Programación por Restricciones
Planificación de los ensayos en una telenovela

Victor Hugo Ortega Gomez
1532342
Alejandro Orozco Hurtado
1744439
David Santiago Cortés
1745133

Junio 2022

Índice

1. Introducción

Como proyecto final de la materia Programación por Restricciones de la Escuela de Ingeniería de Sistemas, Universidad del Valle. Los estudiantes deberán estudiar el problema de programación de ensayos de una telenovela, conocido en la literatura como *Talent Scheduling Problem / Scene Allocation Problem* y además resolverlo modelandolo como un problema de satisfacción de restricciones formalmente, para luego implementarlo en Minizinc y analizar su desempeño frente a distintas entradas.

El problema consiste en determinar el orden en el cual se deben ensayar un conjunto de escenas de una telenovela, de tal forma que el costo total del proyecto sea el mínimo posible, teniendo en cuenta que cada actor cobra por cada unidad de tiempo presente en el set y además este mismo no se puede ir hasta que haya ensayado todas las escenas donde participa.

A continuación se presentarán los modelos propuestos, desde el más sencillo hasta el más completo, junto con sus resultados y análisis respectivos.

2. Modelo Básico

El primer modelo que se presenta es un modelo preliminar que no pretende eliminar simetrías, tan solo cumple con la restricción fundamental del problema: un actor determinado debe estar en el set hasta que ya no tenga más escenas por ensayar.

2.1. Parámetros

- (Parámetro implícito) n, número de actores ≥ 1.
- (Parámetro implícito) m, número de escenas ≥ 1.
- **ACTORES**: enumeración de tamaño n que contiene los nombres de cada actor.
- **Escenas**: matriz de n filas por $m+1$ columnas. $\forall x \in Escenas\ x = [0 \dots \infty]$. Cada fila representa un actor y las primeras m columnas una escena, luego cada celda (i, j) indica si el actor i debe ensayar (1) o no (0) la escena j. La columna $m+1$ específica el costo por unidad de tiempo del actor i. $i = [1 \dots n]$ y $j = [1 \dots m]$.
- **Duración**: arreglo de tamaño m el cual muestra la duración en unidades de tiempo t para cada escena, con $t \in \mathbf{N}$.

2.2. Variables

- ***Orden***: arreglo de tamaño m que especifica el orden en el que deben ensayarse las escenas: La escena $Orden[i]$ debe ensayarse en el i-ésimo puesto. $i = [1 \dots m]$. Naturalmente, cada $Orden[i] \in [1 \dots m]$

- ***Costo***: variable entera que indica el costo total de la solución. Su cota inferior y superior se definen así:

$$\text{Cota inferior: } \sum_{i=1}^{n}(\sum_{k=1}^{m} Duracion[k] * Escenas[i, m+1] * Escenas[i, k]) \quad (1)$$

$$\text{Cota superior: } \sum_{i=1}^{n}((\sum_{t \in \text{Duracion}} t) * Escenas[i, m+1]) \quad (2)$$

  La ecuación (1) corresponde al mejor caso posible, todos los actores están en el set únicamente durante las escenas que deben ensayar.

  La ecuación (2) muestra el peor caso, todos los actores están en el set durante toda la duración de los ensayos. Es decir, se les debe pagar por la totalidad de la duración del proyecto.

- ***Intervalos***: matriz de n filas y 2 columnas: Cada elemento toma valores de 1 a m y cada fila índica el *indice* de la primera y última escena en la que el actor i actúa, de acuerdo a su posición en $Orden$. Para un actor i arbitrario:

$$intervalos[i, 1] = min(\{k | k \in [1 \dots m] \wedge Escenas[i, Orden[k]] = 1\}) \quad (3)$$

$$intervalos[i, 2] = max(\{k | k \in [1 \dots m] \wedge Escenas[i, Orden[k]] = 1\}) \quad (4)$$

  La ecuación (3) es el índice de la primera escena dónde el actor i está en el set, según el orden propuesto

  La ecuación(4) es el índice de la última escena hasta que el actor i ya puede irse, según el orden propuesto.

2.3. Restricciones

1. Todas las escenas deben ensayarse exactamente una vez: $|Set(Orden)| = m$
2. El actor i solo se puede ir del set hasta que ya no le queden más escenas por ensayar, según el orden propuesto. Sea $l = intervalos[i, 1]$ y $l' = intervalos[i, 2]$:

$$costo = \sum_{i=1}^{n} \sum_{j=l}^{l'} Duracion[orden[j]] * Escenas[i, m+1] \quad (5)$$

2.4. Función objetivo

- Minimizar $costo = \sum_{i=1}^{n} \sum_{j=l}^{l'} Duracion[orden[j]] * Escenas[i, m+1]$

2.5. Implementación

2.5.1. Notas sobre la implementación

- Para facilitar la lectura del modelo, se crearon dos parametros extra (*rangoActores* y *rangoEscenas*), que son solamente el rango de 1 hasta n y m respectivamente.
- Se crearon dos funciones, c/u para extraer el primer y último índice l y l' mencionado anteriormente.

2.5.2. Código

```
include "alldifferent.mzn";

% ------------------------Parametros de entrada----------------------------
enum ACTORES;
array[int,int] of int: Escenas;
array[int] of int: Duracion;

% ------------------------Constantes---------------------------------------
int: n = max(rangoActores); % Número de actores.
int: m = length(Duracion); % Número de escenas

set of int: rangoActores = index_set(ACTORES);
set of int: rangoEscenas = 1..m;

% ------------------------Variables----------------------------------------

% Orden en el que deben ensayarse las escenas.
array[rangoEscenas] of var rangoEscenas: orden;

% Indices de la primera y última escena de cada actor.
array[rangoActores, 1..2] of var rangoEscenas: intervalos;

% u. de tiempo en set por cada actor
array[rangoActores] of var 0..max(Duracion)*m: tiemposxActor;

% Costo total
var int: costo = sum(i in rangoActores) ((tiemposxActor[i] * Escenas[i, m+1]));


% ----------------------------------------Funciones-----------------------------------
function var int: getLowerBound(int: actor) =
min([ k | k in rangoEscenas where Escenas[actor,orden[k]] == 1]);

function var int: getUpperBound(int: actor) =
```

```
max([ k | k in rangoEscenas where Escenas[actor, orden[k]]== 1]);
% ---------------------------------------Restricciones------------------------------

% Cada escena se graba una vez.
constraint alldifferent(orden);

constraint forall(i in rangoActores)
(tiemposxActor[i] = sum(j in intervalos[i,1]..intervalos[i,2])(Duracion[orden[j]]));

% Se llena la matriz intervalos de acuerdo al primer y
% último indice de la primer y ultima escena respectivamente.
constraint forall(i in rangoActores)
(intervalos[i,1] = getLowerBound(i) / intervalos[i,2] = getUpperBound(i));

constraint costo <= sum(i in rangoActores) (sum(Duracion) * Escenas[i, numEscenas+1]);

% En el mejor de los casos, ningún actor está en el set sin hacer nada.
constraint costo >= sum(i in rangoActores) (
sum(k in rangoEscenas where Escenas[i, k] > 0)
(Duracion[k]*Escenas[i,m+1])
);

% ----------------------------------Objetivo-------------------------------------

solve minimize costo;
%solve :: int_search(orden, dom_w_deg, indomain_min) minimize costo;
```

2.6. Resultados

Para probar el modelo se utilizaron los siguientes archivos de entrada, en orden creciente de complejidad.

- Trivial1.dzn
- Trivial1-2.dzn
- Trivial1-3.dzn
- Trivial1-4.dzn
- Desenfreno1.dzn

2.7. Análisis

2.7.1. Selección de distintas estrategias de búsqueda

El modelo se probó con los mismos datos de entrada, pero con distintas estrategias de búsqueda para comparar los tamaños de los árboles de búsqueda generados y así encontrar

Datos	Tiempo Gecode	Tiempo Chuffed	Costo Gecode	Costo Chuffed
Trivial1	133ms	128ms	255	255
Trivial1-2	236ms	187ms	405	405
Trivial1-3	17,943s	273ms	580	640
Trivial1-4	2m 36s	336ms	766	1016
Desenfreno1	20h	1m 7s	961	1523

Tabla 1: Tiempos de ejecución sin estrategia de búsqueda

la estrategia más favorable: aquella que reduzca el espacio de búsqueda y posiblemente el tiempo de ejecución. Para ello se utilizó una búsqueda de enteros (dada la forma en que se definió el modelo) y el solver **Gecode Gist** sobre la instancia del problema Trivial1.dzn.

2.7.2. Baseline, el árbol sin ninguna estrategia:

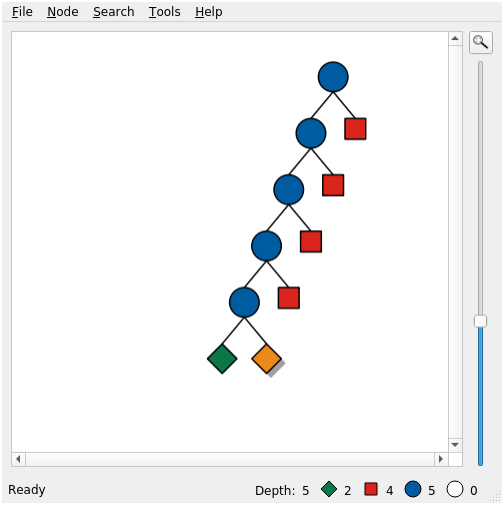


Figura 1: Arbol generado sobre Trivial1.dzn (sin estrategias)

2.7.3. Escoger variable con el dominio más pequeño y asignarle el valor más pequeño del mismo

```
int_search(orden, first_fail, indomain_min)
```

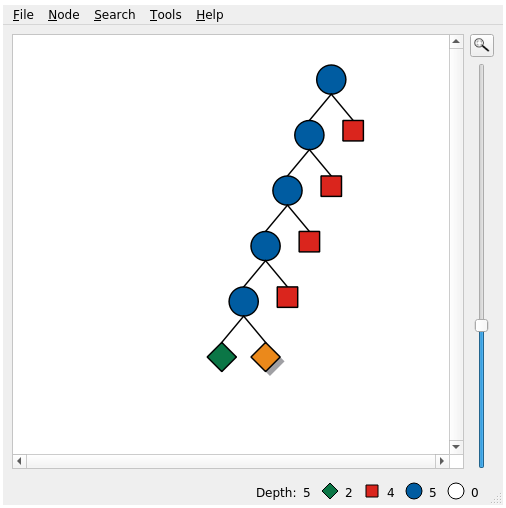


Figura 2: Arbol generado con first_fail, indomain_min

No presenta ninguna diferencia notable respecto al baseline.

2.7.4. Escoger variable con el dominio más pequeño y asignarle la mediana

```
int_search(orden, first_fail, indomain_median)
```

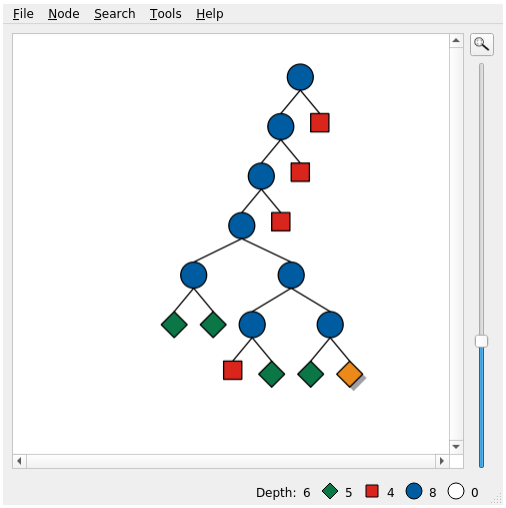


Figura 3: Arbol generado con first_fail, indomain_median

Respecto al caso base, se puede notar un árbol con mayor profundidad e incluso mayor número de soluciones generadas. La solución óptima que encuentra tiene el mismo costo, pero el orden es distinto, de hecho todas tienen un orden distinto, lo que empieza a dar pistas sobre posibles simetrías en las soluciones.

2.7.5. Dominio con pesos y dividir el dominio

```
int_search(dom_w_deg, indomain_split)
```

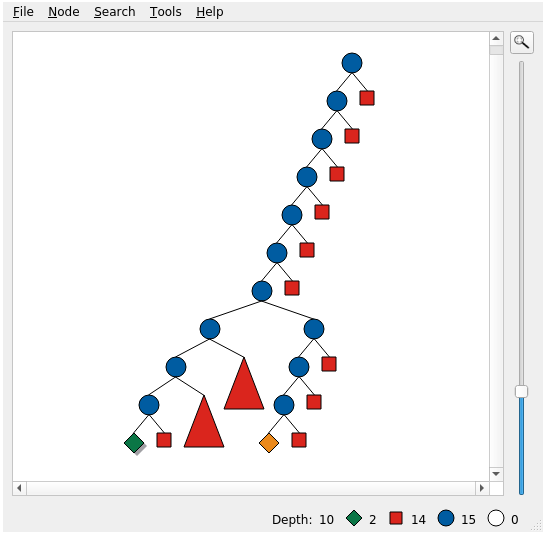


Figura 4: Árbol generado con dom_w_deg, indomain_split

Esta estrategia genera un árbol más grande que el caso base y la solución óptima que encuentra es una simetría del caso base, se invierte el orden.

2.7.6. Escoger la variable con el valor más pequeño en su dominio

```
int_search(smallest, indomain_min)
```

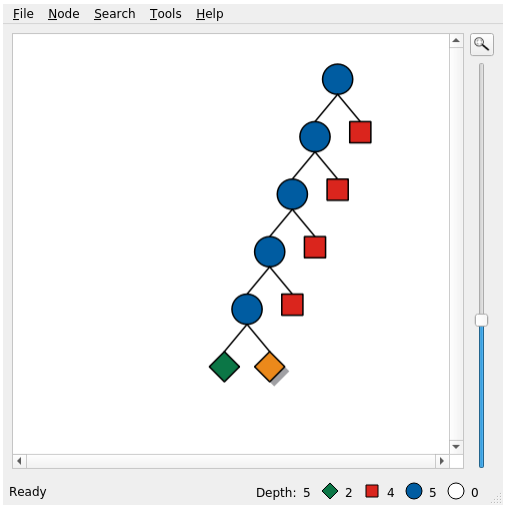


Figura 5: Árbol generado con smallest, indomain_min

La profundidad y cantidad de soluciones generadas son iguales al caso base, sin embargo, la solución óptima encontrada es la misma simetría de la estrategia anterior

Las ejecuciones se realizaron con Trivial1 para poder visualizar mejor el árbol generado, aún así consideramos que la ejecución se puede extrapolar a instancias más complejas debido a que el modelo aquí propuesto es bastante general/simple.

3. Modelo Básico con eliminación de simetrías

Este modelo pretende extender el anterior añadiendo restricciones extras que buscan eliminar las simetrías detectadas previamente. Esto se logra restringiendo deliberadamente el espacio de búsqueda, de tal forma que no se exploren algunas ramas que llevan a soluciones capaces de ser generadas a partir de otras, esto con el fin de tener un modelo más robusto y rápido.

A continuación se introducen todos los elementos *extras* que se añaden al modelo básico.

3.1. Restricciones

1. La primera restricción tiene que ver con escenas dónde ningún actor tiene parte. Este tipo de escenas sin costo alguno pueden ubicarse al inicio o al final de la obra sin afectar el costo de la misma, introducimos una restricción que obliga al modelo a poner este tipo de escenas al inicio del arreglo $Orden$.
   Se identifican porque son columnas solo con ceros, si sabemos que hay k escenas de este tipo entonces:

   $$zeroedColums = \{j \mid j \in 1..m \wedge \sum Escenas[.., j] = 0\} \quad \text{(Guardar las columnas cero)}$$
   $$orden[d] = zeroedColumns[d],\ \forall d \in 1..k$$

   Esto se logró en Minizinc con las constantes `numZeroedColumns` y `zeroedColumns`:
   Hace las veces de k:

   ```
   int: numZeroedColumns = sum(
       [1 | j in rangoEscenas where sum(Escenas[..,j]) == 0]
   );
   ```

   ```
   array[1..numZeroedColumns] of rangoEscenas: zeroedColumns = [
       j | j in rangoEscenas where sum(Escenas[..,j]) == 0
   ];
   ```

   La restricción, los primeras k posiciones de $Orden$ corresponden a escenas-cero.

   ```
   constraint symmetry_breaking_constraint(
       forall(k in 1..numZeroedColumns)
           (orden[k] = zeroedColumns[k])
   );
   ```

2. La segunda restricción detecta simetrías producto de escenas idénticas, es decir, actúan los mismos actores y además tienen la misma duración. Luego cualquier permutación de estas escenas no tiene efecto sobre el costo total del proyecto.

Concretamente, dos columnas j y j', $j, j' = [1..m]$ que pueden ser o no contiguas, son idénticas ssi:

$$\begin{aligned} \forall i \in [1..n]\ Escenas[i,j] &= Escenas[i,j'] && \text{(Todos sus elementos son iguales)} \\ Duracion[j] &= Duracion[j'] && \text{(Tienen la misma duración)} \\ \sum_{i=1}^{n} Escenas[i,j] &\neq 0 && \text{(No son columnas-cero)} \end{aligned}$$

Para detectar ambos casos (contiguas o no contiguas) se propuso la siguiente restricción:

```
constraint symmetry_breaking_constraint(
forall(j in 1..m-1, k in j+1..m) (
    if Escenas[..,j] = Escenas[..,k]  && Duracion[j] = Duracion[k] &&
        sum(Escenas[..,j]) != 0
    then orden[j] < orden[k]
    endif
));
```

La cual define dos índices $j = [1..m-1]$ y $k = [j+1..m]$ para recorrer las columnas de la matriz *Escenas* de tal forma que nunca se comparen la misma escena consigo misma. Dado que j va detrás de k por una unidad, es posible comparar escenas contiguas y a su vez, dado que k avanza "más rápido", es decir, j avanza cada que k completa un ciclo, entonces también es posible comparar escenas/columnas, que no están contiguas.

Con esto en mente, cuando se encuentran dos escenas cualesquiera que cumplen con la definición de escenas idénticas, se establece una restricción de ordenamiento que las obliga a ocupar un puesto específico en el arreglo final *orden*, efectivamente, reduciendo el espacio de búsqueda considerablemente, pues estas escenas ya no podrán permutar libremente.

3.1.1. Recuperar soluciones

- Para la primera restricción, si se obtiene una solución $orden = [l, \ldots]$ dónde l sabemos es una escena cero entonces:

  $$orden' = [\ldots, l]$$

  sería la solución que se perdió y que se recupera a partir de la que se obtuvo, ubicando la escena l en la última posición del arreglo.

- Si se tiene una solución del tipo

  $$orden = [\ldots, j, k, \ldots]$$

  donde j y k son escenas idénticas, entonces la solución que se perdió se recupera intercambiando su orden de aparición, sin afectar las escenas colindantes a ellas:

  $$orden' = [\ldots, k, j, \ldots]$$

- En el caso de una solucipon del tipo

$$orden = [\ldots, j, \ldots, k, \ldots]$$

j y k escenas idénticas que evidentemente no son consecutivas. Entonces, de forma analoga, si se intercambian sus ubicaciones en el arreglo, se recuperaría la solución eliminada por las restricciones:

$$orden = [\ldots, k, \ldots, j, \ldots]$$

- En el caso de que hubiera n escenas idénticas, contiguas o no, cada permutación de ellas y solo entre ellas, diferente a la solución propuesta, sería entonces una solución simétrica que se recupera.

3.2. Resultados

Para analizar la efectividad de las restricciones introducidas se utilizaron las instancias que tienen un potencial para generar simetrías:

- Trivial1
- Trivial1-4
- Trivial1-5
- Desenfreno1

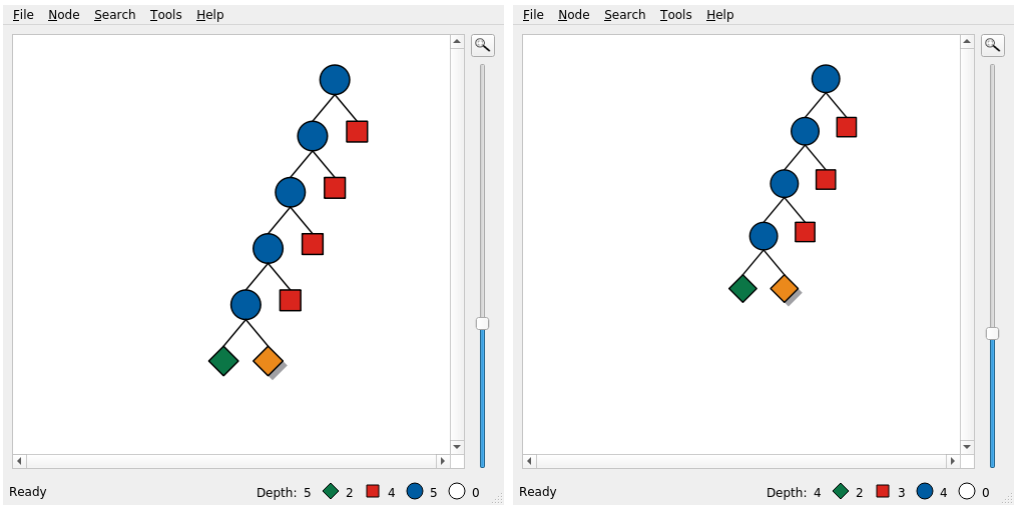
3.2.1. Tiempos

Datos	Tiempo Gecode	Tiempo Chuffed	Costo Gecode	Costo Chuffed
Trivial1	133ms	150ms	255	255
Trivial1-4	1m 4s	273ms	766	950
Trivial1-5	200ms	200ms	216	216
Desenfreno1	20h	-	961	-

Tabla 2: Tiempos de ejecución sin estrategia de búsqueda

3.2.2. Árboles

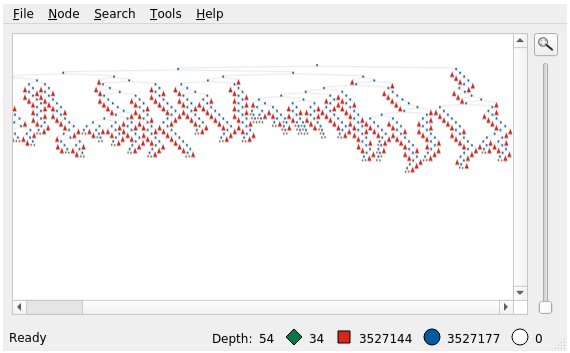
Dado que el principal interés de eliminar simetrías es revisar si en verdad se redujo el espacio de búsqueda, a continuación se muestran los árboles generados antes y después de eliminar simetrías.



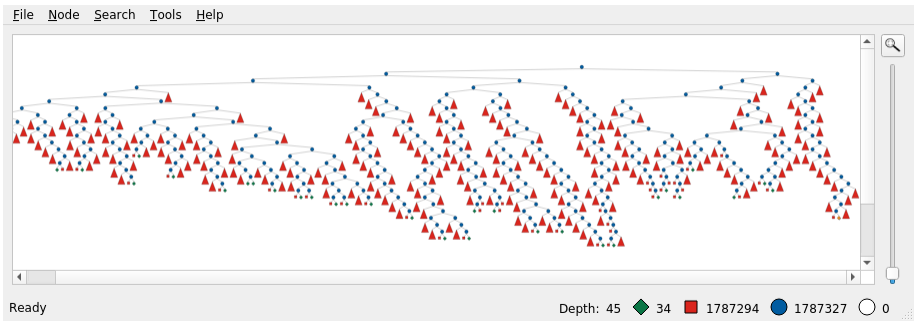
(a) Antes (b) Después

Figura 6: Árboles Trivial1

Como se observa, el árbol generado después de aplicar las restricciones de simetrías es un nivel más pequeño al original, esto debido a que la instancia contiene simetrías producto de columnas cero y además dos columnas idénticas.



(a) Antes

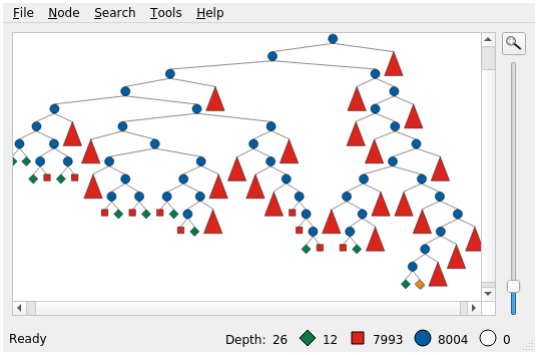


(b) Después

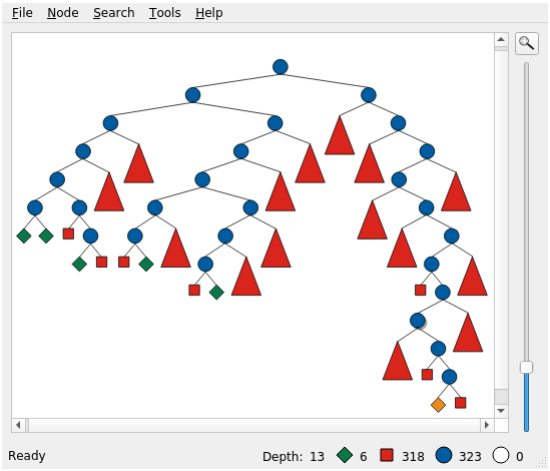
Figura 7: Árboles Trivial1-4

El árbol después explora alrededor de 1 millón ochocientos menos nodos que el árbol

original.



(a) Antes



(b) Después

Figura 8: Árboles Trivial1-5

La instancia Trivial1-5 fue creada con muchas simetrías para obtener una evaluación clara del desempeño, acá se puede observar una reducción del 96 % en la cantidad de nodos explorados, se redujo la profundidad a la mitad y se eliminaron la mitad de las soluciones encontradas anteriormente sin haber perdido el óptimo.

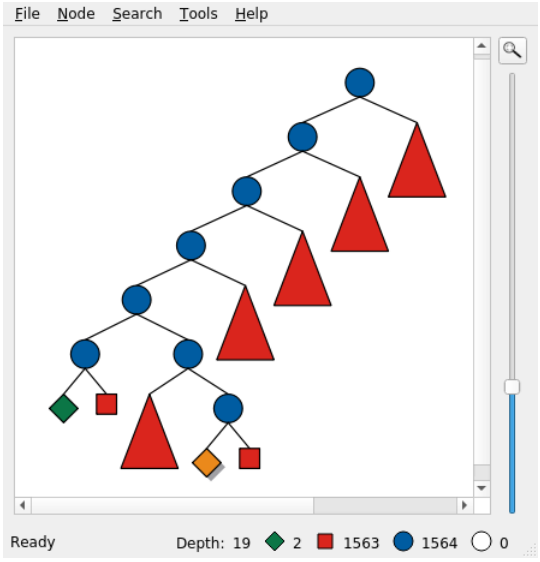
De las anteriores ejecuciones concluimos que las restricciones que se añadieron son buenas para reducir el tamaño de los árboles de búsqueda, e incluso, reducir el tiempo de ejecución, como fue el caso de Trivial1-4.

3.3. Análisis

A continuación se estudiarán los árboles generados con distintas estrategias de búsqueda, para determinar si existe alguna que sea mejor en general, o si la mejor estrategia depende de cada instancia específica. Para ello se utilizó la instancia Trivial1-2 y la solución encontrada por Desenfreno1 después de cinco minutos de ejecución.

3.3.1. Baseline

Sin ninguna estrategia de búsqueda, pero con eliminación de simetrías, Gecode Gist genera el árbol siguiente



Desenfreno1 después de cinco minutos:

```
Orden de las escenas:
[19, 18, 16, 17, 14, 15, 20, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Costo: 961
```

3.3.2. Dominio con pesos y dividir el dominio

`int_search(dom_w_deg, indomain_reverse_split)`

Lo que esta estrategia busca es escoger la variable con el dominio más pequeño dividido por el grado ponderado, que es la cantidad de veces que ha estado en una restricción que causó la falla anteriormente en la búsqueda. Luego a esa variable escogida se le asigna el valor producto de bisectar el dominio en dos partes y excluir la mitad inferior.

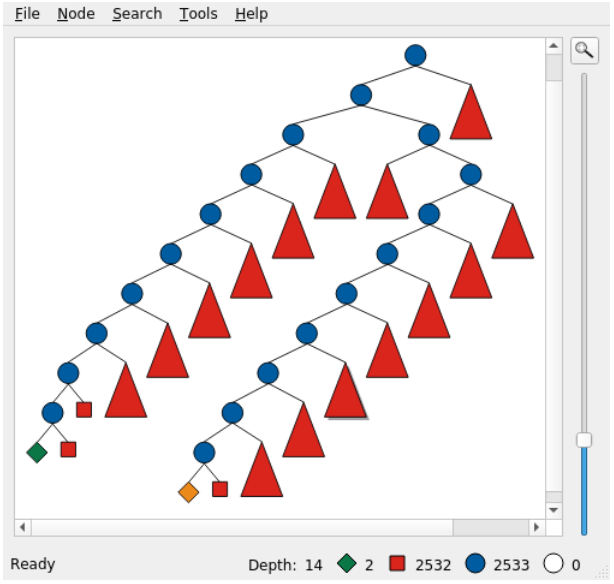


Figura 9: Trivial1-2 domw_deg, indomain_reverse_split

Desenfreno1 después de cinco minutos:

```
Orden de las escenas:
[20, 19, 18, 17, 16, 15, 14, 13, 12, 5, 3, 2, 8, 7, 9, 6, 10, 11, 1, 4]
Costo: 903
```

Si bien el arbol generado explora alrededor de mil nodos más, a su vez encuentra las mismas dos soluciones y tiene dos niveles menos que el caso base. Este comportamiento suele suceder, ya que las anotaciones guían la búsqueda en un orden específico y suelen generarse árboles un poco más grandes (o más pequeños) que encuentran la misma solución.

Algo interesante de esta estrategia es que logró hallar una mejor solución a Desenfreno1 en tan solo 5 minutos, comparado con las 10 horas del caso base. Esto indica que posiblemente nuestro modelo se está quedando "estancado" en ciertas regiones del árbol sin llegar a explorar otras que puede que tengan un valor menor, tal como se ha evidenciado en este ejemplo.

3.4. Variable con más restricciones y el valor más pequeño

`int search(occurrence, indomain_min)`

Esta estrategia hace la propagación escogiendo la variable con más restricciones sobre ella y le asigna el valor más pequeño del dominio.

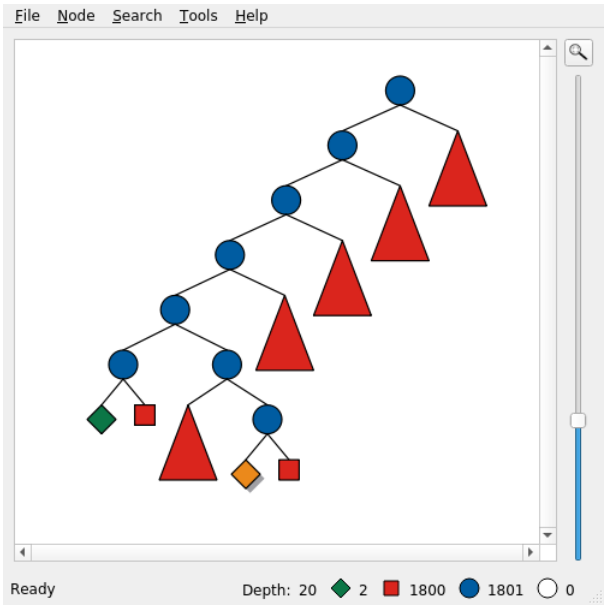


Figura 10: Trivial1-2occurrence, indomain_min

De igual forma que con la estrategia anterior, el árbol es ligeramente más grande, pero sin presentar mayor cambio en el tiempo o en el costo de la solución encontrada.
Desenfreno1 después de cinco minutos:

```
Orden de las escenas:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 20, 15, 14, 17, 16, 18, 19]
Costo: 961
```

En este caso, la solución hallada es la misma que el caso base.

3.5. Variable con el dominio más grande

`int_search(orden, anti_first_fail, indomain)`
Propaga de acuerdo a la variable con el dominio más grande y le asigna un valor del dominio de acuerdo a un orden ascendente.

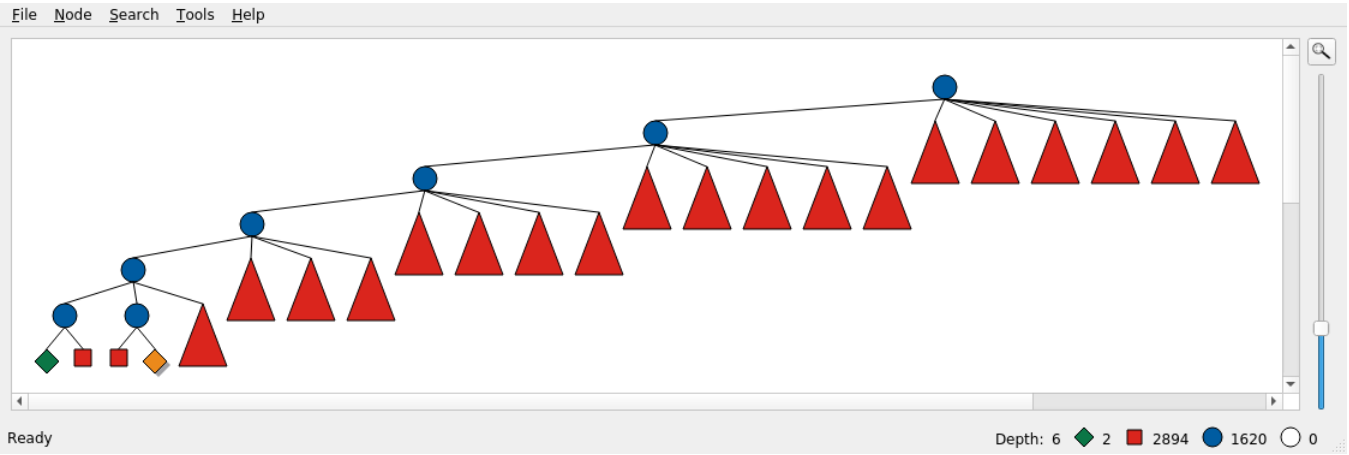


Figura 11: Trivial1-2 (orden, anti_first_fail, indomain

Con esta estrategia de búsqueda el árbol es menos profundo, pero al mismo tiempo explora más nodos, es más ancho. La búsqueda es en amplitud más que en profundidad. Puede ser una buena estrategia para otras instancias del problema, donde la solución se encuentre en posiciones que una búsqueda por profundidad se demore mucho en explorar.

```
Orden de las escenas:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 20, 15, 14, 17, 16, 18, 19]
Costo: 961
```

En este caso, la solución hallada es la misma que el caso base.

3.6. Conclusión

Esta ronda de pruebas sirvió para determinar que debido a la naturaleza combinatoria del problema, las soluciones que se encuentran y el número de nodos explorados/expandidos dependen en gran medida del orden en el que se hace la propagación, como se hace la búsqueda y el valor que se asigna a las variables.

Al mismo tiempo, es posible que la búsqueda se estanque en óptimos locales cuando se trata de instancias grandes del problema, pues los árboles generados son de más de 10 millones de nodos. Las anotaciones de búsqueda buscan ayudar en este aspecto y en efecto fue posible hallar una mejor solución a Desenfreno1 en 5 minutos, que la propuesta por el modelo inicialmente.

Sin embargo, ninguna de las estrategias escogidas (a excepción del caso Desenfreno1) generó un árbol más pequeño (tanto en nodos como en número de soluciones al mismo tiempo), por ende la estrategia de búsqueda para este modelo corresponde a la por defecto de Minizinc: búsqueda por profundidad.

3.7. Extra

Dado el relativo buen resultado que dio la primera estrategia con Desenfreno1 decidimos realizarla de nuevo, pero con la capacidad de devolverse utilizando la anotación `restart`, sin embargo, no se encontró una solución con costo menor a 901.

4. Modelo Extendido

En el modelo extendido se agrega dos restricciones nuevas:

1. Disponibilidad de los actores: en general todos los actores deben estar disponibles todo el tiempo, pero hay ciertos actores que pueden imponer restricciones como por ejemplo un máximo número de unidades de tiempo en el estudio. Por ejemplo, Actor 1 podría disponer de solo 10 unidades de tiempo para estar en el estudio.

2. Puede haber actores que no la van bien con otros actores y sería ideal que el tiempo que estuvieran en el estudio al mismo tiempo fuera el menor posible (pero tenga en cuenta que la prioridad es reducir costo).

4.1. Parámetros

- (Parámetro implícito) c, número de conflictos ≥ 1.
- **Disponibilidad**: matriz de n filas por 2 columnas. Por cada fila la primera columna representa el actor y la segunda columna el tiempo $t \in \mathbf{N}$ que tienen disponible para estar en el set. Si el tiempo que tienen disponible es 0 significa que no tienen restricción de tiempo.
- **Evitar**: matriz de c filas por 2 columnas. Cada dupla representa actores que prefieren evitarse entre ellos.

4.2. Variables

- ***tiemposxActor***: arreglo de tamaño n que representa cuanto tiempo $t \in \mathbf{N}$ pasa cada actor en set.
- ***compartido***: arreglo de tamaño c que representa cuantas escenas $\in [0..m]$ comparten en el set actores que se quieren evitar.

- ***objetivo***: variable entera $\in \mathbf{N}$ que se busca optimizar, está compuesta por la variable *costo* y la cantidad de escenas que comparten los actores que se quieren evitar.
  Su funcionamiento se explicará a más profundidad en la sección 4.4. Función objetivo.

4.3. Restricciones

1. Se verifica que la matriz tiemposxActor esté llena con el tiempo que dura un actor en set, desde su primera escena p hasta su última escena u.

$$tiemposxActor[i] = \sum_{i=p}^{u} Duracion[orden[i]] \tag{6}$$

2. Se verifica para cada actor su tiempo en set $tset$ sea menor o igual a el tiempo que tienen disponible tdi.

$$\forall i \in [1..n]\ (if\ tdi_i == \ 0\ then\ TRUE\ else\ tset_i <= tdi_i) \tag{7}$$

4.4. Función objetivo

Buscamos minimizar el costo de producción y minimizar la cantidad de escenas en set que comparten actores que se quieran evitar, debido a que reducir el costo de producción tiene prioridad podemos hacer uso del método lexicográfico en el cual las preferencias se imponen ordenando las funciones objetivo según su importancia, en lugar de asignar pesos.

Escalamos el primer objetivo(costo) por un valor que sea al menos como el máximo del segundo objetivo(actores que se quieran evitar en set). Esto asegura que cualquier mejora en el primer objetivo triunfe sobre cualquier mejora en el segundo objetivo. El resultado sería entonces el óptimo lexicográfico.

$$objetivo = costo * (c * m + 1) + \sum_{i=1}^{c} compartido[i]$$

- Minimizar *objetivo*

4.5. Implementación

4.5.1. Notas sobre la implementación

- Para facilitar la lectura del modelo, se creo un parametros extra ($rangoConflictos$), que es solamente el rango de 1 hasta c.

- Se creó una función que recibe una dupla de los actores que prefieren evitarse, de esta dupla se obtiene de *Intervalos* la primera p_i y la última u_i escena de la dupla de actores que quieren evitarse. Se cuentan las veces(escenas) que están juntos en el set.

$$sum\ (p_2 < i < u_2 \mid i\ \in\ [p_1..u_1])$$

4.6. Resultados

Para probar el modelo extendido se utilizaron los siguientes archivos de entrada, cada archivo representa un caso donde se muestra en funcionamiento las restricciones adicionales.

- Trivial2.dzn
  Caso base con el objetivo de probar el modelo en general.

- Trivial2-2.dzn
  El tiempo disponible de un actor es menor al tiempo que se necesita al actor en set.
  Caso para comprobar el funcionamiento de la restricción de tiempo.

- Trivial2-3.dzn
  Múltiples posibles soluciones del orden de las escenas con un mismo costo de producción pero con diferentes tiempos compartido entre actores que se quieren evitar.
  Caso para comprobar el funcionamiento de la función de optimización lexicográfica.

- Desenfreno2.dzn

Datos	**Costo Gecode**	**Costo Chuffed**	**Costo COIN-BC**
Trivial2	450	450	450
Trivial2-2	UNSATISFIABLE	UNSATISFIABLE	UNSATISFIABLE
Trivial2-3	360	360	360
Desenfreno2	1269	1040	1173

Tabla 3: Variable costos sin estrategia de búsqueda

Datos	**Costo Gecode**	**Costo Chuffed**	**Costo COIN-BC**
Trivial2	5853	5853	5853
Trivial2-2	UNSATISFIABLE	UNSATISFIABLE	UNSATISFIABLE
Trivial2-3	4680	4680	4680
Desenfreno2	102826	84277	95059

Tabla 4: Variable objetivo sin estrategia de búsqueda

Datos	**Tiempo Gecode**	**Tiempo Chuffed**	**Tiempo COIN-BC**
Trivial2	336msec	333msec	6s 491msec
Trivial2-2	353msec	356msec	1s 661msec
Trivial2-3	451msec	342msec	4s 694msec
Desenfreno2	Stopped 25m	1m 30s	Stopped 25m

Tabla 5: Tiempos de ejecución sin estrategia de búsqueda

4.7. Análisis

Comparando los resultados obtenidos de cada Solver se puede notar que para problemas triviales, es decir problemas con pocos factores (actores, escenas) el resultado obtenido es el mismo pero en el tiempo que se demoran en ejecutarse se nota que COIN-BC toma particularmente un largo tiempo en terminar, a diferencia de Geocode o Chuffed los cuales comparten un similar tiempo de ejecución.

Para el caso Desenfreno2 el cual es un caso con gran cantidad de factores (8 actores y 20 escenas) Chuffed llegó a una solución en relativamente poco tiempo, y una respuesta mejor que las que ofrecían COIN-BC y Geocode durante el tiempo que estuvieron corriendo. Es posible que COIN-BC y Geocode cuando terminaran de ejecutar ofrecieran una mejor solución pero que esto suceda no es una certeza, si se requiere obtener una solución aceptable y se dispone de poco tiempo puede que lo recomendable sea usar Chuffed.

5. Implementación

5.1. Código modelo básico

```
include "alldifferent.mzn";

% ------------------------Parametros de entrada----------------------------
enum ACTORES;
array[int,int] of int: Escenas;
array[int] of int: Duracion;

% ------------------------Constantes---------------------------------------
int: n = max(rangoActores); % Número de actores.
int: m = length(Duracion); % Número de escenas

set of int: rangoActores = index_set(ACTORES);
set of int: rangoEscenas = 1..m;

int: peorCaso = sum(i in rangoActores) (sum(Duracion) * Escenas[i, m+1]);
int: mejorCaso = sum(i in rangoActores) (
sum(k in rangoEscenas where Escenas[i, k] == 1)
(Duracion[k]*Escenas[i,m+1])
);

int: numZeroedColumns = sum([1 | j in rangoEscenas where sum(Escenas[..,j]) == 0]);
array[1..numZeroedColumns] of rangoEscenas : zeroedColumns = [
j | j in rangoEscenas where sum(Escenas[..,j]) == 0
];

% ------------------------Variables----------------------------------------
```

```
% Orden en el que deben ensayarse las escenas.
array[rangoEscenas] of var rangoEscenas: orden;

% Indices de la primera y última escena de cada actor.
array[rangoActores, 1..2] of var rangoEscenas: intervalos;

% u. de tiempo en set por cada actor
array[rangoActores] of var 0..infinity: tiemposxActor;

var mejorCaso..peorCaso: costo;

% --------------------------------Funciones----------------------------------------
function var int: getLowerBound(int: actor) =
min([ k | k in rangoEscenas where Escenas[actor,orden[k]] == 1]);

function var int: getUpperBound(int: actor) =
max([ k | k in rangoEscenas where Escenas[actor, orden[k]] == 1]);

% --------------------------------Restricciones-------------------------------------

% Cada escena se graba una vez.
constraint alldifferent(orden);

% Se llena la matriz intervalos de acuerdo al primer y ultimo indice de la primer y
%ultima escena respectivamente.
constraint forall(i in rangoActores)
(intervalos[i,1] = getLowerBound(i) && intervalos[i,2] = getUpperBound(i));

constraint costo = sum(i in rangoActores) (
                    sum(j in intervalos[i,1]..intervalos[i,2])
                      (Duracion[orden[j]])*Escenas[i,m+1]);

% Todas las escenas que valgan cero van de primero en el orden.
constraint symmetry_breaking_constraint(
forall(k in 1..numZeroedColumns) (orden[k] = zeroedColumns[k]));

% Forzar un orden en aquellas escenas no contiguas que sean iguales y que no sean
columnas cero.
constraint symmetry_breaking_constraint(
forall(j in 1..m-1, k in j+1..m) (
    if Escenas[..,j] = Escenas[..,k] && Duracion[j] = Duracion[k] &&
        sum(Escenas[..,j]) != 0
    then orden[k] < orden[j]
    endif));
```

```
% ------------------------------------Objetivo---------------------------------------
solve minimize costo;

output ["Orden de las escenas: (orden)"] ++
["Costo: (costo)"]++
["peor: (peorCaso)"]++ ["mejor: (mejorCaso)"];
```

5.2. Codigo modelo extendido

```
include "alldifferent.mzn";

% ------------------------Parametros de entrada---------------------------------
enum ACTORES;
array[int,int] of int: Escenas;
array[int] of int: Duracion;
array[ACTORES, 1..2] of int: Disponibilidad;
array[int, 1..2] of int: Evitar;

% ------------------------Constantes--------------------------------------------
int: n = max(rangoActores); % Número de actores.
int: m = length(Duracion); % Número de escenas
int: c = length(Evitar[..,1]); % Número de conflictos

set of int: rangoActores = index_set(ACTORES);
set of int: rangoEscenas = 1..m;
set of int: rangoConflictos = 1..c;

int: peorCaso = sum(i in rangoActores) (sum(Duracion) * Escenas[i, m+1]);
int: mejorCaso = sum(i in rangoActores) (
sum(k in rangoEscenas where Escenas[i, k] == 1)
(Duracion[k]*Escenas[i,m+1])
);

int: numZeroedColumns = sum([1 | j in rangoEscenas where sum(Escenas[..,j]) == 0]);
array[1..numZeroedColumns] of rangoEscenas : zeroedColumns =
    [ j | j in rangoEscenas where sum(Escenas[..,j]) == 0];

% ------------------------------------Variables---------------------------------------

% Orden en el que deben ensayarse las escenas.
array[rangoEscenas] of var rangoEscenas: orden;

% Indices de la primera y última escena de cada actor.
array[rangoActores, 1..2] of var rangoEscenas: intervalos;
```

```
% u. de tiempo en set por cada actor
array[rangoActores] of var 0..infinity: tiemposxActor;

var mejorCaso..peorCaso: costo;

% Costo por escena
array[rangoEscenas] of var 0..infinity: costoxEscena;

%Escenas compartidas entre actores que se quieren evitar
array[rangoConflictos] of var int: compartido =
    [getTimeSpend(Evitar[i,..])| i in rangoConflictos];
% ------------------------------------------------Funciones-------------------------
function var int: getLowerBound(int: actor) =
min([ k | k in rangoEscenas where Escenas[actor,orden[k]] == 1]);

function var int: getUpperBound(int: actor) =
max([ k | k in rangoEscenas where Escenas[actor, orden[k]] == 1]);

function var int: getTimeSpend(array[int] of int: evitan) =
  sum(i in intervalos[evitan[1],1]..intervalos[evitan[1],2])
  (intervalos[evitan[2],1] <= i / i <= intervalos[evitan[2],2]);

% ----------------------------------Restricciones--------------------------------

% Cada escena se graba una vez.
constraint alldifferent(orden);

constraint forall(i in rangoActores)
(tiemposxActor[i] = sum(j in intervalos[i,1]..intervalos[i,2])
(Duracion[orden[j]]));

% Se verifica la matriz intervalos de acuerdo al primer y ultimo indice de la primer
%y ultima escena respectivamente.
constraint forall(i in rangoActores)
(intervalos[i,1] = getLowerBound(i) / intervalos[i,2] = getUpperBound(i));

% Definición de costoxEscena
constraint forall(k in rangoEscenas, l in m..2)
(costoxEscena[k] = sum(i in rangoActores)
(col(Escenas,k)[i]*Escenas[i,m+1])*Duracion[k]);

constraint costo = sum(i in rangoActores) (
sum(j in intervalos[i,1]..intervalos[i,2]) (Duracion[orden[j]])*Escenas[i,m+1]);
```

```
% Todas las escenas que valgan cero van de primero en el orden.
constraint symmetry_breaking_constraint(
forall(k in 1..numZeroedColumns) (orden[k] = zeroedColumns[k])
);

% Forzar un orden en aquellas escenas no contiguas que sean iguales
%y que no sean columnas cero.
constraint symmetry_breaking_constraint(
forall(j in 1..m-1, k in j+1..m) (
if Escenas[..,j] = Escenas[..,k]  / Duracion[j] = Duracion[k]
/ sum(Escenas[..,j]) != 0
then orden[j] < orden[k]
endif
));

%Se verifica para cada actor que se cumpla sus restricciones de tiempo en set.
constraint forall(i in rangoActores)
(if Disponibilidad[ACTORES[i],2] = 0 then
true
else tiemposxActor[i] <= Disponibilidad[ACTORES[i],2] endif);
% -----------------------------------Objetivo-----------------------------------
%Escalamos el primer objetivo(costo) por un valor que sea al menos como el maximo
%del segundo objetivo(actores que se quieren evitar en set).
%El resultado seria entonces el optimo lexicografico.

var int: objetivo = costo*(c*m + 1) + sum(i in rangoConflictos)
    (getTimeSpend(Evitar[i,..]));

solve :: int_search(orden, smallest, indomain_min) minimize objetivo;

%solve :: int_search(orden, smallest, indomain_min) minimize costo;
```

6. Conclusiones

El problema de planificación de ensayos de una telenovela es un problema NP-Hard (Garcia de la Banda, Stuckey y Chu [1] y Kochetov [2]), sin embargo, para instancias no muy grandes (menos de 20 escenas y 10 actores) nuestro modelo propuesto en MiniZinc es capaz de hallar una solución óptima. En el caso del problema de la vida real (Desenfreno1) nos acercamos bastante a la solución más óptima (903 vs. 871) en menos de 10 minutos, lo cual nos da algo de tranquilidad en que quizá con más tiempo o con restricciones redundantes será posible hallar la solución más óptima.

Adicionalmente, logramos identificar que el parámetro que más hace crecer la complejidad del problema es el número de escenas, cada escena extra aumenta exponencialmente el tamaño del árbol y en consecuencia el tiempo de ejecución.

Como estudiantes de la materia Programación Por Restricciones nos pareció un problema interesante de abordar, dado que integró todos los conocimientos teórico-prácticos de la misma.

Repositorio de GitHub [3]

Referencias

[1] Maria Garcia de la Banda, Peter J Stuckey y Geoffrey Chu. "Solving talent scheduling with dynamic programming". En: *INFORMS Journal on Computing* 23.1 (2011), págs. 120-137.

[2] Yury Kochetov. "Iterative local search methods for the talent scheduling problem". En: *Proceedings of 1st international symposium and 10th Balkan conference on operational research, September*. Vol. 22. 2011, págs. 282-288.

[3] David Cortés Victor Hugo Alejandro Orozco. *PlanificacionEnsayosTelenovela*. Jul. de 2022. URL: `https://github.com/VictorHOG/PlanificacionEnsayosTelenovela` (visitado 26-07-2022).