Modelamiento e Implementación de CSPs

Taller 1

Juan Sebastián Estupiñán Cifuentes (201924846)
Juan David Soto Carmona (201958813)

Programación por Restricciones

Juan Francisco Díaz, Ph.D

Escuela de Ingeniería de Sistemas y Computación
Facultad de Ingeniería
Universidad del Valle

16 de abril de 2024

Tabla de contenidos

1. Sudoku

1.1. Modelo

1.1.1. Parámetros

$sudoku\_inicial$: es una matriz 9x9 que representa el estado inicial del Sudoku. Los valores diferentes de cero en esta matriz indican las celdas predefinidas del tablero inicial.

$sudoku\_inicial_{i,j}: i \in \{1, ..., 9\}, j \in \{1, ..., 9\}$.

1.1.2. Variables

$sudoku$: también es una matriz 9x9, pero esta vez contiene variables de decisión. Estas variables representan los valores que deben asignarse a las celdas del Sudoku para completar el tablero.

Dominio: $sudoku_{i,j}: \forall i \in \{1, ..., 9\} \land \forall j \in \{1, ..., 9\}, sudoku_{i,j} \in \{1, ..., 9\}$.

1.1.3. Restricciones

- **Asignación de valores iniciales:** la primera restricción garantiza que las celdas predefinidas del Sudoku (aquellas con valores diferentes de cero en $sudoku\_inicial$), mantengan sus valores originales.

  $$\forall i \in \{1, ..., 9\} \land \forall j \in \{1, ..., 9\}, sudoku\_inicial_{i,j} \neq 0 \rightarrow sudoku_{i,j} = sudoku\_inicial_{i,j}$$

- **Filas y columnas:** aseguran que en cada fila y en cada columna de la cuadrícula, no haya números repetidos.

  $$\forall i \in \{1, ..., 9\} \land \forall x, y \in \{1, ..., 9\}, x \neq y, sudoku_{i,x} \neq sudoku_{i,y}$$

  $$\forall j \in \{1, ..., 9\} \land \forall x, y \in \{1, ..., 9\}, x \neq y, sudoku_{x,j} \neq sudoku_{y,j}$$

- **Cuadrículas 3x3:** por último, esta restricción garantiza que en cada subcuadrícula de 3x3 no haya números repetidos.

  $$\forall i \in \{1, 4, 7\} \land \forall j \in \{1, 4, 7\} \land \forall k, a \in \{0, 1, 2\} \land \forall l, b \in \{0, 1, 2\}, k \neq a, l \neq b,$$

  $$sudoku_{i+k,j+l} \neq sudoku_{i+a,j+b}$$

1.1.4. Anotaciones de búsqueda

$solve :: int\_search(sudoku, first\_fail, indomain\_random)\ satisfy;$

Elige primero las celdas con menos opciones ya que en un sudoku son estas celdas las primeras en colocarse. Luego asigna valores de forma aleatoria ya que la casilla podría ser cualquier valor del dominio (Ver Figura 1).

Se comparó con $solve :: int\_search(sudoku, smallest, indomain\_random)\ satisfy;$

Aquí se eligen primero las celdas con menor valor en su dominio independientemente de la cantidad de opciones, haciendo que aumente la ramificación (Ver Figura 2).

1.2. Árboles de búsqueda

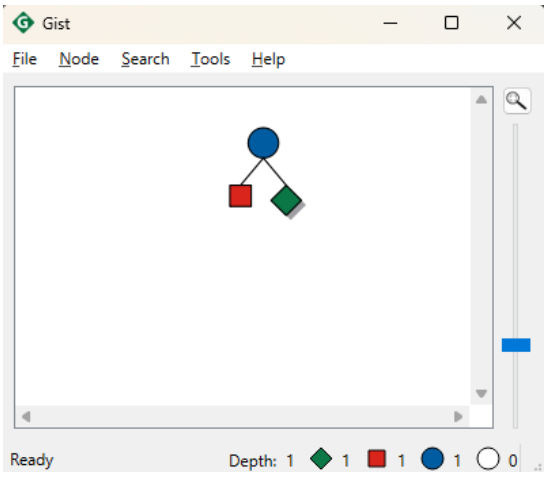

Figura 1: Árbol de búsqueda de la prueba 1 - sudoku.

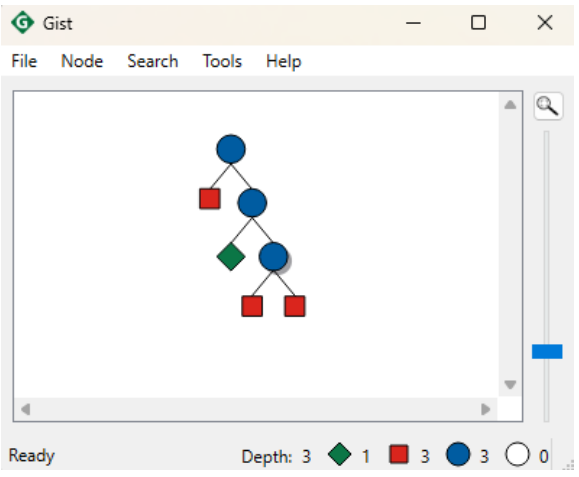

Figura 2: Árbol de búsqueda de la prueba 1 - sudoku.

1.3. Pruebas

1.3.1. Prueba 1

Entrada:

```
sudoku_inicial = [|
3,2,0,0,0,7,0,0,1|
0,7,0,0,0,2,0,0,9|
0,8,0,0,0,0,0,0,4|
0,0,0,0,0,0,1,0,6|
0,0,0,5,4,0,0,8,0|
```

Salida:

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

0,4,0,0,0,1,0,0,0| 0,5,0,0,0,0,0,6,0| 6,9,0,0,8,0,7,0,0| 2,0,0,0,6,0,0,0,5|];	7 5 3 2 1 9 4 6 8 6 9 4 3 8 5 7 1 2 2 1 8 7 6 4 3 9 5

1.3.2. Prueba 2

Entrada:	Salida:
sudoku_inicial = [| 4,0,8,0,7,0,0,5,2| 0,0,0,9,0,0,1,0,7| 0,0,6,0,0,0,8,0,0| 0,0,0,0,5,0,0,0,6| 5,0,4,0,0,0,0,0,8| 0,8,0,2,0,7,0,0,0| 0,0,0,0,0,0,0,0,0| 9,0,0,0,0,0,3,0,0| 0,2,1,3,0,5,9,0,0|];	4 9 8 1 7 3 6 5 2 2 5 3 9 8 6 1 4 7 1 7 6 5 2 4 8 3 9 7 3 2 8 5 1 4 9 6 5 1 4 6 3 9 7 2 8 6 8 9 2 4 7 5 1 3 3 4 5 7 9 8 2 6 1 9 6 7 4 1 2 3 8 5 8 2 1 3 6 5 9 7 4

1.3.3. Prueba 3

Entrada:	Salida:
sudoku_inicial = [| 0,7,9,0,0,0,2,0,0| 0,8,0,0,4,5,0,3,0| 0,0,0,0,3,0,1,0,8| 0,2,1,0,0,0,0,0,0| 4,0,6,0,0,3,0,1,0| 0,5,0,0,0,0,0,2,0| 0,0,0,0,0,0,0,0,0| 0,4,0,0,9,0,6,0,0| 0,0,5,6,7,2,0,0,1|];	3 7 9 1 6 8 2 4 5 1 8 2 9 4 5 7 3 6 5 6 4 2 3 7 1 9 8 8 2 1 7 5 9 3 6 4 4 9 6 8 2 3 5 1 7 7 5 3 4 1 6 8 2 9 6 1 7 3 8 4 9 5 2 2 4 8 5 9 1 6 7 3 9 3 5 6 7 2 4 8 1

1.4. Conclusiones

El modelo para resolver un sudoku funciona correcta y eficazmente, dando una solución única para una entrada válida. El tiempo de ejecución es mínimo debido a que un sudoku tiene una única solución y las restricciones permiten encontrarla rápidamente.

2. Kakuro

2.1. Modelo

2.1.1. Parámetros

n: representa la longitud del tablero, excluyendo la primera columna.

m: representa el ancho del tablero, excluyendo la primera fila.

$kakuro$: un arreglo bidimensional que indica las coordenadas de las celdas disponibles para poner números que deben sumarse para obtener el resultado correspondiente en la última celda de esa fila.

Nota: la coordenada (0,0) sirve para rellenar la fila cuando no hay más números disponibles para sumar.

$blanks$: un arreglo de enteros que indica las coordenadas de las celdas deshabilitadas (casillas pintadas o con resultados a sumar) en el tablero.

2.1.2. Variables

x: un arreglo bidimensional de variables enteras entre 0 y 9 que representa los valores de las celdas del tablero. El cero significa que esa casilla no está disponible (casillas pintadas o con los resultados de las sumas).

Dominio: $\forall i \in \{1, ..., n\} \land \forall j \in \{1, ..., m\}, x_{i,j} \in \{0, ..., 9\}$.

2.1.3. Restricciones

- **Asignación de valores iniciales:** asignar ceros en las posiciones de $blanks$ en x.

  $Sea\ b = |blanks| \div 2,$

  $\forall i \in \{1, ..., b\}, x_{b_{2*(i-1)+1}, b_{2*(i-1)+2}} = 0$

- **Suma:** garantiza que para cada fila o columna en el tablero, la suma de los valores de las celdas correspondientes sea igual al valor objetivo especificado.

  $Sea\ kakuro_{i,j}: k, l \in Z, i \in \{1, ..., k\} \land j \in \{1, ..., l\},$

  $a = (l - 1) \div 2,$

  $\forall p \in \{1, ..., k\}, \sum_{i=1}^{a} (kakuro_{p,2*(i-1)+1} > 0 \land x_{kakuro_{p,2*(i-1)+1}, kakuro_{p,2*(i-1)+2}}) = kakuro_{p,l}$

- **Valores diferentes:** asegurar que los valores que componen una suma sean distintos entre sí.

$$Sea\ kakuro_{i,j}: k, l\ \in Z, i \in \{1, ..., k\} \wedge j \in \{1, ..., l\},$$

$$a = (l - 1) \div 2,$$

$$\forall z, y \in \{1, ..., k\}, z \neq y, \forall i \in \{1, ..., a\}, (kakuro_{z,2*(i-1)+1} > 0 \wedge kakuro_{y,2*(i-1)+1} > 0) \rightarrow$$

$$x_{kakuro_{z,2*(i-1)+1},kakuro_{z,2*(i-1)+2}} \neq x_{kakuro_{y,2*(i-1)+1},kakuro_{y,2*(i-1)+2}}$$

- **Valores mayores a cero:** asegurar que los valores que componen una suma sean distintos a cero.

$$Sea\ kakuro_{i,j}: k, l\ \in Z, i \in \{1, ..., k\} \wedge j \in \{1, ..., l\},$$

$$a = (l - 1) \div 2,$$

$$\forall p \in \{1, ..., k\}, \forall i \in \{1, ..., a\}, kakuro_{p,2*(i-1)+1} > 0 \rightarrow x_{kakuro_{p,2*(i-1)+1},kakuro_{p,2*(i-1)+2}} > 0$$

2.1.4. Anotaciones de búsqueda

$solve :: int\_search(x,\ first\_fail,\ indomain\_min)\ satisfy;$

Elige primero las celdas con menos opciones ya que al igual que en un sudoku, son estas celdas las primeras en colocarse. Luego el orden de la asignación puede ser cualquiera (Ver Figura 3).

Se comparó con $solve :: int\_search(x,\ smallest,\ indomain\_min)\ satisfy;$

Aquí se eligen primero las celdas con menor valor en su dominio independientemente de la cantidad de opciones, haciendo que aumente la ramificación (Ver Figura 4).

Se utiliza la estrategia de búsqueda int_search, seleccionando las variables de forma que se prueben primero aquellas que tienen menor dominio, y dentro de ellas, se elige el valor más pequeño disponible.

2.2. Árboles de búsqueda

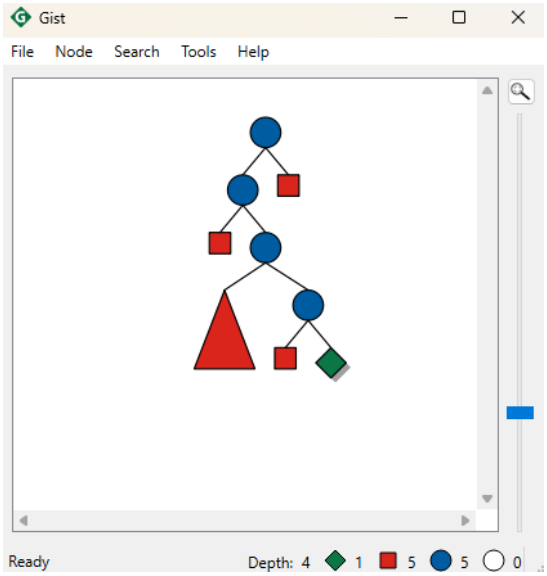

Figura 3: Árbol de búsqueda de la prueba 1 - kakuro.

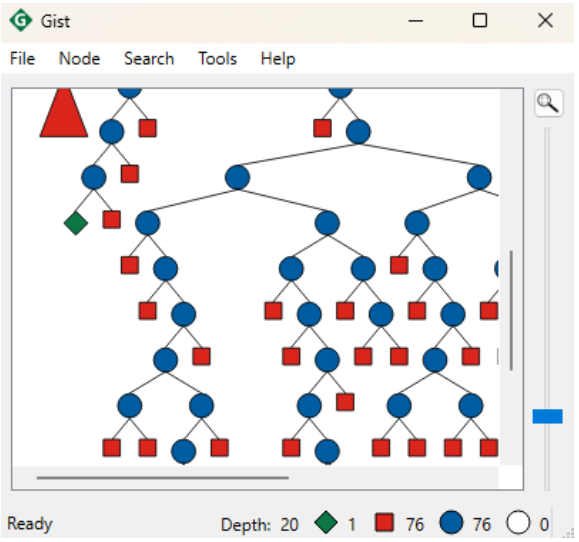

Figura 4: Árbol de búsqueda de la prueba 1 - kakuro.

2.3. Pruebas

2.3.1. Prueba 1

Entrada:

```
n=7;
m=7;

blanks = [
 1,3, 1,4,
 2,3,
 3,6, 3,7,
 4,1, 4,4, 4,7,
 5,1, 5,2,
 6,5,
 7,4, 7,5
 ];

kakuro = [|
 1,1, 1,2, 0,0, 0,0, 0,0,  16|
 1,5, 1,6, 1,7, 0,0, 0,0,  24|
 2,1, 2,2, 0,0, 0,0, 0,0,  17|
 2,4, 2,5, 2,6, 2,7, 0,0,  29|
 3,1, 3,2, 3,3, 3,4, 3,5,  35|
 4,2, 4,3, 0,0, 0,0, 0,0,   7|
 4,5, 4,6, 0,0, 0,0, 0,0,   8|
 5,3, 5,4, 5,5, 5,6, 5,7,  16|
 6,1, 6,2, 6,3, 6,4, 0,0,  21|
 6,6, 6,7, 0,0, 0,0, 0,0,   5|
 7,1, 7,2, 7,3, 0,0, 0,0,   6|
 7,6, 7,7, 0,0, 0,0, 0,0,   3|

 1,1, 2,1, 3,1, 0,0, 0,0,  23|
 1,2, 2,2, 3,2, 4,2, 0,0,  30|
 1,5, 2,5, 3,5, 4,5, 5,5,  27|
 1,6, 2,6, 0,0, 0,0, 0,0,  12|
 1,7, 2,7, 0,0, 0,0, 0,0,  16|
 2,4, 3,4, 0,0, 0,0, 0,0,  17|
 3,3, 4,3, 5,3, 6,3, 7,3,  15|
 4,6, 5,6, 6,6, 7,6, 0,0,  12|
 5,4, 6,4, 0,0, 0,0, 0,0,   7|
 5,7, 6,7, 7,7, 0,0, 0,0,   7|
 6,1, 7,1, 0,0, 0,0, 0,0,  11|
 6,2, 7,2, 0,0, 0,0, 0,0,  10|];
```

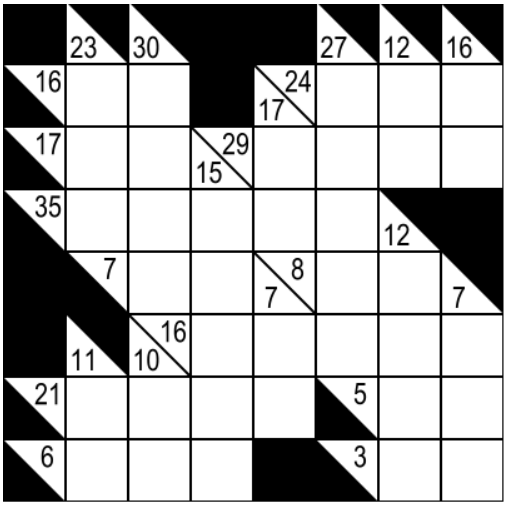

Figura 5: Tablero inicial prueba 1 - kakuro

Salida:

```
9 7 0 0 8 7 9
8 9 0 8 9 5 7
6 8 5 9 7 0 0
0 6 1 0 2 6 0
0 0 4 6 1 3 2
8 9 3 1 0 1 4
3 1 2 0 0 2 1
```

2.3.2. Prueba 2

Entrada:

```
n=5;
m=4;

blanks = [
 1,4,
 2,4,
 3,3, 3,4,
 4,1,
 5,1
];

kakuro = [|
 1,1, 1,2, 1,3, 0,0, 0,0,  9|
 2,1, 2,2, 2,3, 0,0, 0,0,  13|
 3,1, 3,2, 0,0, 0,0, 0,0,  13|
 4,2, 4,3, 4,4, 0,0, 0,0,  7|
 5,2, 5,3, 5,4, 0,0, 0,0,  19|

 1,1, 2,1, 3,1, 0,0, 0,0,  9|
 1,2, 2,2, 3,2, 4,2, 5,2,  34|
 1,3, 2,3, 0,0, 0,0, 0,0,  4|
 4,3, 5,3, 0,0, 0,0, 0,0,  11|
 4,4, 5,4, 0,0, 0,0, 0,0,  3|];
```

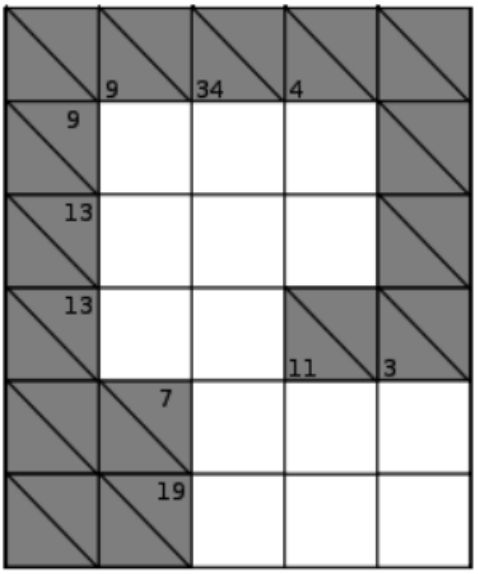

Figura 6: Tablero inicial prueba 2 - kakuro

Salida:

```
2 6 1 0
1 9 3 0
6 7 0 0
0 4 2 1
0 8 9 2
```

2.4. Conclusiones

El modelo para resolver un kakuro funciona correcta y eficazmente al tener una única solución al igual que un sudoku. Acepta cualquier configuración y resuelve tableros cuadrados o rectangulares.

3. Secuencia mágica

3.1. Modelo

3.1.1. Parámetros

n: representa la longitud de la secuencia mágica.

3.1.2. Variables

x: arreglo que representa la secuencia mágica.

Dominio: $\forall i \in \{0, ..., n-1\},\ x_i \geq 0 \ \wedge \ x_i \leq n-1$

3.1.3. Restricciones

- x_i es un entero entre 0 y n − 1:

  $\forall i \in \{0, ..., n-1\},\ x_i \geq 0 \ \wedge \ x_i \leq n-1$

- El número i ocurre exactamente x_i veces en la secuencia:

  $\forall i \in \{0, ..., n-1\},\ x_i = \sum_{j=0}^{n-1} (x_j = i)$

- Restricciones redundantes:

  - La suma de todo los elementos x_i de la secuencia mágica es igual a la longitud n de la secuencia:

    $$\sum_{i=0}^{n-1} x_i = n$$

  - La suma de cada x_i multiplicado por el índice i anterior es igual a 0**.**

    $$\sum_{i=0}^{n-1} x_i * (i-1) = 0$$

    Las restricciones son redundantes porque se deducen de la condición que establece que el número i ocurre exactamente x_i veces en la secuencia.

Dado de que cada x_i debe ser igual a la suma de las veces en que aparece i en la secuencia, la suma de todos los x_i debe ser n, lo que da lugar a la primera condición redundante:

$$\sum_{i=0}^{n-1} x_i = n \,.$$

Además, la suma de los índices multiplicados por la cantidad de veces que aparece cada número también es igual a n**:**

$$\sum_{i=0}^{n-1} x_i * i = n \,,$$

Como,

$$\sum_{i=0}^{n-1} x_i = \sum_{i=0}^{n-1} x_i * i \,,$$

$$\sum_{i=0}^{n-1} x_i * i - \sum_{i=0}^{n-1} x_i = 0$$

o, de otra forma:

$$\sum_{i=0}^{n-1} x_i * (i - 1) = 0$$

Lo que da lugar a la segunda restricción redundante.

Al añadir restricciones redundantes al modelo se mejoró el rendimiento en la mayoría de las pruebas, ya que al poner a disposición del solucionador más información, se reduce el espacio de búsqueda más rápido.

En la siguiente tabla se muestran los tiempos de ejecución de ambas implementaciones del modelo. Se puede observar como el tiempo de ejecución se reduce al añadir las condiciones redundantes, observe por ejemplo, la prueba con $n = 500$ en la que el tiempo de ejecución se reduce casi a la mitad:

	Tiempo de ejecución	
Prueba	**Sin condiciones redundantes (ms)**	**Con condiciones redundantes (ms)**
n = 7	144	144
n = 11	280	159

n = 20	282	175
n = 100	1252	1188
n = 500	44056	23896

3.1.4. Anotaciones de búsqueda

$solve :: int\_search(x, input\_order, indomain\_max)\ satisfy;$

Selecciona las variables en el orden en que están definidas en el modelo, sin embargo este orden no importa mucho, lo que sí importa es el orden de asignación; asignar el valor más grande el dominio puede ayudar a satisfacer la restricción que obliga a que *la suma de todos los valores de* x *debe ser igual a* n, más rápidamente (Ver Figura 7).

Se comparó con $solve :: int\_search(x, input\_order, indomain\_min)\ satisfy;$

Aquí se asignan primero los valores más pequeños del dominio, haciendo que aumente la ramificación (Ver Figura 8).

3.2. Árboles de búsqueda

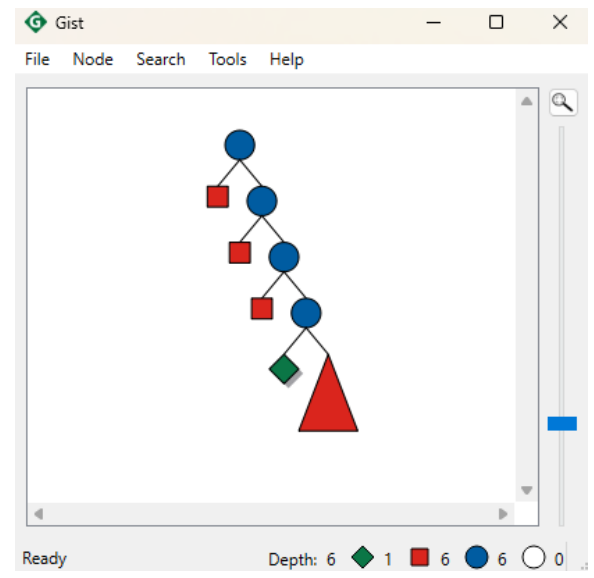

Figura 7: Árbol de búsqueda de la prueba 1 - secuencia mágica.

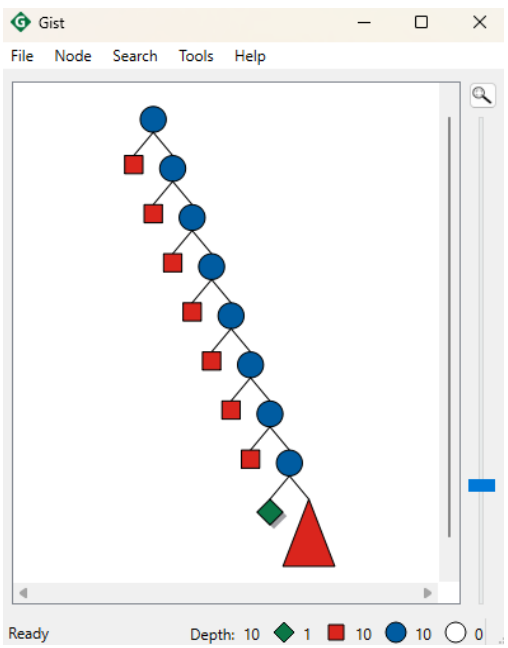

Figura 8: Árbol de búsqueda de la prueba 1 - secuencia mágica.

3.3. Pruebas

3.3.1. Prueba 1

Entrada:	Salida:
`n = 7;`	`7 = [3, 2, 1, 1, 0, 0, 0]`

3.3.2. Prueba 2

Entrada:	Salida:
`n = 11;`	`11 = [7, 2, 1, 0, 0, 0, 0, 1, 0, 0, 0]`

3.3.3. Prueba 3

Entrada:	Salida:
`n = 20;`	`20 = [16, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]`

3.3.4. Prueba 4

Entrada:	Salida:
`n = 100;`	`100 = [96, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]`

3.3.5. Prueba 5

Entrada:	Salida:
`n = 500;`	`500 = [496, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,`

	0, 1, 0, 0, 0]

3.4. Conclusiones

En el problema de la secuencia mágica la inclusión de condiciones redundantes puede disminuir el tiempo de ejecución. Estas condiciones redundantes actúan como restricciones adicionales que limitan aún más el espacio de búsqueda, lo que puede reducir el tiempo de búsqueda de soluciones y mejorar la eficiencia del modelo.

4. Acertijo Lógico

4.1. Modelo

4.1.1. Parámetros

Ninguno.

4.1.2. Variables

$Oscar$: representa la edad de Oscar, $Oscar \in \{24, 25, 26\}$.
$Dario$: representa la edad de Dario, $Dario \in \{24, 25, 26\}$.
$Juan$: representa la edad de Juan, $Juan \in \{24, 25, 26\}$.

$Lopez$: representa la edad de Lopez, $Lopez \in \{24, 25, 26\}$.
$Garcia$: representa la edad de Garcia, $Garcia \in \{24, 25, 26\}$.
$Gonzalez$: representa la edad de Gonzalez, $Gonzalez \in \{24, 25, 26\}$.

$Clasica$: representa la edad de la persona a la que le gusta la música clásica, $Clasica \in \{24, 25, 26\}$.
$Jazz$: representa la edad de la persona a la que le gusta la música Jazz, $Jazz \in \{24, 25, 26\}$.
Pop: representa la edad de la persona a la que le gusta la música Pop, $Pop \in \{24, 25, 26\}$.

$nombres$: arreglo donde se ponen los nombres de las tres personas,

$\forall i \in \{1, 2, 3\},\ nombres_i \in \{24, 25, 26\}$

$apellidos$: arreglo donde se ponen los apellidos de las tres personas,

$\forall i \in \{1, 2, 3\},\ apellidos_i \in \{24, 25, 26\}$

$musicas$: arreglo donde se pone la música favorita de cada una de las tres personas,

$\forall i \in \{1, 2, 3\},\ musicas_i \in \{24, 25, 26\}$

4.1.3. Restricciones

Se añadieron restricciones para que los valores de cada arreglo sean diferentes entre sí:

$\forall x \in nombres \land \forall y \in nombres, x \neq y$

$\forall x \in apellidos \land \forall y \in apellidos, x \neq y$

$\forall x \in musicas \land \forall y \in musicas, x \neq y$

Asimismo se agregaron restricciones para cumplir con las siguientes pistas del acertijo:

- Juan es más viejo que González, a quien le gusta la música clásica.
  $Juan > Gonzalez \land Gonzalez = Clasica$
- El fan de la música pop, que no es García, no tiene 24.
  $Pop \neq Garcia \land Pop \neq 24$
- Oscar, quién no es López, tiene 25.
  $Oscar \neq Lopez \land Oscar = 25$
- La música favorita de Darío no es el jazz.
  $Dario \neq Jazz$
- Las edades de los 3 amigos están entre 24 y 26 años.
  $24 \leq Juan \leq 26$
  $24 \leq Oscar \leq 26$
  $24 \leq Dario \leq 26$

4.1.4. Anotaciones de búsqueda

$solve\ satisfy$: Esta declaración indica que se busca una solución satisfactoria para el problema del acertijo, es decir, una solución donde se cumplan todas las restricciones (Ver Figura 9).

4.2. Árboles de búsqueda

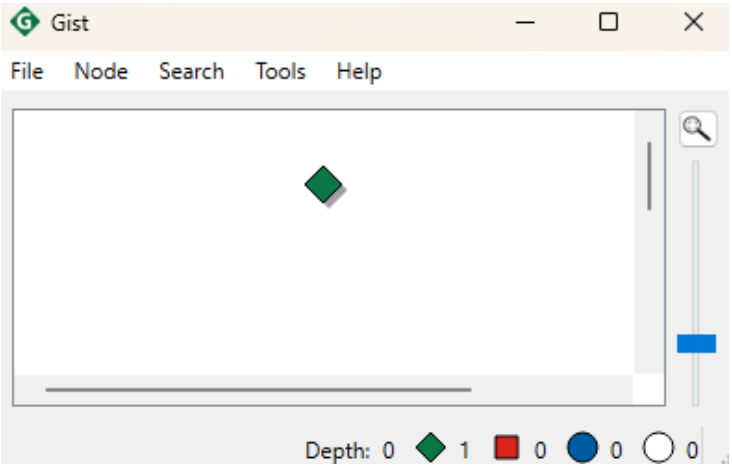

Figura 9: Árbol de búsqueda - acertijo lógico.

4.3. Pruebas

Entrada:	Salida:
No hay entrada debido a qué el programa no recibe ningún parámetro y tiene una solución única.	[Oscar, Dario, Juan] - [25, 24, 26] [Lopez, Garcia, Gonzalez] - [26, 25, 24] [Clasica, Jazz, Pop] - [24, 25, 26]

La solución es entonces:

Dario Gonzalez, 24, clásica

Oscar Garcia, 25, jazz

Juan Lopez, 26, pop

4.4. Conclusiones

El modelo encuentra una solución única y satisfactoria que cumple con todas las restricciones impuestas. Es el problema más claro de todos debido a que las restricciones se escriben en lenguaje natural sin tener que usar condiciones avanzadas del lenguaje o iteraciones más complicadas.

La forma en que se abordó el problema fue definiendo todas las variables asignándoles valores que representaban la edad y de esa forma saber que por ejemplo, el nombre, el apellido y la música de la persona de 25 años eran los que tenían valores 25 en cada una de las tres variables.

5. Ubicación de personas en una reunión

5.1. Modelo

5.1.1. Parámetros

$personas$: arreglo que contiene los nombres de las personas

$next$: arreglo multidimensional que indica qué personas deben ir juntas en la fila

$|next| \ mod \ 2 \ = 0$

$separate$: arreglo multidimensional que indica qué personas deben ir separadas en la fila

$|separate| \ mod \ 2 \ = 0$

$distance$: arreglo multidimensional que indica qué personas deben ir separadas en la fila señalando el máximo de personas que pueden estar ubicadas entre ellas.

$|distance| \ mod \ 3 \ = 0$

5.1.2. Variables

$fila$: arreglo que indica en qué posición se debe colocar a cada persona.

$\text{Sea } n = |personas|, \forall i \in \{1, ..., n\}, \ fila_i \in \{1, ..., n\}$

5.1.3. Restricciones

Restricción para que las parejas de personas indicadas en $next$ aparezcan juntas en $fila$.

$Sea \ m = |next| \div 2, \ n = |personas|$

$\forall j \in \{1, ..., m\}, \forall i \in \{1, ..., n\}, fila_i = next_{j,1} \rightarrow (i < n \wedge fila_{i+1} = next_{j,2}) \vee (i > 1 \wedge fila_{i-1} = next_{j,2})$

Restricción para que las parejas de personas indicadas en $separate$ aparezcan separadas en $fila$.

$Sea \ m = |separate| \div 2, \ n = |personas|$

$\forall j \in \{1, ..., m\}, \forall i \in \{1, ..., n\}, fila_i = separate_{j,1} \rightarrow (i < n \wedge fila_{i+1} \neq separate_{j,2}) \wedge$

$(i > 1 \wedge fila_{i-1} \neq separate_{j,2})$

Restricción para que las parejas de personas indicadas en $distance$ aparezcan separadas en $fila$.

$Sea\ m = |distance| \div 3,\ n = |personas|$

$\forall j \in \{1, \ldots, m\}, \forall i \in \{1, \ldots, n\}, fila_i = distance_{j,1} \rightarrow (i < n \wedge fila_{i+1} \neq distance_{j,2}) \wedge$

$(i > 1 \wedge fila_{i-1} \neq distance_{j,2})$

Restricción para que las parejas de personas indicadas en $distance$ no puedan aparecer separadas por más del número máximo de personas indicadas en $distance$, en $fila$**.**

$Sea\ m = |distance| \div 3,\ n = |personas|$

$\forall j \in \{1, \ldots, m\}, \forall i \in \{1, \ldots, n\}, fila_i = distance_{j,1} \rightarrow$

$(\forall k \in \{i + distance_{j,3} + 2, \ldots, n\}, fila_k \neq distance_{j,2}) \wedge$

$(\forall l \in \{1, \ldots, i - distance_{j,3} - 2\}, fila_k \neq distance_{j,2})$

Restricción para que los valores en $fila$ sean diferentes entre sí.

$\forall x \in fila \wedge \forall y \in fila,\ x \neq y$

5.1.4. Anotaciones de búsqueda

$solve :: int\_search(fila,\ first\_fail,\ indomain\_min)\ satisfy;$

Elegir primero las variables con menor dominio significa posicionar primero a las personas que están más condicionadas, podando el árbol de búsqueda eficazmente.(Ver Figura 10).

Se comparó con $solve :: int\_search(x,\ input\_order,\ indomain\_min)\ satisfy;$

Aquí se eligen las variables en el orden en que están definidas en el modelo, lo cuál es ineficiente ya que para el problema elegir las variables con menos opciones ayuda a encontrar las soluciones más rápido (Ver Figura 11).

5.2. Árboles de búsqueda

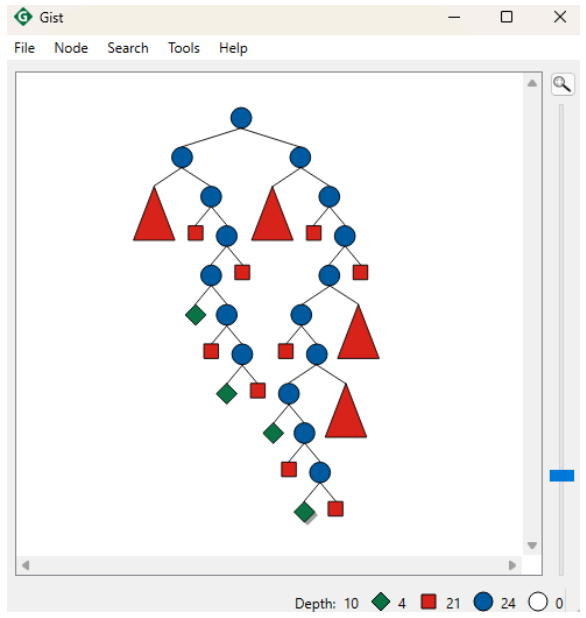

Figura 10: Árbol de búsqueda de la prueba 1 - ubicación de personas en una reunión.

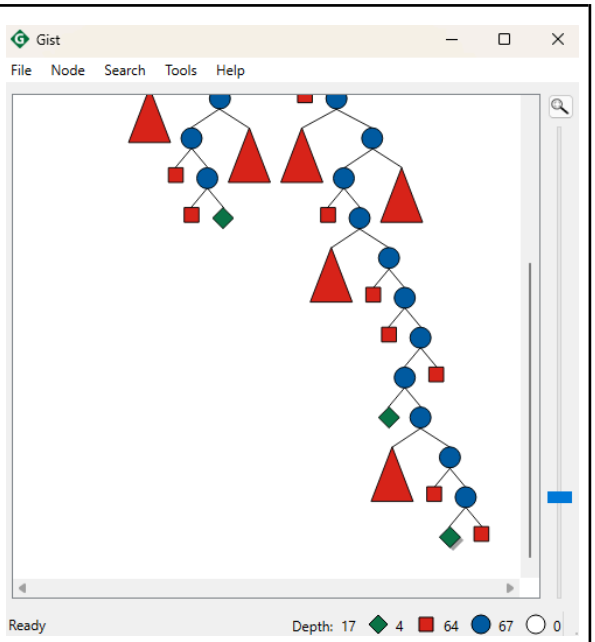

Figura 11: Árbol de búsqueda de la prueba 1 - ubicación de personas en una reunión.

5.3. Pruebas

5.3.1. Prueba 1

Entrada:

```
personas =
["jose","julian","diana","juan","david","claudia","edgar","ma
rio"];
next = [|4, 6| 8, 4| 1, 8| 3, 5|];
separate = [|6, 5|];
distance = [|1, 6, 3| 7, 5, 1|];
```

Salida:

```
personas = [julian, jose, mario, juan, claudia, edgar, diana,
david]
next = [ juan y claudia, mario y juan, jose y mario, diana y
david,]
separate = [ claudia y david,]
distance = [ jose y claudia, M=3, edgar y david, M=1,]
```

5.3.2. Prueba 2

Entrada:

```
personas =
["jose","julian","diana","juan","david","claudia","edgar","ma
rio"];
next = [|5, 8| 5, 2| 2, 4| 4, 6|];
separate = [|4, 8| 3, 5|];
distance = [|1, 2, 3|];
```

Salida:

```
personas = [claudia, juan, julian, david, mario, jose, diana,
edgar]
next = [ david y mario, david y julian, julian y juan, juan y
claudia,]
separate = [ juan y mario, diana y david,]
distance = [ jose y julian, M=3,]
```

5.3.3. Prueba 3

Entrada:

```
personas =
["marta","pedro","camila","andrea","juan","sara","lina"];
next = [|1, 2| 2, 3| 3, 4| 5, 6| 6, 7| 7, 1|];
separate = [|6, 1| 6, 4|];
distance = [|7, 5, 3| 1, 4, 2|];
```

Salida:

```
personas = [andrea, camila, pedro, marta, lina, sara, juan]
next = [ marta y pedro, pedro y camila, camila y andrea, juan
y sara, sara y lina, lina y marta,]
separate = [ sara y marta, sara y andrea,]
distance = [ lina y juan, M=3, marta y andrea, M=2,]
```

5.4. Conclusiones

Para este problema pueden existir una o varias soluciones así como también puede no haber ninguna solución; para ambos casos, el programa funciona a la perfección gracias a las restricciones impuestas.

6. Construcción de un rectángulo

6.1. Modelo

6.1.1. Parámetros

$cuadrados$: arreglo donde cada valor representa la medida del lado de un cuadrado.
$ancho$: el ancho del rectángulo.
$alto$: el alto del rectángulo.

6.1.2. Variables

x: representa la posición en el eje x de cada cuadrado para formar el rectángulo.

$\text{Sea } n = |cuadrados|, \forall i \in \{1, ..., n\}, \ x_i \in Z$

y: representa la posición en el eje y de cada cuadrado para formar el rectángulo.

$\text{Sea } n = |cuadrados|, \forall i \in \{1, ..., n\}, \ y_i \in Z$

6.1.3. Restricciones

Restricción que verifica que la posición del cuadrado en x sea mayor a cero y menor al $ancho$ menos la longitud del lado del cuadrado.

$\text{Sea } n = |cuadrados|, \forall i \in \{1, ..., n\}, x_i \geq 0 \wedge x_i \leq ancho - cuadrados_i$

Restricción que verifica que la posición del cuadrado en y sea mayor a cero y menor al $alto$ menos la longitud del lado del cuadrado.

$\text{Sea } n = |cuadrados|, \forall i \in \{1, ..., n\}, y_i \geq 0 \wedge y_i \leq alto - cuadrados_i$

Restricción que asegura que la suma del área de todos los cuadrados sea igual al área del rectángulo.

$$Sea\ n = |cuadrados|, \sum_{i=1}^{n} (cuadrados_i * cuadrados_i) = ancho * alto$$

Restricción que verifica que ningún cuadrado se sobreponga a otro.

$$solapar(i,j) = ((x_i \leq x_j \ \wedge \ x_j < x_i + cuadrados_i) \ \vee \ (x_j \leq x_i \ \wedge \ x_i < x_j + cuadrados_j))$$

$$\wedge \ ((y_i \leq y_j \ \wedge \ y_j < y_i + cuadrados_i) \ \vee \ (y_j \leq y_i \ \wedge \ y_i < y_j + cuadrados_j))$$

$\text{Sea } n = |cuadrados|, \forall i, j \in \{1, ..., n\}, i < j, \neg solapar(i,j)$

Restricción que establece un orden lógico para los valores de x y y.

$$orden(i,j) = (cuadrados_i = cuadrados_j) \rightarrow (x_i < x_j \ \vee \ (x_i = x_j \ \wedge \ y_i < y_j))$$

$$\text{Sea } n = |cuadrados|, \forall i,j \in \{1, ..., n\}, i < j, \ orden(i,j)$$

6.1.4. Anotaciones de búsqueda

$$solve :: int\_search(x, \ first\_fail, \ indomain\_split) \ satisfy;$$

En este problema, donde las variables en x representan las coordenadas horizontales de los cuadrados, seleccionar primero las variables con un dominio más pequeño puede ayudar a reducir la ramificación del árbol de búsqueda y a encontrar soluciones más rápidamente.

6.2. Árboles de búsqueda

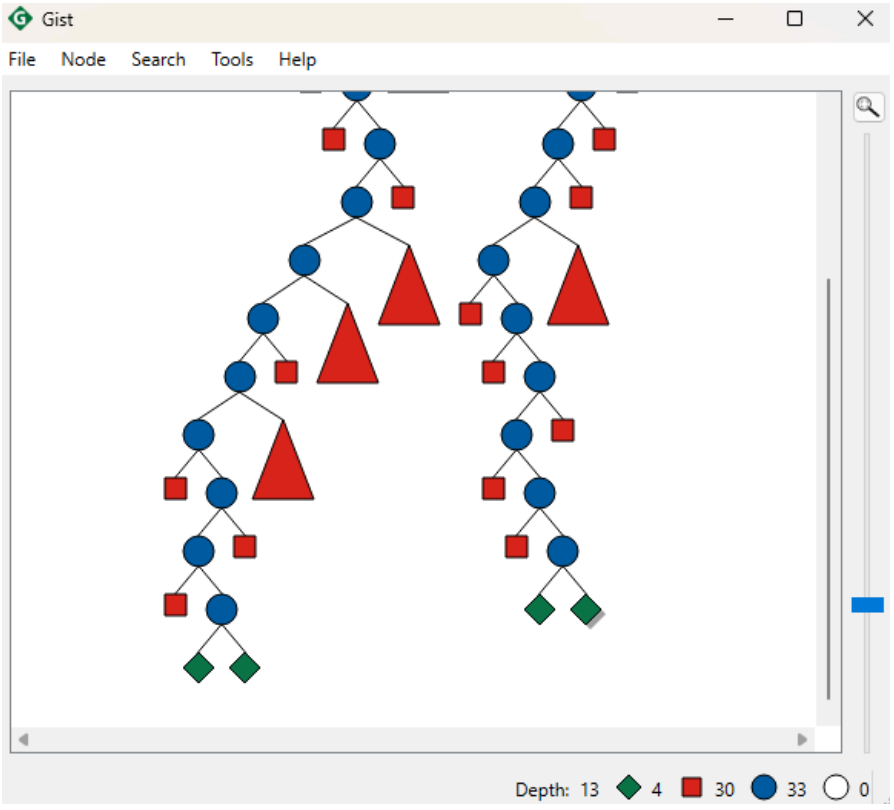

Figura 12: Árbol de búsqueda de la prueba 1 - construcción de un rectángulo.

6.3. Pruebas

6.3.1. Prueba 1

Entrada:

```
cuadrados = [3,2,2,1,1,1];
ancho = 5;
alto = 4;
```

Salida:

```
   3       2       2       1       1       1
[ 0,0 ][ 3,0 ][ 3,2 ][ 0,3 ][ 1,3 ][ 2,3 ]
```

6.4. Conclusiones

La implementación del modelo en MiniZinc ha sido adecuada para resolver el problema de la construcción de un rectángulo. La solución obtenida cumple con las restricciones de ancho y alto del rectángulo y la de utilizar todos los cuadrados para formar el rectángulo.

La definición correcta de las demás restricciones, como la de que no puede haber superposición entre cuadrados y la de que todas las coordenadas de los cuadrados deben estar dentro del área del rectángulo, fueron claves para disminuir el árbol de búsqueda y encontrar soluciones correctas.

La posibilidad de definir restricciones y predicados en MiniZinc en un lenguaje matemático sencillo ha facilitado el proceso de modelado y resolución del problema.

El modelo implementado es general lo que permite adaptarse a distintas configuraciones de entrada, como el número y tamaño de los cuadrados, así como las dimensiones deseadas del rectángulo.