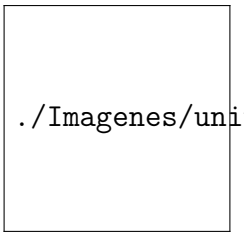

Taller 1: Modelamiento e Implementación de CSPs

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.

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