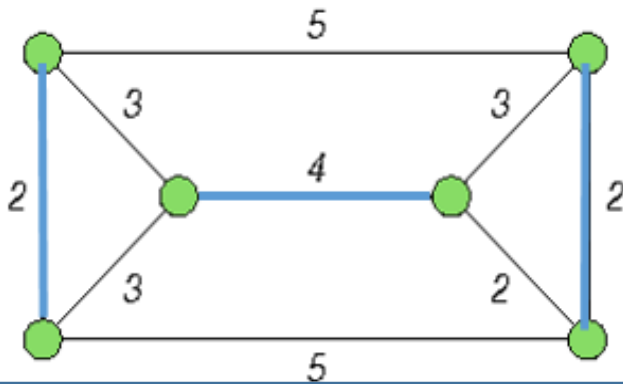


Algoritmos y Programación

Programación con
Restricciones 6.2

VPL: emparejamiento_perfecto

Dado un grafo, con un número par de vértices diseñar un modelo que nos permita obtener el emparejamiento perfecto de coste mínimo. El coste entre un par de vértices (coste de la arista) viene determinado en una tabla. (Ejemplo, coste mínimo 8)



[El problema del emparejamiento perfecto asociado a un grafo $G=(N,M)$ con un número par de vértices y unos costes c , consiste en emparejar cada vértice i de N con un y sólo un vértice j de N a coste mínimo, $\forall i, j \in N, i \neq j$]

Parámetros

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

```
int: n = 6;
```

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

```
array[VERTEX, VERTEX] of int: weights = [|  
10000, 5, 3, 10000, 2, 10000|  
5, 10000, 10000, 3, 10000, 2|  
3, 10000, 10000, 4, 3, 10000|  
10000, 3, 4, 10000, 10000, 2|  
2, 10000, 3, 10000, 10000, 5|  
10000, 2, 10000, 2, 5, 10000|  
|];
```

Definición de predicados

- Los predicados permiten la definición de nuevos tipos de restricciones de alto nivel
- Permite un modelado más modular
- Permite la reutilización

! Predicate definitions

Predicates are defined by a statement of the form

```
predicate <pred-name> ( <arg-def>, ..., <arg-def> ) = <bool-exp>
```

The <pred-name> must be a valid MiniZinc identifier, and each <arg-def> is a valid MiniZinc type declaration.

Definición de funciones

❗ Function definitions

Functions are defined by a statement of the form

```
function <ret-type> : <func-name> ( <arg-def>, ..., <arg-def> ) = <exp>
```


The *<func-name>* must be a valid MiniZinc identifier, and each *<arg-def>* is a valid MiniZinc type declaration. The *<ret-type>* is the return type of the function which must be the type of *<exp>*. Arguments have the same restrictions as in predicate definitions.

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

Variables locales, let

```
1 var 1..3:a;  
2 predicate bmenor(var int:a) =  
3   let {var 1..5:b} in b < a;  
4  
5 constraint bmenor(a);  
6  
7 solve satisfy;  
8  
9 output([show(a)]);
```

```
Running untitled_model.mzn  
2  
-----  
3  
-----  
=====  
Finished in 126msec
```



Ejemplo de
las 8 reinas
con minizinc

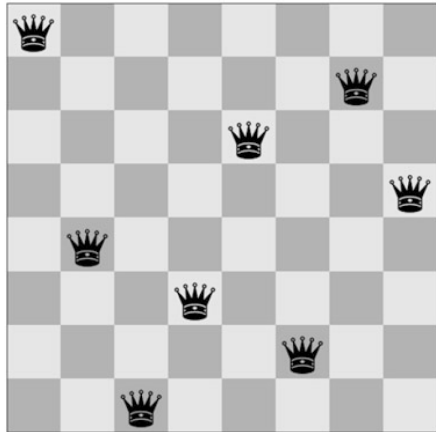
```
include "globals.mzn";
```

```
set of int: R = 1..8;  
array[R] of var R: row;
```

```
constraint alldifferent(row);  
constraint all_different([row[i]+i | i in R]);  
constraint all_different([row[i]-i | i in R]);
```

```
solve :: int_search(row,  
                    first_fail,  
                    indomain_min|  
                    satisfy;
```

Combinación de heurísticas de búsqueda



Q4,Q5,Q3,Q6,Q2,Q7,Q1,Q8

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

Search steps	Naive	(A) Middle-out Var	(B) First fail	(A+B)	(A+B)+ Middle-out Val
8-queens	10	0	17	0	5
16-queens	542	28	4	0	7
32-queens	Timeout	Timeout	9	1	15
64-queens	Timeout	Timeout	365	3	2
128-queens	Timeout	Timeout	Timeout	Timeout	0

- Naive: **orden de variables:** $Q1, Q2, \dots, Q8$, **orden de valores:** 1,2,...8
- A: **orden de variables:** Las variables se cogen del centro hacia los lados, **orden de valores:** 1,2...8
- B: **orden de variables:** First fail, **orden de valores:** 1,2, ...8
- A+B: Igual a B, cuando 2 variables tienen la misma cardinalidad coge la más cercana al centro
- A+B+middle-out-val: Igual a (A+B) pero el orden de valores sigue la fila más cercana al centro del tablero, 4,5,3,6,2,7,1,8
-

Solución

```
1|include "globals.mzn";
2
3% sign(k) = 1 si k es par, y sign(k) = -1 si k es impar
4function int:sign(int:j) =
5    2*((j+1)mod 2) -1 ;
6
7% middle_out mapea 5 a [3,4,2,5,1] y mapea 6 a [3,4,2,5,1,6]
8function array [int] of int: middle_out(int:n) =
9    let { int:mid = (n+1) div 2 } in
10    [mid + sign(i)*(i div 2) | i in 1..n];
```



```
13 int: n=128;
14 array [1..n] of var 1..n: rows;
15
16 constraint alldifferent(rows);
17 constraint alldifferent(i in 1..n)(rows[i] + i);
18 constraint alldifferent(i in 1..n)(rows[i] - i);
19
20 solve
21 :: int_search([rows[i] | i in middle_out(n)], first_fail, indomain_median)
22 satisfy ;
```

Combinación de heurísticas de búsqueda, seq_search, múltiples variables

```
int: n ;  
int: m ;  
  
array [1..n] of var 1..n: X ;  
array [1..m] of var 1..m: Y ;  
  
solve  
  :: seq_search([int_search(X,first_fail,indomain_max),  
                int_search(Y,first_fail,indomain_min)  
                ])  
  maximize sum(X)-sum(Y) ;
```

Solución:

```
X = array1d(1..4, [4, 4, 4, 4]);  
Y = array1d(1..5, [1, 1, 1, 1, 1]);  
-----  
=====
```

→ No hay soluciones intermedias

Combinación de heurísticas de búsqueda, seq_search, múltiples variables

```
1 int: n= 4;  
2 int: m= 5;  
3 array [1..n] of var 1..n: X ;  
4 array [1..m] of var 1..m: Y ;  
5 solve  
6 :: seq_search([int_search(X,first_fail,indomain_min),  
7 int_search(Y,first_fail,indomain_max)|  
8 ])  
9 maximize sum(X)-sum(Y) ;
```

Solución:

```
X = array1d(1..4, [1, 1, 1, 1]);  
Y = array1d(1..5, [5, 5, 5, 5, 5]);  
-----  
X = array1d(1..4, [1, 1, 1, 1]);  
Y = array1d(1..5, [5, 5, 5, 5, 4]);  
-----  
X = array1d(1..4, [1, 1, 1, 1]);  
Y = array1d(1..5, [5, 5, 5, 5, 3]);  
-----  
X = array1d(1..4, [1, 1, 1, 1]);  
Y = array1d(1..5, [5, 5, 5, 5, 2]);  
-----  
X = array1d(1..4, [1, 1, 1, 1]);  
Y = array1d(1..5, [5, 5, 5, 5, 1]);  
-----  
X = array1d(1..4, [1, 1, 1, 1]);  
Y = array1d(1..5, [5, 5, 5, 4, 1]);
```


...

```
X = array1d(1..4, [3, 4, 4, 4]);  
Y = array1d(1..5, [1, 1, 1, 1, 1]);  
-----  
X = array1d(1..4, [4, 4, 4, 4]);  
Y = array1d(1..5, [1, 1, 1, 1, 1]);  
-----  
=====
```

```

2 include "globals.mzn";
3
4 % sign(k) = 1 if k is even, and sign(k) = -1 if k is odd
5 function int:sign(int:j) =
6     2*((j+1)mod 2) -1 ;
7
8 % middle_out maps 5 to [3,4,2,5,1] and maps 6 to [3,4,2,5,1,6]
9 function array [int] of int: middle_out(int:n) =
10     let { int:mid = (n+1) div 2 } in
11     [mid + sign(i)*(i div 2) | i in 1..n];
12
13
14 int: n=128;
15 array [1..n] of var 1..n: rows;
16
17 constraint alldifferent(rows);
18 constraint alldifferent(i in 1..n)(rows[i] + i);
19 constraint alldifferent(i in 1..n)(rows[i] - i);
20
21
22 solve
23 :: seq_search(
24 [int_search([rows[i] | i in middle_out(n)], first_fail, indomain_median),
25 int_search([rows[i] | i in middle_out(n)], first_fail, indomain_random)]):restart_constant(10000)
26 satisfy ;

```

Restricción
global:
Regular

Regular

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

It constrains that the sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $\langle 1..Q, 1..S \rangle$ to $0..Q$) and initial state $q0$ (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state.

Ejemplo trabajo por turnos

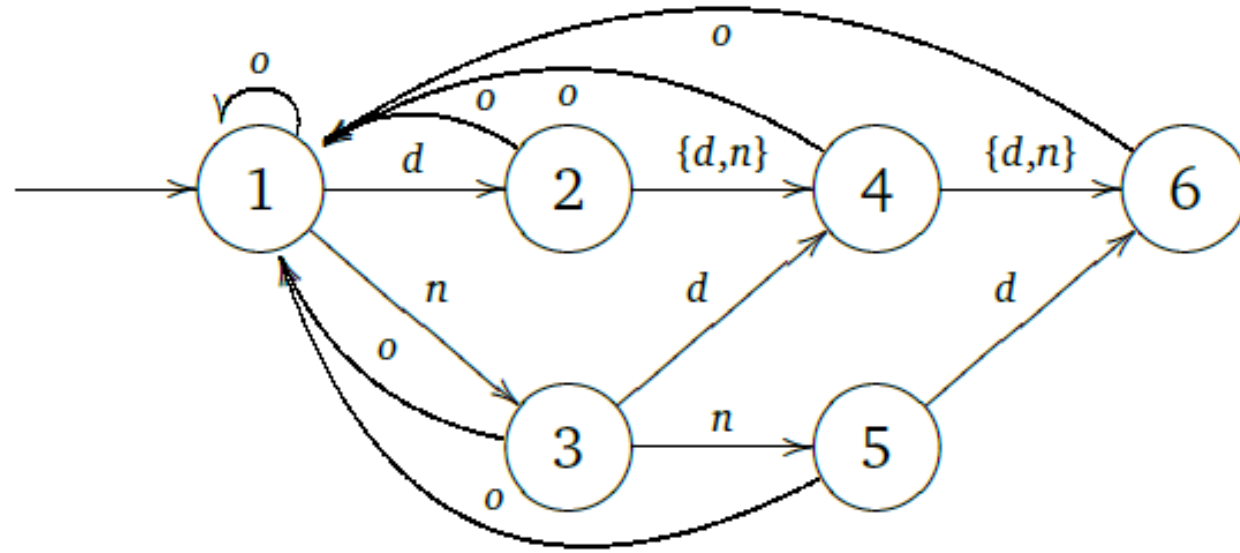
- Establecer las distintas posibilidad para **10 días** de un trabajo por turno de **7 enfermeros/as** de un hospital
- Los turnos pueden ser de día (**d**), noche(**n**), o día libre (**o**,off): {d,n,-}
- Por cada período de 3 días, un trabajador debe tener al menos un día libre.
- No se pueden hacer más de 2 turnos de noche seguidos
- (Definimos el Autómata Finito Determinista)
- Cada día debe tener
 - 3 turnos de día, **req_day**
 - 2 turnos de noche, **req_night**
- Cada trabajador debe hacer al menos
 - 2 turnos de noche, **min_night**

		día: 1..10									
Trabajador: 1..7	1	d	d	d	-	d	d	d	-	n	n
	2	d	d	d	-	d	d	d	-	n	n
	3	d	d	-	d	d	n	-	n	d	d
	4	n	n	-	d	n	d	-	d	d	d
	5	n	-	d	n	n	-	d	d	d	-
	6	-	n	n	d	-	n	n	d	-	d
	7	-	-	n	n	-	-	n	n	-	-

AFD

AP (JQG)

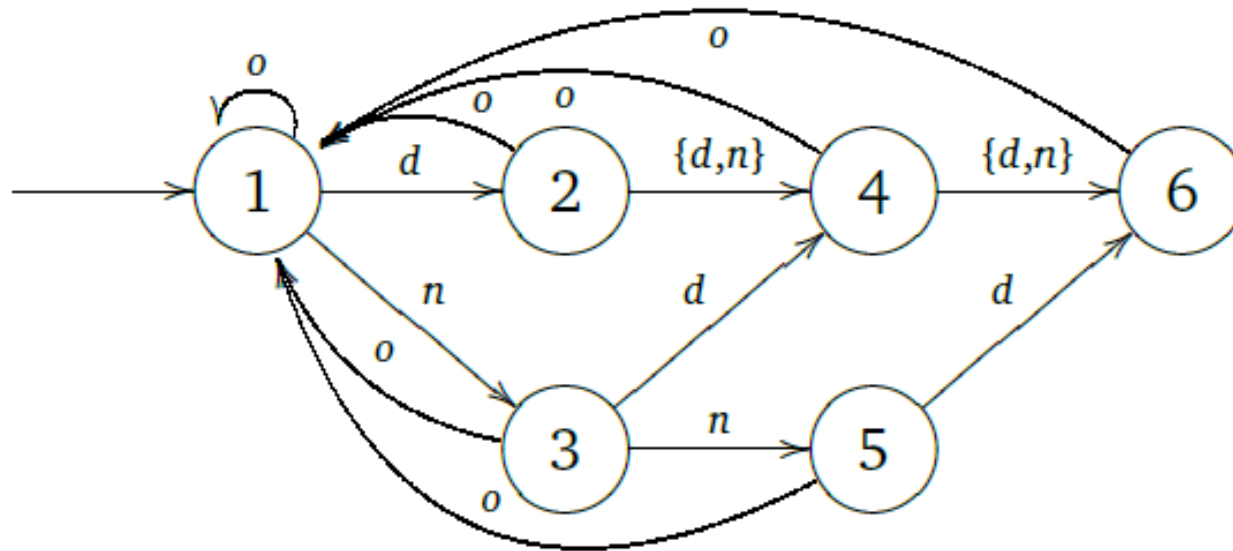
AFD



AFD

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

It constrains that the sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $\langle 1..Q, 1..S \rangle$ to $0..Q$) and initial state $q0$ (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state.



	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

0-> Estado de error

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

It constrains that the sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $\langle 1..Q, 1..S \rangle$ to $0..Q$) and initial state $q0$ (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state.

	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

```
int: S = 3;
set of int: SHIFTS = 1..S;
int: Q = 6;
set of int: STATES = 1..Q;

int: q0 = 1;
```

-> F

estados de aceptación únicamente, en este caso particular son todos, pero no tendría que ser así siempre

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

It constrains that the sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $\langle 1..Q, 1..S \rangle$ to $0..Q$) and initial state $q0$ (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state.

	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

```
array[STATES,SHIFTS] of int: t =  
    [| 2, 3, 1    % state 1  
     | 4, 4, 1    % state 2  
     | 4, 5, 1    % state 3  
     | 6, 6, 1    % state 4  
     | 6, 0, 1    % state 5  
     | 0, 0, 1|]; % state 6
```

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

It constrains that the sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $\langle 1..Q, 1..S \rangle$ to $0..Q$) and initial state $q0$ (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state.

```
array[NURSES,DAYS] of var SHIFTS: roster;  
constraint forall(i in NURSES)(  
    regular([roster[i,j] | j in DAYS], Q, S, t, q0, STATES)  
);
```

Código completo. Parámetros

```
NURSE ≡  
% Simple nurse rostering  
include "regular.mzn";  
int: num_nurses;  
set of int: NURSES = 1..num_nurses;  
int: num_days;  
set of int: DAYS = 1..num_days;  
int: req_day;  
int: req_night;  
int: min_night;  
  
int: S = 3;  
set of int: SHIFTS = 1..S;  
int: d = 1; int: n = 2; int: o = 3;  
array[SHIFTS] of string: name = ["d", "n", "-"];  
  
int: Q = 6; int: q0 = 1; set of int: STATES = 1..Q;  
array[STATES, SHIFTS] of int: t =  
    [| 2, 3, 1    % state 1  
     | 4, 4, 1    % state 2  
     | 4, 5, 1    % state 3  
     | 6, 6, 1    % state 4  
     | 6, 0, 1    % state 5  
     | 0, 0, 1|]; % state 6
```

Variable de decisión

```
array[NURSES,DAYS] of var SHIFTS: roster;
```

Restricciones

```
constraint forall(j in DAYS)(  
    sum(i in NURSES)(bool2int(roster[i,j] == d)) == req_day /\  
    sum(i in NURSES)(bool2int(roster[i,j] == n)) == req_night  
);  
constraint forall(i in NURSES)(  
    regular([roster[i,j] | j in DAYS], Q, S, t, q0, STATES) /\  
    sum(j in DAYS)(bool2int(roster[i,j] == n)) >= min_night  
);  
  
solve satisfy;
```