Informe: Proyecto Programación por Restricciones

Juan Marcos Caicedo Mejía (1730504)
David Gaona Medina (1729009)

Martes 28 de Abril del 2020

Asignatura: Programación Por Restricciones
Código: 750067M
Profesores: Robinson Duque, Juan Francisco Díaz

Cali, Colombia
Universidad del Valle
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación

Tabla de Contenidos

1 Abstract

This article contains the proposal of a solution to the problem of scheduling the shooting of scenes in a soap opera optimally (original idea by Camilo Rueda), given in the final project of the subject of Constraint Programming imparted by professors Robinson Duque and Juan Francisco Díaz at Universidad del Valle, during 2019-2 semester (October 2019 - May 2020). Since this problem has a combinatorial nature, it's a pretty difficult problem, specially because it's a COP (Constraint Optimization Problem), were it has to search for the optimal ordering of scenes to be shot in a set so that the total cost of production is minimal. The cost of the production of the soap opera depends on the total time that an actor is on set (time between the first and last scene that the actor participates in), multiplied by the cost per time unit that each actor charges.

Este artículo contiene una propuesta a la solución del problema de planificación óptima de ensayos de una telenovela (idea original por Camilo Rueda), dado en el proyecto final de la materia Programación por Restricciones dada por los profesores Robinson Duque y Juan Francisco Díaz en la Universidad del Valle, durante el semestre 2019-2 (October 2019 - May 2020). Como este problema tiene una naturaleza combinatoria, resulta un problema bastante difícil, especialmente por ser un COP, donde tiene que buscar el orden óptimo de las escenas a grabar en un set donde el costo de producción total sea el mínimo. Dicho costo de producción de la novela depende del tiempo total en el que el actor se encuentre en el set (el tiempo entre la primera y última escena en la que participa el actor), multiplicado por lo que cobra cada actor por unidad de tiempo.

2 Parte 1 - El problema de la planificación de los ensayos en una telenovela

2.1 Modelo Inicial

El modelo inicial se empeña en ajustar datos de entrada, establecer variables de decisión y dominios de las mismas, junto con imposición de restricciones en un modelo escrito en el lenguaje de modelamiento de MiniZinc que permita dar respuesta al problema indicado.

2.1.1 Datos de entrada

El modelo recibe, esencialmente, una entrada normalizada consistente en:

- Un tipo enumerado *ACTORES* (enumeration-type en MiniZinc) que representa a los Actores que participan en el ensayo: $\{Actor_1, Actor_2, Actor_3...Actor_n\}$. Definición en MiniZinc:

  ```
  enum ACTORES;
  ```

- Una matriz $n \times m + 1$, *Escenas*, donde n es el número de Actores que hacen parte del ensayo y m es el número de Escenas a ensayar. La columna adicional contiene la información que cobra cada Actor por unidad de tiempo en el set. Definición en MiniZinc:

  ```
  array[int, int] of int: Escenas;
  ```

- Una lista (arreglo/array en programación) no necesariamente ordenada de longitud m, *Duracion*, que corresponde a la duración en unidades de tiempo de cada escena. Por ejemplo, el elemento i-ésimo de dicha lista corresponde a la duración en unidades de tiempo de la i-ésima escena. Definición en MiniZinc:

  ```
  array[int] of int: Duracion;
  ```

Adicionalmente a la entrada normalizada del proyecto, antes de empezar a jugar con variables de decisión se declaran algunos datos y variables que son útiles para el resto del modelo, que son definidas de manera total dependiendo de la entrada y no se utilizan en la resolución del problema. Dichos datos adicionales son:

- El número *rows* (en español: Filas), que es igual a la longitud del tipo enumerado *ACTORES*, se utiliza para llevar un seguimiento al número de actores (número de filas en la matriz). Definición en MiniZinc:

  ```
  int: rows = length(ACTORES);
  ```

- El número *cols* (en español: Columnas), que es igual a la longitud de la lista *Duracion* más 1, se utiliza para poder obtener la información de costos individuales de los actores. Definición en MiniZinc:

  ```
  int: cols = length(Duracion) + 1;
  ```

- El número *number_of_scenes* (en español: Número de escenas), que es igual a la longitud de la lista *Duracion*, se utiliza para poder hacer seguimiento al número real de Escenas del ensayo, que viene siendo el número de Columnas en la Matriz que da solución al problema (Matriz que no incluye la última columna de Costos individuales de actores). Definición en MiniZinc:

  ```
  int: number_of_scenes = length(Duracion);
  ```

- La lista *individual_costs* (en español: Costos individuales), de longitud *rows*, que guarda la información únicamente de los costos individuales de cada actor. Útil en el cálculo de la función objetivo. Definición en MiniZinc:

  ```
  array[1..rows] of int: individual_costs = [Escenas[i, cols] | i in 1..rows];
  ```

2.1.2 Variables de decisión

Dentro de la solución del problema es necesario utilizar y jugar con datos que contienen variables de decisión, pues es en éstos que se imponen las restricciones y que se logra encaminar el modelamiento del problema para que su ejecución resulte en la solución óptima al planteamiento del problema.

Las variables de decisión utilizadas son:

- La lista (o arreglo) *scenes_variable_order*, de longitud *number_of_scenes*, que es una de las piezas claves del modelo, pues esta lista contiene el Orden de las Escenas que permite ser modificado y manipulado por las distintas restricciones, para poder hallar el ordenamiento óptimo que

permite a las Escenas tener el costo mínimo del ensayo de la telenovela. Este arreglo está conectado intrínsecamente con la matriz de decisión usada para el cálculo del costo de la obra. Definición en MiniZinc:

```
array[1..number_of_scenes] of var 1..number_of_scenes: scenes_variable_order;
```

- La lista (o arreglo) *variable_duration*, de longitud *number_of_scenes*, que contiene los mismos valores de el arreglo *Duracion*, solo que cambian las posiciones y se mantiene consistente con el arreglo *scenes_variable_order* conforme va cambiando. Definición en MiniZinc:

```
array[1..number_of_scenes] of var int: variable_duration =

[Duracion[scenes_variable_order[i]] | i in 1..number_of_scenes];
```

- La matriz (array2d), de dimensiones *rows* × *number_of_scenes*, llamada *only_scenes_decision_variable*, que es otra parte clave en la resolución del problema, pues esta es la verdadera Matriz de Variables de Decisión con la cual se calcula el costo total de la obra. Esta matriz a diferencia de la matriz de la entrada *Escenas* no posee la columna correspondiente a la información de los costos de cada autor, pues ya está almacenada en la lista (arreglo) *individual_costs*. Esta matriz está conectada con el arreglo *scenes_variable_order*, ya que coloca las Columnas de la entrada *Escenas* en función del ordenamiento que posea el arreglo *scenes_variable_order*. Definición en MiniZinc:

```
array[1..rows, 1..number_of_scenes] of var 0..1: only_scenes_decision_variable =

array2d(1..rows, 1..number_of_scenes,

[Escenas[i,scenes_variable_order[j]] | i in 1..rows, j in 1..number_of_scenes]);
```

2.1.3 Restricciones

En este apartado se explican las restricciones que fueron impuestas al modelo para poder obtener la forma de la solución óptima, es decir, en estas restricciones se le indica al solver el **cómo** (la forma) de la solución que necesitamos.

Para poder incluir estas restricciones se escribieron una serie de funciones que facilitan la imposición de éstas (además de que ayuda a la lectura de código).

Primero se explicarán las funciones de manera individual y posteriormente se explicarán las restricciones (que hacen uso de las funciones).

Las funciones usadas son:

- Función *getIndexOfFirstOcurrence* (en español: Obtener índice de primera ocurrencia), recibe un número *num* (constante, parámetro) y un arreglo de variables de decisión enteras. Devuelve la posición de la primera ocurrencia del número *num* en el arreglo dado. Código en MiniZinc:

```
function var int: getIndexOfFirstOcurrence(int: num,
                                array[int] of var int: arr) =
min([if num = arr[i] then i else length(arr) endif | i in index_set(arr)]);
```

- Función *getIndexOfLastOcurrence* (en español: Obtener índice de última ocurrencia), recibe un número *num* (constante, parámetro) y un arreglo de variables de decisión enteras. Devuelve la posición de la última ocurrencia del número *num* en el arreglo dado. Código en MiniZinc:

```
function var int: getIndexOfLastOcurrence(int: num,
                                array[int] of var int: arr) =
max([if num = arr[i] then i else -length(arr) endif | i in index_set(arr)]);
```

Ambas funciones son usadas para poder determinar los rangos o intervalos de las Escenas en las que un Actor cualquiera está presente en el Set. Se usa para, dada la fila correspondiente a un Actor, calcular su primera aparición (primer 1) y su última aparición (último 1), y con ambos poder construir el intervalo de Escenas en el que permanece en el Set.

- Función *getSetOfOne* (en español: Obtener conjunto de uno), recibe un arreglo de variables de decisión enteras. Esta es la función que recibe la fila particular de un Actor, y retorna el intervalo de Escenas que ha permanecido en el Set. Usa las 2 funciones anteriormente mencionadas. Código en MiniZinc:

```
function var set of int: getSetOfOne(array[int] of var int: arr) =
     getIndexOfFirstOcurrence(1, arr)..getIndexOfLastOcurrence(1, arr);
```

- Función *getSetsOfOnes* (en español: Obtener conjuntos de unos), recibe una matriz, que viene siendo la matriz de variables de decisión (sin columna de costos), y retorna, por cada actor (cada fila), el intervalo

de Escenas en el que este permanece en el Set. Devuelve esto en un arreglo, correspondiendo el i-ésimo elemento del arreglo (i-ésimo intervalo de escenas) con el i-ésimo Actor. Utiliza la función anteriormente mencionada. Código en MiniZinc:

```
function array[int] of var set of int: getSetsOfOnes(array[int, int] of var int: matrix) =
  [getSetOfOne([matrix[i,j] | j in 1..number_of_scenes]) | i in 1..rows];
```

Estas dos fucniones, *getSetOfOne* y *getSetsOfOnes*, sirven para encontrar los intervalos de Escenas en los que los Actores están presentes en el Set, dada la Matriz que contiene el ordenamiento de las Escenas.

- Función *getTotalTimes* (en español: Obtener tiempos totales), que recibe un arreglo que posee los intervalos de escenas en los que participan todos los actores, y otro arreglo que posee la duración de cada escena. Esta función, dependiendo de el intervalo de escenas de cada Actor, calcula para cada Actor su tiempo total en el set, basándose en el tiempo de cada escena proveído por el arreglo que tiene la información de duración de todas las escenas. Dicho tiempo total de cada Actor lo retorna en un arreglo, relacionando el i-ésimo elemento (tiempo total) con el i-ésimo actor. Código en MiniZinc:

```
function array[int] of var int: getTotalTimes(array[int] of var set of int: set_times,
array[int] of var int: duration) =
  [sum([duration[j] | j in set_times[i] where j > 0]) | i in 1..rows];
```

Estas son las funciones que se escribieron, para ser invocadas y finalmente ser usadas cuando se calcula (y minimiza) la función objetivo.

Para poder garantizar esto se hizo lo siguiente:

- Se instanció el arreglo *array_of_intervals_of_each_actor_in_set*, que viene siendo el resultado de ejecutar la función *getSetsOfOnes* con el argumento *only_scenes_decision_variable*. Es decir, aquí se le pasa la Matriz de Variables de Decisión, y calcula los intervalos de las escenas en las que permanece cada Actor. Código en MiniZinc:

```
array[int] of var set of int: array_of_intervals_of_each_actor_in_set =
getSetsOfOnes(only_scenes_decision_variable);
```

- Se instanció el arreglo *total_time_in_set_for_each_actor*, que es el resultado de ejecutar la función *getTotalTimes* con el anterior arreglo *array_of_intervals_of_each_actor_in_set* y la duración variable *variable_duration*. Entonces en el arreglo resultante de la invocación de la función estarán los tiempos totales en los que permanece en el set cada Actor. Código en MiniZinc:

  ```
  array[int] of var int: total_time_in_set_for_each_actor =
  getTotalTimes(array_of_intervals_of_each_actor_in_set, variable_duration);
  ```

2.1.4 Función objetivo

En la anterior sección habíamos calculado el tiempo total que cada Actor permanece en el set y lo habíamos guardado en el arreglo *total_time_in_set_for_each_actor*. Como la función objetivo, matemáticamente, es multiplicar cada tiempo total del Actor i-ésimo por la cantidad que cobra por unidad de tiempo el Actor i-ésimo, hacer esto para todos los Actores y sumar, se vería algo así:

Llamémosle t al arreglo de tiempos totales de cada actor y c al arreglo de costos por unidad de tiempo de cada actor. La función objetivo f se vería así:

$$f = \sum_{i=1}^{n} t_i * c_i$$

Donde n sería el número de actores que participan en el ensayo.

Escribir esto en MiniZinc con los datos que tenemos en el modelo hasta ahora es posible: el arreglo t está representado por *total_time_in_set_for_each_actor* y el arreglo c está representado por *individual_costs*. Así se escribiría la función objetivo f:

```
var int: f =
sum([total_time_in_set_for_each_actor[i] * individual_costs[i] | i in 1..rows]);
```

Dicha función objetivo f finalmente representa el costo total de la obra, y es lo que queremos minimizar, entonces se lo decimos a MiniZinc:

```
solve minimize f;
```

2.1.5 Resultados experimentales con distintas instancias del problema

Para el Modelo Inicial se probaron los siguientes archivos de entrada:

- Trivial1.dzn
- Med1-1.dzn
- Med1-2.dzn
- Med1-3.dzn
- Med1-4.dzn

También se probaron en los 2 solvers: **Gecode** y **Chuffed**.

Los atributos Tiempo Esperado y Costo Esperado corresponden a los valores proveídos en los Input para el Proyecto.

⇒ **Gecode:**

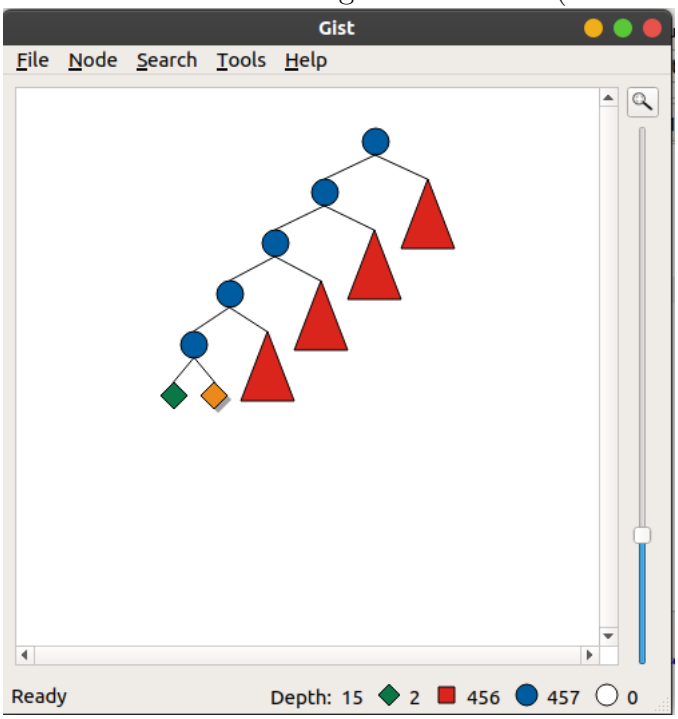
	Tiempo	Tiempo Esperado	Costo	Costo Esperado
Trivial1	143msec	137msec	255	255
Med1-1	162msec	126msec	330	330
Med1-2	209msec	15s 887msec	390	390
Med1-3	229msec	2m 2s	415	415
Med1-4	464msec	7m 29s	490	490

⇒ **Chuffed:**

	Tiempo	Tiempo Esperado	Costo	Costo Esperado
Trivial1	153msec	137msec	255	255
Med1-1	170msec	126msec	330	330
Med1-2	172msec	15s 887msec	390	390
Med1-3	184msec	2m 2s	415	415
Med1-4	368msec	7m 29s	490	490

2.1.6 Probando distintas estrategias de búsqueda

Adicionalmente, probamos con distintas estrategias de búsqueda propuestas para comparar cantidad de nodos explorados en el problema, y poder encontrar una estrategia de búsqueda que favoreciera la cantidad de nodos explorados y comparados, y poder bajar un poco más la cota del tiempo. Partiendo de una solución *default*, es decir, sin estrategias de búsqueda, usando **Gecode Gist** obtenemos el gráfico de árbol (sobre la instancia Trivial1.dzn):

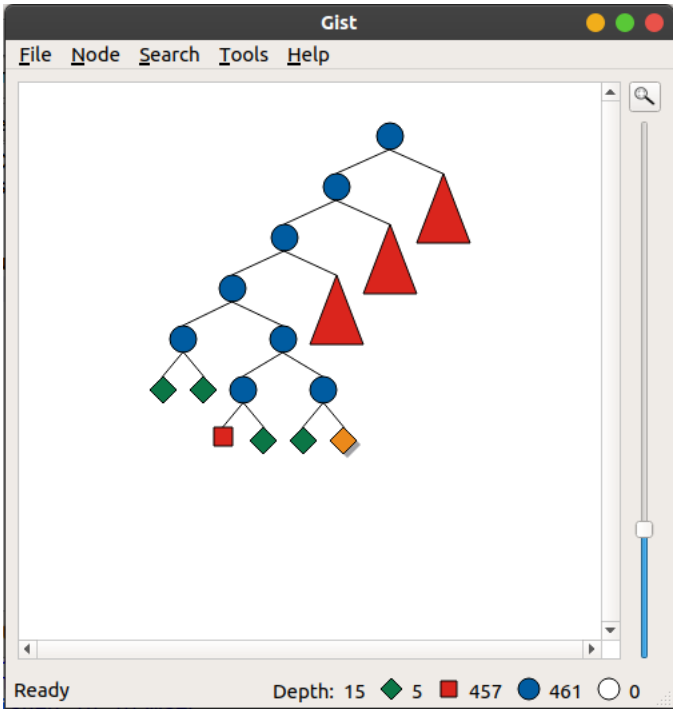


Propusimos probar 3 estrategias de búsqueda:

- input_order, indomain_median
- first_fail, indomain_split
- smallest, indomain_min

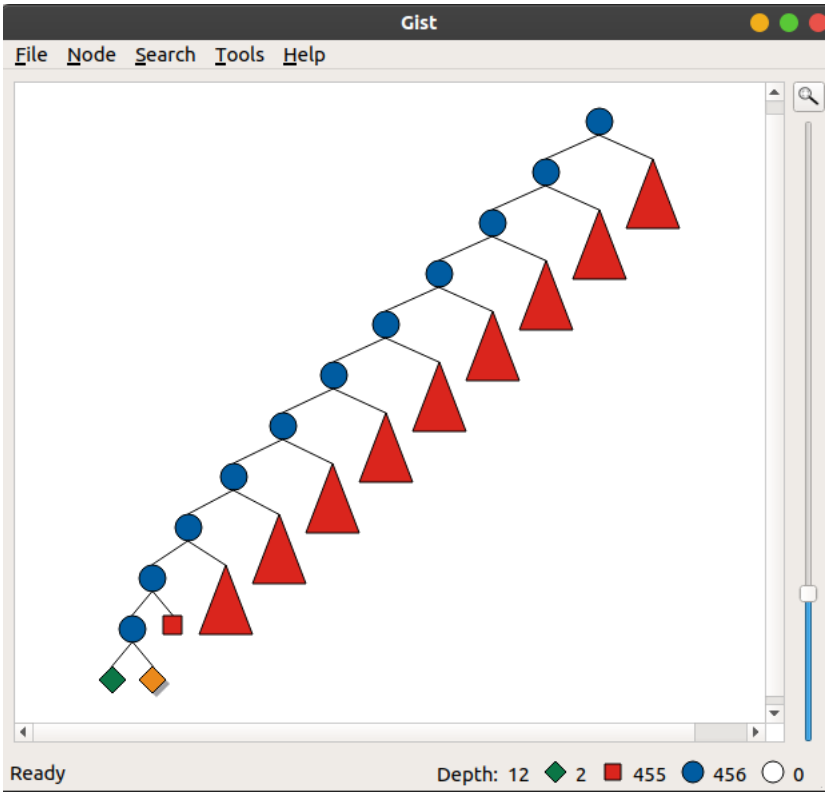
Todas, estrategias de búsqueda entera (int_search) que actúan sobre el array variable de decisión de el orden de las escenas: *scenes_variable_order*

⇒ Aplicando input_order, indomain_median obtuvimos el siguiente árbol de siguiente:



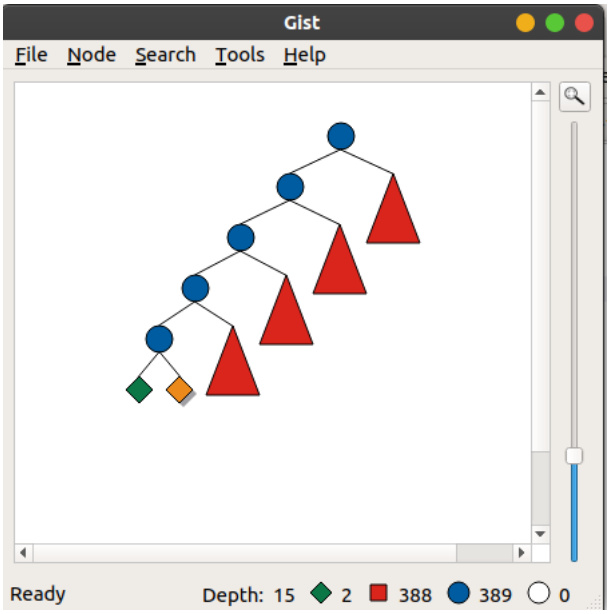
Que realmente mejora muy poco con respecto al *default*.

⇒ Luego aplicando first_fail, indomain_split obtuvimos el siguiente árbol de siguiente:



Que tampoco mejora con respecto a *default*, es muy similar.

⇒ Finalmente aplicando smallest, indomain_min obtuvimos el siguiente árbol de siguiente:



Que posee una ligera mejora con respecto a *default*, y podría ser una mejor estrategia de búsqueda para este modelo en particular.

Todas estas pruebas fueron sobre la instancia Trivial1.dzn para facilitar la visualización del árbol, sin embargo, el comportamiento de la estrategia de búsqueda se puede extrapolar a instancias más grandes y más complejas.

3 Parte 2 - El problema de la planificación de los ensayos en una telenovela, con restricciones adicionales y eliminación de simetrías

3.1 Modelo con Eliminación de Simetrías

Una extensión del Modelo Inicial consiste en añadirle ciertas restricciones que permitan eliminar simetrías y hacer que el solver no las explore, es decir, que solo busque el conjunto de soluciones no-simétricas o que encuentre una y a partir de ella saber calcular las simétricas, para achicar el espacio de búsqueda y probablemente reducir el tiempo de respuesta. En esta sección se explicarán dichas restricciones y se harán pruebas experimentales para compararlo con el modelo sin esta clase de restricciones.

3.1.1 Restricciones que eliminan simetrías (breaking symmetry constraints)

En síntesis, se añadieron un total de 4 restricciones que rompen simetrías para casos en los que:

- Hay columnas iguales en la matriz (en dos escenas en particular, participan los mismos actores).
- Existen escenas donde no participa ningún actor.

Las 4 restricciones que se añadieron son:

1. Si dos columnas adyacentes son iguales, es decir, columna i es igual a la columna $i+1$, y la columna i representa a la escena j, y la columna $i+1$ representa a la escena k, entonces en toda la permutación de los ordenes de las escenas habrán dos soluciones:

   $$...j, k, ...$$

   y

   $$...k, j...$$

   Los puntos suspensivos indican el resto del ordenamiento de las escenas a ambos lados. Estos dos grupos de soluciones son idénticos en

cuanto a que la escena j es igual a la escena k, y son ambos adyacentes. Si imponemos un orden, de que se debe preservar la solución que ordena ambas escenas, es decir, si $j < k$ entonces solo conservará la primera posibilidad, y si $k < j$ solo explorará el segundo grupo e ignorará el resto de soluciones que no cumplan con esta restricción de ordenamiento. Así, eventualmente podamos y logramos que no explore tantos nodos. Código en MiniZinc:

```
constraint forall(i in 1..number_of_scenes-1)
(if [only_scenes_decision_variable[j,i] | j in 1..rows] =
[only_scenes_decision_variable[j,i+1] | j in 1..rows] then
scenes_variable_order[i] < scenes_variable_order[i+1] endif);
```

Es decir, en el orden de las escenas sólo considerará las soluciones que cumplan con dicha restricción de ordenamiento.

2. Si dos columnas cualesquiera no adyacentes son iguales, es decir, columna i es igual a la columna j, y la columna i representa a la escena k, y la columna j representa a la escena l, entonces en toda la permutación de los ordenes de las escenas habrán dos soluciones:

$$...k, ..., l, ...$$

y

$$...l, ..., k, ...$$

Los puntos suspensivos indican el resto del ordenamiento de las escenas. Estos dos grupos de soluciones son idénticos en cuanto a que la escena k es igual a la escena l. Si imponemos un orden, de que se debe preservar la solución que ordena ambas escenas, es decir, si $k < l$ entonces solo conservará la primera posibilidad, y si $l < k$ solo explorará el segundo grupo e ignorará el resto de soluciones que no cumplan con esta restricción de ordenamiento. Así, eventualmente podamos y logramos que no explore tantos nodos. Código en MiniZinc:

```
constraint forall(i in 1..number_of_scenes-1, j in i+1..number_of_scenes)
(if [only_scenes_decision_variable[k,i] | k in 1..rows] =
[only_scenes_decision_variable[k,j] | k in 1..rows] then
scenes_variable_order[i] < scenes_variable_order[j] endif);
```

Es decir, en el orden de las escenas sólo considerará las soluciones que cumplan con dicha restricción de ordenamiento.

3. Otra restricción es poder ver a columnas iguales $i, j, k, ...$ como una especie de bloques, es decir, detectar columnas iguales en una Matriz de escenas y solo permitir las permutaciones que las contengan seguidas una detrás de otra y todas juntas, para poder ahorrar otras permutaciones de soluciones que al final van a terminar acabando por pegar estas columnas, y lo mejor es hacerlo como restricción para que el solver no pierda tiempo llegando a este estado de la solución. Esto debido a que siempre sucede que en la Matriz Solución Óptima al problema, las columnas que son iguales terminan juntas en la solución óptima. Esto se hace, primero, ordenando lexicográficamente la Matriz de la Entrada, y con ello se van a poner seguidas unas tras otras las columnas que lleguen a ser iguales, y luego hacer un recorrido sobre esta Matriz ordenada lexicograficamente e imponer columna a columna la restricción de que si son iguales, deberán estar seguidas (adyacentes) en el orden de escenas definitivo (solución óptima). Esto se hace con ayuda de una función que calcula la distancia entre dos escenas en el arreglo de orden de escenas (si su distancia es 1 es porque son adyacentes):

```
function var int: getDistanceBetweenTwoScenes
(array[int] of var int: scene_order,
var int: scene1, var int: scene2) =
  abs(getIndexOfFirstOcurrenceVar(scene1, scene_order) -
  getIndexOfFirstOcurrenceVar(scene2, scene_order));
```

Junto con una copia de el orden de escenas para poder analizar el orden ordenado lexicograficamente y también una copia de la Matriz para ordenar lexicograficamente:

```
array[1..number_of_scenes] of var 1..number_of_scenes:
scenes_variable_order_lex;

array[1..rows, 1..number_of_scenes] of var 0..1:
only_scenes_decision_variable_lex =
array2d(1..rows, 1..number_of_scenes,
[Escenas[i,scenes_variable_order_lex[j]] |
i in 1..rows, j in 1..number_of_scenes]);
```

Después ordenamos dicha Matriz lexicograficamente:

```
constraint forall(i in 1..number_of_scenes-1, j in i+1..number_of_scenes)
(lex_lesseq([only_scenes_decision_variable_lex[k,i] | k in 1..rows],
[only_scenes_decision_variable_lex[k,j] | k in 1..rows]));
```

Tras haberla ordenado lexicograficamente la recorremos, y si vemos alguna pareja de columnas iguales, obligamos a que en el orden original sean adyacentes usando la función de distancia:

```
constraint forall(i in 1..number_of_scenes-1)
(if [only_scenes_decision_variable_lex[k,i] | k in 1..rows] =
 [only_scenes_decision_variable_lex[k,i+1] | k in 1..rows] then
  getDistanceBetweenTwoScenes(scenes_variable_order,
  scenes_variable_order_lex[i], scenes_variable_order_lex[i+1])
  = 1 endif);
```

4. La última restricción tiene que ver con las columnas que son cero-columnas, es decir, columnas donde todos sus elementos son 0. Estas columnas (escenas) al no tener Actores que participen en ellas, realmente no agregan valor al costo de la función, así que las únicas dos posibilidades es que estas columnas (escenas) sean ubicadas al inicio o al final de el orden de las escenas. Como ambas soluciones, al inicio o al final, dan el mismo costo, lo mejor sería dejar esto fijo al modelo, es decir, establecer si encuentra cero-columnas y de una vez darles el espacio al inicio o al final, y que el Solver se ahorre el tiempo y el esfuerzo de explorar ambas posibilidades, ya que en teoría son lo mismo. En ese caso entonces, en nuestro caso decidimos hacer funciones y restricciones que detectaran cero-columnas y las pusiera siempre al inicio de el orden, así el Solver no explora las soluciones que las ponen al final. Lo hacemos con dos funciones: una que calcula el numero de cero-columnas en la Matriz de la entrada:

```
function int: getNumberOfZeroColumnsInMatrix(array[int,int] of int: arr,
int: n_scenes, int: n_rows) =
  sum([if [arr[j,i] | j in 1..n_rows]
 = [0 | j in 1..n_rows] then 1 else 0 endif| i in 1..n_scenes]);
```

Esto lo obtenemos en una constante:

```
int: number_of_zero_columns =
getNumberOfZeroColumnsInMatrix(only_scenes_initial, number_of_scenes, rows);
```

(only_scenes_initial es una copia de la matriz del input). Luego usamos otra función que averigüa cuales son los números de las columnas (escenas) que son cero-columnas:

```
function array[1..number_of_zero_columns] of int:
getWhichColumnsAreZeroColumns(array[int,int] of int: arr,
int: n_scenes, int: n_rows) =
  [i | i in 1..n_scenes where [arr[j,i] | j in 1..n_rows]
  = [0 | j in 1..n_rows]];
```

Para luego guardarlo en un arreglo:

```
array[1..number_of_zero_columns] of int: array_of_zero_columns =
getWhichColumnsAreZeroColumns(only_scenes_initial, number_of_scenes, rows);
```

E imponer dicha restricción sobre dicho arreglo, y la restricción que se impone es que siempre al inicio del orden de las escenas se van a colocar las escenas en las que no participan actores:

```
constraint forall(i in 1..number_of_zero_columns)
(scenes_variable_order[i] = array_of_zero_columns[i]);
```

3.1.2 Resultados experimentales con distintas instancias del problema

Para el Modelo extendido con Eliminación de Simetrías se probaron los siguientes archivos de entrada:

- Trivial1.dzn
- Med1-1.dzn
- Med1-2.dzn
- Med1-3.dzn
- Med1-4.dzn

También se probaron en los 2 solvers: **Gecode** y **Chuffed**.

Los atributos Tiempo Esperado y Costo Esperado corresponden a los valores proveídos en los Input para el Proyecto. Se añade el atributo Tiempo M.I.,que corresponde al Tiempo del Modelo Inicial, para ser contrastado.

⇒ **Gecode:**

	Tiempo	Tiempo M.I.	Tiempo Esperado	Costo	Costo Esperado
Trivial1	139msec	143msec	137msec	255	255
Med1-1	154msec	162msec	126msec	330	330
Med1-2	195msec	209msec	15s 887msec	390	390
Med1-3	204msec	229msec	2m 2s	415	415
Med1-4	421msec	464msec	7m 29s	490	490

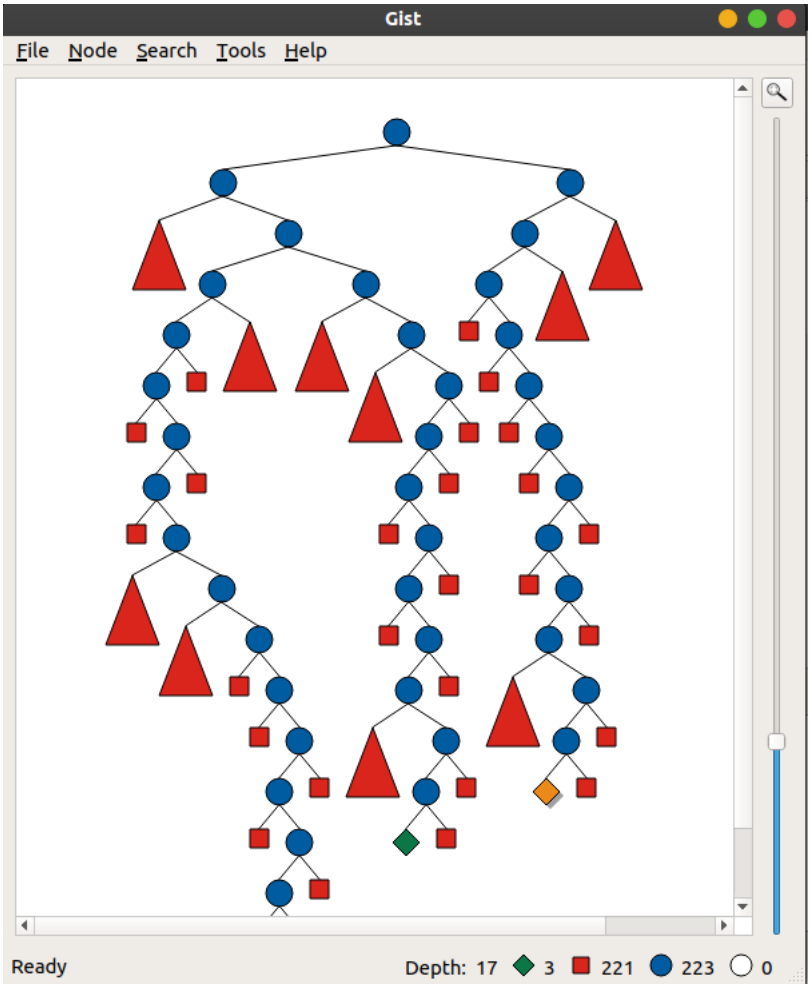
⇒ **Chuffed:**

	Tiempo	Tiempo M.I.	Tiempo Esperado	Costo	Costo Esperado
Trivial1	131msec	153msec	137msec	255	255
Med1-1	132msec	170msec	126msec	330	330
Med1-2	159msec	172msec	15s 887msec	390	390
Med1-3	169msec	184msec	2m 2s	415	415
Med1-4	312msec	368msec	7m 29s	490	490

Aquí podemos evidenciar una ligera mejoría en los tiempos de ejecución con el modelo que posee eliminación de simetrías.

3.1.3 Probando distintas estrategias de búsqueda

Adicionalmente, probamos con distintas estrategias de búsqueda propuestas para comparar cantidad de nodos explorados en el problema, y poder encontrar una estrategia de búsqueda que favoreciera la cantidad de nodos explorados y comparados, y poder bajar un poco más la cota del tiempo. Partiendo de una solución *default*, es decir, sin estrategias de búsqueda, usando **Gecode Gist** obtenemos el gráfico de árbol (sobre la instancia Trivial1.dzn):



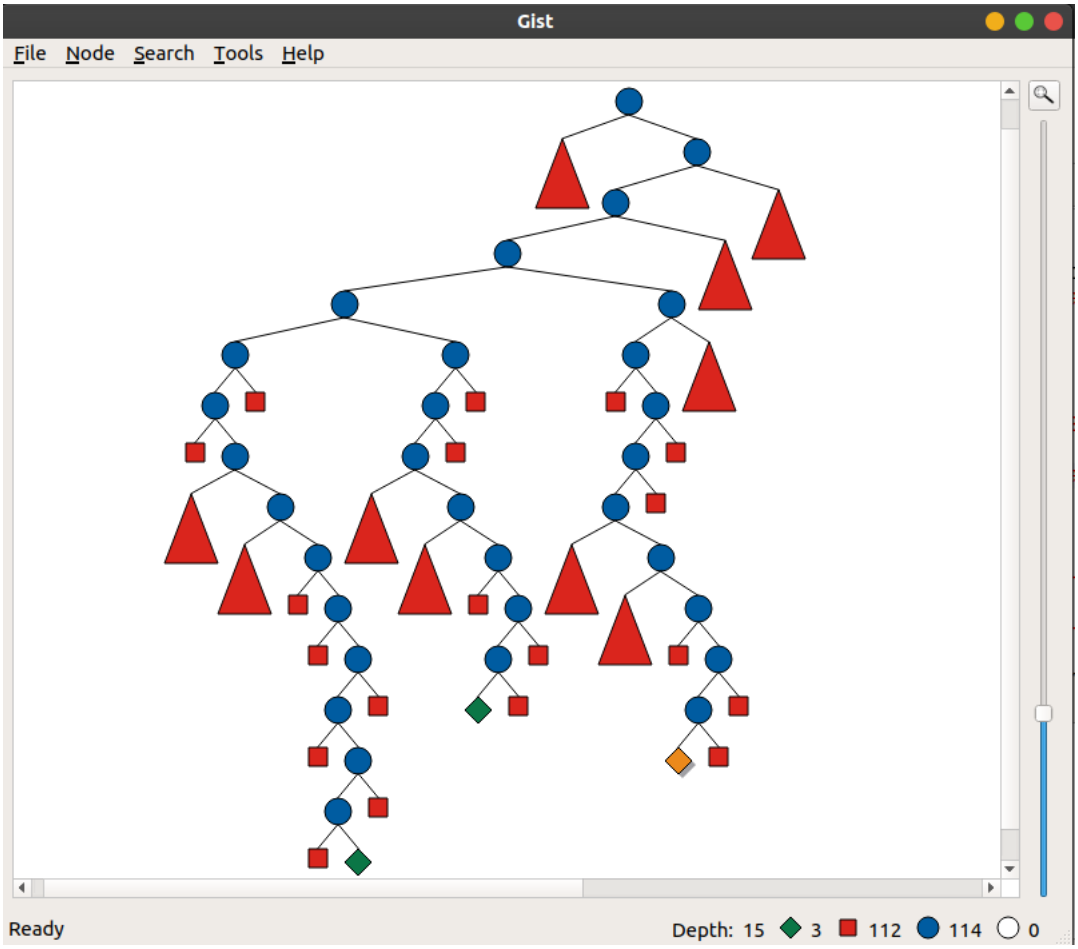
A pesar de no tener estrategia de búsqueda, explora en 223 nodos, a diferencia de los 457 nodos que explora sin eliminación de simetrías (modelo inicial).

Propusimos probar 3 estrategias de búsqueda:

- input_order, indomain_median
- first_fail, indomain_split
- smallest, indomain_min

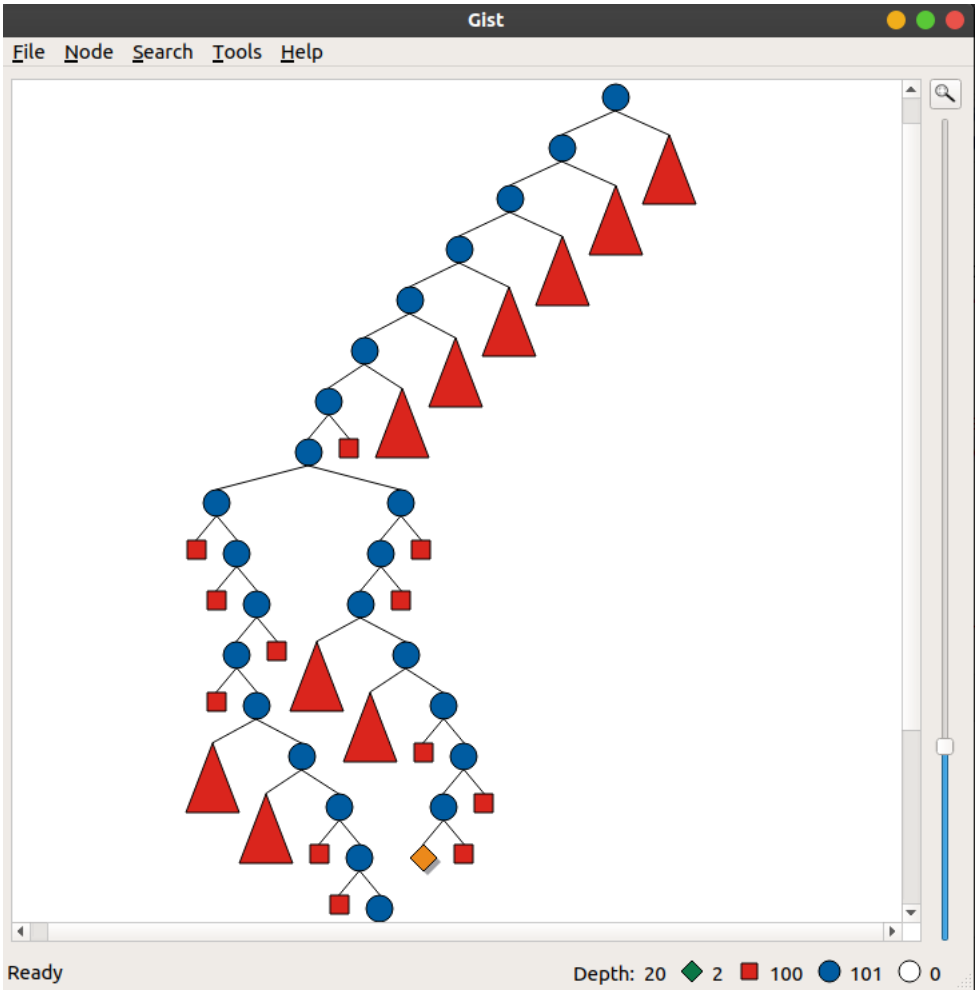
Todas, estrategias de búsqueda entera (int_search) que actúan sobre el array variable de decisión de el orden de las escenas: *scenes_variable_order*

⇒ Aplicando input_order, indomain_median obtuvimos el siguiente árbol de siguiente:



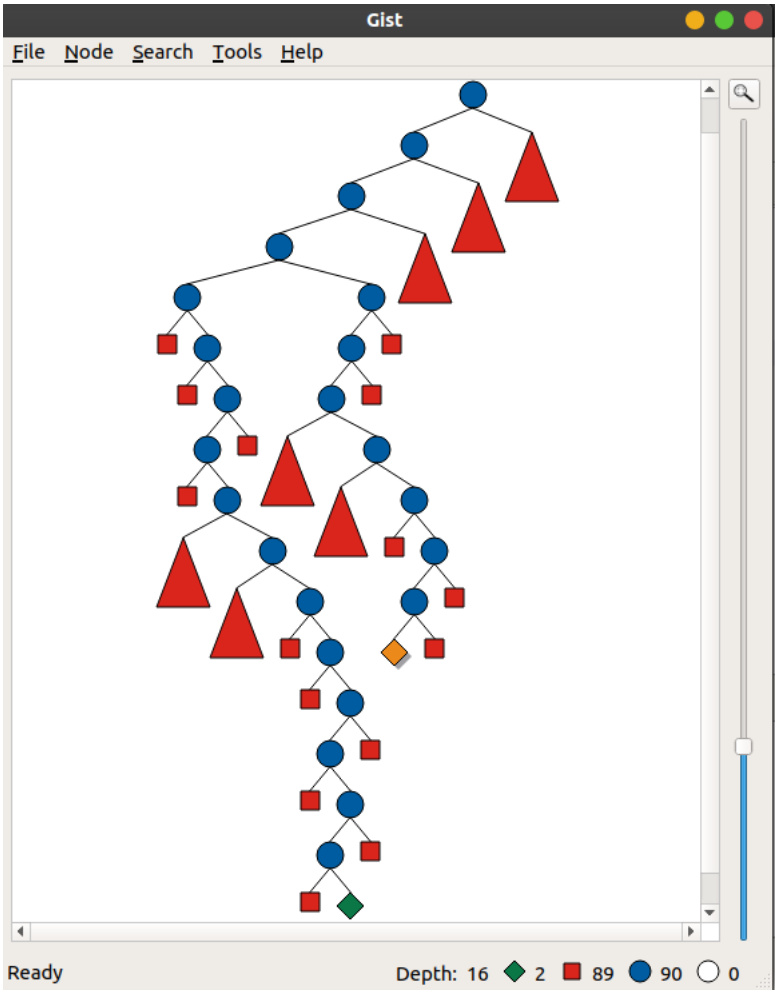
Que realmente mejora considerablemente con respecto a *default*.

⇒ Luego aplicando first_fail, indomain_split obtuvimos el siguiente árbol de siguiente:



Que mejora un poco con respecto a la anterior.

⇒ Finalmente aplicando smallest, indomain_min obtuvimos el siguiente árbol de siguiente:



Que posee una ligera mejora con respecto a la anterior, y comparando con *default* podría ser la mejor estrategia de búsqueda para el modelo.

Todas estas pruebas fueron sobre la instancia Trivial1.dzn para facilitar la visualización del árbol, sin embargo, el comportamiento de la estrategia de búsqueda se puede extrapolar a instancias más grandes y más complejas.

3.2 Modelo con Eliminación de Simetrías y Restricciones Adicionales

Esta última extensión del Modelo añade las restricciones adicionales de:

- Cota de tiempo máximo que pueden pasar los Actores en el Set.
- Minimizar el tiempo que comparten ciertos Actores que se evitan entre ellos.

3.2.1 Restricciones adicionales y nueva función objetivo

Aquí entran a jugar dos nuevos elementos en la entrada:

- Matriz *Disponibilidad*: Relaciona cada Actor con un máximo tiempo de disponibilidad para estar en el Set. Si es 0, no tiene límite
- Matriz *Evitar*: Relaciona un Actor con otro indicando que ambos Actores no se llevan bien y que lo mejor es minimizar el tiempo compartido por estos Actores que no se llevan bien en el Set.

Para poder garantizar la cota de tiempo máximo que pueden pasar los Actores en el Set realizamos la siguiente restricción:

```
constraint forall(i in 1..rows) (if Disponibilidad[i,2] > 0 then
total_time_in_set_for_each_actor[i] <= Disponibilidad[i,2] endif);
```

Esto lo que quiere decir es que recorrerá elemento por elemento la información proveída por la Matriz *Disponibilidad*, si es mayor a 0 entonces impondrá la restricción de que su tiempo total no debe superar dicho elemento, en otro caso (igual a 0) no impone dicha restricción.

Luego para poder garantizar que se comparta el mínimo de tiempo entre los Actores que no se llevan bien, escribimos algunas funciones que nos ayudan a lograr el objetivo:

- Una función *getSharedTime* (en español: Obtener tiempo compartido), que recibe un arreglo de 2 elementos que contiene la pareja de Actores que se evitan, y retorna la **intersección** de sus intervalos de escenas en los que cada uno pertenece, para así conocer exactamente en qué escenas llegan a estar al mismo tiempo en el set. Código en MiniZinc:

```
function var set of int: getSharedTime(array[int] of var int: pair) =
array_of_intervals_of_each_actor_in_set[pair[1]]
intersect array_of_intervals_of_each_actor_in_set[pair[2]];
```

- Una función *getArrayOfSharedTimes* (en español: Obtener arreglo de tiempos compartidos), que recibe la Matriz de *Evitar* y devuelve los tiempos (intersecciones) que cada pareja de Actores que se evitan tienen compartido en el set. Esto lo retorna en un array relacionando cada pareja de *Evitar* con cada elemento. Utiliza la anterior función. Código en MiniZinc:

  ```
  function array[int] of var set of int:
  getArrayOfSharedTimes(array[int, int] of var int: avoid_matrix) =
  [getSharedTime([Evitar[i,j] | j in 1..2]) | i in 1..length(Evitar) div 2];
  ```

Usando estas funciones instanciamos un arreglo de intervalos de tiempo compartido por actores que se evitan:

```
array[int] of var set of int:
array_of_intervals_shared_time_of_actors_that_avoid_themselves =
getArrayOfSharedTimes(Evitar);
```

Luego escribiendo otras dos funciones:

- Una función *getIntegerTimeShared* (en español: Obtener tiempo compartido entero) que, recibe un intervalo (posiblemente la intersección anteriormente encontrada) y retorna el total del tiempo en duración de escenas que representa dicho intervalo. Sirve para concretar en unidades de tiempo el intervalo que comparten dos actores que se evitan. Código en MiniZinc:

  ```
  function var int: getIntegerTimeShared(var set of int: interval) =
    sum([variable_duration[i] | i in interval where i > 0]);
  ```

- Una función *getTotalIntegerTimeShared* (en español: Obtener tiempo total compartido entero), que recibe un arreglo de intervalos y devuelve así mismo un arreglo de unidades de tiempo (duración) enteras, cada elemento representando cada pareja de Actores que se evitan. Utiliza la función anterior. Código en MiniZinc:

```
function array[int] of var int:
getTotalIntegerTimeShared(array[int] of var set of int: intervals_shared) =
[getIntegerTimeShared(intervals_shared[i]) | i in 1..length(Evitar) div 2];
```

Con esto ya podremos tener el tiempo compartido total entre Actores que se evitan. Lo hacemos instanciando primero el arreglo que obtiene los tiempos que comparten en unidades de tiempo enteras:

```
array[int] of var int: array_of_total_integer_shared_time_of_actors_that_avoid_themselves =
getTotalIntegerTimeShared(array_of_intervals_shared_time_of_actors_that_avoid_themselves);
```

Y luego, con dicho array instanciado, lo único que hacemos es denotar el **nuevo** componente de la función objetivo, esta pequeña función g, que representa el tiempo compartido entre actores que se evitan:

```
var int: g = sum(array_of_total_integer_shared_time_of_actors_that_avoid_themselves);
```

Al ser esto un componente que también se busca minimizar, lo que se hace es que se incluye en la función a optimizar, es decir que se le dice a MiniZinc lo siguiente:

```
solve minimize f+g;
```

Recordemos que f es la función que determina el costo total de la obra. Como le pedimos que minimice en conjunto $f + g$, lo que sucede es que en este caso el término que más valor aporta es f, el costo, y va a minimizar primero esta parte y luego intentará minimizar g.

Así ya hemos terminado todas las especificaciones y restricciones del modelo.

3.2.2 Resultados experimentales con distintas instancias del problema

Para el Modelo con Restricciones Adicionales y Eliminación de Simetrías se probaron los siguientes archivos de entrada:

- Trivial2.dzn
- Med2-1.dzn
- Med2-2.dzn

- Med2-3.dzn
- Med2-4.dzn
- Med2-5.dzn

También se probaron en los 2 solvers: **Gecode** y **Chuffed**.

El atributo nuevo T.C. significa Tiempo Compartido (tiempo compartido entre Actores que se evitan). Los atributos Tiempo Esperado, Costo Esperado, T.C. (Tiempo Compartido) Esperado corresponden a los valores proveídos en los Input para el Proyecto.

⇒ **Gecode:**

	Tiempo	Tiempo Esperado	Costo	Costo Esperado	T.C.	T.C. Esperado
Trivial2	162msec	140msec	450	450	6	6
Med2-1	169msec	260msec	330	330	8	8
Med2-2	219msec	164msec	390	390	10	10
Med2-3	1s 833msec	31s 398msec	520	520	11	11
Med2-4	3s 314msec	1m 46s	729	729	21	21
Med2-5	4s 49msec	4m 16s	841	841	21	21

⇒ **Chuffed:**

	Tiempo	Tiempo Esperado	Costo	Costo Esperado	T.C.	T.C. Esperado
Trivial2	176msec	140msec	450	450	6	6
Med2-1	181msec	260msec	330	330	8	8
Med2-2	203msec	164msec	390	390	10	10
Med2-3	321msec	31s 398msec	520	520	11	11
Med2-4	460msec	1m 46s	729	729	21	21
Med2-5	581msec	4m 16s	841	841	21	21

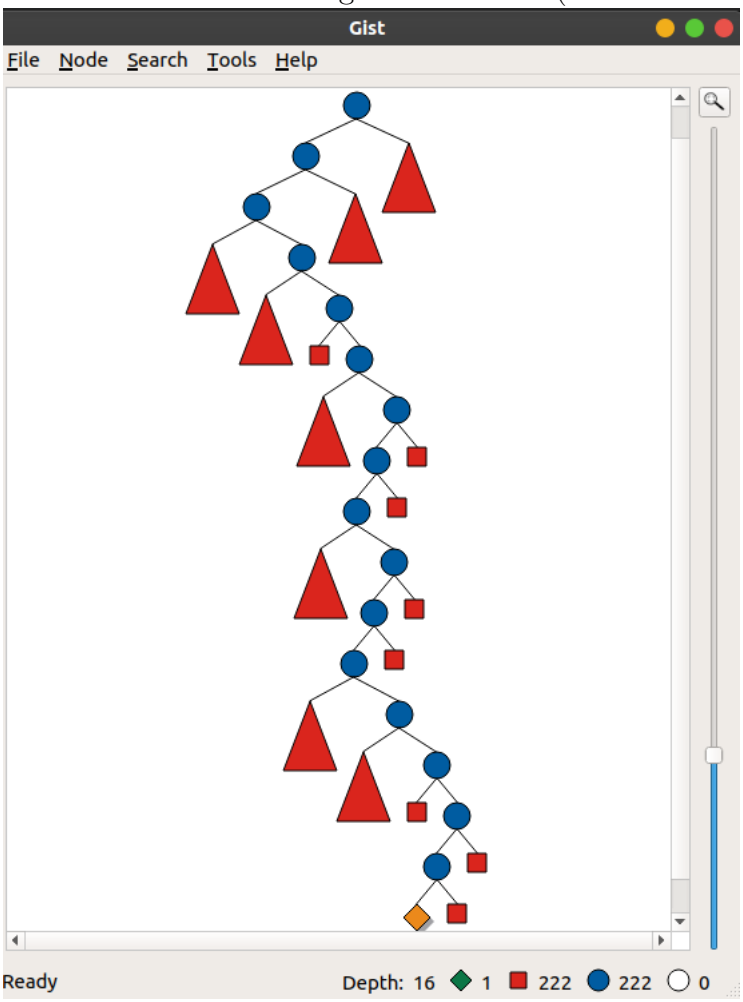
Como algunas conclusiones acerca de los resultados experimentales de los 3 modelos:

- El planteamiento del modelo es correcto y resuelve en buen tiempo instancias del problema
- Las restricciones que eliminan simetrías fueron útiles

- El solver Gecode es eficiente para entradas pequeñas pero no tanto para entradas grandes, en cambio, Chuffed es al revés: es mucho mejor para entradas grandes que para instancias pequeñas.

3.2.3 Probando distintas estrategias de búsqueda

Adicionalmente, probamos con distintas estrategias de búsqueda propuestas para comparar cantidad de nodos explorados en el problema, y poder encontrar una estrategia de búsqueda que favoreciera la cantidad de nodos explorados y comparados, y poder bajar un poco más la cota del tiempo. Partiendo de una solución *default*, es decir, sin estrategias de búsqueda, usando **Gecode Gist** obtenemos el gráfico de árbol (sobre la instancia Trivial2.dzn):

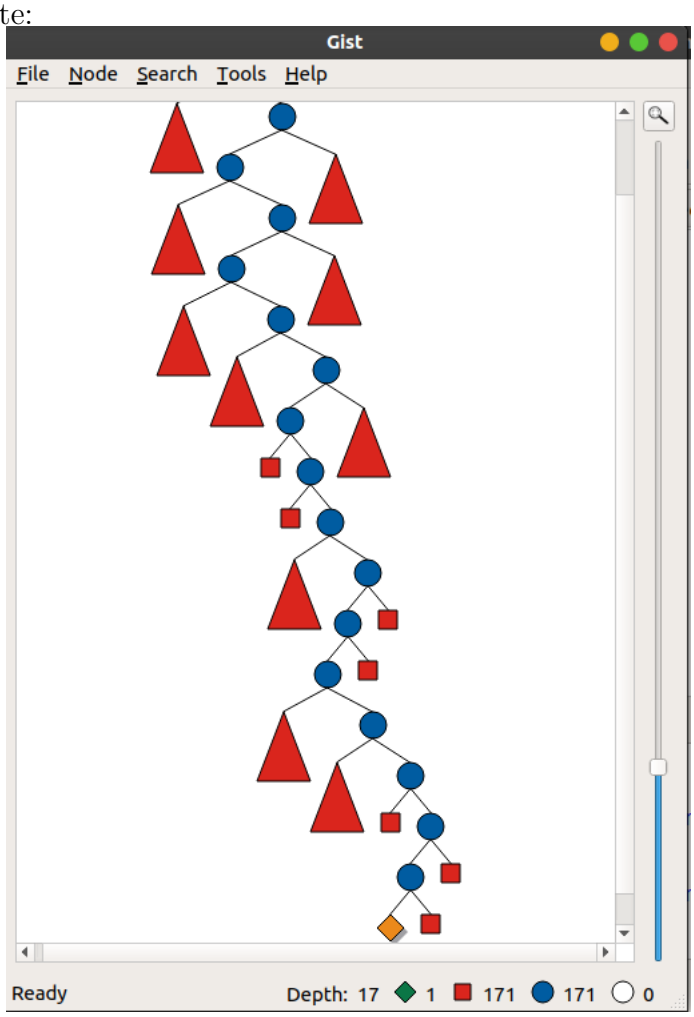


Propusimos probar 3 estrategias de búsqueda:

- smallest, indomain_min
- dom_w_deg, indomain_min
- dom_w_deg, indomain_random

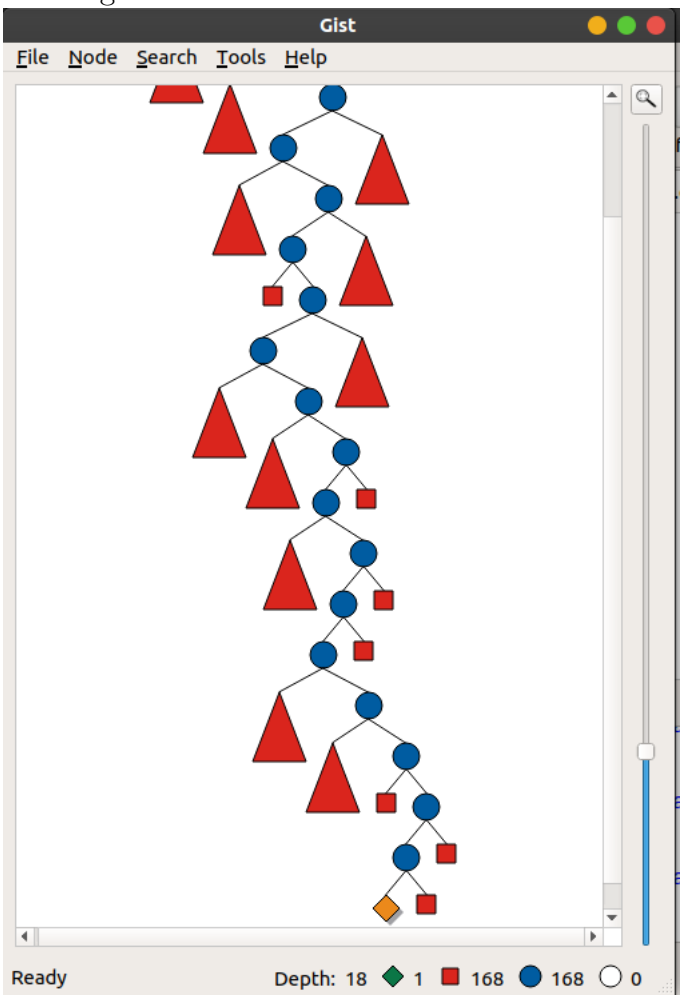
Todas, estrategias de búsqueda entera (int_search) que actúan sobre el array variable de decisión de el orden de las escenas: *scenes_variable_order*

⇒ Aplicando smallest, indomain_min obtuvimos el siguiente árbol de siguiente:



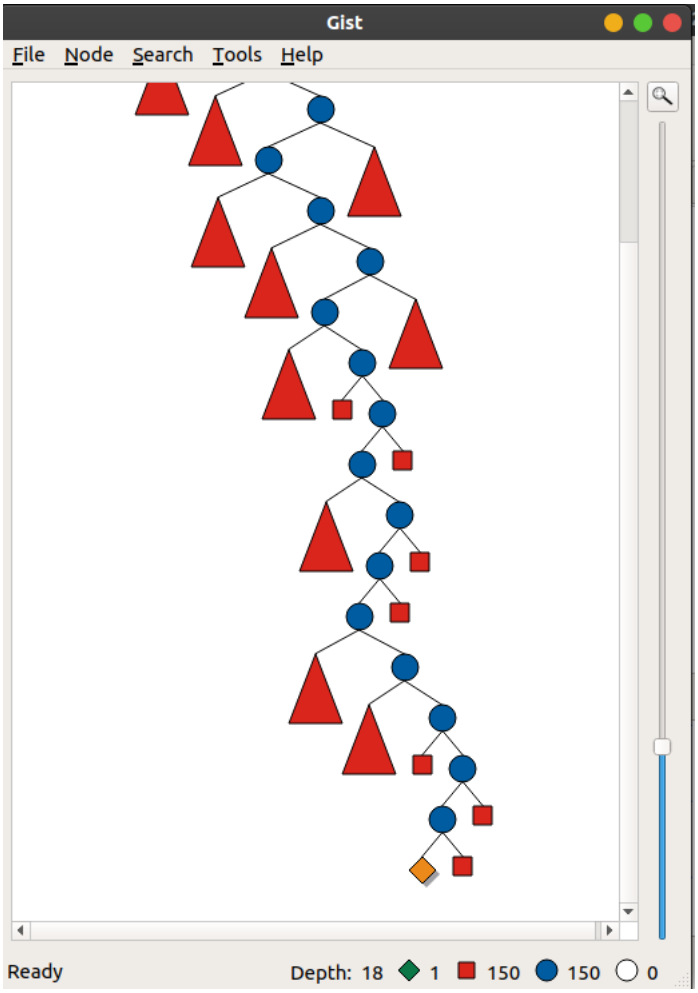
Que es ligeramente mejor a *default*.

⇒ Luego aplicando dom_w_deg, indomain_min obtuvimos el siguiente árbol de siguiente:



Que es un poco mejor comparado con la anterior.

⇒ Finalmente aplicando dom_w_deg, indomain_random obtuvimos el siguiente árbol de siguiente:



Que entre todas las estrategias probadas, daba los mejores resultados, de menor cantidad de nodos explorados.

Todas estas pruebas fueron sobre la instancia Trivial2.dzn para facilitar la visualización del árbol, sin embargo, el comportamiento de la estrategia de búsqueda se puede extrapolar a instancias más grandes y más complejas.

Una de las conclusiones de probar distintas estrategias de búsqueda con los 3 modelos es que algunas búsquedas se ajustan mejor que otras en cada caso particular de los 3 modelos, y usando la mejor estrategia en cada caso ocasiona una menor exploración de nodos y más rapidez en la respuesta.