有七种类型的项。

- 引用项允许另外一个文件的内容被插入模型中。它们有以下形式：

  ```
  include ⟨ 文件名⟩;
  ```

  其中文件名是一个字符串常量。它们使得大的模型可以被分为小的子模型以及包含库文件中定义的约束。我们会在图 11中看到一个例子。

- 变量声明声明新的变量。这种变量是全局变量，可以在模型中的任何地方被提到。变量有两种。在模型中被赋一个固定值的参数变量以及只有在模型被求解的时候

才会被赋值的决策变量。我们称参数是固定的，决策变量是 不固定的。变量可以选择性地被赋一个值来作为声明的一部分。形式是：

```
⟨ 类型 -实例化表达式⟩: ⟨ 变量⟩ [ = ⟨ 表达式⟩];
```

类型 -实例化表达式给了变量的类型和实例化。这些是 MiniZinc 比较复杂的其中一面。用 par来实例化声明参数，用 var来实例化声明决策变量。如果没有明确的实例化声明，则变量是一个参数。类型可以为基类型，一个整数或者浮点数范围，或者一个数组或集合。基类型有 float，int，string，bool，ann。其中只有 float，int和 bool可以被决策变量使用。基类型 ann是一个注解—我们会在节 6中讨论注解。整数范围表达式可以被用来代替类型 int。类似的，浮点数范围表达式可以被用来代替类型 float。这些通常被用来定义一个整型决策变量的定义域，但也可以被用来限制一个整型参数的范围。变量声明的另外一个用处是定义枚举类型—我们会在小节 3.4中讨论。

- 赋值项给一个变量赋一个值。它们有以下形式：

```
⟨ 变量⟩ = ⟨ 表达式⟩;
```

数值可以被赋给决策变量。在这种情况下，赋值相当于加入 constraint ⟨ 变量⟩ = ⟨ 表达式⟩;

- 约束项是模型的心脏。它们有以下形式：

```
constraint ⟨ 布尔型表达式⟩;
```

我们已经看到了使用算术比较的简单约束以及内建函数 *assert*操作符。在下一节我们会看到更加复杂的约束例子。

- 求解项详细说明了到底要找哪种类型的解。正如我们看到的，它们有以下三种形式：

```
solve satisfy;
solve maximize ⟨ 算术表达式⟩;
solve minimize ⟨ 算术表达式⟩;
```

一个模型必须有且只有一个求解项。

- 输出项用来恰当的呈现模型运行后的结果。它们有下面的形式：

```
output [⟨ 字符串表达式⟩, ··· , ⟨ 字符串表达式⟩ ];
```

如果没有输出项，MiniZinc 会默认输出所有没有被以赋值项的形式赋值的决策变量值。

- 谓词函数和测试项被用来定义新的约束，函数和布尔测试。我们会在节 4中讨论。
- 注解项用来定义一个新的注解。我们会在节 6中讨论。

3 更多复杂模型

在上一节中，我们介绍了 MiniZinc 模型的基本结构。在这一节中，我们介绍数组和集合数据结构，枚举类型，以及更加复杂的约束。

3.1 数组和集合

在绝大多数情况下，我们都是有兴趣建一个约束和变量的个数依赖于输入数据的模型。为了达到此目的，我们通常会使用数组。

考虑一个关于金属矩形板温度的简单有限元素模型。通过把矩形板在 2 维的矩阵上分成有限个的元素，我们近似计算矩形板上的温度。一个模型在图 8中给出。它声明了有限元素模型的宽 w和高 h。声明

```
arraydec ≡
  set of int: HEIGHT = 0..h;
  set of int: CHEIGHT = 1..h-1;
  set of int: WIDTH = 0..w;
  set of int: CWIDTH = 1..w-1;
  array[HEIGHT,WIDTH] of var float: t; %  在点（i,j）处的温度
```

声明了四个固定的整型集合来描述有限元素模型的尺寸：HEIGHT是整个模型的整体高度，而 CHEIGHT是省略了顶部和底部的中心高度，WIDTH是模型的整体宽度，而 CWIDTH是省略了左侧和右侧的中心宽度。最后，声明了一个浮点型变量组成的行编号从 0 到 w，列编号从 0 到 h 的两维数组 t用来表示金属板上每一点的温度。我们可以用表达式 t[i , j]来得到数组中第 i 行和第 j 列的元素。

拉普拉斯方程规定当金属板达到一个稳定状态时，每一个内部点的温度是它的正交相邻点的平均值。约束

```
equation ≡
  % 拉普拉斯方程：每一个内部点温度是它相邻点的平均值
  constraint forall(i in CHEIGHT, j in CWIDTH)(
                4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);
```

保证了每一个内部点（i,j）是它的四个正交相邻点的平均值。约束

laplace ≡ [download]

```
int: w = 4;
int: h = 4;

►arraydec
var float: left;   %  左侧温度
var float: right;  %  右侧温度
var float: top;    %  顶部温度
var float: bottom; %  底部温度

►equation
►sides
►corners
left = 0.0;
right = 0.0;
top = 100.0;
bottom = 0.0;

solve satisfy;

output [ show_float(6, 2, t[i,j]) ++
         if j == h then "\n" else " " endif |
         i in HEIGHT, j in WIDTH
];
```

图 8: 决定稳定状态温度的有限元素金属板模型（laplace.mzn）.

sides ≡

```
% 边约束
constraint forall(i in CHEIGHT)(t[i,0] = left);
constraint forall(i in CHEIGHT)(t[i,w] = right);
constraint forall(j in CWIDTH)(t[0,j] = top);
constraint forall(j in CWIDTH)(t[h,j] = bottom);
```

限制了每一个边的温度必须是相等的，并且给了这些温度名字：left，right，top和bottom。而约束

```
corners ≡
  % 角约束
  constraint t[0,0]=0.0;
  constraint t[0,w]=0.0;
  constraint t[h,0]=0.0;
  constraint t[h,w]=0.0;
```

确保了角的温度（这些是不相干的）被设置为 0.0。我们可以用图 8中给出的模型来决定一个被分成 5 × 5 个元素的金属板的温度。其中左右下侧的温度为 0，上侧的温度为 100.

运行命令

```
$ mzn-g12mip laplace.mzn
```

得到输出

```
  0.00 100.00 100.00 100.00   0.00
  0.00  42.86  52.68  42.86   0.00
  0.00  18.75  25.00  18.75   0.00
  0.00   7.14   9.82   7.14   0.00
  0.00   0.00   0.00   0.00   0.00
----------
```

集合

集合变量用以下方式声明

`set of` ⟨ 类型 -实例化⟩ : ⟨ 变量名⟩ ;

整型，枚举型（参见后面），浮点型和布尔型集合都可以定义。决策变量集合只可以是类型为整型或者枚举型的变量集合。集合常量有以下形式

{ ⟨ 表达式 $_1$⟩, ... , ⟨ 表达式 $_n$⟩ }

或者是以下形式的整型，枚举型或浮点型范围表达式

⟨ 表达式 $_1$⟩ .. ⟨ 表达式 $_2$⟩

标准的集合操作符有：元素属于（in），集合包含（subset, superset），并集（union），交集（intersect），集合差运算（diff），集合对称差（symdiff）和集合元素的个数（card）。

我们已经看到集合变量和集合常量（包含范围）可以被用来作为变量声明时的隐式类型。在这种情况下变量拥有集合元素中的类型并且被隐式地约束为集合中的一个成员。

我们的烤蛋糕问题是一个非常简单的批量生产计划问题例子。在这类问题中，我们希望去决定每种类型的产品要制造多少来最大化利润。同时制造一个产品会消耗不同数量固定的资源。我们可以扩展图 4中的 MiniZinc 模型为一个不限制资源和产品类型的模型去处理这种类型的问题。这个模型在图 9中给出。一个（烤蛋糕问题的）数据文件例子在图 10中给出。

这个模型的新特征是只用枚举类型。这使得我们可以把资源和产品的选择作为模型的参数。模型的第一个项

```
enum Products;
```

声明 Products为未知的产品集合。

simple-prod-planning ≡ [download]

```
% 要制造的产品
enum Products;
% 每种产品的单位利润
array[Products] of int: profit;
% 用到的资源
enum Resources;
% 每种资源可获得的数量
array[Resources] of int: capacity;

% 制造一个单位的产品需要的资源单位量
array[Products, Resources] of int: consumption;
constraint assert(forall (r in Resources, p in Products)
          (consumption[p,r] >= 0), "Error: negative consumption");

% 产品数量的界
int: mproducts = max (p in Products)
                     (min (r in Resources where consumption[p,r] > 0)
                          (capacity[r] div consumption[p,r]));

% 变量: 每种产品我们需要制造多少
array[Products] of var 0..mproducts: produce;
array[Resources] of var 0..max(capacity): used;

% 产量不可以使用超过可获得的资源量:
constraint forall (r in Resources) (
      used[r] = sum (p in Products)(consumption[p, r] * produce[p])
      /\ used[r] <= capacity[r]
);

% 最大化利润
solve maximize sum (p in Products) (profit[p]*produce[p]);

output [ "\(p) = \(produce[p]);\n" | p in Products ] ++
       [ "\(r) = \(used[r]);\n" | r in Resources ];
```

图 9: 简单批量生产计划模型（simple-prod-planning.mzn）。

simple-prod-planning-data ≡ [download]

```
% 简单批量生产计划模型的数据文件
Products = { BananaCake, ChocolateCake };
profit = [400, 450]; % 以分为单位

Resources = { Flour, Banana, Sugar, Butter, Cocoa };
capacity = [4000, 6, 2000, 500, 500];

consumption= [| 250, 2, 75, 100, 0,
              | 200, 0, 150, 150, 75 |];
```

图 10: 简单批量生产计划模型的数据文件例子。

枚举类型

枚举类型，我们称为 enums，用以下方式声明

enum ⟨ 变量名⟩;

一个枚举类型用以下赋值的方式定义

⟨ 变量名⟩ = { ⟨ 变量名 $_1$⟩, … , ⟨ 变量名 $_n$⟩ } ;

其中变量名 $_1$，…，变量名 $_n$ 是名为变量名的枚举类型中的元素。通过这个定义，枚举类型中的每个元素也被有效地声明为这个类型的一个新的常量。声明和定义可以像往常一样结合为一行。

第二个项声明了一个整型数组：

```
array[Products] of int: profit;
```

数组 profit的下标集合是 Products。理想情况下，这种声明方式表明只有集合Products中的元素才能被用来做数组的下标。但是 MiniZinc 中的枚举类型被看做跟整型一样来处理，所以目前唯一的保障是只有 $1, 2, ..., |Products|$ 是合法的数组下标。访问数组 profit[i]可以得到产品 i的利润。

有 n 个元素组成的枚举类型中的元素的行为方式和整数 $1..n$ 的行为方式很像。它们可以被比较，它们可以按照它们出现在枚举类型定义中的顺序被排序，它们可以遍历，它们可以作为数组的下标，实际上，它们可以出现在一个整数可以出现的任何地方。

在数据文件例子中，我们用一列整数来初始化数组

```
Products = { BananaCake, ChocolateCake };
profit = [400,450];
```

意思是香蕉蛋糕的利润是 400，而巧克力蛋糕的利润是 450。在内部，BananaCake会被看成是像整数 1 一样，而 ChocolateCake会被看成像整数 2 一样。MiniZinc 虽然不提供明确的列表类型，但用 $1..n$ 为下标集合的一维数组表现起来就像列表。我们有时候也会称它们为列表。

根据同样的方法，接下来的两项中我们声明了一个资源集合 Resources，一个表明每种资源可获得量的数组 capacity。

更有趣的是项

```
array[Products, Resources] of int: consumption;
```

声明了一个两维数组 consumption。consumption[p,r]的值是制造一单位的产品 p所需要的资源 r的数量。其中第一个下标是行下标，而第二个下标是列下标。

数据文件包含了一个两维数组的初始化例子

```
consumption= [| 250, 2, 75,  100, 0,
              | 200, 0, 150, 150, 75 |];
```

注意分隔符 |是怎样被用来分隔行的。

数组

因此，MiniZinc 提供一维和多维数组。它们用以下类型来声明：

array[⟨ 下标集合 $_1$⟩, ..., ⟨ 下标集合 $_n$⟩] **of** ⟨ 类型 -实例化⟩

MiniZinc 要求数组声明要给出每一维的下标集合。下标集合或者是一个整型范围，一个被初始化为整型范围的集合变量，或者是一个枚举类型。数组可以是所有的基类型：整型，枚举型，布尔型，浮点型或者字符串型。这些可以是固定的或者不固定的，除了字符串型，它只可以是参数。数组也可以作用于集合但是不可以作用于数组。

一维数组常量有以下格式

[⟨ 表达式 $_1$⟩, ... , ⟨ 表达式 $_n$⟩]

而二维数组常量有以下格式

[| ⟨ 表达式 $_{1,1}$⟩, ... , ⟨ 表达式 $_{1,n}$⟩, | ..., | ⟨ 表达式 $_{m,1}$⟩, ... , ⟨ 表达式 $_{m,n}$⟩ |]

其中这个数组有 m 行 n 列。

内建函数 array1d, array2d等家族可以被用来从一个列表（或者更准确的说是一个一维数组）去实例化任何维度的数组。调用

array$n$**d**(⟨ 下标集合 $_1$⟩, ..., ⟨ 下标集合 $_n$⟩, ⟨ 列表⟩)

返回一个 n 维的数组，它的下标集合在前 n 个参数给出，最后一个参数包含了数组的元素。例如 **array2d**(1..3, 1..2, [1, 2, 3, 4, 5, 6])和 [|1, 2 |3, 4 |5, 6|])是相等的。

数组元素按照通常的方式获取：a[i,j]给出第 i 行第 j 列的元素。

串联操作符‘++’可以被用来串联两个一维的数组。结果得到一个列表，即一个元素从 1 索引的一维数组。例如 [4000, 6] ++ [2000, 500, 500]求得 [4000, 6, 2000, 500, 500]。内建函数 length返回一维数组的长度。

模型的下一项定义了参数 `mproducts`。它被设为可以生产出的任何类型产品的数量上限。这个确实是一个复杂的内嵌数组推导式和聚合操作符例子。在我们试图理解这些项和剩下的模型之前，我们应该先介绍一下它们。

首先，MiniZinc 提供了在很多函数式编程语言都提供的列表推导式。例如，列表推导式 `[i + j | i, j in 1..3 where j < i]` 算得 `[1 + 2, 1 + 3, 2 + 3]`等同于 `[3, 4, 5]`。`[3, 4, 5]`只是一个下标集合为 1..3 的数组。

MiniZinc 同时也提供了集合推导式，它有类似的语法：例如 `{i + j | i, j in 1..3 where j < i}` 计算得到集合 `{3, 4, 5}`。

列表和集合推导式

列表推导式的一般格式是

[⟨ 表达式⟩ | ⟨ 生成元表达式⟩]

⟨ 表达式⟩ 指明了如何从 ⟨ 生成元表达式⟩ 产生的元素输出列表中创建元素。生成元 ⟨ 生成元表达式⟩ 由逗号分开的一列生成元表达式组成，选择性地跟着一个布尔型表达式。两种格式是

⟨ 生成元⟩, ..., ⟨ 生成元⟩
⟨ 生成元⟩, ..., ⟨ 生成元⟩ **where** ⟨ 布尔表达式⟩

第二种格式中的可选择的 ⟨ 布尔型表达式⟩ 被用作生成元表达式的过滤器：只有满足布尔型表达式的输出列表中的元素才被用来构建元素。⟨ 生成元⟩有以下格式

⟨ 标识符⟩,..., ⟨ 标识符⟩ **in** ⟨ 数组表达式⟩

每一个标识符是一个迭代器，轮流从数值表达式中取值，最后一个标识符变化的最迅速。

列表推导式的生成元和 ⟨ 布尔型表达式⟩ 通常不会涉及决策变量。如果它们确实涉及了决策变量，那么产生的列表是一列 `var opt` T，其中 T 是 ⟨ 表达式⟩ 的类型。更多细节，请参考　　中有关选项类型的论述。

集合推导式几乎和列表推导式一样：唯一的不同是这里使用‘{’和‘}’括住表达式而不是‘[’和‘]’。集合推导式生成的元素必须是固定的，即不能是决策变量。类似的，集合推导式的生成元和可选择的 ⟨ 布尔型表达式⟩ 必须是固定的。

第二，MiniZinc 提供了一系列的可以把一维数组的元素聚合起来的内建函数。它们中最有用的可能是 `forall`。它接收一个布尔型表达式数组（即，约束），返回单个布尔型表达式，它是对数组中的布尔型表达式的逻辑合取。

例如，以下表达式

```
forall( [a[i] != a[j] | i,j in 1..3 where i < j])
```

其中 a是一个下标集合为 1..3 的算术数组。它约束了 a中的元素是互相不相同的。列表推导式计算得到 [a[1] != a[2], a[1] != a[3], a[2] != a[3]]，所以 forall函数返回逻辑合取 a[1] != a[2] ∧ a[1] != a[3] ∧ a[2] != a[3]。

聚合函数

算术数组的聚合函数有：sum 把元素加起来，product把元素乘起来，和 min和max各自返回数组中的最小和最大元素。当作用于一个空的数组时，min和 max返回一个运行错误，sum返回 0，product返回 1。

MiniZinc 为数组提供了包含有布尔型表达式的四个聚合函数。正如我们看到的，它们中的第一个是 forall，它返回一个等于多个约束的逻辑合取的单个约束。第二个函数，exists，返回多个约束的逻辑析取。因此 forall强制数组中的所有约束都满足，而 exists确保至少有一个约束满足。第三个函数，xorall 确保奇数个约束满足。第四个函数，iffall确保偶数个约束满足。

第三个，也是难点的最后一个部分是当使用数组推导式时，MiniZinc 允许使用一个特别的聚合函数的语法。建模者不仅仅可以用

```
forall( [a[i] != a[j] | i,j in 1..3 where i < j])
```

也可以用一个更加数学的表示

```
forall (i,j in 1..3 where i < j) (a[i] != a[j])
```

两种表达方式是完全相等的：建模者可以自由使用任何一个他们认为更自然的表达方式。

生成表达式

一个生成表达式有以下格式

⟨ 聚合函数⟩ (⟨ 生成元表达式⟩) (⟨ 表达式⟩)

圆括号内的 ⟨ 生成元表达式⟩ 以及构造表达式 ⟨ 表达式⟩ 是非选择性的：它们必须存在。它等同于

⟨ 聚合函数⟩([⟨ 表达式⟩ | ⟨ 生成元表达式⟩])

⟨ 聚合函数⟩ 可以是 MiniZinc 的任何由单个数组作为其参数的函数。

接下来我们就来了解图 9中的简单批量生产计划模型剩余的部分。现在请暂时忽略定义 mproducts的这部分。接下来的项：

```
array[Products] of var 0..mproducts: produce;
```

定义了一个一维的决策变量数组 produce。produce[p]的值代表了最优解中产品 p 的数量。下一项

```
array[Resources] of var 0..max(capacity): used;
```

定义了一个辅助变量集合来记录每一种资源的使用量。下面的两个约束

```
constraint forall (r in Resources)
           (used[r] = sum (p in Products) (consumption[p, r] * produce[p]));
constraint forall (r in Resources)(used[r] <= capacity[r] );
```

使用 used[r]计算资源 r的总体消耗以及保证它是少于可获得的资源 r的量。最后，项

```
solve maximize sum (p in Products) (profit[p]*produce[p]);
```

表明这是一个最大化问题以及最大化的目标是全部利润。

现在我们回到 mproducts的定义。对每个产品 p，表达式

```
(min (r in Resources where consumption[p,r] > 0)
                                (capacity[r] div consumption[p,r])
```

决定了在考虑了每种资源 r的数量以及制造产品 p需要的 r量的情况下，p 可以生产的最大量。注意过滤器 where consumption[p,r] > 0的使用保证了只有此产品需要的资源才会考虑，因此避免了出现除数为零的错误。所以，完整的表达式

```
int: mproducts = max (p in Products)
                     (min (r in Resources where consumption[p,r] > 0)
                                (capacity[r] div consumption[p,r]));
```

计算任何产品可以被制造的最大量，因此它可以被作为 produce中的决策变量定义域的上限。

最后，注意输出项是比较复杂的，并且使用了列表推导式去创建一个易于理解的输出。运行

```
$ mzn-g12fd simple-prod-planning.mzn simple-prod-planning-data.dzn
```

输出得到如下结果

send-more-money ≡ [download]

```
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 alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;

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

图 11: SEND+MORE=MONEY 算式谜题模型（`send-more-money.mzn`）。

```
BananaCake = 2;
ChocolateCake = 2;
Flour = 900;
Banana = 4;
Sugar = 450;
Butter = 500;
Cocoa = 150;
----------
```

3.2 全局约束

MiniZinc 包含了一个全局约束的库，这些全局约束也可以被用来定义模型。一个例子是`alldifferent`约束，它要求所有参数中的变量都必须是互相不相等的。

SEND+MORE=MONEY 问题要求给每一个字母赋不同的数值使得此算术约束满足。

图 11中的模型使用 `alldifferent([S,E,N,D,M,O,R,Y])`约束表达式来保证每个字母有不同的数字值。在使用了引用项

```
include "alldifferent.mzn";
```

后，此全局约束 `alldifferent`可以在模型中使用。我们可以用以下代替此行

```
include "globals.mzn";
```

它包含了所有的全局约束。

一系列所有在 MiniZinc 中定义了的全局约束都被包含在了发布的文档中。对一些重要的全局约束的描述，请参见小节 4.1。

3.3 条件表达式

MiniZinc 提供了一个条件表达式 *if-then-else-endif*。它的一个使用例子如下

```
int: r = if y != 0 then x div y else 0 endif;
```

若 y 不是零，则 r 设为 x 除以 y，否则则设为零。

条件表达式

条件表达式的格式是

`if` ⟨ 布尔型表达式⟩ `then` ⟨ 表达式 $_1$⟩ `else` ⟨ 表达式 $_2$⟩ `endif`

它是一个真表达式而不是一个控制流语句，所以它可以被用于其他表达式中。如果 ⟨ 布尔型表达式⟩ 是真，则它取值 ⟨ 表达式 $_1$⟩，否则则是 ⟨ 表达式 $_2$⟩。条件表达式的类型是 ⟨ 表达式 $_1$⟩ 和 ⟨ 表达式 $_2$⟩ 的类型，而它们俩必须有相同的类型。

如果 ⟨ 布尔型表达式⟩ 包含决策变量，则表达式的类型 -实例化是 `var` T，其中 T 是 ⟨ 表达式 $_1$⟩ 和 ⟨ 表达式 $_2$⟩ 的类型，就算是在两个表达式都已经固定了的情况下也是如此。

在创建复杂模型或者复杂输出时，条件表达式是非常有用的。我们来看下图 12中的数独问题模型。板的初始位置在参数 `start`中给出，其中 0 代表了一个空的板位置。通过使用以下条件表达式

```
constraint forall(i,j in PuzzleRange)(
     if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );
```

它被转换为对决策变量 `puzzle`的约束。

在定义复杂输出时，条件表达式也很有用。在数独模型图 12中，表达式

```
if j mod S == 0 then " " else "" endif
```

sudoku ≡ [download]

```
include "alldifferent.mzn";

int: S;
int: N = S * S;
int: digs = ceil(log(10.0,int2float(N))); % 输出的数字

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

array[1..N,1..N] of 0..N: start; %  板初始 0 = 空
array[1..N,1..N] of var PuzzleRange: puzzle;

% 填充初始板
constraint forall(i,j in PuzzleRange)(
    if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );

% 每行中取值各不相同
constraint forall (i in PuzzleRange) (
                  alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% 每列中取值各不相同
constraint forall (j in PuzzleRange) (
                  alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% 每个子方格块中取值各不相同
constraint
        forall (a, o in SubSquareRange)(
                alldifferent( [ puzzle[(a-1) *S + a1, (o-1)*S + o1] |
                                    a1, o1 in SubSquareRange ] ) );

solve satisfy;

output  [ show_int(digs,puzzle[i,j]) ++ " " ++
         if j mod S == 0 then " " else "" endif ++
         if j == N /\ i != N then
             if i mod S == 0 then "\n\n" else "\n" endif
         else "" endif
         | i,j in PuzzleRange ] ++ ["\n"];
```

图 12: 广义数独问题的模型（sudoku.mzn）。

sudoku.dzn ≡ [download]

```
S=3;
start=[|
0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 6, 8, 4, 0, 1, 0, 7, 0|
0, 0, 0, 0, 8, 5, 0, 3, 0|
0, 2, 6, 8, 0, 9, 0, 4, 0|
0, 0, 7, 0, 0, 0, 9, 0, 0|
0, 5, 0, 1, 0, 6, 3, 2, 0|
0, 4, 0, 6, 1, 0, 0, 0, 0|
0, 3, 0, 2, 0, 7, 6, 9, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0|];
```

	6	8	4		1		7	
				8	5		3	
	2	6	8		9		4	
		7				9		
	5		1		6	3	2	
	4		6	1				
	3		2		7	6	9	

图 13: 广义数独问题的数据文件例子（sudoku.dzn）以及它代表的问题。

在大小为 S 的组群之间插入了一个额外的空格。输出表达式同时也使用条件表达式来在每 S 行后面加入一个空白行。这样得到的输出有很高的可读性。

剩下的约束保证了每行中，每列中以及每 $S \times S$ 子方格块中的值都是互相不相同的。

通过使用标示 -a或 --all-solutions，我们可以用 MiniZinc 求解得到一个满足问题（solve satisfy）的所有解。

```
$ mzn-g12fd --all-solutions sudoku.mzn sudoku.dzn
```

得到

aust-enum ≡ [download]

```
enum Color;
var Color: wa;
var Color: nt;
var Color: sa;
var Color: q;
var Color: nsw;
var Color: v;
var Color: t;
constraint wa != nt /\ wa != sa /\ nt != sa /\ nt != q /\ sa != q;
constraint sa != nsw /\ sa != v /\ q != nsw /\ nsw != v;
solve satisfy;
```

图 14: 使用枚举类型的澳大利亚涂色模型（aust-enum.mzn）。

```
 5 9 3  7 6 2  8 1 4
 2 6 8  4 3 1  5 7 9
 7 1 4  9 8 5  2 3 6

 3 2 6  8 5 9  1 4 7
 1 8 7  3 2 4  9 6 5
 4 5 9  1 7 6  3 2 8

 9 4 2  6 1 8  7 5 3
 8 3 5  2 4 7  6 9 1
 6 7 1  5 9 3  4 8 2
----------
==========
```

当系统输出完所有可能的解之后，此行 `==========` 被输出。在这里则表明了此问题只有一个解。

3.4 枚举类型

枚举类型允许我们根据一个或者是数据中的一部分，或者在模型中被命名的对象集合来创建模型。这样一来，模型就更容易被理解和调试。我们之前已经简单介绍了枚举类型或者 enums。在这一小分段，我们会探索如何可以全面地使用它们，并且给出一些处理枚举类型的内建函数。

让我们重新回顾一下节 2中的给澳大利亚涂色问题。

图 14中的模型声明了一个枚举类型 Color，而它必须在数据文件中被定义。每一个州变量被声明为从此枚举类型中取一个值。使用以下方式运行这个程序

```
$ minizinc -D"Color = { red, yellow, blue };" aust-enum.mzn
```

可能会得到输出

```
wa = yellow;
nt = blue;
sa = red;
q = yellow;
nsw = blue;
v = yellow;
t = red;
```

枚举类型变量声明

一个枚举类型参数变量被声明为以下两种方式：

⟨ 枚举名⟩ : ⟨ 变量名⟩

⟨l⟩ .. ⟨u⟩ : ⟨ 变量名⟩

其中⟨ 枚举名⟩是枚举类型的名字，l和 u是此枚举类型的固定枚举类型表达式。

一个枚举类型决策变量被声明为以下两种方式：

var ⟨ 枚举名⟩ : ⟨ 变量名⟩

var ⟨l⟩ .. ⟨u⟩ : ⟨ 变量名⟩

其中⟨ 枚举名⟩是枚举类型的名字，l和 u是此枚举类型的固定枚举类型表达式。

枚举类型一个重要的行为是，当它们出现的位置所期望的是整数时，它们会自动地强制转换为整数。这样一来，这就允许我们使用定义在整数上的全局变量，例如

```
global_cardinality_low_up([wa,nt,sa,q,nsw,v,t],
                          [red,yellow,blue],[2,2,2],[2,2,3]);
```

要求每种颜色至少有两个州涂上并且有三个州被涂了蓝色。

枚举类型操作符

有一系列关于枚举类型的内部操作符：

- `enum_next`(x)：返回枚举类型中 x 后的下一个值。
- `enum_prev`(x)：返回枚举类型中 x 前的上一个值。
- `to_enum`($Enum$,i)：把一个整型表达式 i 映射到枚举类型 $Enum$ 中的一个枚举类型值。若 i 大于 $Enum$ 中元素的个数，则映射失败。

注意，一些标准函数也是可以应用于枚举类型上

- `card`($Enum$)：返回枚举类型 $Enum$ 的势。
- `min`($Enum$)：返回枚举类型 $Enum$ 中最小的元素。
- `max`($Enum$)：返回枚举类型 $Enum$ 中最大的元素。

3.5 复杂约束

约束是 MiniZinc 模型的核心。我们已经看到了简单关系表达式，但是约束其实是比这更加强大的。一个约束可以是任何布尔型表达式。想象一个包含两个时间上不能重叠的任务的调度问题。如果 `s1`和 `s2`是相对应的起始时间，`d1`和 `d2`是相对应的持续时间，我们可以表达约束为：

```
constraint s1 + d1 <= s2  \/ s2 + d2 <= s1;
```

来保证任务之间互相不会重叠。

布尔型

MiniZinc 中的布尔型表达式可以按照标准的数学语法来书写。布尔常量是真或假，布尔型操作符有合取，即，与（`/\`），析取，即，或（`\/`），必要条件蕴含（`<-`），充分条件蕴含（`->`），充分必要蕴含（`<->`）以及非（`not`）。内建函数 `bool2int`强制转换布尔型为整型：如果参数为真，它返回 1，否则返回 0。

图 15中的车间作业调度模型给出了一个使用析取建模功能的现实例子。车间作业调度问题中，我们有一个作业集合，每一个包含一系列的在不同机器上的任务：任务 $[i,j]$ 是在第 i 个作业中运行在第 j 个机器上的任务。每列任务必须按照顺序完成，并且运行在同一个机器上的任何两个任务在时间上都不能重叠。就算是对这个问题的小的实例找最优解都会是很有挑战性的。

命令

```
$ mzn-g12fd --all-solutions jobshop.mzn jobshop.dzn
```

jobshop ≡ [download]

```
enum JOB;
enum TASK;
TASK: last = max(TASK);
array [JOB,TASK] of int: d;                    %  任务持续时间
int: total = sum(i in JOB, j in TASK)(d[i,j]);%  总持续时间
int: digs = ceil(log(10.0,int2float(total))); %  输出的数值
array [JOB,TASK] of var 0..total: s;           %  起始时间
var 0..total: end;                             %  总结束时间


constraint %% 保证任务按照顺序出现
    forall(i in JOB) (
        forall(j in TASK where j < last)
            (s[i,j] + d[i,j] <= s[i,enum_next(j)]) /\
        s[i,last)] + d[i,last] <= end
    );

constraint %% 保证任务之间没有重叠
    forall(j in TASK) (
        forall(i,k in JOB where i < k) (
            s[i,j] + d[i,j] <= s[k,j] \/
            s[k,j] + d[k,j] <= s[i,j]
        )
    );

solve minimize end;

output ["end = \(end)\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == last then "\n" else "" endif |
         i in JOB, j in TASK ];
```

图 15: 车间作业调度问题模型（jobshop.mzn）。

求解了一个小的车间作业调度问题，并且显示了优化问题在 all-solutions下的表现。在这里，求解器只有当找到一个更好的解时才会输出它，而不是输出所有的可能最优解。这个命令下的（部分）输出是：

```
end = 31
 0  3  7 12 18
 6  9 19 26 28
 2 11 15 19 24
 1  2  3  4 10
 9 16 26 28 30
----------
end = 30
 1  2  6 11 17
 6 10 15 22 23
 2  6 11 15 25
 0  1  2  3  9
 9 16 22 24 29
----------
==========
```

```
end = 41
 0  1  5 10 13
 5  8 10 25 26
 1 10 17 21 28
 8 14 21 26 32
 9 16 22 32 40
----------
```

然后等相当一段时间后，得到更多的解。然后最终：

表明一个结束时间为 30 的最优解终于被找到，并且被证明为是最优的。通过加一个约束 `end = 30`，并且把求解项改为 `solve satisfy`，然后运行

```
$ mzn-g12fd --all-solutions jobshop.mzn jobshop.dzn
```

我们可以得到所有的最优解。这个问题有非常多的最优解。

MiniZinc 中的另外一个强大的建模特征是决策变量可以被用来访问数组。作为一个例子，考虑（老式的）稳定婚姻问题。我们有 n 个（直）女以及 n 个（直）男。每一个男士有一个女士排行榜，女士也是。我们想给每一个女士/男士找一个丈夫/妻子来使得所有的婚姻按以下意义上来说都是稳定的：

- 每当 m 喜欢另外一个女士 o 多过他的妻子 w 时，o 喜欢她的丈夫多过 m，以及
- 每当 w 喜欢另外一个男士 o 多过她的丈夫 m 时，o 喜欢他的妻子多过 w。

stable-marriage ≡ [download]

```
int: n;

enum Men = anon_enum(n);
enum Women = anon_enum(n);

array[Women, Men] of int: rankWomen;
array[Men, Women] of int: rankMen;

array[Men] of var Women: wife;
array[Women] of var Men: husband;

► assignment
► ranking
solve satisfy;

output ["wives= \(wife)\nhusbands= \(husband)\n"];
```

图 16: 稳定婚姻问题模型（stable-marriage.mzn）。

这个问题可以很优雅地在 MiniZinc 中建模。模型和数据例子在图 16和??中分别被给出。

模型中的前三项声明了男士/女士的数量以及男士和女士的集合。在这里我们介绍匿名枚举类型的使用。Men和 Women都是大小为 n 的集合，但是我们不希望把它们混合到一起，所以我们使用了一个匿名枚举类型。这就允许 MiniZinc 检测到使用 Men为 Women或者反之的建模错误。

矩阵 rankWomen和 rankMen分别给出了男士们的女士排行以及女士们的男士排行。因此，项 rankWomen[w,m]给出了女士 w的关于男士 m的排行。在排行中的数目越小，此男士或者女士被选择的倾向越大。

有两个决策变量的数组：wife和 husband。这两个分别代表了每个男士的妻子和每个女士的丈夫。

前两个约束

assignment ≡

```
constraint forall (m in Men) (husband[wife[m]]=m);
constraint forall (w in Women) (wife[husband[w]]=w);
```

确保了丈夫和妻子的分配是一致的：w 是 m 的妻子蕴含了 m 是 w 的丈夫，反之亦然。注意在 husband[wife[m]]中，下标表达式 wife[m]是一个决策变量，而不是一个参数。

接下来的两个约束是稳定条件的直接编码：

stable-marriage.dzn ≡ [download]

```
n = 5;
rankWomen =
 [| 1, 2, 4, 3, 5,
  | 3, 5, 1, 2, 4,
  | 5, 4, 2, 1, 3,
  | 1, 3, 5, 4, 2,
  | 4, 2, 3, 5, 1 |];

rankMen =
 [| 5, 1, 2, 4, 3,
  | 4, 1, 3, 2, 5,
  | 5, 3, 2, 4, 1,
  | 1, 5, 4, 3, 2,
  | 4, 3, 2, 1, 5 |];
```

图 17: 图 16中的稳定婚姻问题模型的数据文件例子。

ranking ≡

```
constraint forall (m in Men, o in Women) (
     rankMen[m,o] < rankMen[m,wife[m]] ->
         rankWomen[o,husband[o]] < rankWomen[o,m] );

constraint forall (w in Women, o in Men) (
     rankWomen[w,o] < rankWomen[w,husband[w]] ->
         rankMen[o,wife[o]] < rankMen[o,w] );
```

在有了用决策变量作为数组的下标和用标准的布尔型连接符构建约束的功能后，稳定婚姻问题的自然建模才变得可行。敏锐的读者可能会在这时产生疑问，如果数组下标变量取了一个超出数组下标集合的值，会产生什么情况。MiniZinc 把这种情况看做失败：一个数组访问 `a[e]`在其周围最近的布尔型语境中隐含地加入了约束 `e in index_set(a)`，其中 `index_set(a)`给出了 `a` 的下标集合。

匿名枚举类型

一个匿名枚举类型表达式有格式 `enum_anon(` n `)`，其中 n 是一个固定的整型表达式，它定义了枚举类型的大小。

除了其中的元素没有名字，匿名枚举类型和其他的枚举类型一样。当被输出时，它们根据枚举类型的名字被给定独有的名字。

例如，如下的变量声明

magic-series ≡ [download]

```
int: n;
array[0..n-1] of var 0..n: s;

constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));

solve satisfy;

output [ "s = \(s);\n" ] ;
```

图 18: 魔术串问题模型（magic-series.mzn）。

```
array[1..2] of int: a= [2,3];
var 0..2: x;
var 2..3: y;
```

约束 a[x] = y会在 $x=1 \wedge y=2$ 和 $x=2 \wedge y=3$ 时得到满足。约束 not a[x] = y会在 $x=0 \wedge y=2$，$x=0 \wedge y=3$，$x=1 \wedge y=3$ 和 $x=2 \wedge y=2$ 时得到满足。

当参数无效访问数组时，正式的 MiniZinc 语义会把此情况看成失败来确保参数和决策变量的处理方式是一致的，但是会发出警告，因为这种情况下几乎总是会有错误出现。

强制转换函数 bool2int可以被任何布尔型表达式调用。这就使得 MiniZinc 建模者可以使用所谓的高价约束。举个简单的例子，请看魔术串问题：找到一列数字 $s=[s_0,\ldots,s_{n-1}]$ 使得 s_i 是数字 i 出现在 s 的次数。一个解的例子是 $s=[1,2,1,0]$。

这个问题的一个 MiniZinc 模型在图 18中给出。bool2int的使用使得我们可以把函数 s[j]=i 满足的次数加起来。运行命令

```
$ mzn-g12fd --all-solutions magic-series.mzn -D "n=4;"
```

得到输出

```
s = [1, 2, 1, 0];
----------
s = [2, 0, 2, 0];
----------
==========
```

确切地显示出这个问题的两个解。

注意当有需要的时候，MiniZinc 会自动地强制转换布尔型为整型以及整型为浮点型。我们可以把图 18中的约束项替换为

```
constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(s[j]=i)));
```

由于 MiniZinc 系统实际上会自动地加入缺失的 `bool2int`，布尔型表达式 `s[j] = i`会被自动地强制转换为整型，所以会得到同样的结果。

强制转换

MiniZinc 中，通过使用函数 `bool2int`，我们可以把一个布尔型数值强制转换为一个整型数值。同样地，通过使用函数 `int2float`，我们也可以把一个整型数值强制转换为一个浮点型数值。被强制转换的数值的实例化和原数值一样。例如，`par bool`被强制转换为 `par int`，而 `var bool`被强制转换为 `var int`。

通过适当地在模型中加入 `bool2int`和 `int2float`，MiniZinc 会自动地强制转换布尔型表达式为整型表达式，以及整型表达式为浮点型表达式。注意通过两步转换，它也会强制转换布尔型为浮点型。

3.6 集合约束

MiniZinc 另外一个强大的建模特征是它允许包含整数的集合是决策变量：这表示当模型被评估时，求解器会查找哪些元素在集合中。

举个简单的例子，*0/1* 背包问题。这个问题是背包问题的局限版本，即我们或者选择把物品放入背包或者不放。每一个物品有一个重量和一个利润，在受限制于背包不能太满的条件下，我们想找到选取哪些物品会得到最大化利润。

很自然地，我们在 MiniZinc 中使用单个的决策变量来建模：其中 `ITEM`是可放置的物品集合。如果数组 `weight[i]`和 `profit[i]`分别给出物品 `i`的重量和利润，以及背包可以装载的最大重量是 `capacity`，则一个自然的模型在图 19中给出。

注意，关键字 `var`出现在 `set`声明之前，表明这个集合本身是决策变量。这就和一个 `var`关键字描述其中元素而不是数组自身的数组形成对比，因为此时数组的基本结构，即它的下标集合，是固定了的。

我们来看一个更复杂的关于集合约束的例子，图 20中给出的高尔夫联谊问题。这个问题的目的是给 `groups` × `size`个高尔夫手在 `weeks`时间内安排一个高尔夫联赛。每一周我们需要安排 `groups`个大小为 `size`的不同的组。任何一对高尔夫手都不能一起出现于两个组中进行比赛。

模型中的变量是第 i 周第 j 组的高尔夫手 `Sched[i,j]`组成的集合。

图 21中的约束首先对每一周的第一个集合进行一个排序来去除掉周之间可以互相调换的对称。然后它对每一周内的集合进行了一个排序，同时使得每一个集合的势为 `size`。接下来通过使用全局约束 `partition_set`，确保了每一周都是对高尔夫手集合的一个划分。最后一个约束确保了任何两个高尔夫手都不会一起在两个组内比赛（因为任何两个组的交集的势最多都是 1）。

knapsack ≡ [download]

```
enum ITEM;
int: capacity;

array[ITEM] of int: profits;
array[ITEM] of int: weights;

var set of ITEM: knapsack;

constraint sum (i in knapsack) (weights[i]) <= capacity;

solve maximize sum (i in knapsack) (profits[i]) ;

output ["knapsack = \(knapsack)\n"];
```

图 19: 0/1 背包问题模型（knapsack.mzn）。

在图 22中，我们也给出了去对称初始化约束：第一周被固定为所有的高尔夫手都按顺序排列；第二周的第一组被规定为是由第一周的前几组的第一个选手组成；最后，对于剩下的周，模型规定第一个 size内的高尔夫手们出现在他们相对应的组数中。

运行命令

```
$ mzn-g12fd social-golfers.mzn social-golfers.dzn
```

其中数据文件定义了一个周数为 4，大小为 3，组数为 4 的问题，得到如下结果

```
1..3 4..6 7..9 10..12
{ 1, 4, 7 } { 2, 5, 10 } { 3, 9, 11 } { 6, 8, 12 }
{ 1, 5, 8 } { 2, 6, 11 } { 3, 7, 12 } { 4, 9, 10 }
{ 1, 6, 9 } { 2, 4, 12 } { 3, 8, 10 } { 5, 7, 11 }
----------
```

注意范围集合是如何以范围格式输出的。

3.7 汇总

我们以一个可以阐释这一章介绍的大部分特征的复杂例子来结束这一节，包括枚举类型，复杂约束，全局约束以及复杂输出。

图 23中的模型安排婚礼桌上的座位。这个桌子有 12 个编码的顺序排列的座位，每边有 6 个。男士必须坐奇数号码的座位，女士坐偶数。Ed 由于恐惧症不能坐在桌子的边缘，新郎和新

social-golfers ≡ [download]

```
include "partition_set.mzn";
int: weeks;    set of int: WEEK = 1..weeks;
int: groups;   set of int: GROUP = 1..groups;
int: size;     set of int: SIZE = 1..size;
int: ngolfers = groups*size;
set of int: GOLFER = 1..ngolfers;

array[WEEK,GROUP] of var set of GOLFER: Sched;

▶ constraints
▶ symmetry

solve satisfy;

output [ show(Sched[i,j]) ++ " " ++
         if j == groups then "\n" else "" endif |
         i in WEEK, j in GROUP ];
```

图 20: 高尔夫联谊问题模型（social-golfers.mzn）。

娘必须坐在彼此旁边。我们的目的是最大化已知的互相憎恶的人之间的距离。如果在同一边，座位之间的距离是座位号码之间的差，否则则是和其对面座位的距离 + 1。

注意在输出语句中我们观察每个座位 s 来找一个客人 g 分配给此座位。我们利用内建函数 fix，它检查一个决策变量是否是固定的以及输出它的固定值，否则的话中断。在输出语句中使用此函数总是安全的，因为当输出语句被运行的时候，所有的决策变量都应该是固定了的。

运行

```
$ mzn-g12fd wedding.mzn
```

得到输出

```
ted bride groom rona bob carol ron alice ed bridesmaid bestman clara
----------

==========
```

最终得到的座位安排在图 24中给出。其中连线表示互相憎恶，总的距离是 22.

固定

输出项中，内建函数 fix检查一个决策变量的值是否固定，然后把决策变量的实例化强制转换为参数。

```
constraints ≡
  constraint
      forall (i in 1..weeks-1) (
                 Sched[i,1] < Sched[i+1,1]
      ) /\
      forall (i in WEEK, j in GROUP) (
             card(Sched[i,j]) = size
          /\ forall (k in j+1..groups) (
                   Sched[i,j] < Sched[i,k]
                /\ Sched[i,j] intersect Sched[i,k] = {}
             )
      ) /\
      forall (i in WEEK) (
            partition_set([Sched[i,j] | j in GROUP], GOLFER)
         /\ forall (j in 1..groups-1) (
                 Sched[i,j] < Sched[i,j+1]
            )
      ) /\
      forall (i in 1..weeks-1, j in i+1..weeks) (
          forall (x,y in GROUP) (
              card(Sched[i,x] intersect Sched[j,y]) <= 1
          )
      );
```

图 21: 高尔夫联谊问题的约束。

```
symmetry ≡
  constraint
    % 固定第一周 %
    forall (i in GROUP, j in SIZE) (
        ((i-1)*size + j) in Sched[1,i]
    ) /\
    % 固定第二周的第一组 %
    forall (i in SIZE) (
        ((i-1)*size + 1) in Sched[2,1]
    ) /\
    % 固定第一个‘size’内的高尔夫手 %
    forall (w in 2..weeks, p in SIZE) (
        p in Sched[w,p]
    );
```

图 22: 高尔夫联谊问题的去对称约束。

wedding ≡ [download]

```
enum Guests = { bride, groom, bestman, bridesmaid, bob, carol,
  ted, alice, ron, rona, ed, clara};
set of int: Seats = 1..12;
set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];
set of Guests: Males = {groom, bestman, bob, ted, ron,ed};
set of Guests: Females = {bride,bridesmaid,carol,alice,rona,clara};

array[Guests] of var Seats: pos;  %  客人的座位
array[Hatreds] of var Seats: p1;  %  互相憎恶的客人 1的座位
array[Hatreds] of var Seats: p2;  %  互相憎恶的客人 2的座位
array[Hatreds] of var 0..1: sameside;  %  互相憎恶的客人是否坐在同一
边
array[Hatreds] of var Seats: cost;  %  互相憎恶的客人的距离

include "alldifferent.mzn";
constraint alldifferent(pos);
constraint forall(g in Males)( pos[g] mod 2 == 1 );
constraint forall(g in Females)( pos[g] mod 2 == 0 );
constraint not (pos[ed] in {1,6,7,12});
constraint abs(pos[bride] - pos[groom]) <= 1 /\
           (pos[bride] <= 6 <-> pos[groom] <= 6);
constraint forall(h in Hatreds)(
              p1[h] = pos[h1[h]] /\
              p2[h] = pos[h2[h]] /\
              sameside[h] = bool2int(p1[h] <= 6 <-> p2[h] <= 6) /\
              cost[h] = sameside[h] * abs(p1[h] - p2[h]) +
                 (1 - sameside[h]) * (abs(13 - p1[h] - p2[h]) + 1) );

solve maximize sum(h in Hatreds)(cost[h]);

output [ show(g)++" " | s in Seats,g in Guests where fix(pos[g]) == s]
       ++ ["\n"];
```

图 23: 使用枚举类型规划婚礼座位（wedding.mzn）

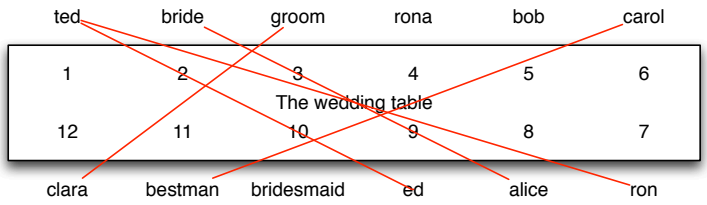


图 24: 婚礼桌上座位的安排

4 谓词和函数

MiniZinc 中的谓词允许我们用简洁的方法来表达模型中的复杂约束。MiniZinc 中的谓词利用预先定义好的全局约束建模，同时也让建模者获取以及定义新的复杂约束。MiniZinc 中的函数用来捕捉模型中的共同结构。实际上，一个谓词就是一个输出类型为 `var bool`的函数。

4.1 全局约束

MiniZinc 中定义了很多可以在建模中使用的全局约束。由于全局约束的列表一直在慢慢增加，最终确定的列表可以在发布的文档中找到。下面我们讨论一些最重要的全局约束。

4.1.1 Alldifferent

约束 `alldifferent`的输入为一个变量数组，它约束了这些变量取不同的值。`alldifferent`的使用有以下格式

```
alldifferent(array[int] of var int: x)
```

即，参数是一个整型变量数组。

Alldifferent 是约束规划中被最多研究以及使用的全局约束之一。它被用来定义分配子问题，人们也给出了 `alldifferent`的高效全局传播器。`send-more-money.mzn` (图 11) 和 `sudoku.mzn` (图 12) 是使用 `alldifferent`的模型例子。

4.1.2 Cumulative

约束 `cumulative`被用来描述资源累积使用情况。

```
cumulative(array[int] of var int: s, array[int] of var int: d,
           array[int] of var int: r, var int: b)
```

规定对于一个起始时间为 s，持续时间为 d 以及资源需求量为 r 的任务集合，在任何时间对资源的需求量都不能超过一个全局资源量界限 b。

图 25中的模型为搬运家具规划一个行程表使得每一份家具在搬运的过程中都有足够的搬用工和足够的手推车可以使用。允许的时间，可以使用的搬运工以及手推车被给出，每个物体的搬运持续时间，需要的搬运工和手推车的数量等数据也被给出。使用图 26中的数据，命令

```
$ mzn-g12fd moving.mzn moving.dzn
```

可能会得到如下输出

moving ≡ [download]

```
include "cumulative.mzn";

enum OBJECTS;
array[OBJECTS] of int: duration; %  移动的持续时间
array[OBJECTS] of int: handlers; %  需要的搬运工的数量
array[OBJECTS] of int: trolleys; %  需要的手推车的数量

int: available_handlers;
int: available_trolleys;
int: available_time;

array[OBJECTS] of var 0..available_time: start;
var 0..available_time: end;

constraint cumulative(start, duration, handlers, available_handlers);
constraint cumulative(start, duration, trolleys, available_trolleys);

constraint forall(o in OBJECTS)(start[o] +duration[o] <= end);

solve minimize end;

output [ "start = \(start)\nend = \(end)\n"];
```

图 25: 使用 cumulative来建模搬运家具问题的模型（moving.mzn）。

moving.dzn ≡ [download]

```
OBJECTS = { piano, fridge, doublebed, singlebed,
            wardrobe, chair1, chair2, table };

duration = [60, 45, 30, 30, 20, 15, 15, 15];
handlers = [3, 2, 2, 1, 2, 1, 1, 2];
trolleys = [2, 1, 2, 2, 2, 0, 0, 1];

available_time = 180;
available_handlers = 4;
available_trolleys = 3;
```

图 26: 使用 cumulative来建模搬运家具问题的数据（moving.dzn）。

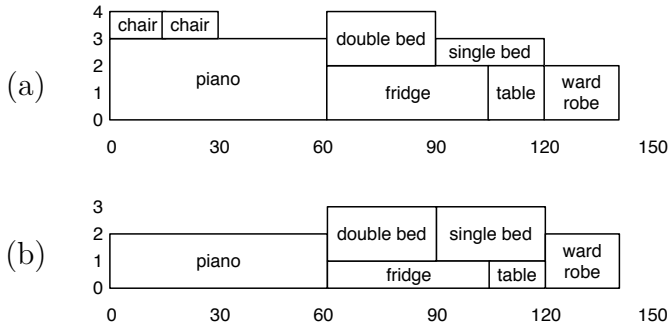


图 27: 搬运时 (a) 搬运工和 (b) 手推车使用量直方图。

```
start = [0, 60, 60, 90, 120, 0, 15, 105]
end = 140
----------
==========
```

图 27(a) and 图 27(b) 给出了这个解中搬运时每个时间点所需要的搬运工和手推车。

4.1.3 Table

约束 `table`强制变量元组从一个元组集合中取值。由于 MiniZinc 中没有元组，我们用数组来描述它。根据元组是布尔型还是整型，`table`的使用有以下两种格式

```
table(array[int] of var bool: x, array[int, int] of bool: t)
table(array[int] of var int:  x, array[int, int] of int:  t)
```

强制约束了 $x \in t$，其中 x 和 t 中的每一行是元组，t 是一个元组集合。

图 28中的模型寻找均衡的膳食。每一个食物项都有一个名字（用整数表示），卡路里数，蛋白质克数，盐毫克数，脂肪克数以及单位为分的价钱。这些个项之间的关系用一个 `table`约束来描述。模型寻找拥有最小花费，最少卡路里数 min_energy，最少蛋白质量 $min\_protein$，最大盐分 max_salt 以及脂肪 max_fat 的膳食。

4.1.4 Regular

约束 `regular`用来约束一系列的变量取有限自动机定义的值。`Regular`的使用有以下方式

```
regular(array[int] of var int: x, int: Q, int: S,
        array[int,int] of int: d, int: q0, set of int: F)
```

meal ≡ [download]

```
% 规划均衡的膳食
include "table.mzn";
int: min_energy;
int: min_protein;
int: max_salt;
int: max_fat;
set of FOOD: desserts;
set of FOOD: mains;
set of FOOD: sides;
enum FEATURE = { name, energy, protein, salt, fat, cost};
enum FOOD;
array[FOOD,FEATURE] of int: dd; %  食物数据库

array[FEATURE] of var int: main;
array[FEATURE] of var int: side;
array[FEATURE] of var int: dessert;
var int: budget;

constraint main[name] in mains;
constraint side[name] in sides;
constraint dessert[name] in desserts;
constraint table(main, dd);
constraint table(side, dd);
constraint table(dessert, dd);
constraint main[energy] + side[energy] + dessert[energy] >=min_energy;
constraint main[protein]+side[protein]+dessert[protein] >=min_protein;
constraint main[salt] + side[salt] + dessert[salt] <= max_salt;
constraint main[fat] + side[fat] + dessert[fat] <= max_fat;
constraint budget = main[cost] + side[cost] + dessert[cost];

solve minimize budget;

output ["main = ",show(to_enum(FOOD,main[name])),",
        ", side = ",show(to_enum(FOOD,side[name])),
        ", dessert = ",show(to_enum(FOOD,dessert[name]))],
        ", cost = ",show(budget), "\n"];
```

图 28: 使用 table约束来建模食物规划问题的模型（meal.mzn）。

meal.dzn ≡ [download]

```
FOODS = { icecream, banana, chocolatecake, lasagna,
          steak, rice, chips, brocolli, beans} ;

dd = [| icecream,      1200,  50,  10, 120,  400     %  冰淇淋
      | banana,         800, 120,   5,  20,  120     %  香蕉
      | chocolatecake, 2500, 400,  20, 100,  600     %  巧克力蛋糕
      | lasagna,       3000, 200, 100, 250,  450     %  千层面
      | steak,         1800, 800,  50, 100, 1200     %  牛排
      | rice,          1200,  50,   5,  20,  100     %  米饭
      | chips,         2000,  50, 200, 200,  250     %  薯条
      | brocolli,       700, 100,  10,  10,  125     %  花椰菜
      | beans,         1900, 250,  60,  90,  150 |]; %  黄豆

min_energy = 3300;
min_protein = 500;
max_salt = 180;
max_fat = 320;
desserts = { icecream, banana, chocolotecake };
mains = { lasagna, steak, rice };
sides = { chips, brocolli, beans };
```

图 29: 定义 table的食物规划的数据（meal.dzn）。

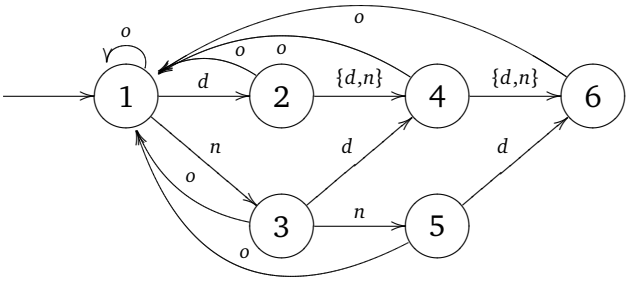


图 30: 判定正确排班的 DFA。

它约束了 x 中的一列值（它们必须是在范围 $1..S$ 内）被一个有 Q 个状态，输入为 $1..S$，转换函数为 d（$< 1..Q, 1..S >$ 映射到 $0..Q$），初始状态为 $q0$（必须在 $1..Q$ 中）和接受状态为 F（必须在 $1..Q$ 中）的 DFA接受。状态 0 被保留为总是会失败的状态。

我们来看下护士排班问题。每一个护士每一天被安排为以下其中一种：(d) 白班 (n) 夜班或者 (o) 休息。每四天，护士必须有至少一天的休息。每个护士都不可以被安排为连续三天夜班。这个问题可以使用 图 30中的不完全 DFA 来表示。我们可以把这个 DFA 表示为初始状态是 1，

结束状态是 1 .. 6，转换函数为

	d	n	o
1	2	3	1
2	4	4	1
3	4	5	1
4	6	6	1
5	6	0	1
6	0	0	1

注意状态表中的状态 0 代表一个错误状态。图 31中给出的模型为 num_nurses 个护士 num_days 天寻找一个排班，其中我们要求白天有 req_day 个护士值班，晚上有 req_night 个护士值班，以及每个护士至少有 min_night 个夜班。

运行命令

```
$ mzn-g12fd nurse.mzn nurse.dzn
```

找到一个给 7 个护士 10 天的排班，要求白天有 3 个人值班，夜晚有 2 个人值班，以及每个护士最少有 2 个夜班。一个可能的结果是

```
o d n n o n n d o o
d o n d o d n n o n
o d d o d o d n n o
d d d o n n d o n n
d o d n n o d o d d
n n o d d d o d d d
n n o d d d o d d d
----------
```

另外一种 regular 约束是 `regular_nfa`。它使用 NFA（没有 ϵ 弧）来定义 regular 表达式。此约束有以下格式

```
regular_nfa(array[int] of var int: x, int: Q, int: S,
        array[int,int] of set of int: d, int: q0, set of int: F)
```

它约束了数组 x 中的数值序列（必须在范围 1..S 中）被含有 Q 个状态，输入为 1..S，转换函数为 d（映射 $<1..Q, 1..S>$ 到 1..Q 的子集），初始状态为 $q0$（必须在范围 1..Q 中）以及接受状态为 F （必须在范围 1..Q 中）的 NFA接受。在这里，我们没必要再给出失败状态 0，因为转换函数可以映射到一个状态的空集。

nurse ≡ [download]

```
% 简单护士排班问题
include "regular.mzn";
enum NURSE;
enum DAY;
int: req_day;
int: req_night;
int: min_night;

enum SHIFT = { d, n, o };
int: S = card(SHIFT);

int: Q = 6; int: q0 = 1; set of int: STATE = 1..Q;
array[STATE,SHIFT] of int: t =
     [| 2, 3, 1    %  状态 1
      | 4, 4, 1    %  状态 2
      | 4, 5, 1    %  状态 3
      | 6, 6, 1    %  状态 4
      | 6, 0, 1    %  状态 5
      | 0, 0, 1|]; %  状态 6

array[NURSE,DAY] of var SHIFT: roster;

constraint forall(j in DAY)(
             sum(i in NURSE)(roster[i,j] == d) == req_day /\
             sum(i in NURSE)(roster[i,j] == n) == req_night
           );
constraint forall(i in NURSE)(
             regular([roster[i,j] | j in DAY], Q, S, t, q0, STATE) /\
             sum(j in DAY)(roster[i,j] == n) >= min_night
           );

solve satisfy;

output [ show(roster[i,j]) ++ if j==card(DAY) then "\n" else " " endif
       | i in NURSE, j in DAY ];
```

图 31: 使用 regular约束来建模的护士排班问题模型（nurse.mzn）。

4.2 定义谓词

MiniZinc 的其中一个最强大的建模特征是建模者可以定义他们自己的高级约束。这就使得他们可以对模型进行抽象化和模块化。也允许了在不同的模型之间重新利用约束以及促使了用来定义标准约束和类型的特殊库应用的发展。

我们用一个简单的例子开始，回顾下前面章节中的车间作业调度问题。这个模型在图 32中给出。我们感兴趣的项是谓词项：

nooverlap ≡

```
predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;
```

它定义了一个新的约束用来约束起始时间为 s1，持续时间为 d1的任务不能和起始时间为 s2，持续时间为 d2的任务重叠。它可以在模型的任何（包含决策变量的）布尔型表达式 可以出现的地方使用。

和谓词一样，建模者也可以定义只涉及到参数的新的约束。和谓词不一样的是，它们可以被用在条件表达式的测试中。它们被关键字 test定义。例如

```
test even(int:x) = x mod 2 = 0;
```

jobshop2 ≡ [download]

```
int: jobs;                                          %  作业的数量
set of int: JOB = 1..jobs;
int: tasks;                                         %  每个作业的任务数量
set of int: TASK = 1..tasks;
array [JOB,TASK] of int: d;                         %  任务持续时间
int: total = sum(i in JOB, j in TASK)(d[i,j]);%  总持续时间
int: digs = ceil(log(10.0,total));                  %  输出的数值

array [JOB,TASK] of var 0..total: s;                %  起始时间
var 0..total: end;                                  %  总结束时间

► nooverlap

constraint %% 保证任务按照顺序出现
    forall(i in JOB) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );

constraint %% 保证任务之间没有重叠
    forall(j in TASK) (
        forall(i,k in JOB where i < k) (
            no_overlap(s[i,j], d[i,j], s[k,j], d[k,j])
        )
    );

solve minimize end;

output ["end = \(end)\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in JOB, j in TASK ];
```

图 32: 使用谓词的车间作业调度问题模型（jobshop2.mzn）。

谓词定义

使用以下形式的语句，我们可以定义谓词

```
predicate ⟨ 谓词名⟩ ( ⟨ 参数定义⟩, ..., ⟨ 参数定义⟩ ) = ⟨ 布尔表达式⟩
```

⟨ 谓词名⟩ 必须是一个合法的 MiniZinc 标识符，每个 ⟨ 参数定义⟩ 都是一个合法的 MiniZinc 类型 声明。

参数定义的一个松弛是数组的索引类型可以是没有限制地 写为 `int`。

类似的，使用以下形式的语句，我们定义测试

```
test ⟨ 谓词名⟩ ( ⟨ 参数定义⟩, ..., ⟨ 参数定义⟩ ) = ⟨ 布尔表达式⟩
```

其中的 ⟨ 布尔表达式⟩ 必须是固定的。

另外我们介绍一个谓词中使用到的 `assert`命令的新形式。

```
assert ( ⟨ 布尔表达式⟩, ⟨ 字符串表达式⟩, ⟨ 表达式⟩ )
```

`assert`表达式的类型和最后一个参数的类型一样。`assert`表达式检测第一个参数是否为假，如果是则输出第二个参数字符串。如果第一个参数是真，则输出第三个参数。

注意 assert 表达式中的第三个参数是延迟的，即如果第一个参数是假，它就不会被评估。所以它可以被用来检查

```
predicate lookup(array[int] of var int:x, int: i, var int: y) =
    assert(i in index_set(x), "index out of range in lookup",
           y = x[i]
    );
```

此代码在 i 超出数组 x 的范围时不会计算 $x[i]$。

4.3 定义函数

MiniZinc 中的函数和谓词一样定义，但是它有一个更一般的返回类型。

下面的函数定义了一个数独矩阵中的第 a 个子方块的第 $a1$ 行。

```
function int: posn(int: a, int: a1) = (a-1) * S + a1;
```

有了这个定义之后，我们可以把图 12中的数独问题的最后一个约束替换为

```
constraint forall(a, o in SubSquareRange)(
                  alldifferent([ puzzle [ posn(a,a0), posn(o,o1) ] |
                                         a1,o1 in SubSquareRange ] ) );
```

函数对于描述模型中经常用到的复杂表达式非常有用。例如，想象下在 $n \times n$ 的方格的不同位置上放置数字 1 到 n 使得任何两个数字 i 和 j 之间的曼哈顿距离比这两个数字其中最大的值减一还要大。我们的目的是最小化数组对之间的总的曼哈顿距离。曼哈顿距离函数可以表达为：

manf ≡
```
function var int: manhattan(var int: x1, var int: y1,
                            var int: x2, var int: y2) =
         abs(x1 - x2) + abs(y1 - y2);
```

完整的模型在图 33中给出。

函数定义

函数用以下格式的语句定义

`function` ⟨ 返回类型⟩ : ⟨ 函数名⟩ (⟨ 参数定义⟩, ..., ⟨ 参数定义⟩) = ⟨ 表达式⟩

⟨ 函数名⟩ 必须是一个合法的 MiniZinc 标识符。每一个 ⟨ 参数定义⟩ 是一个合法的 MiniZinc 类型声明。⟨ 返回类型⟩ 是函数的返回类型，它必须是 ⟨ 表达式⟩ 的类型。参数和谓词定义中的参数有一样的限制。

MiniZinc 中的函数可以有任何返回类型，而不只是固定的返回类型。在定义和记录多次出现在模型中的复杂表达式时，函数是非常有用的。

4.4 反射函数

为了方便写出一般性的测试和谓词，各种反射函数会返回数组的下标集合，var 集合的定义域以及决策变量范围的信息。关于下标集合的有以下反射函数 `index_set(`⟨*1-D array*⟩`)`，`index_set_1of2(`⟨*2-D array*⟩`)`，`index_set_2of2(`⟨*2-D array*⟩`)` 以及关于更高维数组的反射函数。

车间作业问题的一个更好的模型是把所有的对于同一个机器上的不重叠约束结合为一个单个的析取约束。这个方法的一个优点是虽然我们只是初始地把它建模成一个不重叠部分的结合，但是如果下层的求解器对于解决析取约束有一个更好的方法，在对我们的模型最小改变的情况下，我们可以直接使用它。这个模型在 图 34中给出。

约束 `disjunctive` 获取每个任务的开始时间数组以及它们的持续时间数组，确保每次只有一个任务是被激活的。我们定义析取约束为一个有以下特征的谓词

```
predicate disjunctive(array[int] of var int:s, array[int] of int:d);
```

在图 34中，我们可以用这个析取约束定义任务之间不重叠。我们假设 `disjunctive`谓词的定义已经在模型中引用的文件 `disjunctive.mzn`中给出。如果下层的系统直接支持 `disjunctive`，则会在它的全局目录下包含一个 `disjunctive.mzn`文件（拥有上述特征定义内

manhattan ≡ [download]

```
int: n;
set of int: NUM = 1..n;

array[NUM] of var NUM: x;
array[NUM] of var NUM: y;
array[NUM,NUM] of var 0..2*n-2: dist =
     array2d(NUM,NUM,[
     if i < j then manhattan(x[i],y[i],x[j],y[j]) else 0 endif
                    | i,j in NUM ]);

► manf

constraint forall(i,j in NUM where i < j)
                 (dist[i,j] >= max(i,j)-1);

var int: obj = sum(i,j in NUM where i < j)(dist[i,j]);
solve minimize obj;

% 简单地显示结果
include "alldifferent_except_0.mzn";
array[NUM,NUM] of var 0..n: grid;
constraint forall(i in NUM)(grid[x[i],y[i]] = i);
constraint alldifferent_except_0([grid[i,j] | i,j in NUM]);

output ["obj = \(obj);\n"] ++
       [ if fix(grid[i,j]) > 0 then show(grid[i,j]) else "." endif
         ++ if j = n  then "\n" else "" endif
       | i,j in NUM ];
```

图 33: 阐释如何使用函数的数字放置问题模型（manhattan.mzn）。

容）。如果我们使用的系统不直接支持析取，通过创建文件 disjunctive.mzn，我们可以给出我们自己的定义。最简单的实现是单单使用上面定义的不重叠谓词。一个更好的实现是利用全局约束 cumulative，假如下层求解器支持它的话。图 35给出了一个 disjunctive的实现。注意我们使用 index_set反射函数来（a）检查 disjunctive的参数是有意义的，以及（b）构建 cumulative的合适大小的资源利用数组。另外注意这里我们使用了 assert的三元组版本。

jobshop3 ≡ [download]

```
include "disjunctive.mzn";

int: jobs;                                          %  作业的数量
set of int: JOB = 1..jobs;
int: tasks;                                         %  每个作业的任务数量
set of int: TASK = 1..tasks;
array [JOB,TASK] of int: d;                         %  任务持续时间
int: total = sum(i in JOB, j in TASK)(d[i,j]);%  总持续时间
int: digs = ceil(log(10.0,total));                  %  输出的数值
array [JOB,TASK] of var 0..total: s;                %  起始时间
var 0..total: end;                                  %  总结束时间

constraint %% 保证任务按照顺序出现
    forall(i in JOB) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );

constraint %% 保证任务之间没有重叠
    forall(j in TASK) (
        disjunctive([s[i,j] | i in JOB], [d[i,j] | i in JOB])
    );

solve minimize end;

output ["end = \(end)\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in JOB, j in TASK ];
```

图 34: 使用 disjunctive谓词的车间作业调度问题模型（jobshop3.mzn）。

4.5 局部变量

在谓词，函数或者测试中，引进局部变量总是非常有用的。表达式 let允许你去这样做。它可以被用来引进决策变量和参数，但是参数必须被初始化。例如：

```
var s..e: x;
let {int: l = s div 2; int: u = e div 2; var l .. u: y;} in x = 2*y
```

disjunctive ≡ [download]

```
include "cumulative.mzn";

predicate disjunctive(array[int] of var int:s, array[int] of int:d) =
          assert(index_set(s) == index_set(d), "disjunctive: " ++
          "first and second arguments must have the same index set",
            cumulative(s, d, [ 1 | i in index_set(s) ], 1)
          );
```

图 35: 使用 cumulative来定义一个 disjunctive谓词（disjunctive.mzn）。

引进了参数 l 和 u 以及变量 y。Let 表达式虽然在谓词，函数和测试定义中最有用，它也可以被用在其他的表达式中。例如，来消除共同的子表达式：

```
constraint let { var int: s = x1 + x2 + x3 + x4 } in
           l <= s /\ s <= u;
```

局部变量可以被用在任何地方，在简化复杂表达式时也很有用。通过使用局部变量来定义目标函数而不是显式地加入很多个变量，图 36给出了稳定婚姻模型的一个改进版本。

一个局限是谓词和函数包含的决策变量如果没有在声明的时候初始化，则不能被用在否定语境下 。下面的例子是不合法的

```
predicate even(var int:x) =
          let { var int: y } in x = 2 * y;

constraint not even(z);
```

如果局部变量被赋了值，则可以用在否定语境中。下面的例子是合法的

```
predicate even(var int:x) =
          let { var int: y = x div 2; } in x = 2 * y;

constraint not even(z);
```

注意 even的意思是正确的，因为如果 x 是偶数，则 $x = 2 * (x \text{ div } 2)$。

4.6 语境

有一个局限，即含有决策变量并且在声明时没有初始化的谓词和函数不可以被用在一个否定语境下。下面例子是非法的

wedding2 ≡ [download]

```
enum Guests = { bride, groom, bestman, bridesmaid, bob, carol,
  ted, alice, ron, rona, ed, clara};
set of int: Seats = 1..12;
set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];
set of Guests: Males = {groom, bestman, bob, ted, ron,ed};
set of Guests: Females = {bride,bridesmaid,carol,alice,rona,clara};

array[Guests] of var Seats: pos; %  客人的座位

include "alldifferent.mzn";
constraint alldifferent(pos);

constraint forall(g in Males)( pos[g] mod 2 == 1 );
constraint forall(g in Females)( pos[g] mod 2 == 0 );

constraint not (pos[ed] in {1,6,7,12});
constraint abs(pos[bride] - pos[groom]) <= 1 /\
           (pos[bride] <= 6 <-> pos[groom] <= 6);

solve maximize sum(h in Hatreds)(
          let {  var Seats: p1 = pos[h1[h]];
                 var Seats: p2 = pos[h2[h]];
                 var 0..1: same = bool2int(p1 <= 6 <-> p2 <= 6); } in
          same * abs(p1 - p2) + (1-same) * (abs(13 - p1 - p2) + 1));

output [ show(g)++" " | s in Seats,g in Guests where fix(pos[g]) == s]
       ++ ["\n"];
```

图 36: 使用局部变量来定义一个复杂的目标函数（wedding2.mzn）。

```
predicate even(var int:x) =
          let { var int: y } in x = 2 * y;


constraint not even(z);
```

原因是求解器只解决存在约束的问题。如果我们在否定语境下引入了一个局部变量，则此变量是普遍地量化了，因此超出下层求解器的解决范围。例如，$\neg even(z)$ 等价于 $\neg\exists y.z = 2y$，然后等价于 $\forall y.z \neq 2y$。

如果局部变量被赋了值，则它们可以被用在否定语境中。下面的例子是合法的

```
predicate even(var int:x) =
          let { var int: y = x div 2; } in x = 2 * y;

constraint not even(z);
```

注意，现在 even的意思是正确的，因为如果 x 是偶数，则 $x = 2*(x \text{ div } 2)$。由于 y 被 z 功能性定义了，$\neg even(z)$ 等价于 $\neg\exists y.y = z\ div\ 2 \wedge z = 2y$，同时等价于 $\exists y.y = z\ div\ 2 \wedge \neg z \neq 2y$。

MiniZinc 中的任意表达式都出现在以下四种语境中的一种中：根，肯定，否定，或者混合。非布尔型表达式的语境直接地为包含其最近的布尔型表达式的语境。唯一的例外是目标表达式出现在一个根语境下（由于它没有包含其的布尔型表达式）。

为了方便定义语境，我们把蕴含表达式 e -> e' 等价地写为 not e \/ e'，e <- e' 等价地写为 e -> not e'。

一个布尔型表达式的语境可能有：

根 根语境是任何作为 constraint的参数或者作为一个赋值项出现的表达式 e 的语境，或者作为一个出现在根语境中的 e /\ e' 的子表达式 e 或 e' 的语境。

根语境下的布尔型表达式必须在问题的任何模型中都满足。

肯定 肯定语境是任何作为一个出现在根语境或者肯定语境中的 e \/ e' 的子表达式 e 或 e' 的语境，或者是作为一个出现在肯定语境中的 e /\ e' 的子表达式 e 或 e' 的语境，或者是作为一个出现在否定语境中的 not e 的子表达式 e 的语境。

肯定语境下的布尔型表达式不是必须要在模型中满足，但是满足它们会增加包含其的约束被满足的可能性。对于一个肯定语境下的表达式，从包含其的根语境到此表达式有偶数个否定。

否定 否定语境是任何作为一个出现在根语境或者否定语境中的 e \/ e' 或 e /\ e' 的子表达式 e 或 e'，或者是作为一个出现在肯定语境中的 not e 的子表达式 e 的语境。

否定语境下的布尔型表达式不是必须要满足，但是让它们成假会增加包含其的约束被满足的可能性。对于一个否定语境下的表达式，从包含其的根语境到此表达式有奇数个否定。

混合 混合语境是任何作为一个出现在 e <-> e'，e = e' 或者 bool2int(e) 中的子表达式 e 或 e' 的语境。

混合语境下的表达式实际上既是肯定也是否定的。通过以下可以看出：$e \leftrightarrow e'$ 等价于 $(e \wedge e') \vee (\neg e \vee \neq e')$ 以及 bool2int(e) 等价于 $(e \wedge x = 1) \vee (\neg e \wedge x = 0)$。

观察以下代码段

```
constraint x > 0 /\ (i <= 4 -> x + bool2int(x > i) = 5);
```

其中 `x > 0`在根语境中，`i >= 4`在否定语境中，`x + bool2int(b) = 5`在肯定语境中，`x > i`在混合语境中。

4.7 局部约束

Let 表达式也可以被用来引入局部约束，通常用来约束局部变量的行为。例如，考虑只利用乘法来定义开根号函数：

```
function var float: mysqrt(var float:x) =
         let { var float: y;
               constraint y >= 0;
               constraint x = y * y; } in y;
```

局部约束确保了 `y`取正确的值；而此值则会被函数返回。

局部约束可以在 let 表达式中使用，尽管最普遍的应用是在定义函数时。

Let 表达式

局部变量可以在任何以下格式的 *let 表达式*中引入：

`let {⟨声明⟩; ...⟨声明⟩; } in ⟨表达式⟩`

⟨声明⟩可以是决策变量或者参数（此时必须被初始化）或者约束项的声明。任何声明都不能在一个新的声明变量还没有引进时使用它。

注意局部变量和约束不可以出现在测试中。局部变量不可以出现在否定语境下的谓词和函数中，除非这个变量是用表达式定义的。

4.8 定义域反射函数

其他重要的反射函数有允许我们对变量定义域进行访问的函数。表达式 `lb(x)` 返回一个小于等于 x 在一个问题的解中可能取的值的数值。通常它会是 x 声明的下限。如果 x 被声明为一个非有限类型，例如，只是 `var int`，则它是错误的。类似地，表达式 `dom(x)` 返回一个 x 在问题的任何解中的可能值的（非严格）超集。再次，它通常是声明的值，如果它不是被声明为有限则会出现错误。

例如，图 37中的模型或者输出

```
y = -10
D = -10..10
----------
```

或

reflection ≡ [download]

```
var -10..10: x;
constraint x in 0..4;
int: y = lb(x);
set of int: D = dom(x);
solve satisfy;
output ["y = ", show(y), "\nD = ", show(D), "\n"];
```

图 37: 使用反射谓词（reflection.mzn ）

```
y = 0
D = {0, 1, 2, 3, 4}
----------
```

或任何满足 $-10 \leq y \leq 0$ 和 $\{0,\ldots,4\} \subseteq D \subseteq \{-10,\ldots,10\}$ 的答案。

变量定义域反射表达式应该以在任何安全近似下都正确的的方式使用。但是注意这个是没有被检查的！例如加入额外的代码

```
var -10..10: z;
constraint z <= y;
```

不是一个定义域信息的正确应用。因为使用更紧密（正确的）近似会比使用更弱的初始近似产生更多的解。

定义域反射

我们有查询包含变量的表达式的可能值的反射函数：

- dom (⟨ 表达式⟩): 返回 ⟨ 表达式⟩ 所有可能值的安全近似。
- lb (⟨ 表达式⟩): 返回 ⟨ 表达式⟩ 下限值的安全近似。
- ub (⟨ 表达式⟩): 返回 ⟨ 表达式⟩ 上限值的安全近似。

lb和 ub的表达式必须是 int，bool，float或者 set of int类型。dom中表达式的类型不能是 float。如果 ⟨ 表达式⟩ 中的一个变量有一个非有限声明类型（例如，var int 或 var float类型），则会出现一个错误。

我们也有直接作用于表达式数组的版本（有类似的限制）：

- dom_array (⟨array-exp⟩): 返回数组中出现的表达式的所有可能值的并集的一个安全近似。
- lb_array (⟨array-exp⟩): 返回数组中出现的所有表达式的下限的安全近似。
- ub_array (⟨array-exp⟩): 返回数组中出现的所有表达式的下限的安全近似。

cumulative ≡ [download]

```
%-----------------------------------------------------------------------%
% 需要给出一个任务集合，其中起始时间为's',
% 持续时间'd' 以及资源需求量'r',
% 任何时候需求量都不能超过一个全局资源界限'b'。
% 假设:
% - forall i, d[i] >= 0 and r[i] >= 0
%-----------------------------------------------------------------------%
predicate cumulative(array[int] of var int: s,
                     array[int] of var int: d,
                     array[int] of var int: r, var int: b) =
   assert(index_set(s) == index_set(d) /\
          index_set(s) == index_set(r),
     "cumulative: the array arguments must have identical index sets",
   assert(lb_array(d) >= 0 /\ lb_array(r) >= 0,
     "cumulative: durations and resource usages must be non-negative",
          let {
              set of int: times =
                lb_array(s) ..
                max([ ub(s[i]) + ub(d[i]) | i in index_set(s) ])
              }
          in
             forall( t in times ) (
                b >= sum( i in index_set(s) ) (
                   bool2int( s[i] <= t /\ t < s[i] + d[i] ) * r[i]
                )
             )
      )
   );
```

图 38: 利用分解来定义一个 cumulative谓词（cumulative.mzn）。

谓词，局部变量和定义域反射的结合使得复杂全局约束通过分解定义变为可能。利用图 38中的代码，我们可以定义 cumulative约束的根据时间的分解。

这个分解利用 lb和 ub来决定任务可以执行的时间范围集合。接下来，它对 times中的每个时间 t 都断言在此时间 t 激活的所有任务所需要的资源量总和小于界限 b。

scope ≡ [download]

```
int: x = 3;
int: y = 4;
predicate smallx(var int:y) = -x <= y /\ y <= x;
predicate p(int: u, var bool: y) =
      exists(x in 1..u)(y \/ smallx(x));
constraint p(x,false);
solve satisfy;
```

图 39: 阐述变量作用域的模型（scope.mzn）。

4.9 作用域

MiniZinc 中声明的作用域值得我们简单地介绍下。MiniZinc 只有一个 namespace，所以出现在声明中的所有变量都可以在模型中的每个表达式中可见。用以下几个方式，MiniZinc 引进局部作用域变量：

- 推导式表达式中的迭代器
- 使用 let表达式
- 谓词和函数中的参数

任何局部作用域变量都会覆盖同名称的外部作用域变量。

例如，在图 39中给出的模型中，-x <= y中的 x是全局 x，even(x)中的 x是迭代器 x in 1..u，而析取中的 y是谓词的第二个参数。

5 选项类型

选项类型是一个强大的抽象，使得简洁建模成为可能。一个选项类型决策变量代表了一个有其他可能 T 的变量，在 MiniZinc 中表达为 <>，代表了这个变量是缺失的。选项类型变量在建模一个包含在其他变量没做决定之前不会有意义的变量的问题时是很有用的。

5.1 声明和使用选项类型

选项类型变量

一个选项类型变量被声明为：

[var] opt ⟨ 类型⟩ : ⟨ 变量名⟩

其中类型是 int，float或 bool中的一个，或者是一个固定范围的表达式。选项类型变量可以是参数，但是这个不常用。

一个选项类型变量可以有附加值 <>表明它是缺失的。

三个内建函数被提供给选项类型变量：absent(v) 只有在选项类型变量 v 取值 <>时返回 true，occurs(v) 只有在选项类型变量 v 不取值 <>时返回 true，以及 deopt(v) 返回 v 的正常值或者当它取值 <>时返回失败。

选项类型最常被用到的地方是调度中的可选择任务。在灵活的车间作业调度问题中，我们有 n 个在 k 个机器上执行的任务，其中完成每个机器上每个任务的时间可能是不一样的。我们的目标是最小化所有任务的总完成时间。一个使用选项类型来描述问题的模型在图 40中给出。在建模这个问题的时候，我们使用 $n \times k$ 个可选择的任务来代表每个机器上每个任务的可能性。我们使用 alternative全局约束来要求任务的起始时间和它的持续时间跨越了组成它的可选择任务的时间，同时要求只有一个会实际运行。我们使用 disjunctive全局变量在每个机器上最多有一个任务在运行，这里我们延伸到可选择的任务。最后我们约束任何时候最多有 k 个任务在运行，利用一个在实际（不是可选择的）任务上作用的冗余约束。

5.2 隐藏选项类型

当列表推导式是从在变量集合迭代上创建而来，或者 where从句中的表达式还没有固定时，选项类型变量会隐式地出现。

例如，模型片段

```
var set of 1..n: x;
constraint sum(i in x)(i) <= limit;
```

是以下的语法糖

```
var set of 1..n: x;
constraint sum(i in 1..n)(if i in x then i else <> endif) <= limit;
```

内建函数 sum实际上在一列类型 -实例化 var opt int上操作。由于 <>在 + 中表现为标识 0，我们会得到期望的结果。

类似地，模型片段

```
array[1..n] of var int: x;
constraint forall(i in 1..n where x[i] >= 0)(x[i] <= limit);
```

flexible-js ≡ [download]

```
int: horizon;                    %  时间范围
set of int: Time = 0..horizon;
enum Task;
enum Machine;

array[Task,Machine] of int: d; %  每个机器上的持续时间
int: maxd = max([ d[t,m] | t in Task, m in Machine ]);
int: mind = min([ d[t,m] | t in Task, m in Machine ]);

array[Task] of var Time: S;              %  起始时间
array[Task] of var mind..maxd: D;        %  持续时间
array[Task,Machine] of var opt Time: O; %  可选择的任务起始

constraint forall(t in Task)(alternative(S[t],D[t],
                 [O[t,m]|m in Machine],[d[t,m]|m in Machine]));
constraint forall(m in Machine)
                 (disjunctive([O[t,m]|t in Task],[d[t,m]|t in Task]));
constraint cumulative(S,D,[1|i in Task],k);

solve minimize max(t in Task)(S[t] + D[t]);
```

图 40: 使用选项类型的灵活车间作业调度模型（`flexible-js.mzn`）。

是以下的语法糖

```
array[1..n] of var int: x;
constraint forall(i in 1..n)(if x[i] >= 0 then x[i] <= limit else <> endif);
```

同样地，函数 `forall`实际上在一列类型 -实例化 `var opt bool`上操作。由于 `<>`在 $\wedge$ 上表现为标识 `true`，我们可以得到期望的结果。

尽管我们已经很小心了，隐式的使用可能会导致意外的行为。观察

```
var set of 1..9: x;
constraint card(x) <= 4;
constraint length([ i | i in x]) > 5;
solve satisfy;
```

它本应该是一个不可满足的问题。它返回 $x = 1,2,3,4$ 作为一个解例子。这个是正确的因为第二个约束等于

```
constraint length([ if i in x then i else <> endif | i in 1..9 ]) > 5;
```

而可选择整数列表的长度总会是 9，所以这个约束总是会满足。

我们可以通过不在变量集合上创建迭代或者使用不固定的 `where`从句来避免隐式的选项类型。例如，上面的两个例子可以不使用选项类型重写为

```
var set of 1..n: x;
constraint sum(i in 1..n)(bool2int(i in x)*i) <= limit;
```

和

```
array[1..n] of var int: x;
constraint forall(i in 1..n)(x[i] >= 0 -> x[i] <= limit);
```

6 搜索

MiniZinc 默认没有我们想如何搜索解的声明。这就把搜索全部都留给下层的求解器了。但是有些时候，尤其是对组合整数问题，我们或许想规定搜索应该如何去进行。这就需要我们和求解器沟通出一个搜索策略。注意，搜索策略不真的是模型的一部分。实际上，我们不要求每个求解器把所有可能的求解策略都实现了。MiniZinc 通过使用 *annotations*来用一个稳定的方法跟约束求解器沟通额外的信息。

6.1 有限域搜索

利用有限域求解器搜索涉及到检查变量剩余的可能值以及选择进一步地约束一些变量。搜索则会加一个新的约束来限制这个变量的剩余值（实际上猜测解可能存在于哪里），然后使用传播来确定其他的值是否可能存在于解中。为了确保完全性，搜索会保留另外一个选择，而它是新约束的否定。搜索会当有限域求解器发现所有的约束都被满足，此时一个解已经被找到，或者有约束不被满足时停止。当不可满足出现的时候，搜索必须换另外一个不同的选择集合继续下去。通常有限域求解器使用深度优先搜索，它会撤销最后一个做的选择然后尝试做一个新的选择。

nqueens ≡ [download]

```
int: n;
array [1..n] of var 1..n: q; %  i 列的皇后在行 q[i]

include "alldifferent.mzn";

constraint alldifferent(q);                          %  不同行
constraint alldifferent([ q[i] + i | i in 1..n]); %  不同对角线
constraint alldifferent([ q[i] - i | i in 1..n]); %  上 + 下

► search
output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

图 41: n 皇后问题模型（nqueens.mzn）。

有限域问题的一个简单例子是 n 皇后问题，它要求我们放置 n 个皇后在 $n \times n$ 棋盘上使得任何两个都不会互相攻击。变量 $q[i]$ 记录了在 i 列的皇后放置在哪一行上。alldifferent约束确保了任何两个皇后都不会在同一行或者对角线上。图 42的左边给出了 $n = 9$ 的典型（部分）搜索树。我们首选设置 $q[1] = 1$，这样就可以从其他变量的定义域里面移除一些数值，例如 $q[2]$ 不能取值 1 或者 2. 我们接下来设置 $q[2] = 3$，然后进一步地从其他变量的定义域里面移除一些

数值。我们设置 $q[3] = 5$（它最早的可能值）。在这三个决策后，棋盘的状态显示为图 42(a)。其中皇后表示已经固定的皇后位置。星星表示此处放置的皇后会攻击到已经放置的皇后，所以我们不能在此处放置皇后。

一个搜索策略决定要做哪一个选择。我们目前所做的决定都按照一个简单的策略：选择第一个还没有固定的变量，尝试设置它为它的最小可能值。按照这个策略，下一个决策应该是 $q[4] = 7$。变量选择的另外一个策略是选择现在可能值集合（定义域）最小的变量。按照这个所谓最先失败变量选择策略，下一个决策应该是 $q[6] = 4$。如果我们做了这个决策，则初始的传播会去除掉图 42(b) 中显示的额外的值。但是它使得 $q[8]$ 只剩余有一个值。所以 $q[8] = 7$ 被执行。但是这又使得 $q[7]$ 和 $q[9]$ 也只剩余一个值 2。因此这个时候有个约束一定会被违反。我们检测到了不满足性，求解器必须回溯取消最后一个决策 $q[6] = 4$ 并且加入它的否定 $q[6] \neq 4$（引导我们到了图 42中树的状态 (c)），即强制使 $q[6] = 8$。这使得有些值从定义域中被去除。我们接下来继续重新启用搜索策略来决定该怎么做。

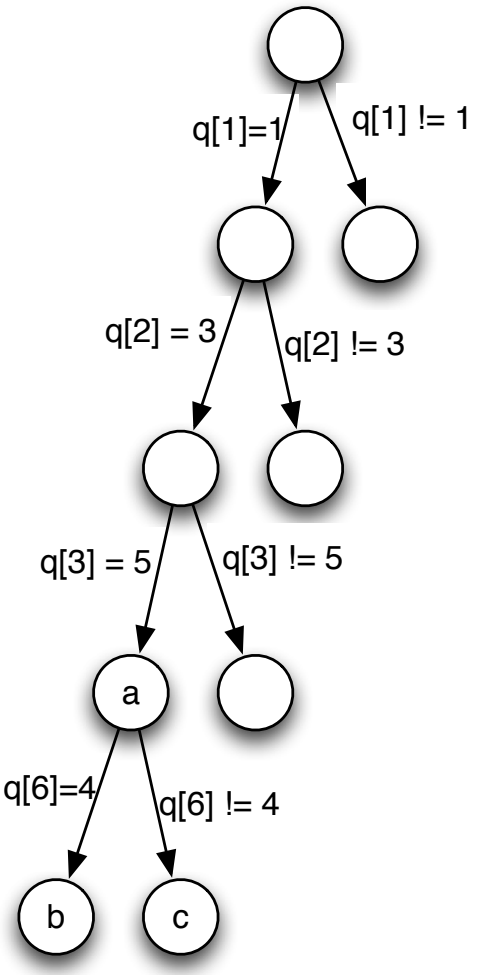
很多有限域搜索被定义为这种方式：选择一个变量来进一步约束，然后选择如何进一步地约束它。

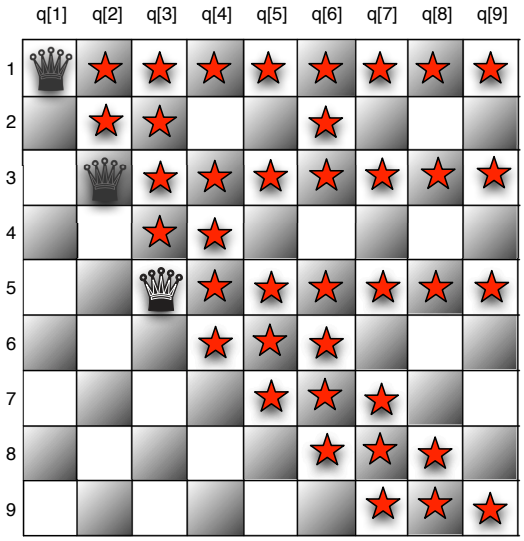
6.2 搜索注解

MiniZinc 中的搜索注解注明了为了找到一个问题的解应如何去搜索。注解附在求解项，在关键字 `solve`之后。搜索注解

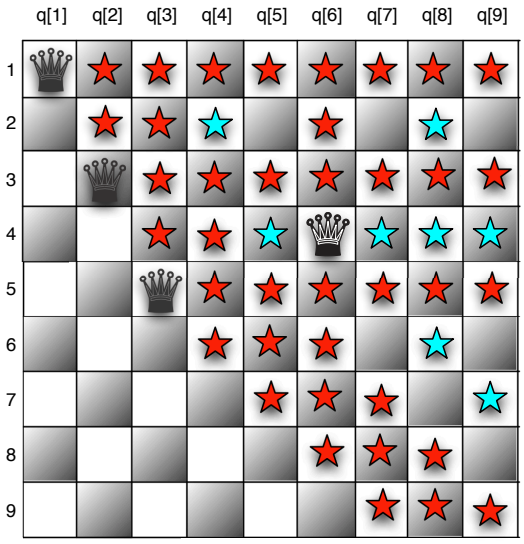
```
search ≡
  solve :: int_search(q, first_fail, indomain_min, complete)
        satisfy;
```

出现在求解项中。注解使用连接符`::`附为模型的一部分。这个搜索注解意思是我们应该按照从整型变量数组 `q`中选择拥有最小现行定义域的变量（这个是最先失败规则），然后尝试设置其为它的最小可能值（`indomain_min`值选择），纵观整个搜索树来搜索（`complete`搜索）。





(a)



(b)

图 42: 9 皇后问题的部分搜索树：(a) 在加入 $q[1]=1$，$q[2]=4$，$q[3]=5$ 后的状态 (b) 在进一步加入 $q[6]=4$ 后的初始传播

基本搜索注解

我们有三个基本搜索注解，相对应于不同的基本搜索类型：

- int_search(变量，变量选择，约束选择，策略) 其中变量是一个 var int类型的一维数组，变量选择是一个接下来会讨论的变量选择注解，约束选择是一个接下来会讨论的如何约束一个变量的选择，策略是一个搜索策略，我们暂时假设为 complete。
- bool_search(变量，变量选择，约束选择，策略) 其中变量是一个 var bool类型的一维数组，剩余的和上面一样。
- set_search(变量，变量选择，约束选择，策略) 其中变量是一个 var set of int类型的一维数组，剩余的和上面一样。

变量选择注解的例子有：input_order 从数组中按照顺序选择，first_fail选择拥有最小定义域大小的变量，以及 smallest选择拥有最小值的变量。约束一个变量的方式有：indomain_min 赋最小的定义域内的值给变量，indomain_median赋定义域内的中间值给变量，indomain_random从定义域中取一个随机的值赋给变量，以及 indomain_split把变量定义域一分为二然后去除掉上半部分。

对于完全搜索，搜索策略基本都是 complete。关于一份完整的变量和约束选择注解，请参看 MiniZinc 参考文档中的 FlatZinc 说明书。

利用搜索构造注解，我们可以创建更加复杂的搜索策略。目前我们只有一个这样的注解。

```
seq_search([ search_ann, …, search_ann ])
```

顺序搜索构造首先执行对列表中的第一个注解所指定的变量的搜索，当这个注解中的所有的变量都固定后，它执行第二个搜索注解，等等。直到所有的搜索注解都完成。

我们来看一下图 34中给出的车间作业调度模型。我们可以替换求解项为

```
solve :: seq_search([
            int_search(s, smallest, indomain_min, complete),
            int_search([end], input_order, indomain_min, complete)])
      minimize end
```

通过选择可以最早开始的作业并设置其为 s，起始时间被设置为 s。当所有的起始时间都设置完后，终止时间 end或许还没有固定。因此我们设置其为它的最小可能取值。

6.3 注解

在 MiniZinc 中，注解是第一类对象。我们可以在模型中声明新的注解，以及声明和赋值给注解变量。

nqueens-ann ≡ [download]

```
annotation bitdomain(int:nwords);

include "alldifferent.mzn";

int: n;
array [1..n] of var 1..n: q :: bitdomain(n div 32);

constraint alldifferent(q) :: domain;
constraint alldifferent([ q[i] + i | i in 1..n]) :: domain;
constraint alldifferent([ q[i] - i | i in 1..n]) :: domain;

ann: search_ann;

solve :: search_ann satisfy;

output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

图 43: n 皇后问题的注解模型（nqueens-ann.mzn）。

注解

注解有一个类型 ann。你可以声明一个注解参数（拥有可选择的赋值）

ann : ⟨ 标识符⟩ [= ⟨ 注解表达式⟩] ;

对注解变量赋值和对其他参数赋值一样操作。

表达式，变量声明，和 solve 表达式项都可以通过使用::操作符来成为注解。

使用注解项，我们可以声明一个新的注解

annotation ⟨ 注解名⟩(⟨ 参数定义⟩, ..., ⟨ 参数定义⟩) ;

图 43中的程序阐述了注解声明，注解和注解变量的使用。我们声明一个新的注解 bitdomain，意思是来告诉求解器变量定义域应该通过大小为 nwords的比特数组来表示。注解附注在变量 q的声明之后。每一个 alldifferent约束都被注解为内部注解 domain，而它指导求解器去使用 alldifferent的定义域传播版本（如果有的话）。一个注解变量 search_ann被声明和使用来定义搜索策略。我们可以在一个单独的数据文件中来给出搜素策略的值。

搜索注解的例子或许有以下几种（我们假设每一行都在一个单独的数据文件中）

```
search_ann = int_search(q, input_order, indomain_min, complete);
search_ann = int_search(q, input_order, indomain_median, complete);
search_ann = int_search(q, first_fail, indomain_min, complete);
search_ann = int_search(q, first_fail, indomain_median, complete);
```

第一个只是按顺序来选择皇后然后设置其为最小值。第二个按顺序来选择皇后，但是设置中间值给它。第三个选择定义域大小最小的皇后，然后设置最小值给它。最后一个策略选择定义域大小最小的皇后，设置中间值给它。

不同的搜索策略对于能多容易找到解有显著的差异。下面的表格给出了一个简单的关于使用 4 种不同的搜索策略找到 n 皇后问题的第一个解所要做的决策个数（其中 —表示超过 100,000 个决策）。很明显地看到，合适的搜索策略会产生显著的提高。

n	input-min	input-median	ff-min	ff-median
10	28	15	16	20
15	248	34	23	15
20	37330	97	114	43
25	7271	846	2637	80
30	—	385	1095	639
35	—	4831	—	240
40	—	—	—	236

grocery ≡ [download]

```
var int: item1;
var int: item2;
var int: item3;
var int: item4;

constraint item1 + item2 + item3 + item4 == 711;
constraint item1 * item2 * item3 * item4 == 711 * 100 * 100 * 100;

constraint 0 < item1 /\ item1 <= item2 /\ item2 <= item3 /\ item3 <= item4;

solve satisfy;

output ["{", show(item1), ",", show(item2), ",", show(item3), ",",
  show(item4),"}\n"];
```

图 44: 含有无界限整数的模型（grocery.mzn）。

7 MiniZinc 中的有效建模实践

对同一个问题，几乎总是存在多种方式来建模。其中一些产生的模型可以很有效地求解，另外一些则不是。通常情况下，我们很难提前判断哪个模型是对解决一个特定的问题最有效的。事实上，这或许十分依赖于我们使用的底层求解器。在这一章中，我们专注于建模实践，来避免产生模型的过程和产生的模型低效。

7.1 变量界限

有限域传播器，是 MiniZinc 所针对的求解器中的核心类型。在当其所涉及到的变量的界限越紧凑时，此传播器越有效。它也会当问题含有会取很大整型数值的子表达式时表现得很差，因为它们可能会隐式地限制整型变量的大小。

注意就算在所有变量都有界的模型中都有可能引入对求解器来说过大的中间变量。

图 44中的杂货店问题要找寻 4 个物品使得它们的价格加起来有 7.11 元并且乘起来也有 7.11 元。变量被声明为无界限。运行

```
$ mzn-g12fd grocery.mzn
```

得到

```
=====UNSATISFIABLE=====
% grocery.fzn:11: warning: model inconsistency detected before search.
```

这是因为乘法中的中间表达式的类型也会是 `var int`，也会被求解器给一个默认的界限 $-1,000,000..1,000,000$。但是这个范围太小了以至于不能承载住中间表达式所可能取的值。

更改模型使得初始变量都被声明为拥有更紧致的界限

```
var 1..711: item1;
var 1..711: item2;
var 1..711: item3;
var 1..711: item4;
```

我们得到一个更好的模型，因为现在 MiniZinc 可以推断出中间表达式的界限，并且使用此界限而不是默认的界限。在做此更改后，求解模型我们得到

```
{120,125,150,316}
----------
```

注意，就算是改善的模型也可能对于某些求解器来说会很难解决。运行

```
$ mzn-g12lazy grocery.mzn
```

不能得到任何结果，因为求解器给中间产生的变量创建了巨大的表示。

给变量加界限

在模型中要尽量使用有界限的变量。当使用 `let`声明来引进新的变量时，始终尽量给它们定义正确的和紧凑的界限。这会使得你的模型更有效率，避免出现意外溢出的可能性。一个例外是当你引进一个新的变量然后立刻定义它等于一个表达式，通常 MiniZinc 都可以从此表达式推断出此变量有效的界限。

7.2 无约束变量

有些时候，当我们建模的时候，很容易引进多于我们实际建模问题所需要的变量。

让我们来看下图 45中的哥隆尺问题模型。一个有 n 个标记的哥隆尺问题要求任何两个标记的绝对距离都是互相不同的。此模型创建了一个含有距离变量的两维数组，但是只是使用满足 $i > j$ 的 `diff[i,j]`。运行模型

```
$ mzn-g12fd golomb.mzn -D "n = 4; m = 6;"
```

得到输出

golomb ≡ [download]

```
include "alldifferent.mzn";

int: n; % 尺子上标记的数量
int: m; % 尺子的最大长度

array[1..n] of var 0..m: mark;
array[1..n,1..n] of var 0..m: diffs;

constraint mark[1] = 0;
constraint forall ( i in 1..n-1 ) ( mark[i] < mark[i+1] );
constraint forall (i,j in 1..n where i > j)          % (diff)
                  (diffs[i,j] = mark[i] - mark[j]);  % (diff)
constraint alldifferent([ diffs[i,j] | i,j in 1..n where i > j]);
constraint diffs[2,1] < diffs[n,n-1];  % symmetry break

solve satisfy;

output ["mark = \(mark);\ndiffs = \(diffs);\n"];
```

图 45: 哥隆尺问题的一个含有无约束变量的模型（golomb.mzn）。

```
mark = [0, 1, 4, 6];
diffs = [0, 0, 0, 0, 1, 0, 0, 0, 4, 3, 0, 0, 6, 5, 2, 0];
----------
```

在这个模型下，似乎一切都看起来是没有问题的。但是如果我们使用以下命令求得所有的解

```
$ mzn-g12fd -a golomb.mzn -D "n = 4; m = 6;"
```

我们会得到一个不会终止的由同一个解组成的列表。

怎么回事？为了让有限域求解器结束，它需要固定所有的变量，包括当 $i \leq j$ 时的变量 diff[i,j]。这意味着存在着数不尽的方式去得到一个解，因为我们只需要简单地改变这些 $i \leq j$ 时的变量取任何的值。

通过更改模型使无约束变量固定为某些值，我们可以避免无约束变量问题。例如，替换图 45中标记为 (diff) 的行为

```
constraint forall(i,j in 1..n)
                 (diffs[i,j] = if (i > j) then mark[i] - mark[j]
                               else 0 endif);
```

保证了额外的变量都被固定为 0。在做此改变后，求解器只返回一个解。

MiniZinc 会自动去除掉没有约束并且没有在输出时被使用的变量。上述问题的另外一个解决方案是通过改变输出语句为如下方式来简单地去除掉 diffs数组的输出

```
output ["mark = \(mark);\n"];
```

在这个更改下，运行

```
$ mzn-g12fd -a golomb.mzn -D "n = 4; m = 6;"
```

只会得到

```
mark = [0, 1, 4, 6];
----------
==========
```

展示了唯一的解。

无约束变量

模型从来不应该有无约束变量。有时候不加非必须的变量是很难去建模一个问题。如果是这种情况下的话，就要记得加入约束来固定非必须变量使得它们不会影响求解。

7.3 有效的生成元

想象下我们想要计算在一个图中出现的三角形的个数（K_3 子图）。假设此图由一个邻接矩阵定义：如果点 i 和 j 邻接，则 $adj[i,j]$ 为真。我们或许可以写成

```
int: count = sum ([ 1 | i,j,k in NODES where i < j  /\ j < k
                        /\ adj[i,j] /\ adj[i,k] /\ adj[j,k]]);
```

这当然是对的，但是它检查了所有点可能组成的三元组。如果此图是稀疏的，在意识到一旦我们选择了 i 和 j，就可以进行一些测试之后，我们可以做得更好。

```
int: count = sum( i,j in NODES where i < j /\ adj[i,j])(
          sum([1 | k in NODES where j < k /\ adj[i,k] /\ adj[j,k]]));
```

你可以使用内建函数 trace来帮助决定在生成元内发生了什么。

追踪

函数 trace(*s*,*e*) 在对表达式 *e* 求值并返回它的值之前就输出字符串 *s*。它可以在任何情境下使用。

例如，我们可以查看在两种计算方式下的内部循环中分别进行了多少次测试。

count1 ≡ [download]

```
int:count = sum([ 1 | i,j,k in NODES where
            trace("+", i<j /\j<k /\ adj[i,j] /\ adj[i,k] /\ adj[j,k]) ]);
adj = [| false, true,  true,  false
       | true,  false, true,  false
       | true,  true,  false, true
       | false, false, true,  false |];
constraint trace("\n",true);
solve satisfy;
```

得到输出：

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
----------
```

表示内部循环进行了 64 次，而

count2 ≡ [download]

```
int: count = sum( i,j in NODES where i < j /\ adj[i,j])(
       sum([1 | k in NODES where trace("+", j < k /\ adj[i,k] /\ adj[j,k])]));
```

Produces the output:

```
++++++++++++++++
----------
```

表示内部循环进行了 16 次。

注意你可以在 trace中使用单独的字符串来帮助你理解模型创建过程中发生了什么。

count3 ≡ [download]

```
int: count = sum( i,j in NODES where i < j /\ adj[i,j])(
       sum([trace("("++show(i)++","++show(j)++","++show(k)++")",1) |
             k in NODES where  j < k /\ adj[i,k] /\ adj[j,k]]));
```

会输出在计算过程中找到的每个三角形。得到输出

```
(1,2,3)
----------
```

magic-series2 ≡ [download]

```
int: n;
array[0..n-1] of var 0..n: s;

constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));
▶ redundant
solve satisfy;

output [ "s = ", show(s), ";\n" ] ;
```

图 46: 使用冗余约束求解魔术串问题模型（magic-series2.mzn）。

7.4 冗余约束

模型的形式会影响约束求解器求解它的效率。在很多情况下加入冗余约束，即被现有的模型逻辑上隐含的约束，可能会让求解器在更早时候产生更多可用的信息从而提高找寻解的搜索效率。

回顾下第小节 3.5节中的魔术串问题。 运行 $n = 16$ 时的模型：

```
$ mzn-g12fd --all-solutions --statistics magic-series.mzn -D "n=16;"
```

可能会得到如下结果

```
s = [12, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0];
----------
==========
```

统计显示需要 174 个决策点。

我们可以在模型中加入冗余约束。由于序列中的每个数字是用来计算某一个数字出现的次数，我们知道它们的和肯定是 n。类似地，由于这个序列是魔术的，我们知道 $s[i] \times i$ 的和肯定也是 n。使用如下方式把这些约束加入我们的模型

redundant ≡

```
constraint sum(i in 0..n-1)(s[i]) = n;
constraint sum(i in 0..n-1)(s[i] * i) = n;
```

就会得到图 46中的模型。

像之前那样求解同一个问题

```
$ mzn-g12fd --all-solutions --statistics magic-series2.mzn -D "n=16;"
```

产生了同样的输出。但是统计显示只搜索了 13 个决策点。这些冗余约束使得求解器更早地去剪枝。

7.5 模型选择

在 MiniZinc 中有很多方式去给同一个问题建模，尽管其中有些模型或许会比另外的一些模型更自然。不同的模型或许会产生不同的求解效率。更糟糕的是，在不同的求解后端中，不同的模型或许会更好或更差。但是，我们还是可以给出一些关于普遍情况下产生更好的模型的指导。

模型之间的选择

一个好的模型倾向于有以下特征

- 更少量的变量，或者至少是更少量的没有被其他变量功能上定义的变量。
- 更小的变量定义域范围
- 模型的约束定义更简洁或者直接
- 尽可能地使用全局约束

实际情况中，所有这些都需要通过检查这个模型的搜索到底多有效率来断定模型好坏。通常除了用实验之外，我们很难判断搜索是否高效。

观察如下问题，我们要找寻 1 到 n 这 n 个数字的排列，使得相邻数字的差值也形成一个 1 到 n 的排列。图 47中给出了一个用直观的方式来建模此问题的模型。注意变量 u 被变量 x 功能性定义。所以最差情况下的搜索空间是 n^n。

在这个模型中，数组 x 代表 n 个数字的排序。约束自然地可用 `alldifferent`来表示。求解模型

```
$ mzn-g12fd -all-solutions --statistics allinterval.mzn -D "n=10;"
```

在 84598 个决策点和 3 秒的时间内找到了所有的解。

另外一个模型是使用数组 y，其中 $y[i]$ 代表数字 i 在序列中的位置。我们同时也使用变量 v 来建模表示差的位置。$v[i]$ 表示了绝对值差 i 在序列出现的位置。如果 $y[i]$ 和 $y[j]$ 差别为一，其中 $j > i$，则代表了它们的位置是相邻的。所以 $v[j-i]$ 被约束为两个位置中最早的那个。我们可以给这个模型加入两个冗余约束：由于我们知道差值 $n-1$ 肯定会产生，我们就可以推断出 1 和 n 的位置必须是相邻的 $|y[1]-y[n]|=1$。同时也告诉我们差值 $n-1$ 的位置就是在 $y[1]$ 和 $y[n]$ 中的最早的那个位置，即

allinterval ≡ [download]

```
include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: x;       %  数字的序列
array[1..n-1] of var 1..n-1: u;   %  差的序列

constraint alldifferent(x);
constraint alldifferent(u);
constraint forall(i in 1..n-1)(u[i] = abs(x[i+1] - x[i]));

solve :: int_search(x, first_fail, indomain_min, complete)
      satisfy;
output ["x = ",show(x),"\n"];
```

图 47: CSPlib 中全区间序列问题 “prob007” 的一个自然模型（`allinterval.mzn`）。

$v[n-1] = min(y[1], y[n])$。有了这些之后，我们可以建模此问题为图 48。输出语句从位置数组 y 里重现了原本的序列 x。

逆向模型跟初始模型有同样的变量和定义域大小。但是相对于给变量 x 和 u 的关系建模，逆向模型使用了一个更加非直接的方式来给变量 y 和 v 的关系建模。所以我们或许期望初始模型更好些。

命令

```
$ mzn-g12fd --all-solutions --statistics allinterval2.mzn -D "n=10;"
```

在 75536 个决策点和 18 秒内找到了所有的解。有趣的是，尽管这个模型不是简洁的，在变量 y 上搜索比在变量 x 上搜索更加有效率。简洁的缺乏意味着尽管搜索需要更少的决策点，但是在时间上实质上会更慢。

7.6 多重建模和连通

当我们对同一个问题有两个模型时，由于每个模型可以给求解器不同的信息，通过把两个模型中的变量系到一起从而同时使用两个模型或许对我们是有帮助的。

图 49给出了一个结合 `allinterval.mzn`和 `allinterval2.mzn`特征的双重模型。模型的开始来自于 `allinterval.mzn`。我们接着介绍了来自于 `allinterval2.mzn`中的变量 y 和 v。我们使用全局约束 `inverse`来把变量绑到一起：$inverse(x, y)$ 约束 y 为 x 的

allinterval2 ≡ [download]

```
include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: y;  %  每个数字的位置
array[1..n-1] of var 1..n-1: v; %  差值 i的位置

constraint alldifferent(y);
constraint alldifferent(v);
constraint forall(i,j in 1..n where i < j)(
      (y[i] - y[j] = 1 -> v[j-i] = y[j]) /\
                (y[j] - y[i] = 1 -> v[j-i] = y[i])
    );

constraint abs(y[1] - y[n]) = 1 /\ v[n-1] = min(y[1], y[n]);

solve :: int_search(y, first_fail, indomain_min, complete)
      satisfy;

output [ "x = [",] ++
       [ show(i) ++ if j == n then "]\n;" else ", " endif
         | j in 1..n, i in 1..n where j == fix(y[i]) ];
```

图 48: CSPlib 中全区间序列问题 “prob007” 的一个逆向模型。（`allinterval2.mzn`）。

逆向函数（反之亦然），即，$x[i] = j \Leftrightarrow y[j] = i$。图 50中给出了它的一个定义。这个模型没有包含把变量 y 和 v 关联起来的约束，它们是冗余的（实际上是传播冗余）。所以它们不会给基于传播的求解器多余的信息。`alldifferent`也不见了。原因是它们被逆向约束变得冗余了（传播冗余）。唯一的约束是关于变量 x 和 u 和关系的约束以及 y 和 v 的冗余约束。

双重模型的一个优点是我们可以有更多的定义不同搜索策略的视角。运行双重模型，

```
$ mzn-g12fd -all-solutions --statistics allinterval3.mzn -D "n=10;"
```

注意它使用逆向模型的搜索策略，标记变量 y，在 1714 决策点和 0.5 秒内找到了所有的解。注意标记变量 x 来运行同样的模型，需要 13142 个决策点和 1.5 秒。

allinterval3 ≡ [download]

```
include "inverse.mzn";

int: n;

array[1..n] of var 1..n: x;  %  数值的序列
array[1..n-1] of var 1..n-1: u;  %  差值的序列

constraint forall(i in 1..n-1)(u[i] = abs(x[i+1] - x[i]));

array[1..n] of var 1..n: y;  %  每个数值的位置
array[1..n-1] of var 1..n-1: v; %  差值 i的位置

constraint inverse(x,y);
constraint inverse(u,v);

constraint abs(y[1] - y[n]) = 1 /\ v[n-1] = min(y[1], y[n]);

solve :: int_search(y, first_fail, indomain_min, complete)
      satisfy;

output ["x = ",show(x),"\n"];
```

图 49: CSPlib 中全区间序列问题 “prob007” 的一个双重模型。（allinterval3.mzn）。

inverse ≡ [download]

```
predicate inverse(array[int] of var int: f,
                  array[int] of var int: invf) =
    forall(j in index_set(invf))(invf[j] in index_set(f)) /\
    forall(i in index_set(f))(
        f[i] in index_set(invf) /\
        forall(j in index_set(invf))(j == f[i] <-> i == invf[j])
    );
```

图 50: 全局约束 inverse的一个定义。（inverse.mzn）。

8 在 MiniZinc 中对布尔可满足性问题建模

MiniZinc 可以被用来给布尔可满足性问题建模，这种问题的变量被限制为是布尔型（bool）。MiniZinc 可以使用有效率的布尔可满足性求解器来有效地解决求得的模型。

8.1 整型建模

很多时候，尽管我们想要使用一个布尔可满足性求解器，我们可能也需要给问题的整数部分建模。

有三种通用的方式使用布尔型变量对定义域为范围 $0..m$ 内的整型变量 I 建模，其中 $m = 2^k - 1$。

- 二元：I 被表示为 k 个二元变量 $i_0, \ldots, i_{k-1}$，其中 $I = 2^{k-1}i_{k-1} + 2^{k-2}i_{k-2} + \cdots + 2i_1 + i_0$。在 MiniZinc 中，这可表示为

```
array[0..k-1]  of var bool: i;
var 0..pow(2,k)-1: I = sum(j in 0..k-1)(bool2int(i[j])*pow(2,j));
```

- 一元：I 被表示为 m 个二元变量 $i_1, \ldots, i_m$ 且 $I = \sum_{j=1}^{m} bool2int(i_j)$。由于在一元表示中有大量的冗余表示，我们通常要求 $i_j \rightarrow i_{j-1}, 1 < j \leq m$。在 MiniZinc 中，这可表示为

```
array[1..m]  of var bool: i;
constraint forall(j in 2..m)(i[j] -> i[j-1]);
var 0..m: I = sum(j in 1..m)(bool2int(i[j]);
```

- 值：其中 I 被表示为 $m+1$ 个二元变量 $i_0, \ldots, i_m$，其中 $I = k \Leftrightarrow i_k$ 并且 $i_0, \ldots, i_m$ 中最多有一个为真。在 MiniZinc 中，这可表示为

```
array[0..m]  of var bool: i;
constraint sum(j in 0..m)(bool2int(i[j]) == 1;
var 0..m: I;
constraint foall(j in 0..m)(I == j <-> i[j]);
```

每种表示都有其优点和缺点。这取决于模型中需要对整数做什么样的操作，而这些操作在哪一种表示上更为方便。

latin ≡ [download]

```
int: n; % 拉丁方的大小
array[1..n,1..n] of var 1..n: a;

include "alldifferent.mzn";
constraint forall(i in 1..n)(
             alldifferent(j in 1..n)(a[i,j]) /\
             alldifferent(j in 1..n)(a[j,i])
           );
solve satisfy;
output [ show(a[i,j]) ++ if j == n then "\n" else " " endif |
         i in 1..n, j in 1..n ];
```

图 51: 拉丁方问题的整数模型（latin.mzn）。

8.2 非等式建模

接下来，让我们考虑如何为一个拉丁方问题建模。一个拉丁方问题是在 $n \times n$ 个网格上放置 $1..n$ 之间的数值使得每个数在每行每列都仅出现一次。图 51中给出了拉丁方问题的的一个整数模型。

整型变量直接的唯一的约束实际上是非等式，而它在约束 alldifferent中被编码。数值表示是表达非等式的最佳方式。图 52给出了一个关于拉丁方问题的只含有布尔型变量的模型。注意每个整型数组元素 $a[i,j]$ 被替换为一个布尔型数组。我们使用谓词 exactlyone来约束每个数值在每行每列都仅出现一次，也用来约束有且仅有一个布尔型变量对应于整型数组元素 $a[i,j]$ 为真。

8.3 势约束建模

让我们来看下如何对点灯游戏建模。这个游戏由一个矩形网格组成，每个网格为空白或者被填充。每个被填充的方格可能包含 1 到 4 之间的数字，或者没有数字。我们的目标是放置灯泡在空白网格使得

- 每个空白的网格是“被照亮的”，也就是说它可以透过一行没有被打断的空白网格看到光亮。
- 任何两个灯泡都不能看到彼此。
- 一个有数值的填充的网格相邻的灯泡个数必须等于这个网格中的数值。

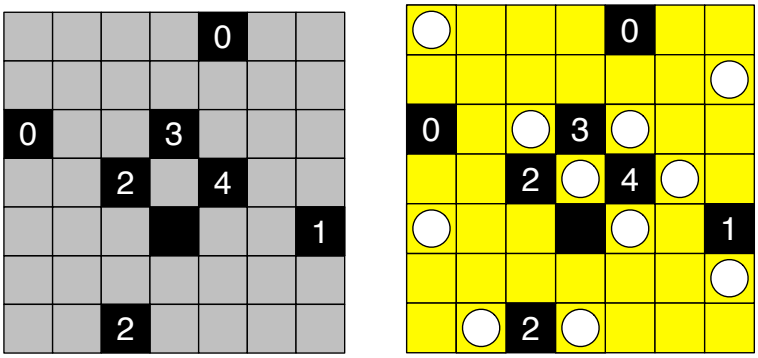
latinbool ≡ [download]

```
int: n; %  拉丁方的大小
array[1..n,1..n,1..n] of var bool: a;

predicate atmostone(array[int] of var bool:x) =
          forall(i,j in index_set(x) where i < j)(
            (not x[i] \/ not x[j]));
predicate exactlyone(array[int] of var bool:x) =
          atmostone(x) /\ exists(x);

constraint forall(i,j in 1..n)(
             exactlyone(k in 1..n)(a[i,j,k]) /\
             exactlyone(k in 1..n)(a[i,k,j]) /\
             exactlyone(k in 1..n)(a[k,i,j])
           );
solve satisfy;
output [ if fix(a[i,j,k]) then
            show(k) ++ if j == n then "\n" else " " endif
         else "" endif | i,j,k in 1..n ];
```

图 52: 拉丁方问题的布尔型模型（latinbool.mzn）。



(a) (b)

图 53: 点灯游戏的一个例子展示（a）初始问题，和（b）完整的解

图 53给出了点灯游戏的一个例子以及它的解。

这个问题很自然地可以使用布尔型变量建模。布尔型变量用来决定哪一个网格包含有一个点灯以及哪一个没有。同时我们也有一些作用于填充的网格上的整数算术运算要考虑。

lightup ≡ [download]

```
int: h; set of int: H = 1..h; %  板高度
int: w; set of int: W = 1..w; %  板宽度
array[H,W] of -1..5: b;       %  板
int: E = -1;                  %  空白格
set of int: N = 0..4;         %  填充并含有数字的网格
int: F =  5;                  %  填充但不含有数字的网格

% 位置 (i1,j1) 对位置 (i2,j2) 可见
test visible(int: i1, int: j1, int: i2, int: j2) =
    ((i1 == i2) /\ forall(j in min(j1,j2)..max(j1,j2))(b[i1,j] == E))
 \/ ((j1 == j2) /\ forall(i in min(i1,i2)..max(i1,i2))(b[i,j1] == E));

array[H,W] of var bool: l; % is there a light

% 填充的网格没有灯泡
constraint forall(i in H, j in W, where b[i,j] != E)(l[i,j] == false);
% 填充且有数字的网格相邻的灯泡数量和此数字相等
include "boolsum.mzn";
constraint forall(i in H, j in W where b[i,j] in N)(
    bool_sum_eq([ l[i1,j1] | i1 in i-1..i+1, j1 in j-1..j+1 where
                             abs(i1 - i) + abs(j1 - j) == 1 /\
                             i1 in H /\ j1 in W ], b[i,j]));
% 每个空白网格是被照亮的
constraint forall(i in H, j in W where b[i,j] == E)(
            exists(j1 in W where visible(i,j,i,j1))(l[i,j1]) \/
            exists(i1 in H where visible(i,j,i1,j))(l[i1,j])
          );
% 任何两个灯泡看不到彼此
constraint forall(i1,i2 in H, j1,j2 in W where
                 (i1 != i2 \/ j1 != j2) /\ b[i1,j1] == E
                 /\ b[i2,j2] == E /\ visible(i1,j1,i2,j2))(
            not l[i1,j1] \/ not l[i2,j2]
          );

solve satisfy;
output [ if b[i,j] != E then show(b[i,j])
         else if fix(l[i,j]) then "L" else "." endif
         endif ++ if j == w then "\n" else " " endif |
         i in H, j in W];
```

图 54: 点灯游戏的 SAT 模型（lightup.mzn）。

lightup.dzn ≡ [download]

```
h = 7;
w = 7;
b = [| -1,-1,-1,-1, 0,-1,-1
     | -1,-1,-1,-1,-1,-1,-1
     |  0,-1,-1, 3,-1,-1,-1
     | -1,-1, 2,-1, 4,-1,-1
     | -1,-1,-1, 5,-1,-1, 1
     | -1,-1,-1,-1,-1,-1,-1
     |  1,-1, 2,-1,-1,-1,-1 |];
```

图 55: 点灯游戏的图 53中实例的数据文件

图 54中给出了这个问题的一个模型。图 53中给出的问题的数据文件在图 55中给出。模型利用了一个布尔型求和谓词

```
predicate bool_sum_eq(array[int] of var bool:x, int:s);
```

使得一个布尔型数组的和等于一个固定的整数。多种方式都能使用布尔型变量给*cardinality*约束建模。

- 加法器网络：我们可以使用包含加法器的一个网络给布尔型总和建立一个二元布尔型表达式
- 排序网络：我们可以通过使用一个排序网络去分类布尔型数组来创建一个布尔型总和的一元表达式
- 二元决策图：我们可以创建一个二维决策图（BDD）来编码势约束。

我们可以使用图 56给出的二元加法器网络代码实现 `bool_sum_eq`。图 57中定义的谓词 `binary_sum`创建了一个 x 总和的二维表示法。它把列表分为两部分，把每一部分分别加起来得到它们的一个二元表示，然后用 `binary_add`把这两个二元数值加起来。如果 x 列大小是奇数，则最后一位被保存起来作为二元加法时的进位来使用。

我们可以使用图 58中给出的一元排序网络代码来实现 `bool_sum_eq`。势约束通过扩展输入 x 长度为 2 的次幂，然后使用奇偶归并排序网络给得到的位排序来实现。奇偶归并排序工作方式在图 59中给出，它递归地把输入列表拆为两部分，给每一部分排序，然后再把有序的两部分归并起来。

bboolsum ≡ [download]

```
% 布尔型变量 x的总和 = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
     let { int: c = length(x) } in
     if s < 0 then false
     elseif s == 0 then
          forall(i in 1..c)(x[i] == false)
     elseif s < c then
          let { % cp =  表示 0..c 所需要的位的数量
                int: cp = floor(log2(int2float(c))),
                % z is sum of x in binary
                array[0..cp] of var bool:z  } in
          binary_sum(x, z) /\
          % z == s
          forall(i in 0..cp)(z[i] == ((s div pow(2,i)) mod 2 == 1))
     elseif s == c then
          forall(i in 1..c)(x[i] == true)
     else false endif;

include "binarysum.mzn";
```

图 56: 使用二元加法器网络表示势约束（bboolsum.mzn）。

我们可以使用图 60中给出的二元决策图代码来实现 bool_sum_eq。势约束被分为两种情况：或者第一个元素 $x[1]$ 为 *true* 并且剩下位的总和是 $s-1$，或者 $x[1]$ 为 *false* 并且剩下位的总和是 s。它的效率的提高依赖于去除共同子表达式来避免产生太多的相同的约束。

binarysum ≡ [download]

```
% 位 x 的的总和 = 二元表示的 s。
%           s[0], s[1], ···, s[k] 其中 2^k >= length(x) > 2^(k-1)
predicate binary_sum(array[int] of var bool:x,
                     array[int] of var bool:s)=
     let { int: l = length(x) } in
     if l == 1 then s[0] = x[1]
     elseif l == 2 then
            s[0] = (x[1] xor x[2]) /\ s[1] = (x[1] /\ x[2])
     else let { int: ll = (l div 2),
                array[1..ll] of var bool: f = [ x[i] | i in 1..ll ],
                array[1..ll] of var bool: t = [ x[i] | i in ll+1..2*ll],
                var bool: b = if ll*2 == l then false else x[l] endif,
                int: cp = floor(log2(int2float(ll))),
                array[0..cp] of var bool: fs,
                array[0..cp] of var bool: ts } in
                binary_sum(f, fs) /\ binary_sum(t, ts) /\
                binary_add(fs, ts, b, s)
     endif;

% 把两个二元数值 x 和 y 加起来，位 ci 用来表示进位，来得到二元 s
predicate binary_add(array[int] of var bool: x,
                     array[int] of var bool: y,
                     var bool: ci,
                     array[int] of var bool: s) =
    let { int:l = length(x),
          int:n = length(s), } in
    assert(l == length(y),
          "length of binary_add input args must be same",
    assert(n == l \/ n == l+1, "length of binary_add output " ++
                        "must be equal or one more than inputs",
    let { array[0..l] of var bool: c } in
    full_adder(x[0], y[0], ci, s[0], c[0]) /\
    forall(i in 1..l)(full_adder(x[i], y[i], c[i-1], s[i], c[i])) /\
    if n > l then s[n] = c[l] else c[l] == false endif ));

predicate full_adder(var bool: x, var bool: y, var bool: ci,
                     var bool: s, var bool: co) =
          let { var bool: xy = x xor y } in
          s = (xy xor ci) /\ co = ((x /\ y) \/ (ci /\ xy));
```

图 57: 创建二元求和网络的代码（binarysum.mzn）。

uboolsum ≡ [download]

```
% 布尔型变量 x的总和 = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
     let { int: c = length(x) } in
     if s < 0 then false
     elseif s == 0 then forall(i in 1..c)(x[i] == false)
     elseif s < c then
          let { % cp =  最接近的 2的次幂 >= c
                int: cp = pow(2,ceil(log2(int2float(c)))),
                array[1..cp] of var bool:y, % y 是 x的扩充版本
                array[1..cp] of var bool:z  } in
          forall(i in 1..c)(y[i] == x[i]) /\
          forall(i in c+1..cp)(y[i] == false) /\
          oesort(y, z) /\ z[s] == true /\ z[s+1] == false
     elseif s == c then forall(i in 1..c)(x[i] == true)
     else false endif;

include "oesort.mzn";
```

图 58: 使用排序网络表示势约束（uboolsum.mzn）。

oesort ≡ [download]

```
%% 奇偶排序
%% y 是 x的有序版本，所有的真都在假之前
predicate oesort(array[int] of var bool:x, array[int] of var bool:y)=
  let { int: c = card(index_set(x)) } in
  if c == 1 then x[1] == y[1]
  elseif c == 2 then comparator(x[1],x[2],y[1],y[2])
  else
    let {
      array[1..c div 2] of var bool:xf = [x[i] | i in 1..c div 2],
      array[1..c div 2] of var bool:xl = [x[i] | i in c div 2 +1..c],
      array[1..c div 2] of var bool:tf,
      array[1..c div 2] of var bool:tl } in
    oesort(xf,tf) /\ oesort(xl,tl) /\ oemerge(tf ++ tl, y)
  endif;

%% 奇偶排序
%% y 是 x的有序版本，所有的真都在假之前
%% 假设 x的前一半是有序的，后一半也是
predicate oemerge(array[int] of var bool:x, array[int] of var bool:y)=
  let { int: c = card(index_set(x)) } in
  if c == 1 then x[1] == y[1]
  elseif c == 2 then comparator(x[1],x[2],y[1],y[2])
  else
    let { array[1..c div 2] of var bool:xo =
            [ x[i] | i in 1..c where i mod 2 == 1],
          array[1..c div 2] of var bool:xe =
            [ x[i] | i in 1..c where i mod 2 == 0],
          array[1..c div 2] of var bool:to,
          array[1..c div 2] of var bool:te } in
      oemerge(xo,to) /\ oemerge(xe,te) /\
      y[1] = to[1] /\
      forall(i in 1..c div 2 -1)(
        comparator(te[i],to[i+1],y[2*i],y[2*i+1])) /\
      y[c] = te[c div 2]
  endif));

% 比较器 o1 = max(i1,i2), o2 = min(i1,i2)
predicate comparator(var bool:i1,var bool:i2,var bool:o1,var bool:o2)=
          (o1 = (i1 \/ i2)) /\ (o2 = (i1 /\ i2));
```

图 59: 奇偶归并排序网络（oesort.mzn）。

bddsum ≡ [download]

```
% 布尔型变量 x的总和 = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
     let { int: c = length(x),
           array[1..c] of var bool: y = [x[i] | i in index_set(x)]
     } in
     rec_bool_sum_eq(y, 1, s);

predicate rec_bool_sum_eq(array[int] of var bool:x, int: f, int:s) =
     let { int: c = length(x) } in
     if s < 0 then false
     elseif s == 0 then
          forall(i in f..c)(x[i] == false)
     elseif s < c - f + 1 then
          (x[f] == true /\ rec_bool_sum_eq(x,f+1,s-1)) \/
          (x[f] == false /\ rec_bool_sum_eq(x,f+1,s))
     elseif s == c - f + 1 then
          forall(i in f..c)(x[i] == true)
     else false endif;
```

图 60: 使用二元决策图表示势约束（bddsum.mzn）。

A MiniZinc 关键字

注意，由于 MiniZinc 和 Zinc 共享一个解析器，所有的 Zinc 关键字同样也不能被用来作为 MiniZinc 的标识符。关键字有：

ann，annotation，any，array，assert，bool，constraint，enum，float，function，in，include，int，list，of，op，output，minimize，maximize，par，predicate，record，set，solve，string，test，tuple，type，var，where。

B MiniZinc 操作符

一元操作符有：not，+和 -。二元操作符有：<->，->，<-，\/，xor，/\，<，>，<=，>=，==，=，!=，in，subset，superset，union，diff，symdiff，..，intersect，++，+，-，*，/，div 和 mod。

C MiniZinc 函数

MiniZinc 中的内建函数有：abort，abs，acos，acosh，array_intersect，array_union，array1d，array2d，array3d，array4d，array5d，array6d，asin，asinh，assert，atan，atanh，bool2int，card，ceil，concat，cos，cosh，dom，dom_array，dom_size，fix，exp，floor，index_set，index_set_1of2，index_set_2of2，index_set_1of3，index_set_2of3，index_set_3of3，int2float，is_fixed，join，lb，lb_array，length，ln，log，log2，log10，min，max，pow，product，round，set2array，show，show_int，show_float，sin，sinh，sqrt，sum，tan，tanh，trace，ub，和 ub_array。