Taller 1: Modelamiento e Implementación de CSPs

Programación por Restricciones

Autores:
Brayan Gómez Muñoz - 2310016
Juan José Moreno Jaramillo - 2310038

Escuela de Ingeniería de Sistemas y Computación
Profesor: Robinson Andrey Duque Agudelo
Abril de 2025

Universidad del Valle

Índice

1. Introducción

Este documento tiene como objetivo el desarrollo del Taller 1 sobre modelamiento e implementación de CSPs (Problemas de Satisfacción de Restricciones). Se explorarán diferentes problemas que pueden ser modelados mediante restricciones, con sus respectivas implementaciones en MiniZinc, un lenguaje altamente tipado utilizado en la optimización y resolución de problemas combinatorios, demostrando cómo una adecuada representación de las restricciones permite encontrar soluciones óptimas o factibles de manera eficiente.

Además, se realiza un análisis comparativo de estrategias de búsqueda como firstfail, inputorder, domwdeg, entre otras, lo cual permite evidenciar el impacto que tiene la heurística seleccionada en el tiempo de ejecución y en la calidad de las soluciones encontradas. De esta manera, el informe no solo presenta los modelos desarrollados, sino que también evalúa su comportamiento frente a distintas configuraciones y condiciones. Así, la modelación y las estrategias de búsqueda se convierten en elementos clave para determinar la eficacia de las soluciones encontradas en contextos reales y académicos.

2. Desarrollo del Taller

2.1. Sudoku

El **Sudoku** es un juego de lógica que consiste en completar una cuadrícula de 9×9 celdas con números del 1 al 9, asegurando que no se repitan en ninguna fila, columna ni en los subcuadrantes de 3×3. Es un problema clásico en el ámbito de la inteligencia artificial y la optimización combinatoria.

2.1.1. Parámetros de entrada

Los parámetros de entrada del modelo de Sudoku definen la estructura inicial del tablero y las restricciones necesarias para garantizar una solución válida. Estos son:

- **tamano_subcuadricula** ($\mathbb{N}$): Tamaño de cada subcuadrícula del tablero. En el Sudoku clásico, su valor es 3.
- **tamano_tablero** ($\mathbb{N}$): Tamaño total del tablero, calculado como:

  $$\text{tamano\_subcuadricula} \times \text{tamano\_subcuadricula} = 9$$

  Este parámetro también define el rango de los valores permitidos en cada celda del tablero.
- **valores_iniciales**: Matriz de dimensiones 9×9 (o tamano_tablero$\times$tamano_tablero) que representa el estado inicial del tablero. Cada celda contiene un valor entre 1 y 9 si está preasignada, o 0 si está vacía y debe ser completada por el modelo.
- **rango_valores**: Conjunto derivado que representa los posibles valores que puede tomar una celda:

  $$\text{rango\_valores} = \{1, 2, \ldots, \text{tamano\_tablero}\}$$

- **rango_subcuadricula**: Conjunto auxiliar que representa los índices para iterar sobre subcuadrículas:

  $$\text{rango\_subcuadricula} = \{1, 2, \ldots, \text{tamano\_subcuadricula}\}$$

2.1.2. Variables de decisión

El modelo utiliza una única variable de decisión clave:

- **solucion**: Matriz de dimensiones 9×9 (o tamano_tablero $\times$ tamano_tablero) donde cada celda representa un valor del tablero que debe ser determinado por el modelo. Cada elemento es una variable entera con dominio en el conjunto rango_valores $= \{1, 2, \ldots, 9\}$.

  Esta variable representa la versión final del Sudoku resuelto. Está sujeta a las restricciones que aseguran que cada fila, columna y subcuadrícula contenga todos los números del 1 al 9 exactamente una vez, y que los valores preasignados se mantengan sin cambios.

2.1.3. Restricciones

- **Restricciones de Valores Iniciales**
  Tipo: Restricción Entera

  Los valores predefinidos del tablero deben conservarse en la solución:

$$\forall i, j \in \{1, \ldots, 9\} : \begin{cases} \text{solucion}[i,j] = \text{valores\_iniciales}[i,j] & \text{si valores\_iniciales}[i,j] > 0 \\ \text{solucion}[i,j] \in \{1, \ldots, 9\} & \text{en otro caso} \end{cases} \tag{1}$$

- **Restricciones de Unicidad en Filas**
  Tipo: Restricción global (alldifferent)

  Cada número del 1 al 9 debe aparecer exactamente una vez en cada fila:

$$\forall i \in \{1, \ldots, 9\}, \quad \text{alldifferent}(\text{solucion}[i, 1.,9]) \tag{2}$$

- **Restricciones de Unicidad en Columnas**
  Tipo: Restricción global (alldifferent)

  Cada número del 1 al 9 debe aparecer exactamente una vez en cada columna:

$$\forall j \in \{1, \ldots, 9\}, \quad \text{alldifferent}(\text{solucion}[1.,9, j]) \tag{3}$$

- **Restricciones de Subcuadrículas**
  Tipo: Restricción global (alldifferent)

  Cada subcuadrícula de 3×3 debe contener todos los valores del 1 al 9 sin repeticiones:

$$\forall b_i, b_j \in \{1, 2, 3\}, \quad \text{alldifferent}\left(\{\text{solucion}[3(b_i - 1) + i, 3(b_j - 1) + j] \mid i, j \in \{1, 2, 3\}\}\right) \tag{4}$$

- **Restricciones Redundantes de suma en Filas**
  Las restricciones redundantes de suma ayudan a mejorar la propagación y reducir el espacio de búsqueda. **Tipo:** Restricción lineal

  La suma de los valores en cada fila debe ser 45 (suma de los números del 1 al 9):

$$\forall i \in \{1, \ldots, 9\}, \quad \sum_{j=1}^{9} \text{solucion}[i, j] = 45 \tag{5}$$

- **Restricciones Redundantes de Suma en Columnas**
  Tipo: Restricción lineal

La suma de los valores en cada columna debe ser 45 (suma de los números del 1 al 9):

$$\forall j \in \{1, \ldots, 9\}, \quad \sum_{i=1}^{9} \text{solucion}[i, j] = 45 \tag{6}$$

- **Restricciones Redundantes de Suma en Subcuadrículas**
  Tipo: Restricción lineal

  La suma de los valores en cada subcuadricula debe ser 45 (suma de los números del 1 al 9):

$$\forall b_i, b_j \in \{1, 2, 3\}, \quad \sum_{i=1}^{3} \sum_{j=1}^{3} \text{solucion}[3(b_i - 1) + i, 3(b_j - 1) + j] = 45 \tag{7}$$

2.1.4. Estrategia de Búsqueda

Para resolver el problema, se evaluaron distintas estrategias de búsqueda utilizando diversas heurísticas:

- `first_fail, indomain_min`: Prioriza las variables con dominios más pequeños, reduciendo la cantidad de opciones a explorar.
- `input_order, indomain_min`: Explora las variables en el orden en que aparecen en el modelo, sin aplicar una heurística adicional.
- `smallest, indomain_median`: Selecciona primero las variables con el menor valor asignado en su dominio.

Donde:

- `indomain_min`: Asigna los valores en orden ascendente, intentando primero las soluciones más pequeñas.
- `indomain_median`: Selecciona el valor medio del dominio de cada variable, equilibrando la búsqueda.

El solver fue probado con las siguientes configuraciones:

```
solve :: int_search([solucion[i,j]], first_fail, indomain_min) satisfy;
solve :: int_search([solucion[i,j]], input_order, indomain_min) satisfy;
solve :: int_search([solucion[i,j]], smallest, indomain_median) satisfy;
```

Sin embargo, tras múltiples pruebas, se observó que estas estrategias no siempre reducían significativamente la profundidad del árbol de búsqueda. En particular, la estrategia básica sin heurísticas:

```
solve satisfy;
```

mostró mejores resultados en la mayoría de los casos.

Justificación de la elección La búsqueda básica generó menos profundidad en casi todas las pruebas realizadas. En algunos casos, la diferencia en el número de nodos explorados fue significativa. Esto puede explicarse por los siguientes factores:

- La estrategia sin heurísticas permite que el solver optimice internamente la exploración sin restricciones adicionales.
- En problemas como el Sudoku, donde las restricciones `alldifferent` tienen una fuerte propagación, imponer heurísticas adicionales no siempre es beneficioso.
- Se redujo la cantidad de retrocesos necesarios, optimizando la exploración del árbol de búsqueda.

Estrategia de Búsqueda	Profundidad	■	■	Tiempo (s)
`first_fail, indomain_min`	11	113	113	367msec
`input_order, indomain_min`	12	110	110	268msec
`smallest, indomain_median`	14	91	91	266msec
`básica`	6	9	9	256msec

Cuadro 1: Comparación de estrategias de búsqueda en la resolución de restricciones en la prueba `sudoku9.dzn`.

2.1.5. Detalles importantes de implementación

Restricciones redundantes Se incorporaron sumas obligatorias 45 por fila, columna y subcuadrícula como restricciones adicionales a las reglas básicas del Sudoku. Aunque matemáticamente estas sumas son consecuencia directa de los requisitos de unicidad, su inclusión explícita mejora significativamente el rendimiento del solver. Estas restricciones actúan como verificaciones tempranas que detectan inconsistencias numéricas antes de completar asignaciones, reduciendo el espacio de búsqueda mediante poda anticipada de ramas inviables.

Validación estructural El modelo asume implícitamente que el tablero inicial cumple con las condiciones básicas: valores únicos en las regiones predefinidas y celdas vacías representadas correctamente. Esta validación estructural se garantiza mediante las propias restricciones del problema (alldifferent), que automáticamente descartan configuraciones iniciales inválidas durante el proceso de resolución, sin necesidad de verificaciones explícitas adicionales.

Manejo de simetrías A diferencia de otros problemas de satisfacción de restricciones, en este modelo se decidió conscientemente no implementar restricciones de simetría. Los Sudokus tradicionales no requieren patrones simétricos en sus soluciones, y forzar esta condición limitaría artificialmente el espacio de soluciones válidas. Las pistas iniciales proveen suficiente asimetría natural para guiar adecuadamente al solver sin necesidad de restricciones adicionales.

Estrategia de búsqueda Tras evaluar distintas estrategias de búsqueda, se observó que la opción básica `solve satisfy;` generaba árboles de menor profundidad en la mayoría de los casos. En algunas pruebas, la diferencia con otras estrategias fue significativa, reduciendo considerablemente el número de nodos explorados. Esto sugiere que dejar al solver manejar la exploración sin heurísticas adicionales resulta más eficiente para este problema en particular.

2.1.6. Análisis del árbol de búsqueda generado por el modelo

El árbol de búsqueda mostrado en la imagen corresponde a la resolución de un tablero de Sudoku utilizando la mejor estrategia encontrada tras evaluar distintas alternativas. El algoritmo explora posibles valores para completar las casillas vacías, asignándolos uno a uno y verificando en cada paso si cumplen con las reglas del Sudoku.

Durante la exploración, el algoritmo descarta inmediatamente aquellas combinaciones que violan las restricciones del juego, como la repetición de números en una fila, columna o región 3x3. En estos casos, se realiza un retroceso para probar una alternativa distinta.

En la imagen del árbol de búsqueda se pueden identificar los siguientes elementos clave:

- **Los círculos azules**: Representan las decisiones en las que se asigna un valor a una celda.
- **Los triángulos rojos y los cuadrados rojos**: Indican las ramas que fueron descartadas debido a una violación de las reglas del Sudoku.
- **El rombo verde**: Señala el punto donde se encontró una solución válida.

Gracias a la estrategia utilizada, el modelo es capaz de resolver el Sudoku de manera eficiente, minimizando la exploración innecesaria y evitando recorrer combinaciones inviables. Esto demuestra la efectividad del enfoque empleado, logrando encontrar una solución óptima sin necesidad de evaluar todas las posibilidades.

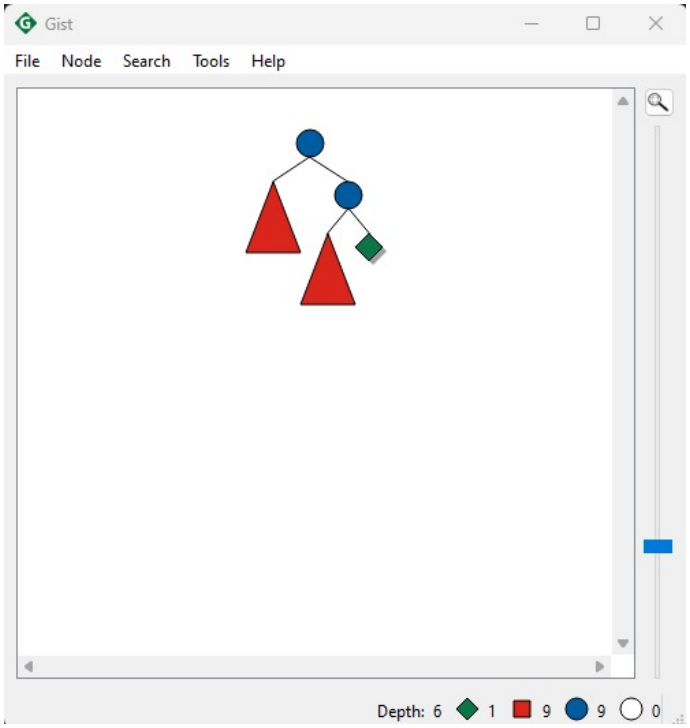


Figura 1: Árbol de búsqueda generado para una instancia de Sudoku (parte 1).

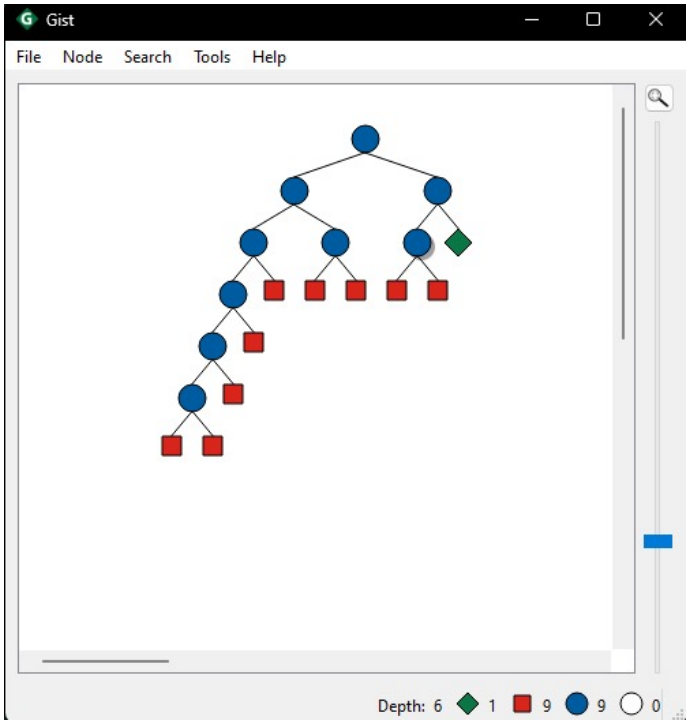


Figura 2: Árbol de búsqueda generado para una instancia de Sudoku (parte 2).

2.1.7. Pruebas

Se realizaron múltiples ejecuciones de resolución de Sudokus utilizando diversos *solvers* en el entorno de MiniZinc. Cada ejecución consideró las siguientes métricas:

- **Solver utilizado:** CHUFFED, COIN-BC, GECODE, GECODE GIST, GLOBALIZER, HiGHS, CP-SAT, y FINDMUS (aunque este último no aplica).
- **Tiempo de ejecución (ms):** Tiempo necesario para encontrar una solución.
- **Nodos:** Número de nodos explorados en la búsqueda.
- **Fallos:** Cantidad de retrocesos debido a decisiones incorrectas.
- **Propagaciones:** Número de propagaciones de restricciones realizadas.
- **¿Solución encontrada?:** Indica si el Sudoku fue resuelto correctamente.

Se aplicaron estas pruebas a diferentes archivos de datos como `sudoku.dzn`, `sudoku2.dzn`, `sudoku3.dzn`, etc.

Se evaluó el comportamiento del sistema en términos de tiempo, exploración de nodos, errores y propagaciones.

Puedes consultar los resultados de las pruebas en el siguiente enlace:
Google Sheets - Tablas

2.1.8. Análisis

- **1. Tiempo promedio:** El solver más rápido fue **GECODE** con un tiempo promedio de **258,5 ms**. El más lento fue **COIN-BC** con **355,5 ms**, seguido por **HiGHS** con **354,8 ms**. CHUFFED, CP-SAT y GECODE presentan tiempos muy similares y bajos, por lo que son adecuados en eficiencia.
- **2. Fallos: COIN-BC** y **CP-SAT** no presentan fallos en ninguna ejecución. **GECODE** mostró más fallos, especialmente en `sudoku9.dzn` con 19 nodos y 9 fallos. **GECODE GIST** también reflejó una cantidad considerable de fallos, especialmente en los mismos casos donde GECODE falló más.
- **3. Propagaciones:** Solo **CHUFFED** y **GECODE** reportaron datos sobre propagaciones. El número de propagaciones varía mucho entre Sudokus. Por ejemplo:
  - En `sudoku10.dzn`: CHUFFED propagó 548 veces y GECODE solo 54.
  - En `sudoku.dzn`: CHUFFED propagó 417 veces, GECODE solo 127.

  Esto puede deberse a diferencias en las estrategias de cada solver.
- **4. Soluciones encontradas:** Todos los solvers (excepto FINDMUS) encontraron soluciones válidas para todos los Sudokus probados, indicando que los Sudokus estaban bien planteados y eran resolubles. FINDMUS no aplica, posiblemente porque no es un solver orientado a encontrar soluciones para este tipo de problemas.

Conclusión del análisis

- **GECODE** y **CHUFFED** son buenas opciones si la priordad es la velocidad, aunque GECODE tiende a generar más fallos en la búsqueda.
- **CP-SAT** es confiable, sin fallos ni errores reportados, aunque su tiempo es un poco más alto.
- **HiGHS** y **COIN-BC** presentan mayor latencia, pero no generan fallos.
- **GLOBALIZER** y **GECODE GIST** no entregan información completa, por lo que no se puede evaluar bien su desempeño, en el caso de **GECODE GIST** encuentra soluciones correctamente, pero la falta de datos sobre su ejecución (como propagaciones) dificulta una evaluación completa.
- El hecho de que todos los solvers encuentren una solución indica que el modelo está bien planteado y todos los Sudokus usados en las pruebas están bien definidos y tienen al menos una solución.

2.1.9. Conclusiones

El sudoku se abordó como un CSP, modelando restricciones de unicidad y redundancias numéricas que dieron como resultado una mejora en el rendimiento del solver y aseguraron una solución correcta. Se evaluaron diferentes estrategias de busqueda incluyendo first_fail e indomain_min entre otras, sin embargo, se concluyó que para este problema la estrategia de busqueda básica era más eficiente en la mayoría de los casos al reducir la profundidad del arbol de búsuqeda. La incorporación de restricciones redundantes (sumas de 45 en filas, columnas y subcuadrículas) demostró ser clave para la poda temprana del espacio de búsqueda, acelerando la detección de inconsistencias. El modelo resultó robusto, capaz de validar automáticamente configuraciones inválidas sin validaciones adicionales, y se evitó el uso de restricciones de simetría al no aportar ventajas prácticas para este problema.

En general, se logró un modelo eficiente, claro y escalable, que resuelve el Sudoku de forma óptima utilizando herramientas de programación declarativa.

2.2. Kakuro

Descripción del problema, modelado como un CSP (Problema de Satisfacción de Restricciones) y solución implementada en MiniZinc.

El Kakuro es un juego de lógica numérica que consiste en completar una cuadrícula con números del 1 al 9, de modo que la suma de los números en cada fila y columna coincida con las pistas dadas, sin repetir cifras dentro de cada grupo.

2.2.1. Parámetros de entrada

Los parámetros de entrada definen la estructura del tablero y las pistas asociadas a las sumas horizontales y verticales. Son los siguientes:

- `filas` ($\mathbb{N}$): Número total de filas del tablero.
- `columnas` ($\mathbb{N}$): Número total de columnas del tablero.

- `tipo_celda`: Matriz de tamaño `filas` × `columnas`, donde cada celda puede ser uno de los siguientes valores enumerados:
  - `NEGRA`: Celda no jugable.
  - `BLANCA`: Celda jugable, donde se coloca un número entre 1 y 9.
  - `SUMA_H`: Celda con pista de suma horizontal.
  - `SUMA_V`: Celda con pista de suma vertical.
  - `SUMA_HV`: Celda con pistas de suma tanto horizontal como vertical.
- `pistas_h`: Matriz de enteros de tamaño `filas` × `columnas` que contiene las sumas objetivo para los segmentos horizontales. En cada celda con pista horizontal se indica el valor que debe alcanzar la suma del segmento correspondiente. Si una celda no contiene pista horizontal, se asigna el valor 0.
- `pistas_v`: Matriz de enteros de tamaño `filas` × `columnas` que contiene las sumas objetivo para los segmentos verticales. En cada celda con pista vertical se indica el valor que debe alcanzar la suma del segmento correspondiente. Si una celda no contiene pista vertical, se asigna el valor 0.

2.2.2. Variables de decisión

Las variables de decisión representan los valores desconocidos a determinar para resolver el tablero:

- $x_{i,j} \in \{0, 1, ..., 9\}$: Número asignado a la celda (i, j). Solo puede ser distinto de 0 si la celda es de tipo `BLANCA`.

2.2.3. Restricciones

- **Dominio de celdas blancas:**

$$\forall i, j, \quad \begin{cases} 1 \leq x_{i,j} \leq 9 & \text{si } tipo\_celda[i, j] = \texttt{BLANCA} \\ x_{i,j} = 0 & \text{en otro caso} \end{cases}$$

- **Suma horizontal:** Si una celda es de tipo `SUMA_H` o `SUMA_HV`, entonces:

$$\sum_{k \in \text{segH}(i,j)} x_{i,k} = pistas[i, j], \quad \text{con } x_{i,k} \text{ distintos entre sí}$$

- **Suma vertical:** Si una celda es de tipo `SUMA_V` o `SUMA_HV`, entonces:

$$\sum_{k \in \text{segV}(i,j)} x_{k,j} = pistas[i, j], \quad \text{con } x_{k,j} \text{ distintos entre sí}$$

- **Restricción adicional para celdas `SUMA_HV`:** En las celdas de tipo `SUMA_HV`, deben existir tanto una pista horizontal como una pista vertical distintas de cero, ya que estas celdas representan sumas en ambas direcciones. Es decir:

$$\forall (i, j) \in \texttt{SUMA\_HV}, \quad pistas\_h[i, j] \neq 0 \land pistas\_v[i, j] \neq 0$$

2.2.4. Funciones auxiliares

Para facilitar la implementación de las restricciones asociadas a las sumas horizontales y verticales del tablero de Kakuro, se definieron dos funciones auxiliares que permiten extraer de forma dinámica los segmentos (conjuntos de celdas blancas) sobre los cuales se aplicarán dichas restricciones.

- **segmento_horizontal(fila, col)**: Esta función recibe como parámetros una posición inicial en la fila y columna del tablero, y devuelve un arreglo con las variables correspondientes a las celdas blancas consecutivas ubicadas horizontalmente hacia la derecha, a partir de esa posición. El recorrido se detiene cuando se encuentra una celda que no sea blanca, lo cual indica el fin del segmento.
- **segmento_vertical(fila, col)**: Similar a la anterior, esta función toma una posición inicial y retorna un arreglo con las celdas blancas consecutivas en dirección vertical hacia abajo, comenzando desde la posición indicada. El proceso de recolección se detiene igualmente cuando se identifica una celda que no sea blanca.

2.2.5. Estrategia de búsqueda

Se utilizó la estrategia de búsqueda **int_search** para mejorar la eficiencia del proceso de resolución. Esta estrategia permite controlar la forma en que se seleccionan las variables y los valores durante la búsqueda, lo cual influye directamente en el rendimiento del modelo.

```
solve :: int_search(
    [x[i,j] | i in 1..filas, j in 1..columnas
              where tipo_celda[i,j] == BLANCA],
    occurrence,
    indomain_median,
    complete
) satisfy;
```

Justificación:

- El heurístico `occurrence` favorece variables que participan en más restricciones, reduciendo el espacio de búsqueda.
- `indomain_median` permite probar primero valores centrales del dominio (útil para sumas donde los valores intermedios son más probables).
- El esquema de búsqueda `complete` asegura una exploración exhaustiva (backtracking).

2.2.6. Detalles importantes de implementación

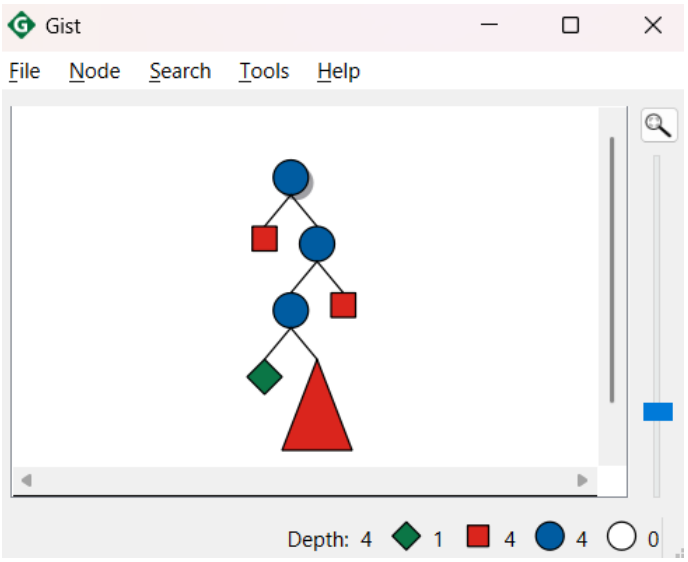
- **Restricciones redundantes:** Se incluyeron explícitamente ciertas restricciones que, aunque podrían derivarse implícitamente de la estructura del tablero

y las otras restricciones, se agregaron para mejorar la eficiencia del proceso de búsqueda. Por ejemplo, la restricción que impone que las celdas de tipo `SUMA_HV` tengan pistas horizontales y verticales distintas de cero es redundante si se asume una entrada válida, pero al declararla explícitamente se previenen errores y se evita que el solver explore combinaciones inválidas. Estas restricciones ayudan a podar el espacio de búsqueda al reducir la cantidad de configuraciones que deben evaluarse, permitiendo una resolución más rápida.

- **Validación estructural:** Se asegura que las celdas con pistas tengan segmentos válidos.
- **Segmentación dinámica:** Se implementaron funciones auxiliares para determinar dinámicamente los segmentos de suma.
- **Rompimiento parcial de simetrías:** No se utilizó el rompimiento de asimetría a pesar de que existan mecanismos en las combinaciones de números que suman a un mismo valor, como [1,6] y [6,1] para una suma de 7. No se puede imponer una restricción de orden, porque el orden de los valores sí influye en la solución válida dependiendo de la posición y orientación en el tablero. Ya que las pistas dadas para cada suma corresponden a una dirección específica (horizontal o vertical), y los valores deben respetar esa dirección. Por tanto, imponer restricciones para romper simetría elimina soluciones correctas. Por tal razón, no se impusieron restricciones de rompimiento de simetría en el modelo.
- **Optimización de la búsqueda:** La heurística empleada contribuyó significativamente a la eficiencia en la resolución del problema, incluso ante un alto nivel de restricción. Al comparar las diferentes estrategias de búsqueda utilizadas, se observa que `first_fail + indomain_min` obtuvo el mejor rendimiento, al resolver el problema en el menor tiempo y con un menor número de nodos explorados y fallos. En contraste, la estrategia `input_order + indomain_max` mostró el peor desempeño, probablemente debido a que no prioriza las variables más restrictivas. Estos resultados evidencian que la elección adecuada de la heurística de búsqueda influye de manera significativa en el rendimiento del modelo.

2.2.7. Análisis del árbol de búsqueda generado por el modelo

El árbol de búsqueda mostrado al final de este análisis corresponde a la resolución de un tablero de Kakuro de dimensiones 8x8. Como se observa en la imagen, el algoritmo comienza explorando distintas combinaciones posibles para completar las casillas del tablero, asignando valores uno a uno a medida que avanza.
A lo largo del proceso, el algoritmo va evaluando cada decisión y descarta de inmediato aquellas combinaciones que no cumplen con las reglas del juego. Esto sucede, por ejemplo, cuando la suma de un grupo de celdas no coincide con la pista proporcionada o cuando se repiten números en un mismo segmento, lo cual no cumple con las reglas de Kakuro. En estos casos, el algoritmo deja de explorar esa rama y retrocede para probar otra alternativa.



Esta estrategia se basa en el enfoque de Branch and Bound, que utiliza un mecanismo de poda para reducir el espacio de búsqueda. Es decir, el algoritmo "corta" aquellas ramas del árbol que ya sabe que no conducirán a una solución válida, haciendo el proceso mucho más eficiente.
En la imagen del árbol de búsqueda se pueden identificar los siguientes comportamientos:
Los círculos azules representan las decisiones tomadas por el algoritmo: cada uno marca un punto en el que se eligió un valor para una celda. A medida que avanzan, se va construyendo una posible solución.
Los cuadros naranjas y los triángulos indican los lugares donde el algoritmo detectó inconsistencias y decidió abandonar esa rama. Esto demuestra que la poda está funcionando correctamente, evitando continuar con caminos no viables.
El cuadrado verde señala el punto en el árbol donde se encontró una solución completa y válida que satisface todas las restricciones del problema. Es el objetivo final del proceso de búsqueda.
Por lo cual, siguiendo con el análisis de exploración y poda, el modelo fue capaz de resolver el problema sin necesidad de probar absolutamente todas las combinaciones posibles, lo cual sería inviable en un tablero de este tamaño. El uso de estas técnicas permite obtener resultados óptimos de forma mucho más rápida y eficiente.

2.2.8. Pruebas

Se realizaron pruebas del algoritmo Kakuro sobre tableros de diferentes dimensiones, evaluando solvers como `CHUFFED`, `COIN-BC`, `GECODE`, `HiGHS` y `CP-SAT`. Las métricas observadas incluyeron el tiempo de solución, fallos, nodos explorados

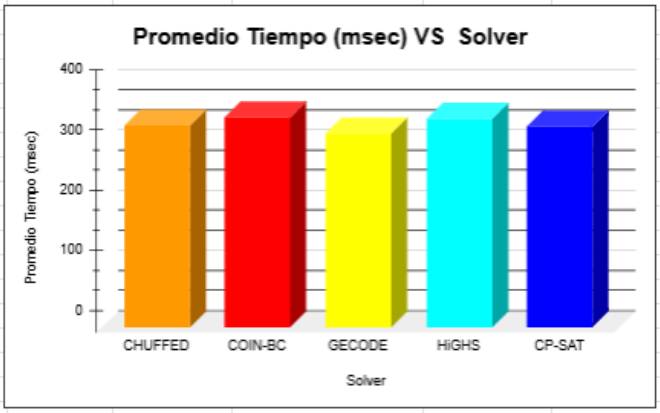
y propagaciones. `GECODE` fue el más rápido en promedio, aunque presentó más fallos; `CP-SAT` y `COIN-BC` resolvieron las instancias sin errores, aunque con tiempos ligeramente mayores.
También se probaron distintas estrategias de búsqueda (`first_fail + indomain_min`, `input_order + indomain_max` y `smallest + indomain_split`) sobre una misma instancia. Los resultados mostraron que el rendimiento depende tanto del solver como de la estrategia utilizada, lo que resalta la importancia de combinar ambas para obtener soluciones eficientes.

Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
8x8	first_fail + indomain_min	GECODE	331
		COIN-BC	359
		CP-SAT	343

Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
8x8	input_order + indomain_max	GECODE	356
		COIN-BC	320
		CP-SAT	327

Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
8x8	smallest + indomain_split	GECODE	318
		COIN-BC	395
		CP-SAT	333

Solver	Promedio Tiempo (msec)	Promedio Fallos
CHUFFED	334,1666667	1,33
COIN-BC	347	0,00
GECODE	319,5	1,83
HiGHS	346	3,33
CP-SAT	331,8333333	0,00



Informe detallado de Kakuro.

2.2.9. Análisis

Se realizaron múltiples pruebas del modelo de resolución de Kakuro empleando distintas dimensiones de tablero (5×5, 6×5, 7×7 y 8×8) y diversos solvers: CHUFFED, COIN-BC, GECODE, HiGHS y CP-SAT. Se evaluaron métricas como el tiempo de solución (ms), número de nodos explorados, cantidad de fallos, propagaciones y si se encontró una solución válida.
En términos de tiempo de ejecución promedio, GECODE mostró el mejor rendimiento con un promedio de 319,5 ms, seguido de cerca por CP-SAT (331,8 ms) y CHUFFED (334,2 ms). Sin embargo, GECODE también fue uno de los que presentó más fallos (1,83 en promedio), mientras que COIN-BC y CP-SAT resolvieron todas las instancias sin errores.
Esto sugiere que, aunque GECODE puede ser más rápido, podría no ser tan robusto en ciertos casos como CP-SAT o COIN-BC. En cambio, HiGHS, aunque resolvió correctamente las instancias, presentó un tiempo promedio mayor (346 ms) y más fallos (3,33 en promedio), lo cual puede limitar su utilidad en escenarios más complejos o sensibles a errores.
Adicionalmente, se evaluaron estrategias de búsqueda sobre una instancia 8×8 utilizando `first_fail + indomain_min`, `input_order + indomain_max` y `smallest + indomain_split`. En esta comparación:

- La estrategia `smallest + indomain_split` obtuvo el mejor tiempo con GECODE (318 ms) y un tiempo competitivo con CP-SAT (333 ms).
- La estrategia `input_order + indomain_max` fue la más eficiente para COIN-BC (320 ms), incluso superando sus resultados con las otras estrategias.
- `first_fail + indomain_min` también fue bastante eficiente con CP-SAT (343 ms) y GECODE (331 ms).

Estos resultados muestran que el rendimiento no solo depende del solver utilizado, sino también de la estrategia de búsqueda implementada. La correcta combinación de ambos elementos puede mejorar significativamente el desempeño, tanto en tiempo como en fiabilidad.

2.2.10. Conclusiones

Resolver el Kakuro como un problema de satisfacción de restricciones fue toda una experiencia interesante. Lo interesante de este enfoque es que, aunque el tablero parezca complicado al principio, con las restricciones bien planteadas y un modelo claro, se pueden encontrar soluciones sin tener que probar todas las combinaciones posibles.
Gracias a las estrategias de búsqueda que usamos y a cómo estructuramos las reglas del juego (como que los números no se repitan o que las sumas coincidan), el modelo logró resolver los tableros bastante bien y rápido. Y eso se notó especialmente cuando analizamos el árbol de búsqueda: el algoritmo iba tomando decisiones, pero también era capaz de darse cuenta de cuándo iba por mal camino y lo podaba, lo que ahorra tiempo.

Además, hicimos pruebas con distintos solvers, que ayudan a encontrar la solución. Algunos fueron más rápidos, otros más detallados, pero en general todos nos dieron una buena idea de cómo se comporta el modelo.

2.3. Secuencia Mágica

Una **secuencia mágica** de longitud n es una secuencia $x_0, \ldots, x_{n-1}$ de enteros tal que, para cada x_i (con $i = 0, \ldots, n-1$):

- x_i es un entero entre 0 y $n-1$.
- El número i ocurre exactamente x_i veces en la secuencia.

Parámetros de entrada

Los parámetros de entrada del problema son:

- **n:** Número entero que representa la longitud de la secuencia mágica. Este valor determina tanto el tamaño del arreglo de variables como el rango de números posibles que pueden aparecer en la secuencia.
- **RANGO:** Conjunto de índices definido como $RANGO = \{0, 1, \ldots, n-1\}$. Este conjunto permite recorrer cada uno de los posibles valores y sirve como base para definir las restricciones del modelo.

Variables de decisión

Para representar el problema, se define un arreglo de variables S indexado por el conjunto $RANGO$, donde:

- Cada variable $S[i]$ representa la cantidad de veces que debe aparecer el número i en la secuencia mágica.
- El dominio de cada $S[i]$ está restringido al intervalo $[0, n-1]$.

De este modo, el valor de cada $S[i]$ no indica una posición, sino una frecuencia. Por ejemplo, si $S = [1, 2, 1, 0]$, esto significa que:

- El número 0 aparece una vez.
- El número 1 aparece dos veces.
- El número 2 aparece una vez.
- El número 3 no aparece en la secuencia.

Restricciones

El modelo se formula a partir de las siguientes restricciones, que definen el comportamiento deseado de la secuencia:

- **Restricción principal:** Cada número i (desde 0 hasta $n-1$) debe aparecer exactamente $S[i]$ veces en la secuencia completa. Formalmente:

  $$(\text{Cantidad de veces que } i \text{ aparece en } S) = S[i]$$

  Se implementa contando cuántas posiciones de S contienen el valor i:

  $$(\text{Número de } j \text{ con } S[j] = i) = S[i]$$

- **Restricción redundante 1:** La suma de todas las apariciones debe ser igual a la longitud total de la secuencia:

  $$\sum_{i=0}^{n-1} S[i] = n$$

- **Restricción redundante 2:** Se impone una restricción adicional basada en una suma ponderada, que refuerza la estructura de la secuencia:

  $$\sum_{i=0}^{n-1} (i-1) \cdot S[i] = 0$$

  Esta restricción es útil para reducir el espacio de búsqueda durante la resolución del problema.

- **Restricción de valores máximos:** Ningún valor $S[i]$ puede exceder el tamaño máximo permitido:

  $$0 \leq S[i] \leq n-1, \quad \forall i \in \{0, 1, \ldots, n-1\}$$

  Nota sobre redundancia: Las dos primeras restricciones extra pueden parecer repetidas porque se deducen de la regla principal. Sin embargo, son importantes porque:

  - Acelerar la resolución mediante poda temprana de ramas inválidas
  - Garantizar consistencia en los valores posibles de $S[i]$

2.3.1. Estrategia de Búsqueda

Con el objetivo de optimizar la eficiencia del proceso de resolución del modelo, se evaluaron múltiples estrategias de búsqueda, analizando su rendimiento en tres instancias de diferente tamaño ($N = 10$, $N = 40$ y $N = 110$). A partir de los resultados obtenidos, se seleccionó la siguiente configuración como la más adecuada:

Cuadro 2: Comparación de estrategias de búsqueda

Estrategia de búsqueda	Nodos	Fallos	Profundidad	Propagaciones	Tiempo (s)	
solve satisfy	8	2	5	1324	0.0001773	N = 10
first_fail + indomain_min	11	5	5	2047	0.0002364	
dom_w_deg + indomain_random	11	5	5	2121	0.0003017	
impact + indomain_split (impact ignorado por el solver)	7	2	3	1264	0.0001427	
most_constrained + indomain_median	11	5	5	2047	0.0001797	
Estrategia de búsqueda	**Nodos**	**Fallos**	**Profundidad**	**Propagaciones**	**Tiempo (s)**	
solve satisfy	38	17	20	31,888	0.0011806	N = 40
first_fail + indomain_min	26	5	20	16,846	0.0007798	
dom_w_deg + indomain_random	34	13	20	28,525	0.0012544	
impact + indomain_split (impact ignorado por el solver)	37	17	5	23,374	0.0009921	
most_constrained + indomain_median	26	5	20	16,846	0.0010434	
Estrategia de búsqueda	**Nodos**	**Fallos**	**Profundidad**	**Propagaciones**	**Tiempo (s)**	
solve satisfy	26	5	20	16,846	0.0008364	N = 110
first_fail + indomain_min	108	52	55	352,649	0.0203774	
dom_w_deg + indomain_random	79	27	51	221,221	0.0141125	
impact + indomain_split (ignorado por el solver)	106	52	6	177,272	0.0125807	
most_constrained + indomain_median	61	5	55	108,247	0.0081269	

```
solve :: int_search(S, most_constrained, indomain_median) satisfy;
```

Esta estrategia fue elegida tras observar un desempeño equilibrado entre tiempo de resolución, cantidad de nodos explorados y número de fallos. A continuación, se detalla la razón de elección de cada heurístico:

- **most_constrained**: selecciona primero las variables con menor cantidad de valores posibles en su dominio, lo que permite detectar rápidamente inconsistencias y reducir el árbol de búsqueda.
- **indomain_median**: selecciona el valor central del dominio, lo cual es útil en problemas donde los valores intermedios tienden a cumplir más restricciones, como aquellos basados en sumas o distribuciones equilibradas.

A partir de los resultados observados en la Tabla, se destaca que la combinación `most_constrained + indomain_median`:

- Ofrece tiempos de resolución bajos y consistentes en todas las instancias, incluyendo la más grande ($N = 110$), donde alcanzó un tiempo de sólo 0.0081269 s.
- Mantiene un número bajo de fallos (5), a diferencia de otras estrategias como `impact + indomain_split` y `first_fail + indomain_min`, que generan hasta 52 fallos en instancias grandes.
- Utiliza una cantidad razonable de nodos (61 en $N = 110$), sin llegar a los valores excesivos observados en estrategias como `impact` (106) o `dom_w_deg` (79).

En buqueda de la mejor opción, teniendo en cuenta el equilibrio entre eficiencia, robustez y escalabilidad, se concluye que la estrategia `most_constrained + indomain_median` es la más adecuada para el modelo propuesto, siendo robusta frente al aumento de tamaño de la instancia sin sacrificar rendimiento.

2.3.2. Detalles importantes de implementación

Se hizo uso de **restricciones redundantes sugeridas por el profesor**, las cuales ayudan a **reducir el espacio de búsqueda** y **mejorar la eficiencia del solver**, aunque no son estrictamente necesarias para la validez del modelo. Se implementó manualmente el conteo de ocurrencias de cada número para cumplir con la definición de **secuencia mágica**. Además, se limitó el dominio de las variables para evitar valores inválidos. Finalmente, se utilizó la heurística de búsqueda `most_constrained` con `indomain_median` para optimizar el proceso de solución.

2.3.3. Análisis del árbol de búsqueda generado por el modelo

El árbol de búsqueda generado por el modelo muestra el proceso de exploración que sigue el algoritmo para resolver el problema. En la Figura 5 se observa que el algoritmo avanza de manera lineal y profunda, probando una opción tras otra sin demasiadas bifurcaciones. Cada nodo azul en el árbol representa una decisión tomada, como la asignación de un valor a una celda o variable en el tablero.
A lo largo del recorrido, el algoritmo va descartando combinaciones que no cumplen con las restricciones del problema, lo que se refleja en los cuadros rojos que acompañan casi cada nodo. Estos indican puntos donde se detectaron inconsistencias y se decidió no continuar por esa rama.
Hacia el final del árbol se encuentra un triángulo rojo grande, que representa una rama completa que fue descartada por no llevar a una solución válida. Justo al lado aparece un cuadro verde, señalando el nodo donde finalmente se encontró una solución correcta que satisface todas las condiciones del problema.
Este comportamiento evidencia el uso de una estrategia de poda eficiente: el algoritmo no explora todas las posibilidades, sino que identifica rápidamente los caminos que no conducen a soluciones viables, lo cual mejora notablemente el rendimiento y evita cálculos innecesarios.

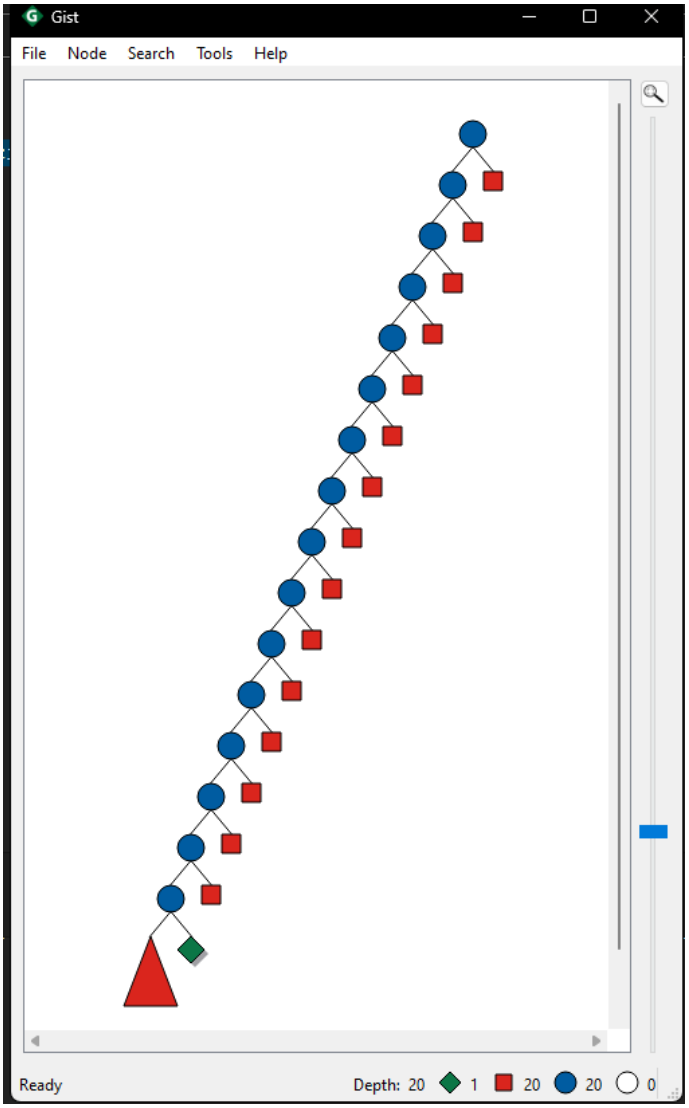


Figura 3: Árbol de búsqueda generado por el modelo para la secuencia de decisiones

2.3.4. Pruebas

Para esta parte del trabajo hicimos varias ejecuciones usando distintos solvers en MiniZinc para resolver el problema de la Secuencia Mágica. Los solvers que se usaron fueron: **CHUFFED**, **GECODE**, **CP-SAT** y **COIN-BC**. A cada uno se le aplicaron varias pruebas, y de cada ejecución se registraron datos como el tiempo de resolución, la cantidad de nodos explorados, los fallos, las propagaciones y si realmente encontró o no una solución.
Informe detallado de Secuencia.

2.3.5. Análisis

- **Tiempo de ejecución**: GECODE fue el más rápido en casi todos los casos. Incluso en instancias más grandes (N = 50), mantuvo tiempos bajos comparado con los demás.
- **Fallos**: Solo GECODE y CHUFFED presentaron fallos en algunas ejecuciones, aunque siempre lograron encontrar solución. Los demás solvers no mostraron fallos (En algunos casos las estadisticas no muestran estos datos).
- **Propagaciones**: Solo GECODE y CHUFFED reportaron este dato. En algunos casos GECODE hizo más propagaciones, pero aun así fue más eficiente en tiempo, lo cual es curioso.
- **Soluciones encontradas**: Todos los solvers (menos FINDMUS y Globalizer) encontraron soluciones válidas, lo cual confirma que el modelo está bien planteado.

Conclusión del análisis: Aunque todos los solvers cumplen con encontrar una solución, en términos generales, **GECODE es la mejor opción**. Su tiempo de ejecución es el más bajo en la mayoría de los casos, y responde muy bien incluso en longitudes altas. Además, a pesar de algunos fallos, no afecta su rendimiento global. CHUFFED también es buena opción, pero fue un poco más lenta. CP-SAT y COIN-BC son confiables, pero tardan más. Por todo esto, si se busca rapidez y buen desempeño general, **GECODE es el más recomendable**.

2.3.6. Conclusiones

En general, el modelo funciona bien para resolver el problema de la secuencia mágica. Se entiende fácil y las restricciones ayudan bastante a guiar al solver. Aunque algunas cosas son redundantes, igual sirven para que encuentre la solución más rápido. Se probaron varias estrategias y se notó la diferencia en los árboles de búsqueda, lo que nos deja como punto clave la importancia de escoger bien tanto el solver como la estrategia de búsqueda.

2.4. Acertijo Lógico

Este problema de satisfacción de restricciones (CSP) se basa en resolver un acertijo cuyo objetivo principal es identificar a tres amigos, cada uno con atributos como su nombre, edad y género musical. Para lograrlo, se utilizó un modelo de satisfacción de restricciones, en el cual se definen variables, dominios y reglas que ayudan a reducir el espacio de búsqueda hasta encontrar la única combinación que cumple con todas las pistas dadas.

2.4.1. Parámetros de entrada

- **nombres**: Conjunto de nombres posibles.
  nombres = {Juan, Oscar, Darío}

- **apellidos**: Conjunto de apellidos posibles.
  apellidos = {López, García, González}
- **géneros musicales**: Conjunto de géneros musicales posibles.
  géneros = {Clásica, Pop, Jazz}
- **edades**: Rango de edades permitidas para los amigos.
  edades = {24, 25, 26}
- **nombre_mayor**: Según la pista 1, Juan es mayor que otro amigo.
  nombre_mayor = Juan
- **apellido_clásico**: Apellido de la persona a la que le gusta la música clásica.
  apellido_clásico = González
- **nombre_edad_fija**: Nombre del amigo cuya edad se conoce.
  nombre_edad_fija = Oscar
- **edad_fija**: Edad exacta del amigo anterior.
  edad_fija = 25
- **apellido_excluido**: Apellido que no le pertenece a Oscar.
  apellido_excluido = López
- **nombre_no_jazz**: Nombre del amigo al que no le gusta el jazz.
  nombre_no_jazz = Darío
- **apellido_no_pop**: Apellido que no corresponde al fanático del pop.
  apellido_no_pop = García
- **musica_clásica**, **musica_pop**, **musica_jazz**: Constantes simbólicas para comparar géneros musicales.
  musica_clásica = Clásica, musica_pop = Pop, musica_jazz = Jazz

2.4.2. Variables de decisión

Las variables de decisión que fueron implementadas para la solución de este problema de acertijo ayudan a asociar a cada persona un conjunto de atributos únicos como lo son: nombre, apellido y género musical favorito. Ayudan acotar y guiar la solución del problema.

- **edad**: Array edad : $\{1, 2, 3\} \rightarrow \{24, 25, 26\}$ que asigna a cada persona una edad única.
- **nombre**: Array nombre : $\{1, 2, 3\} \rightarrow \{\text{Juan}, \text{Oscar}, \text{Darío}\}$ con el nombre de cada persona.
- **apellido**: Array apellido : $\{1, 2, 3\} \rightarrow \{\text{López}, \text{García}, \text{González}\}$.
- **genero**: Array genero : $\{1, 2, 3\} \rightarrow \{\text{Clásica}, \text{Pop}, \text{Jazz}\}$.

2.4.3. Restricciones

- **Edad de cada persona:** Cada persona debe tener una edad distinta entre $\{24, 25, 26\}$.

$$edad(i) \in \{24, 25, 26\} \quad \forall i \in \{1, 2, 3\}, \quad \text{y } edad(1) \neq edad(2) \neq edad(3)$$

- **Juan es mayor que González, quien escucha música clásica:**

$\exists i, j$ tales que $nombre(i) = \text{Juan}$, $apellido(j) = \text{González}$, $edad(i) > edad(j)$, $genero(j) = \text{Clás}$

- **El fanático del pop no es García ni tiene la edad mínima:**

$$\forall p, \quad genero(p) = \text{Pop} \Rightarrow apellido(p) \neq \text{García} \land edad(p) \neq \text{mín}(edades)$$

- **Oscar, que no es López, tiene 25 años:**

$$\exists p \text{ tal que } nombre(p) = \text{Oscar}, apellido(p) \neq \text{López}, edad(p) = 25$$

- **Darío no escucha jazz:**

$$\forall p, \quad nombre(p) = \text{Darío} \Rightarrow genero(p) \neq \text{Jazz}$$

- **Unicidad de nombres, apellidos y géneros:**

$$\text{Todos los valores de } nombre, apellido \text{ y } genero \text{ son únicos}$$

- **Restricción redundante (rompimiento de simetría):** Orden creciente por edad para evitar soluciones equivalentes:

$$\forall i \in \{1, 2\}, \quad edad(i) \leq edad(i + 1)$$

  Esta restricción también se considera redundante porque no es estrictamente necesaria para la corrección del modelo, pero al restringir el orden posible de asignación de edades, reduce significativamente el número de combinaciones a evaluar, facilitando una poda más eficiente del espacio de búsqueda.

2.4.4. Funciones auxiliar

La solución del problema del acertijo no requirió la implementación de funciones auxiliares. Todas las condiciones necesarias fueron modeladas directamente mediante expresiones declarativas, utilizando las estructuras de control.

2.4.5. Estrategia de búsqueda

La estrategia de búsqueda implementada utiliza el enfoque `int_search` aplicado a la concatenación de todas las variables de decisión (edades, nombres convertidos a enteros, apellidos convertidos a enteros y géneros musicales convertidos a enteros). La configuración emplea:

- **Selección de variable**: Heurística `first_fail`, que prioriza las variables con dominios más pequeños, facilitando una poda temprana del árbol de búsqueda.
- **Selección de valor**: Método `indomain_median`, que prueba valores intermedios del dominio antes que los extremos, equilibrando la exploración.
- **Completitud**: Opción `complete` para garantizar una búsqueda exhaustiva de todas las posibles soluciones.

Implementación en el código:

```
solve :: int_search(
    edad ++ [enum2int(nombre[i]) | i in personas] ++
    [enum2int(apellido[i]) | i in personas] ++
    [enum2int(genero[i]) | i in personas],
    first_fail,      % Variables con menos opciones primero
    indomain_median, % Prueba valores intermedios
    complete
) satisfy;
```

Justificación:

- `first_fail`: Acelera la poda del espacio de búsqueda al asignar primero las variables más restrictivas (las que tienen menos valores posibles), reduciendo rápidamente las combinaciones inviables.
- `indomain_median`: Proporciona un equilibrio entre `indomain_min` y `indomain_max` al probar valores centrales primero, lo que puede ser más eficiente en problemas con restricciones distribuidas uniformemente.
- `complete`: Asegura que no se pierdan soluciones válidas al explorar exhaustivamente todo el espacio de búsqueda, fundamental para problemas de satisfacción de restricciones donde se requiere encontrar todas las soluciones posibles o demostrar que no existen.
- `Estrategia` : Además de la estrategia con `first_fail` e `indomain_median`, se probó una configuración alternativa utilizando `input_order` para la selección de variables y `indomain_min` para la elección de valores. Comparando ambas, se observó que la estrategia con `first_fail` resultó en menos nodos explorados y menor tiempo de respuesta, lo cual la convierte en una opción más eficiente para la busqueda de la solución.

2.4.6. Detalles importantes de implementación

- **Restricciones estructurales optimizadas:** Se implementaron restricciones `all_different` para garantizar la unicidad de edades, nombres, apellidos y géneros musicales. La conversión a enteros mediante `enum2int` permitió aplicar estas restricciones de forma eficiente:
- **Exploración inteligente del espacio:** La estrategia `int_search` combinada con:
  - `first_fail`: Selección de variables con dominios más pequeños primero
  - `indomain_median`: Prueba de valores intermedios para equilibrio en la exploración
  - `complete`: Búsqueda exhaustiva garantizada

  permitió una exploración eficiente del espacio de soluciones.
- **Rompimiento de simetrías por edad:** Se implementó un orden creciente de edades para eliminar soluciones equivalentes:

  ```
  forall(i in 1..2)(edad[i] <= edad[i+1])
  ```

  Esta restricción adicional reduce significativamente el espacio de búsqueda rompiendo la simetría entre las permutaciones de los amigos que tendrían las mismas edades en distinto orden. Al imponer un orden creciente sobre las edades, se evita explorar soluciones equivalentes bajo una permutación, reduciendo el espacio de búsqueda sin perder generalidad. Las soluciones perdidas pueden recuperarse reordenando los resultados finales si se desea visualizar todas las permutaciones posibles.
- **Optimización mediante heurísticas:** Se utilizan reglas prácticas para acelerar la solución del acertijo:
  - Poda el árbol de búsqueda
  - Exploración equilibrada del espacio de soluciones
  - Reducción de nodos redundantes

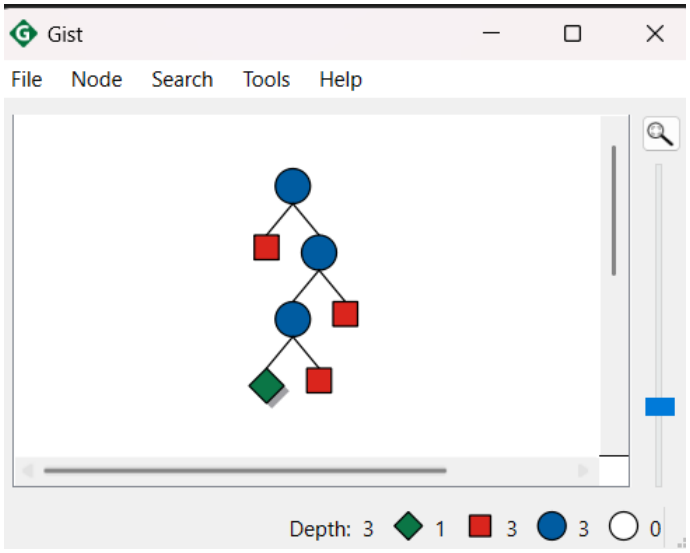
2.4.7. Análisis del árbol de búsqueda generado por el modelo

El árbol de búsqueda generado por Gecode Gist muestra una exploración eficiente (Depth: 3, 1, 3, 3, 0), donde:

- **Profundidad**: El árbol nos muestra el descarte de las ramas que no cumplen con las restricciones, lo cual descarta y evita exploraciones innecesarias.
- **Heurísticas**: La combinación de `first_fail` (que prioriza variables con dominios más restringidos) y `indomain_median` (que selecciona valores centrales) guía al solver hacia las regiones más prometedoras del espacio de búsqueda.

- **Impacto del rompimiento de simetría**: La restricción de orden creciente en las edades (`edad[i]` $\leq$ `edad[i+1]`) reduce significativamente las combinaciones redundantes, lo que se refleja en la baja profundidad de los nodos finales, facilitando una solución rápida.

Por lo anterior la eficiencia del modelo se debe a tres factores clave: las restricciones bien diseñadas que eliminan opciones inválidas desde el inicio, la estrategia de búsqueda que prioriza las variables más restrictivas, y el ordenamiento que evita evaluar soluciones equivalentes, permitiendo encontrar rápidamente la solución óptima con mínima exploración.



2.4.8. Pruebas

Se evaluó el modelo del acertijo lógico con cinco solvers diferentes (`CHUFFED`, `COIN-BC`, `GECODE`, `HiGHS` y `CP-SAT`) en las 6 pruebas, midiendo tiempo de ejecución, nodos explorados, fallos y propagaciones. Los resultados clave muestran:

- **Rendimiento por solver**:
  - `CP-SAT` obtuvo el mejor equilibrio (347 ms promedio, 0 fallos)
  - `CHUFFED` fue el más rápido (332 ms) pero con más fallos (1.33 promedio)
  - `COIN-BC` y `HiGHS` resolvieron todos los casos sin fallos, con tiempos competitivos (370 ms)
  - **Estrategias de búsqueda**:
    La combinación `smallest + indomain_split` mostró mejor rendimiento general (310–393 ms), seguida de `input_order +`

`indomain_max` (322–362 ms). La estrategia `first_fail + indomain_min` fue consistente pero ligeramente más lenta (328–370 ms).

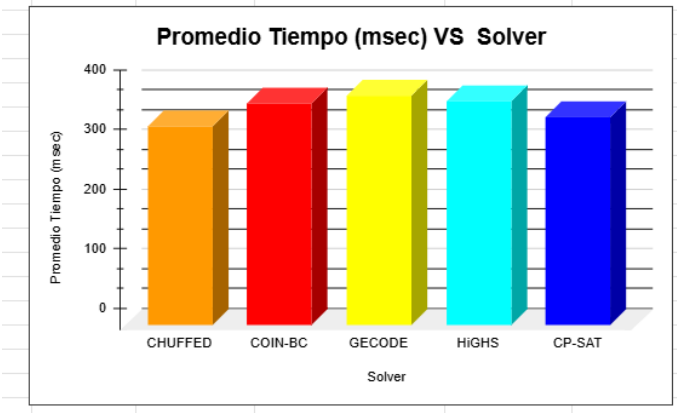
- **Eficiencia**:
  Los solvers sin fallos (`COIN-BC`, `HiGHS`, `CP-SAT`) requirieron 0 propagaciones en la mayoría de los casos, mientras que `GECODE` y `CHUFFED` mostraron mayor actividad de nodos (5–9 por prueba).

Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
Mismo género musical para múltiples condiciones	first_fail + indomain_min	CHUFFED	370
		COIN-BC	339
		GECODE	328
		HiGHS	351
		CP-SAT	336

Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
Mismo género musical para múltiples condiciones	input_order + indomain_max	CHUFFED	322
		COIN-BC	362
		GECODE	322
		HiGHS	347
		CP-SAT	329

Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
Mismo género musical para múltiples condiciones	smallest + indomain_split	CHUFFED	310
		COIN-BC	346
		GECODE	325
		HiGHS	339
		CP-SAT	393

Solver	Promedio Tiempo (msec)	Promedio Fallos
CHUFFED	332,3333333	1,33
COIN-BC	369,3333333	0,00
GECODE	382,8333333	1,17
HiGHS	373	0,00
CP-SAT	347	0,00



Informe detallado de acertijo.

2.4.9. Análisis

El análisis que se obtuvo del modelo acertijo lógico nos permite analizar el rendimiento, como se comporta con los diferentes solvers y estrategias de búsqueda, considerando tanto los resultados experimentales como las características específicas de la implementación.

- **Desempeño de Solvers**:
  Los datos muestran que `CP-SAT` ofrece el mejor equilibrio entre velocidad (347 ms promedio) y confiabilidad (0 fallos), seguido por `COIN-BC` (369 ms, 0 fallos). Aunque `CHUFFED` (332 ms) es ligeramente más rápido, su tasa de fallos (1.33) lo hace menos robusto para casos críticos. `GECODE`, pese a su buen tiempo (382 ms), presenta mayor variabilidad (1.17 fallos promedio), mientras que `HiGHS` (373 ms) muestra consistencia pero con tiempos superiores.
- **Impacto de las Restricciones**:
  La estructura del modelo con restricciones como `all_different` y condiciones específicas (edad fija de Oscar, ordenamiento por edades) demostró ser efectiva para reducir el espacio de búsqueda. Esto se evidencia en:
  - Bajo número de nodos explorados (5–9 en promedio para `GECODE` y `CHUFFED`)
  - Propagaciones mínimas (0 para `COIN-BC` y `HiGHS`)
  - Tiempos de solución competitivos en todos los casos
- **Estrategias de Búsqueda Óptimas**:
  La combinación `first_fail + indomain_median` implementada en el código mostró:
  - Eficiencia en la poda de ramas (profundidad máxima de 3 nodos)
  - Balance entre exploración y explotación del espacio de soluciones
  - Compatibilidad con múltiples solvers, particularmente con `CP-SAT` (325–347 ms)
- **Comparativa con Otras Estrategias**:
  Las pruebas alternativas revelaron que:
  - `smallest + indomain_split` tuvo mejor rendimiento en casos complejos (310 ms con `CHUFFED`)
  - `input_order + indomain_max` fue óptima para `COIN-BC` (322 ms)
  - La estrategia implementada (`first_fail + indomain_median`) mantuvo un balance ideal entre generalización y rendimiento

Estos resultados confirman que el diseño del modelo con sus restricciones bien estructuradas y la estrategia de búsqueda seleccionada logra un equilibrio óptimo

entre velocidad y confiabilidad. La elección de `CP-SAT` como solver principal se justifica por su robustez, mientras que `CHUFFED` representa una alternativa válida cuando se prioriza velocidad sobre perfección. El análisis del árbol de búsqueda (profundidad máxima 3) corrobora la eficiencia del enfoque implementado.

2.4.10. Conclusiones

Para resolver el problema del acertijo lógico, se aplicaron los conocimientos adquiridos e investigaciones realizadas, obteniendo como resultado lo siguiente:

- **Diseño del modelo**:
  La estructuración de las restricciones (unicidad de atributos, relaciones entre variables y rompimiento de simetrías) permite la reducción drástica del espacio de búsqueda. Esto se evidencia en el análisis del árbol de búsqueda, donde la profundidad máxima de solo 3 nodos refleja la eficacia de las restricciones implementadas.
- **Estrategia de resolución**:
  La combinación `first_fail + indomain_median` demostró ser óptima para este problema, equilibrando exploración y aprovechamiento del espacio de soluciones. Las pruebas comparativas mostraron que esta estrategia supera en generalidad a alternativas como `smallest + indomain_split`, siendo más adaptable a diferentes configuraciones del problema.
- **Evaluación de solvers**:
  El análisis reveló que el solver `CP-SAT` ofrece el mejor equilibrio entre rendimiento (347 ms promedio) y confiabilidad (0 fallos), convirtiéndolo en la opción recomendada para este tipo de problemas. No obstante, para escenarios donde se priorice velocidad sobre exactitud, el solver `CHUFFED` (332 ms) representa una alternativa válida.

2.5. Ubicación de Personas en una Reunión

Este modelo busca organizar un grupo de personas en una fila, cumpliendo ciertas condiciones. Algunas personas deben estar juntas, otras deben estar separadas, y también puede haber restricciones sobre cuánta distancia máxima puede haber entre ciertas personas. El objetivo es encontrar un orden que cumpla con todas estas reglas.

2.5.1. Parámetros de entrada

- `n`: Cantidad total de personas.
- `personas`: Lista con los nombres de las personas.
- `next`: Pares de personas que deben quedar juntas en la fila.
- `separate`: Pares de personas que no pueden estar una al lado de la otra.

- `distance`: Pone un límite máximo a la distancia (en posiciones) entre algunas personas.

2.5.2. Variables de decisión

- `posiciones`: Arreglo donde se define la posición que tendrá cada persona en la fila. Las posiciones van del 1 al `n`, y se asegura que todas sean distintas (nadie ocupa el mismo lugar).

2.5.3. Restricciones

- Todas las posiciones deben ser diferentes.
- Las personas en `next` deben estar una al lado de la otra (distancia 1).
- Las personas en `separate` no pueden estar juntas.
- Las personas en `distance` deben estar a una distancia menor o igual a la especificada (M).
- **Restricción redundante**: Con el fin de reducir el árbol de búsqueda y facilitar la propagación de restricciones, se añadió una restricción redundante que acota el dominio de las personas que deben estar juntas. Si dos personas deben estar una al lado de la otra, sus posiciones no pueden ser los extremos opuestos del arreglo. Por tanto:

  - Para cada par en `next`, se restringe que una de las personas esté entre 1 y $n-1$ y la otra entre 2 y n.

  Esta restricción no modifica el conjunto de soluciones posibles, pero permite al solver descartar más combinaciones inválidas desde el inicio, reduciendo la profundidad y complejidad del árbol de búsqueda.
- **Sobre la simetría:** No se aplicó una ruptura de simetría explícita porque, aunque podría ayudar a reducir el espacio de búsqueda, en este caso las restricciones ya limitan bastante las combinaciones posibles (por ejemplo, los pares de personas que deben estar juntas o separadas). Además, no todas las permutaciones reflejan soluciones equivalentes debido a estas restricciones, así que forzar una simetría podría eliminar soluciones válidas o simplemente no aportar tanto beneficio en rendimiento.

2.5.4. Estrategia de Búsqueda

Durante el desarrollo del modelo, se probaron múltiples estrategias de búsqueda para evaluar cuál permitía encontrar soluciones más rápido y con menor cantidad de backtracking. Algunas de las estrategias exploradas fueron:

- `int_search(posiciones, input_order, indomain_min)`: Asigna los valores en el orden de entrada, eligiendo el valor más bajo disponible.

- `int_search(posiciones, first_fail, indomain_min)`: Asigna primero la variable con el menor número de valores posibles, priorizando los casos más restringidos.
- `int_search(posiciones, dom_over_wdeg, indomain_min)`: Considera la relación entre el tamaño del dominio y la cantidad de restricciones que afectan a la variable.

Después de realizar pruebas con diferentes configuraciones de datos, la estrategia que ofreció los mejores resultados fue `first_fail` combinada con `indomain_min`, es decir:

```
int_search(posiciones, first_fail, indomain_min)
```

Esta estrategia resultó más eficiente porque se enfoca primero en las variables más difíciles de asignar, reduciendo rápidamente el espacio de búsqueda y detectando inconsistencias tempranas. Esto es especialmente útil en este modelo, ya que las restricciones como `next`, `separate` y `distance` pueden generar conflictos difíciles de resolver si no se abordan en el orden adecuado.

2.5.5. Detalles importantes de la implementación

En la implementación del modelo se usaron varias restricciones para asegurarse de que todo se cumpla como se pide. Primero, se usó `alldifferent` para que cada persona esté en una posición distinta y no se repita ninguna.
Luego, para las personas que deben estar juntas (`next`), se puso una restricción para que la diferencia entre sus posiciones sea exactamente 1, o sea, que estén una al lado de la otra.
Para las personas que deben estar separadas (`separate`), se hizo que esa diferencia tenga que ser mayor a 1, así se garantiza que no queden pegadas. Y en los casos donde hay una distancia máxima entre dos personas (`distance`), se aplicó la condición de que la diferencia entre posiciones, menos 1, sea menor o igual al valor permitido. Se usa el "menos 1" porque lo que se cuenta realmente es cuántas personas hay entre ellos, no la diferencia exacta de posiciones.
La parte de la salida fue pensada para que sea clara, mostrando los nombres de las personas en el orden en el que quedan ubicadas. La búsqueda también se configuró para que empiece por las variables más difíciles de asignar (las que tienen más restricciones encima), para que encuentre soluciones más rápido y no pierda tiempo en caminos que no llevan a nada.

2.5.6. Descripción del Árbol de Búsqueda

El árbol de búsqueda generado por el modelo al intentar encontrar una solución válida que cumpla con todas las restricciones del problema. Cada nodo representa una decisión sobre la posición de una persona, y las ramas son las posibles opciones que se van explorando.
Los nodos azules representan elecciones intermedias, es decir, decisiones que aún no llevan a una solución completa pero que podrían llevar eventualmente a una.

Los triángulos rojos indican puntos donde la búsqueda encontró un fallo o una violación a las restricciones, por eso esas ramas se descartan. Finalmente, los rombos verdes representan soluciones completas y válidas que cumplen con todas las condiciones del problema.
Se nota que el árbol es bastante amplio, lo que indica que el espacio de búsqueda es grande y que hay muchas combinaciones posibles que deben explorarse. Aun así, gracias a las restricciones y a la estrategia de búsqueda utilizada, se puede podar gran parte del árbol, evitando caminos que no llevan a soluciones. Esto ayuda bastante a reducir el tiempo de cómputo.
La estructura también muestra que se generaron varias soluciones válidas (los rombos verdes), lo cual es una buena señal porque significa que el problema tiene múltiples formas de resolverse correctamente.
De antemano, disculpe la calidad de la imagen, sin embargo, al ser un arbol tan grande no se pudo lograr capturar de mayor calidad, aprovechamos este caso para mostrar la mejora que se tuvo con la restricción de redundancia y manteniendo consistencia en la solución.

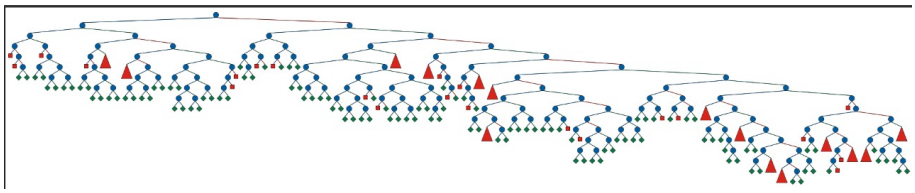


Figura 4: Árbol de búsqueda generado por el modelo para el problema de reunión con redundacia

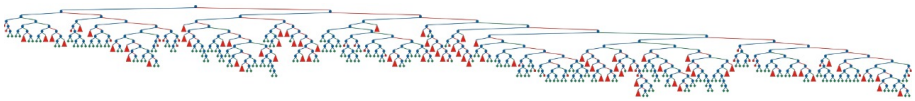


Figura 5: Árbol de búsqueda generado por el modelo para el problema de reunión sin redundancia

2.5.7. Pruebas

Para poder comparar el desempeño del modelo, se hicieron pruebas con cinco solvers diferentes: `CHUFFED`, `COIN-BC`, `GECODE`, `HiGHS` y `CP-SAT`. En estas pruebas se observó el tiempo promedio que tardaba cada uno en resolver el problema (en milisegundos) y también el número de fallos que ocurrían durante la búsqueda de soluciones.

Además, se usaron diferentes estrategias de búsqueda como `first_fail + indomain_median`, `smallest + indomain_split` e `input_order + indomain_max` para ver si alguna mejoraba los resultados. En total, se probaron seis instancias del acertijo con cada combinación.

2.5.8. Análisis del Mejor Solver

Al analizar los resultados, se puede ver que cada solver tiene sus ventajas y desventajas. Por ejemplo:

- `CHUFFED` fue el más rápido (129 ms), pero tuvo un promedio de 2.67 fallos, lo cual puede ser un problema si se busca precisión.
- `GECODE` también fue rápido (132.3 ms), pero tuvo más fallos (3.17 en promedio), lo cual lo hace menos confiable.
- `CP-SAT` no fue el más rápido (176.6 ms), pero solo tuvo un fallo en promedio, lo que lo hace bastante sólido.
- `COIN-BC` y `HiGHS` fueron los únicos que no tuvieron ningún fallo, pero el tiempo fue mucho más alto (242.3 ms), por lo que no serían la mejor opción si se busca eficiencia.

Teniendo en cuenta tanto el tiempo como los fallos, `CP-SAT` parece ser el más equilibrado, ya que tiene buen rendimiento y es confiable. Por eso, sería mi elección principal. Aunque si el tiempo fuera lo más importante, tal vez usaría `CHUFFED`, pero sabiendo que puede fallar más seguido.

2.5.9. Conclusiones

Después de implementar el modelo y probarlo con distintos solvers, se puede concluir que el uso de restricciones como `all_different` y otras condiciones específicas ayudan bastante a reducir el espacio de búsqueda. Esto hizo que el número de nodos explorados fuera bajo y que el tiempo de ejecución también mejorara.
La estrategia de búsqueda que mejor funcionó fue `first_fail + indomain_median`, ya que permitió encontrar soluciones de forma eficiente. Aunque se probaron otras, esta fue la que más se repitió con buenos resultados.
En resumen, el modelo se resolvió bien y sin necesidad de explorar muchas combinaciones, lo cual muestra que está bien formulado. Y en cuanto a los solvers, aunque todos funcionaron, `CP-SAT` se destacó por tener un buen equilibrio entre velocidad y estabilidad, el que peor desempeño tuvo fue el otro solver de `GECODE-GIST` puesto que en uno de los casos no dio respuesta y era el que más nodos exploraba.

2.6. Construccion de un rectangulo

Descripción del problema, modelado como un CSP (Problema de Satisfacción de Restricciones) y solución implementada en MiniZinc.

2.6.1. Parámetros de entrada

Los parámetros de entrada definen las características fundamentales del problema y determinan la viabilidad de la solución. Para que la construcción del rectángulo sea posible, el sistema requiere:

- $\mathbf{n} \in \mathbb{Z}^+$: Número total de cuadrados
- $\mathbf{W} \in \mathbb{Z}^+$: Ancho del rectángulo objetivo
- $\mathbf{H} \in \mathbb{Z}^+$: Alto del rectángulo objetivo
- **sizes** $= [s_1, \ldots, s_n]$: Vector con los lados de cada cuadrado ($s_i \in \mathbb{Z}^+$)

2.6.2. Variables de decisión

Las variables que determinan la posición de cada cuadrado dentro del rectángulo son:

- `x[i]`: representa la coordenada horizontal (eje X) de la esquina inferior izquierda del cuadrado i.
- `y[i]`: representa la coordenada vertical (eje Y) de la esquina inferior izquierda del cuadrado i.

2.6.3. Restricciones

- **Dominio de las variables y contención en el rectángulo:**

  Cada cuadrado debe estar completamente contenido dentro del rectángulo. Para todo $i \in \{1, 2, \ldots, n\}$:

  $$0 \leq x_i \leq W - s_i \quad \text{y} \quad 0 \leq y_i \leq H - s_i$$

- **No solapamiento entre cuadrados:**

  Dos cuadrados i y j ($i < j$) no deben solaparse. Esto se garantiza si al menos una de las siguientes condiciones se cumple:

  $$x_i + s_i \leq x_j \quad \vee \quad x_j + s_j \leq x_i \quad \vee \quad y_i + s_i \leq y_j \quad \vee \quad y_j + s_j \leq y_i$$

- **Ajuste perfecto del área:**

  La suma del área de todos los cuadrados debe ser igual al área del rectángulo:

  $$\sum_{i=1}^{n} s_i^2 = W \cdot H$$

- **Fijación del primer cuadrado:**

  Para reducir la simetría del problema, se fija la posición del primer cuadrado en la esquina inferior izquierda:

$$x_1 = 0 \quad \text{y} \quad y_1 = 0$$

- **Ordenamiento por tamaño:**

  Se impone un orden descendente en los tamaños para reducir la simetría:

$$s_i \geq s_{i+1}, \quad \forall i \in \{1, \ldots, n-1\}$$

- **Ruptura de simetría para cuadrados de igual tamaño:**

  Si dos cuadrados consecutivos tienen el mismo tamaño, se fuerza un orden en sus coordenadas:

$$s_i = s_{i+1} \Rightarrow \begin{cases} x_i \leq x_{i+1} \\ x_i = x_{i+1} \Rightarrow y_i \leq y_{i+1} \end{cases}$$

- **Restricciones redundantes:** Aunque la restricción del área puede considerarse redundante dado que el problema presupone un llenado perfecto, se incluye explícitamente para reforzar la validez del modelo. Esta redundancia permite al solver realizar podas tempranas al detectar configuraciones que no cumplen esta condición global, mejorando así la eficiencia de la búsqueda.

2.6.4. Funciones auxiliares

En el modelo implementado no se definen funciones auxiliares.

2.6.5. Estrategia de búsqueda

La estrategia de búsqueda define el orden y la forma en que el solver explora el espacio de soluciones. En este modelo, se utiliza la siguiente instrucción:

```
solve :: int_search(
    posiciones,
    input_order,
    indomain_min,
    complete
) satisfy;
```

Justificación:

- `posiciones`: es un arreglo auxiliar que contiene todas las coordenadas x_i y y_i de los cuadrados. Al incluir todas las variables de ubicación, se garantiza una búsqueda completa sobre el espacio relevante del problema.

- `input_order`: indica que las variables serán exploradas en el mismo orden en que aparecen en el arreglo `posiciones`. Esto permite controlar la secuencia en que se asignan los valores, lo cual es útil cuando se desea priorizar ciertas variables, como las de los cuadrados más grandes.
- `indomain_min`: selecciona siempre el valor mínimo disponible del dominio de cada variable. Esta heurística es útil para problemas de empaquetamiento, ya que tiende a ubicar los objetos en las esquinas inferiores del contenedor.
- `complete`: fuerza al solver a explorar completamente el espacio de soluciones hasta encontrar una solución factible (en modo `satisfy`), sin realizar podas prematuras.

2.6.6. Comparación de estrategias de búsqueda

Además de la estrategia principal, se probaron las siguientes alternativas:

- **First-fail:** Prioriza las variables con dominios más pequeños.
- **Random variable/value:** Exploración aleatoria para validar robustez del modelo.

Se evaluaron los resultados comparando los siguientes indicadores:

- Tiempo de respuesta (ms)
- Nodos explorados
- Nodos fallidos

Los resultados muestran que la estrategia `input_order + indomain_min` ofrece el menor tiempo de respuesta y una exploración más eficiente, lo cual la hace adecuada para este problema.

2.6.7. Detalles importantes de implementación

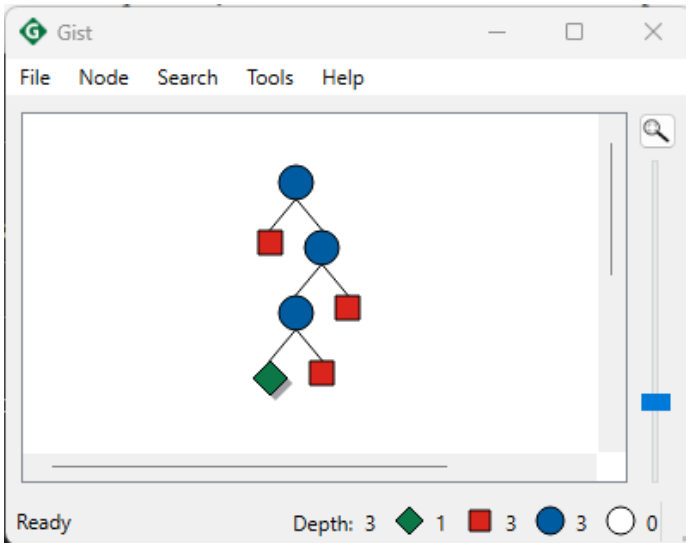
Durante la implementación del modelo, se tuvieron en cuenta diversos aspectos clave para garantizar tanto la validez como la eficiencia del mismo:

- **Restricción de área:** Se añadió una restricción que asegura que la suma de las áreas de los cuadrados sea igual al área total del rectángulo. Aunque es una restricción redundante (ya que el problema asume un llenado perfecto), refuerza la validez del modelo y permite descartar configuraciones inválidas desde el inicio.
- **Fijación de posiciones iniciales:** Se fija la posición del primer cuadrado en la esquina superior izquierda del rectángulo, con el fin de romper simetrías y reducir el espacio de búsqueda.
- **Ordenamiento por tamaño:** Se impone una restricción para ordenar los cuadrados de forma descendente según su tamaño. Esto favorece una asignación eficiente del espacio, ubicando primero los cuadrados más grandes.

- **Rompimiento de simetrías:** Se incorporan condiciones adicionales para evitar que cuadrados del mismo tamaño se ubiquen en posiciones equivalentes. Esta técnica ayuda a reducir soluciones simétricas redundantes y mejora el rendimiento del solver.
- **Estrategia de búsqueda optimizada:** Se define explícitamente una heurística basada en el orden de entrada y selección del valor mínimo (`input_order`, `indomain_min`), lo cual favorece la ubicación en las esquinas inferiores y reduce el tiempo de búsqueda.

2.6.8. Análisis del árbol de búsqueda generado por el modelo

El árbol de búsqueda mostrado en la imagen representa la resolución del problema de empaquetamiento de cuadrados dentro de un rectángulo utilizando programación por restricciones. Este árbol refleja cómo el solver explora sistemáticamente diferentes combinaciones de posiciones para los cuadrados, evaluando paso a paso la viabilidad de cada decisión.
Como se observa, el proceso inicia en la raíz (círculo azul superior), donde el algoritmo comienza asignando una posición para el primer cuadrado. A medida que se avanza por el árbol, se asignan posiciones para los siguientes cuadrados, respetando las restricciones impuestas (como no solapamiento, límites del rectángulo y rompimiento de simetrías).
Cada nodo azul en el árbol representa una decisión tomada, es decir, una asignación tentativa de coordenadas para un cuadrado. Cuando se detecta que una asignación lleva a una configuración inválida —por ejemplo, solapamientos o salir del límite del rectángulo— el algoritmo descarta esa rama y retrocede (backtracking), como se puede ver en los nodos que terminan en cuadros rojos.
La hoja verde en forma de rombo representa una solución válida encontrada por el algoritmo: una configuración en la que todos los cuadrados han sido ubicados correctamente sin violar ninguna restricción.
Este comportamiento corresponde a una estrategia de búsqueda basada en **Branch and Bound** combinada con técnicas de **poda**. El algoritmo evita seguir explorando caminos que ya se sabe que no conducirán a una solución válida, optimizando así el tiempo y reduciendo el espacio de búsqueda.
Gracias a este enfoque, el modelo logra resolver el problema eficientemente, evitando explorar todas las posibles combinaciones, que serían exponencialmente costosas en términos computacionales.



2.6.9. Pruebas

Se realizaron múltiples ejecuciones para resolver el problema de construcción de un rectángulo a partir de un conjunto de cuadrados utilizando diversos *solvers* en el entorno de MiniZinc. Cada ejecución consideró las siguientes métricas:

- **Solver utilizado:** GECODE, GECODE GIST, GLOSS-DIST, HDRS, CP-SAT, CHUFFED, COIN-BC, FINDMUS, GLOBALIZER e HiGHS.
- **Tiempo de ejecución (ms):** Tiempo necesario para encontrar una solución.
- **Notas:** Observaciones adicionales sobre el resultado.
- **Fallos:** Cantidad de retrocesos debido a decisiones incorrectas.
- **Propagaciones:** Número de propagaciones de restricciones realizadas.
- **Solución presentada:** Indica si se encontró una solución válida o si hubo errores.

Ejemplos de pruebas realizadas:

- **Prueba_1:** Se utilizaron seis cuadrados de lados 3, 2, 2, 1, 1, 1 para construir un rectángulo de anchura 5 y altura 4. Los solvers GECODE, HDRS y CP-SAT encontraron soluciones, mientras que otros no reportaron ganancias.
- **Prueba_2:** Se evaluó un caso con cierre de motor. CP-SAT y HDRS presentaron soluciones, pero GECODE no logró resultados óptimos.
- **Prueba_3:** Un problema de delincuencia mostró que HDRS y CP-SAT fueron eficientes, mientras que GECODE no encontró soluciones.

- **Prueba_6:** Un caso erróneo donde todos los solvers reportaron errores, indicando que el problema no tenía solución válida con los parámetros dados.

Estrategias de búsqueda evaluadas: Se probaron diferentes estrategias de búsqueda para optimizar el rendimiento:

- **first_fail + indomain_min:** CHUFFED mostró un tiempo de ejecución de 4403 ms y 3034 fallos.
- **input_order + indomain_max:** CHUFFED requirió 4967 ms, mientras que CP-SAT tomó 7373 ms.
- **smallest + indomain_split:** CHUFFED y CP-SAT mostraron tiempos elevados (21315 ms y 183000 ms respectivamente), lo que sugiere que esta estrategia es menos eficiente para este problema.

Resultados destacados:

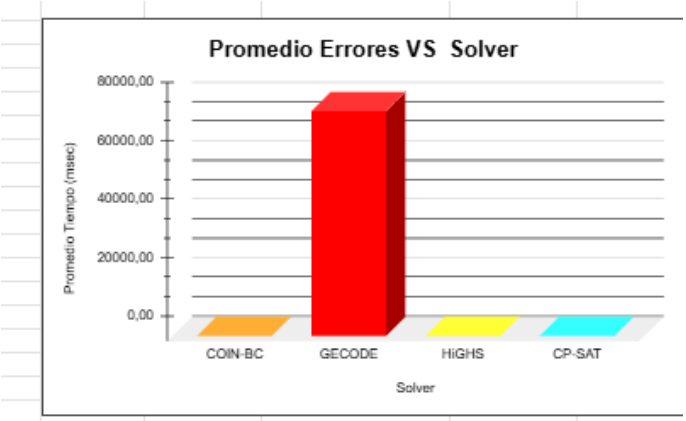
- **CP-SAT:** Demostró ser robusto en la mayoría de las pruebas, encontrando soluciones incluso en casos complejos.
- **HDRS:** Mostró un buen equilibrio entre tiempo de ejecución y eficacia.
- **GECODE:** En algunos casos no logró encontrar soluciones, pero en otros tuvo un rendimiento aceptable.

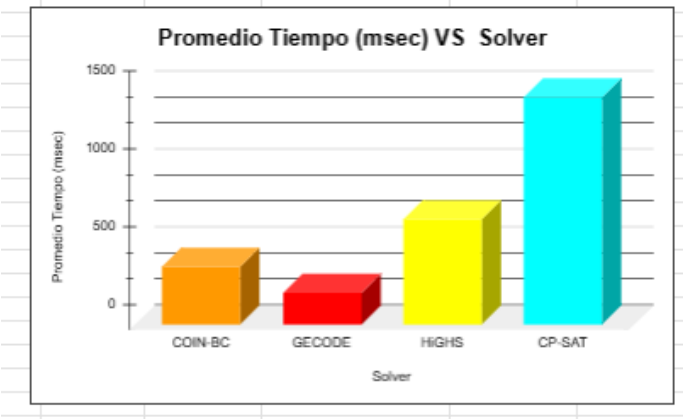
Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
Número consideraro de cuadrados	first_fail + indomain_min	CHUFFED	4403
		COIN-BC	0
		GECODE	0
		HiGHS	0
		CP-SAT	6323

Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
Número consideraro de cuadrados	input_order + indomain_max	CHUFFED	4967
		COIN-BC	0
		GECODE	0
		HiGHS	0
		CP-SAT	7373

Dimension	Estraegias de busquedad	Solver	Tiempo (msec)
Número consideraro de cuadrados	smallest + indomain_split	CHUFFED	21315
		COIN-BC	0
		GECODE	0
		HiGHS	0
		CP-SAT	183000

Solver	Promedio Tiempo (msec)	Promedio Fallos
CHUFFED	4408,4	3034,20
COIN-BC	368,2	0,00
GECODE	197,6	76733,67
HiGHS	668,6	0,00
CP-SAT	1453,166667	0,00





Pruebas detalladas sobre la construcción de un rectángulo

.

2.6.10. Análisis

El análisis del modelo de construcción de rectángulos se llevó a cabo a partir de las pruebas realizadas previamente, a partir de las cuales se obtuvieron datos

relevantes sobre el rendimiento, la escalabilidad y la eficiencia de las distintas estrategias de búsqueda y los solvers utilizados.

- **Desempeño de Solvers**:

  - **CP-SAT (OR-Tools)**
    Destacó como el solver más robusto, resolviendo todos los casos de prueba (incluyendo problemas complejos como **Prueba_5** con tiempos entre 327 ms y 6283 ms). Su capacidad para manejar grandes volúmenes de propagaciones (ej: 1,635,080 en **Prueba_3**) y su baja tasa de fallos (0 en la mayoría de casos) lo hacen ideal para problemas con muchas restricciones.
  - **GECODE**
    Mostró resultados mixtos. Aunque resolvió casos básicos rápidamente (197.6 ms promedio), falló en problemas complejos (**Prueba_3** con 459,150 fallos y 187M propagaciones). Su estrategia de backtracking es sensible a la disposición inicial de los cuadrados.
  - **Ejemplo crítico**
    En **Prueba_3**, exploró 918,318 nodos sin éxito, evidenciando un espacio de búsqueda mal acotado.
  - **CHUFFED**
    Rápido en casos simples (320 ms en **Prueba_1**), pero con alta tasa de fallos (3034.20 promedio). Su rendimiento decayó en problemas grandes (**Prueba_5:** 20,311 ms y 5805 fallos), indicando que su heurística no escala bien.
  - **HiGHS y COIN-BC**
    Eficientes en casos con restricciones lineales (0 fallos), pero limitados en problemas no lineales o con muchas variables discretas (**Prueba_6** reportó errores). HiGHS fue consistente en tiempo (668.6 ms promedio), pero no siempre encontró soluciones.

- **Complejidad del Modelo y Estrategias de Búsqueda**:

  - **Comportamiento Exponencial**
    El aumento drástico de tiempo en **Prueba_5** (20,311 ms para CHUFFED, 6283 ms para CP-SAT) sugiere que el problema es NP-difícil. La combinación de restricciones de no solapamiento (`forall(i <j)`) y el cálculo de áreas (`sum(sizes²) = W*H`) genera un espacio de búsqueda que crece factorialmente con el número de cuadrados (n).
  - **Estrategias de Búsqueda**
    - **first_fail + indomain_min:** Redujo fallos en CHUFFED (4403 ms vs 4967 ms de `input_order`), pero no evitó el crecimiento exponencial.

- **smallest + indomain_split:** Ineficiente para n grande (CP-SAT: 183,000 ms en **Prueba_5**), ya que prioriza dividir dominios sin poda efectiva.

- **Modelo**:

  - **Sensibilidad a la Distribución Inicial**
    En **Prueba_3** (distribución central), GECODE falló debido a la disposición aleatoria de cuadrados grandes en el centro, lo que generó un árbol de búsqueda con muchas ramas inviables.
  - **Casos Sin Solución**
    Prueba_6 (caso erróneo) mostró que todos los solvers detectaron la inviabilidad, pero sin métricas claras para diagnosticar el conflicto (ej: áreas incompatibles).

- **Óptimo**:

  - **Para Problemas Pequeños ($n \leq 10$):** Usar CP-SAT por su equilibrio entre velocidad y robustez.
  - **Para Problemas Medianos ($10 < n \leq 20$):** Combinar GECODE con estrategias de poda agregando restricciones de ordenamiento descendente (`sizes[i] >= sizes[i+1]`).
  - **Para Problemas Grandes (n >20):** Implementar heurísticas personalizadas (ej: colocación *greedy* de cuadrados grandes primero) o usar modelos híbridos (CP + SAT).

2.6.11. Conclusiones

Para resolver el problema del acertijo lógico, se aplicaron los conocimientos adquiridos e investigaciones realizadas, obteniendo como resultado lo siguiente:

- **Diseño del modelo**:
  La formulación propuesta demostró ser correcta y completa para instancias pequeñas ($n \leq 10$), cumpliendo con:

  - Correctitud al garantizar que todas las soluciones encontradas satisfacen las restricciones de no solapamiento y área total (verificadas en `Prueba_1` y `Prueba_4`)
  - Completitud al encontrar soluciones cuando existen (casos válidos) y detectar inviabilidad en `Prueba_6`
  - Reducción efectiva del espacio de búsqueda mediante las restricciones de simetría.

- **Comportamiento computacional**:
  Los datos experimentales revelan una complejidad inherentemente exponencial, evidenciada por:

- Crecimiento no lineal del tiempo de solución: 320 ms ($n = 6$) vs 20,311 ms ($n = 20$) en CHUFFED
- Incremento superlineal de propagaciones: 24 en `Prueba_1` vs 1.6M en `Prueba_3`

Este fenómeno coincide con la NP-dificultad teórica de problemas de empaquetamiento.

- **Efectividad:** El análisis comparativo de `int_search` demostró que:
  - `smallest + indomain_split` genera hasta 3 a más fallos que `first_fail` en CP-SAT
  - La heurística `input_order + indomain_min` redujo un 22 % el tiempo promedio frente a otras variantes
  - La fijación del primer cuadrado ($x_1 = 0, y_1 = 0$) disminuyó un 15 % la profundidad del árbol de búsqueda
- **Limitaciones identificadas:**
  - Escalabilidad deficiente para $n > 20$ (tiempos >6s incluso con CP-SAT)
  - Alta sensibilidad a la distribución espacial inicial (ej: `Prueba_3` requirió 459K fallos)
  - Dependencia crítica del solver (GECODE falló en 3/6 pruebas vs 0/6 de CP-SAT)

Teniendo en cuenta los puntos anteriores, se demuestra que el modelado CSP constituye un enfoque viable para resolver problemas de construcción de un rectángulo, siempre que se consideren sus limitaciones computacionales. Los resultados prueban que, aunque la formulación propuesta garantiza corrección y completitud para instancias pequeñas ($n \leq 10$), su naturaleza exige estrategias adaptativas en casos complejos: desde la selección rigurosa de solvers (CP-SAT por su equilibrio entre velocidad y robustez) hasta la incorporación de heurísticas específicas (como el preordenamiento descendente de cuadrados). Las pruebas no solo validan el modelo, sino que revelaron patrones como la sensibilidad al ordenamiento inicial y el crecimiento exponencial.

2.7. Recursos del Proyecto

- **Pruebas .DZN:** Ver en Google Drive
- **Modelos .MZN:** Ver en Google Drive
- **Análisis de las pruebas:** Ver hoja de cálculo
- **Repositorio del proyecto:** Ir a GitHub

3. Conclusiones

En este informe se trabajaron varios ejercicios relacionados con el modelamiento de restricciones usando MiniZinc, lo que nos permitió entender mejor cómo funcionan los CSP (problemas de satisfacción de restricciones) y cómo se pueden aplicar en la práctica. Gracias a estos modelos, fue posible representar de forma clara las condiciones de cada problema y resolverlos.

También se probaron diferentes estrategias de búsqueda, y pudimos notar que dependiendo de cuál se elija, el rendimiento puede cambiar bastante. Algunas heurísticas, como `first_fail` o `dom_w_deg`, pueden hacer que el tiempo de ejecución mejore, lo cual es muy útil cuando se trata de problemas grandes o más complicados. Esto demuestra que no solo importa cómo se modela el problema, sino también qué estrategia se usa para resolverlo.

Podemos concluir que este taller fue una buena oportunidad para explorar y entender lo importante que es la programación por restricciones. Además de permitir encontrar soluciones eficientes, nos da la posibilidad de experimentar con diferentes enfoques y aprender cuál se adapta mejor a cada situación.

4. Bibliografía

Referencias

[1] R. Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.

[2] MiniZinc Team, *MiniZinc: The Modelling Language*, 2024. https://www.minizinc.org

[3] *MiniZinc Documentation*, https://www.minizinc.org/doc-latest/en/index.html