Inteligencia Artificial

Informe Final: Car Sequencing Problem

Sebastián Torrico D.

October 26, 2019

Abstract

En este paper se presenta un acercamiento y resolución para el problema del tipo *Car Sequencing Problem* (CarSP), cuyo objetivo consiste en encontrar una combinación óptima en las líneas de ensamblado para la producción de vehículos, sujeta a restricciones dadas por una fábrica de ensamblaje. En este caso, las fábricas necesitan agregar distintos elementos dependiendo del tipo de auto y el enfoque de las compañías. Sin embargo, el CarSP pertenece al conjunto NP-Complejo, por tanto, se requiere un análisis profundo para encontrar una forma de abarcarlo. Para entender y buscar estrategias de resolución se presentará el estado del arte proponiendo distintos enfoques, variaciones y técnicas que se utilizan para resolver éste tipo de problemas. Finalmente, se implementará un algortimo basado en Forward Checking + GBJ, mostrando y analizando las ventajas y desventajas de estas técnicas en base al CarSP.

1 Introducción

En la industria de los automóviles siempre es deseado encontrar una manera rentable de producir vehículos con el fin de mejorar la eficiencia, controlar los costos y aumentar la cálidad del producto. A pesar de que algunos autos puedan tener características similares, cada uno podría requerir que se instalen distintos elementos en una misma estación de ensamblaje, ya sea, aire acondicionado, sun-roofs, GPS, entre otros. El problema que enfrentan las fábricas para encontrar una combinación optima de tipos de autos en las lineas de ensamblajes según diversas restricciones se conoce como *Car Sequencing Problem* (CarSP). Por tanto, resolver el CarSP reduciría considerablemente los costos y se aumentaría la flexibilidad que requieren actualmente las plantas de fabricación para satisfacer la demanda en una época de creciente tendencia de la personalización en masa.

El propósito de este documento es realizar un acercamiento y resolver el CarSP, mostrando los distintos enfoques, variaciones, técnicas que se utilizan y resultados en base a la implementación de FC+GBJ para resolver un problema de esta categoría. Para ello, se definirá el problema de manera general explicando el objetivo, las variables y las restricciones que tiene. Luego, se presentará el estado del arte para describir las soluciones existentes hasta el momento, esto es, mostrar los métodos, algoritmos y heurísticas para resolverlo. Después, se escribirá un modelo matemático para representar y entender el problema. Con todo esto, se analizará la implementación con la técnica de búsqueda completa, para así, presentar las conclusiones relevantes del estudio realizado y resultados obtenidos.

2 Definición del Problema

El *Car Sequencing Problem* (CarSP) consiste en la programación de los autos en una línea de ensamblaje para instalar piezas en ellos, tales como, sun-roof, airbag, radio, gps, entre otros. A pesar de que algunos autos puedan tener características similares, cada uno podría requerir la instalación de distintos elementos en una misma estación de ensamblaje. Además, cada estación esta diseñada para mantener a lo más un cierto porcentaje de los vehículos que pasan a lo largo de la línea, de modo que los autos que llegan a una deben organizarse de manera que no exceda la capacidad de la estación.

A lo largo del tiempo, una de las dificultades relacionadas al CarSP ha sido determinar que la complejidad pertenece al conjunto de los problemas tipo NP; es complicado encontrar documentos que certifiquen lo anterior para instanciaciones generales. Sin embargo, recientemente han salido algunas pruebas que verifican que pertenecen tal conjunto, más específicamente, a los de tipo NP-Complejo (NP-Hard) cuando la cantidad de autos es arbitrariamente grande. Ver, por ejemplo, Tamás [11] y Estellon y Gardi [7].

2.1 Definición del modelo

En [13] se define el siguiente modelo como una fábrica en donde hay un conjunto de vehículos que pasan por las líneas de ensamblaje. Existe también conjunto de diferentes opciones para implementar, estos son, los elementos a instalar a cada vehículo (GPS, aire acondicionado y frenos ABS). Las opciones son instaladas en distintas estaciones con capacidades limitadas; lo que significa que existe un máximo de autos que se pueden atender en una estación por sub-secuencia de autos. Por último, existen diversas clases de vehículos y se requiere producir una cantidad determinada de cada una.

Las restricciones son que cada sub-secuencia de autos consecutivos debe lograr ser atendida por una estación y no puede exceder la capacidad de esta. Además, a cada automóvil se le debe instalar todas las opciones que requiere.

El objetivo del problema es encontrar una combinación óptima de secuencias de autos que las estaciones puedan procesar para mejorar la eficiencia, controlar los costos y mejorar la calidad del producto. La idea es encontrar el objetivo logrando minimizar la cantidad de restricciones infringidas para cada secuencia en cada proceso de ensamblaje.

2.2 Variantes del problema

La versión más básica del problema no considera los colores de los autos, las restricciones procuran evitar que la capacidad de las estaciones sean excedidas y el objetivo es minimizar la cantidad de restricciones que son infringidas. Sin embargo, existe una versión más completa que considera los cambios de colores en los autos, de modo que se debe asegurar que una determinada clase de vehículos de cierto color pueda ser asignada a cada posición de una sub-secuencia. Además, se deben añadir un conjunto de restricciones para asegurar que las estaciones realicen sus procedimientos según los colores de los autos. De este modo, se puede cambiar la función objetivo de manera que se busque minimizar el número de cambios de colores.

Una variante común es considerar como entrada a los autos agrupados según distintas configuraciones de opciones. Esto es, que el input este compuesto de $n_1, ..., n_k$ autos de k configuraciones distintas, en vez de los n autos arbitrarios. Por otra parte, una variante más específica es la propuesta por Bautista [2] que utiliza flotas de vehículos especiales como demanda; estos

tienen distintas características y debe ser formulado un módelo de optimización de programación lineal entera mixta satisfacer el máximo número de restricciones. Con esta propuesta transforman el CarSP en un caso párticular, donde solo hay un plan de demanda.

3 Estado del Arte

El *Car Sequencing Problem* fue descrito por primera vez por Parrello et al. [13] en 1986 como una versión del *Job-Shop Scheduling Problem* (JSP), conocido como NP-Completo. Surge debido a las consideraciones económicas subyacentes de esa época para producir autos con distintos conjuntos de opciones. Este documento se enfoca en buscar, analizar y encontrar aproximaciones para formar las secuencias de autos más valiosas a ingresar en las lineas de ensamblaje de las fábricas. Parrello propone utilizar el área de *razonamiento automatizado* (AR) como asistencia al problema, pues contribuyen a un estudio efectivo para problemas del tipo JSP. El AR ofrece ofrece una variedad de maneras para razonar, formar estrategias para controlar el razonamiento y ayudar a los procedimientos que contribuyen al estudio efectivo a problemas como el CarSP. Sin embargo, las soluciones a este problema se desarrollarían posteriormente con distintos enfoques y técnicas que serán explicadas a continuación.

3.1 Enfoques exactos para resolver el Car Sequencing Problem

3.1.1 Programación con restricciones (Constraint programming)

En 1988, Dincbas et al. [5] modelan el caso como un problema de satisfacción de restricciones (CSP); su formulación está basada en que la estación de ensamblaje tiene cinco posibles opciones, las ranuras de la secuencia son las variables y los autos sus valores. El procedimiento comienza agrupando los autos en clases, tales que, todos los autos en cada clase requieran la misma opción. Para lo anterior se utiliza una matriz de tamaño igual a la cantidad de clases por el número de opciones y sus elementos binarios especifican cuales de las opciones están en cada clase; luego, se formulan restricciones de capacidad para que la secuencia de autos no exceda el límite de una estación. Las restricciones de capacidad están formalizadas usando límites de la forma q_i/p_i, las cuales indican que la unidad es capaz de producir a lo más q_i autos con la opción i para cada secuencia de p_i autos. Sin embargo, destacan que támpoco pueden estar por debajo de tal capacidad, pues harían imposible el hecho de evitar exceder alguna capacidad de otra estación de ensamblaje por dejar pasar un vehículo. Además, en ese mismo documento sugieren agregar restricciones implícitas para permitir detectar fallas tempranamente. Finalmente utilizan CHIP, un lenguaje de programación para la *programación con restricciones*, y así resolver el problema. Posteriormente se utilizarían otros lenguajes de este tipo para resolver el problema, tales como cc(FD) [16] y CLP(FD) [3].

3.2 Programación en enteros (Integer programming)

Otro método para resolver el CarSP es propuesto por Drexl y Kimms [6] en el año 2001, donde modelan el problema como *programación en enteros* (ILP). Este modelo se basa en definir variables binarias; C_{ij} corresponde a la clase de auto tipo i en cada posición j, cuyo valor es 1 en caso de que el auto en la posición j es del tipo i y 0 en caso contrario. Las ventajas de considerar este problema con formulación es que aseguran que exactamente una clase de auto es asignada a cada posición, todos los autos de cada clase son asignados a alguna posición y que todas las restricciones de capacidad son cumplidas para cada opción. También, plantean otro modelo basado en una cantidad finita de variables y_k que toman el valor 1 si la secuencia k es escogida, la cual está relacionada a una clase que tiene las posiciones de los vehículos que pertenecen a esta; luego, utilizan el método de generación de columnas para computar los limites superiores

e inferiores. Los resultados son presentados como un conjunto de las instancias generadas, es decir, se muestran las mejores combinaciones. Así, existen una variedad de propuestas utilizando programación entera. Véase Gravel et al. [10].

3.3 Heurísticas para resolver el Car Sequencing Problem

Existen diversos enfoques incompletos para encontrar soluciones óptimas al problema.

3.3.1 Enfoque Greedy

Una de ellas es usar un *enfoque greedy*, esto es, estrategias de búsquedas por la cual se elige una opción óptima en cada paso local con la esperanza de llegar a una solución óptima local. Para efectos del problema, se comienza desde una secuencia vacía e iterativamente se van agregando nuevos autos al final de la secuencia. Gottlieb et al. [9] propone en el 2003 una de las mejores heurísticas en base a eficiencia; su estrategia consiste en agregar en el primer paso el auto que maximice la tasa de utilización de las opciones requeridas y esta se actualiza dinámicamente cada vez que un nuevo auto es agregado al final de la secuencia. Con este enfoque greedy, Gottlieb demuestra que, combinandolo con una cantidad media de aleatorismo y multiples reinicios, puede encontrar rápidamente soluciones óptimas locales para *todas* las instancias de pruebas suministradas por Lee [12] con CSPLib.

3.3.2 Enfoques de Búsqueda Local

Otra idea es utilizar *búsqueda local* para mejorar la secuencia mediante el analisis de sus vecinos. Dada una secuencia inicial, el espacio de busqueda es explorado de vecino a vecino hasta encontrar una secuencia optima o hasta haber realizado el número maximo de movimientos.

Para construir la secuencia inicial se puede realizar permutaciones aleatorias al conjunto de vehículos a producir. Sin embargo, Gottlieb et al. [9] propuso una manera más eficiente de inicialización mediante el uso de técnicas greedy, demostrando así una mejora en el proceso de resolución.

Existen distintos tipos de movimientos para alcanzar los vecinos. Los movimientos tipo *swap* intercambian pares de autos que requieren distintas opciones de configuración. Sin embargo, Puchta et al. [9] propone otros cinco tipos de movimientos: *Forward/backward Insert*, remueve el vehículo de la posición i para insertarlo en otra posición de la secuencia que no sea i; *SwapS*, intercambia dos autos cuyas opciones de requerimiento son distinto por uno o dos opciones; *SwapT*, intercambia dos autos consecutivos; *Lin2Opt*, ordena inversamente los autos en una sub-secuencia; *Shuffle*, baraja aleatoriamente una sub-secuencia.

Las *estrategias de busqueda* son determinadas por distintas heuristicas para la eleccion del movimiento a realizar en cada iteracion. Las más relevantes son las propuestas por Lee et al. [12] y Davenport et al. [4], quienes consideran la heurística de *min-conflict*, un algoritmo de *hill climbing*, para proponer un escape del mínimo local a medida que incrementan las restricciones infringidas. Michael y van Hentenryck [1] consideran un enfoque usando *tabu search*, donde el tamaño de la lista tabú se adapta con respecto a la necesidad de diversificación. En el documento de Puchta et al. [9] se escoge aleatoriamente uno de los movimientos escritos anteriormente y se evalúa; si el movimiento no empeora la calidad de la solución, se acepta y aplica; en otro caso, se rechaza y se intenta otro movimiento para la solución actual. Sin embargo, este enfoque no siempre permite diversificar y podría significar encontrar el mejor óptimo local pero no uno que se aproxime al global.

3.3.3 Algoritmos Genéticos

Los *algoritmos genéticos* también pueden ser utilizados para resolver el CarSP. Este enfoque se inspira en la evolución biológica y permiten hacer evolucionar una población de individuos sometiéndolas a acciones aleatorias como las mutaciones, recombinaciones genéticas y selección. En este caso, las secuencias seleccionadas en cada generación son combinadas mediante cruces (recombinación genética); si la descendencia creada no satisface la restricción global de permutación, se repara vorazmente; luego, cada una es escalada mediante un *swap*. Este es un enfoque que puede resolver instancias simples del Car Sequencing Problem, siempre que sus tasas de utilización sean bajas. Sin embargo, cuando las tasas de utilización son altas, el número de ejecuciones exitosas decrece considerablemente.

3.3.4 Algoritmo de la Colonia de Hormigas

La optimización por colonia de hormigas (ACO) se utiliza como técnica para solucionar problemas mediante la búsqueda de los caminos más cortos en grafos. Está inspirado en el mundo natural donde las hormigas vagan de manera aleatoria para encontrar comida, una vez que la encuentran regresan a su colonia dejando un rastro de feromonas. Si otra hormiga encuentra el rastro, hay una probabilidad de que estas no sigan caminando aleatoriamente, que sigan el rastro, regresen y lo refuercen si encuentran comida. Los rastros de feromonas desaparecen progresivamente por evaporación. Intuitivamente, se utilizan hormigas artificiales que dejan rastros de feromonas en el grafo y escogen respecto a las probabilidades de las rutas de feromonas. En el ámbito del CarSP, Solnon [14] propone el primer algoritmo ACO para satisfacer la restricción de permutaciones del problema. Las feromonas son dejadas en pares de autos consecutivos con el fin de aprender sub-secuencias de autos convenientes. Posteriormente, en [9] se mejoraría implementando heurísticas greedy. Comparado con el enfoque de búsqueda local, los algoritmos ACO son levemente mejores para límites de tiempo de la CPU pequeños; para límites largos, ambos enfoques mantienen una calidad de solución similar.

3.4 Mejores representaciones y algoritmos

En el ROADEF'2005 [15], una competencia organizada cada dos años por la French Society of Operation Research and Decision-Making Aid, proponen un evento por RENAULT que concierne un CarSP. A los participantes se les proveían datos desde la fábrica de RENAULT y las soluciones realizadas por un software de ellos. Tres tipos de set se usaron para la competencia:

- Set A: El primer set con 16 instancias para la etapa de clasificación.
- Set B: Segundo set con 45 instancias para los equipos clasificados.
- Set X: Último set con 19 instancias para el final de la competencia.

En resumen, ningún método basado en CSP pasó la etapa de clasificación. Todos los métodos clasificados utilizados construyeron la solución inicial con un método simple y rápido, basado en ILP, cuando la función objetivo era minimizar el número de cambios de colores, debido a que el problema era polinomial. Sin embargo, cuando la función objetivo era minimizar la proporción de la cantidad de restricciones infringidas, la gran mayoría de los trabajos propuestos por los competidores desarrollaron heurísticas greedy para generar una buena solución inicial, lo que generó soluciones mejores que las propuestas por RENAULT.

Los mejores resultados fueron obtenidos utilizando búsqueda local con los siguientes operadores (déscritos en la Sección 3.3.2):

- Swap entre dos sub-secuencias (Grup Swap)

- Swap K pares de vehículos (K Swap)
- Inserción de sub-secuencias antes/después de la posición j (Forward/backward Insert)
- Inversión de una sub-secuencia en donde el primer y último vehículo no tenienen el mismo color, son equivalentes en termino de prioridad de las restricciones, o son los primeros o últimos vehículos de corridas del mismo o distinto color (Invert same type).

Cabe destacar que la competencia tenía límites de tiempo y los datos de las instancias eran grandes. El equipo ganador implementó técnicas de búsqueda local que le permitió realizar en promedio 170 millones de evaluaciones de vecinos durante 10 minutos.

Por tanto, la tendencia de resolución al problema consiste mayoritariamente en modelar el CarSP como ILP y luego utilizar una de las cuatro heurísticas descritas anteriormente, siendo la búsqueda local la que ha dado mejores resultados en este contexto.

4 Modelo Matemático (desactualizado)

El siguiente modelo es presentado en [8] como una formulación basada en la programación lineal entera. Una de las ventajas de este modelo es que permite enlazar dos períodos de producción consecutivos (entiéndase como período de producción al proceso de ensamblado en una estación), de modo que las restricciones sobre las capacidades en las estaciones de ensamblaje puedan producir soluciones factibles. Éstas restricciones son expresadas como una proporción máxima de vehículos en una secuencia que pueden tener una opción dada. Básicamente, en una secuencia de s_k vehículos consecutivos, a lo más r_k de éstos pueden tener la opción k.

Como entrada a las lineas de ensamblaje, la formulación agrupa los vehículos en clases de autos que tienen exactamente las mismas opciones (elementos a ser instalados).

Parámetros

$\text{opt} =$ Cantidad total de opciones

$\text{cl} =$ Cantidad total de clases

$n_i =$ Cantidad total de autos en la clase i

$nc =$ Cantidad total de autos

$$o_{ik} = \begin{cases} 1, & \text{si el auto de clase } i \text{ requiere la opción } k \\ 0, & \text{en otro caso} \end{cases} \tag{1}$$

$s_k =$ Largo de una secuencia consecutiva de autos, donde algunos requieren la opción k

$r_k =$ Máximo de autos que pueden tener la opción k en una secuencia consecutiva de s_k autos

$M =$ Valor muy grande para representar $(s_k - r_k)$

Variables

$$
\begin{aligned}
C_{ij} &= \begin{cases} 1, & \text{Si el auto de la clase } i \text{ es asignado a la posición } j \text{ en la secuencia} \\ 0, & \text{En otro caso} \end{cases} \\
Y_{kj} &= \begin{cases} 1, & \text{Si el número de autos que requieren la opción } k \text{ en una secuencia de } s_k \text{ autos que comienza en la posición } j \text{ exceden } r_k \\ 0, & \text{En otro caso} \end{cases} \\
cpp_{mk} &= \begin{cases} 1, & \text{Si el m-ésimo auto del período anterior tenía la opción } k \\ 0, & \text{En otro caso} \end{cases} \\
Z_{kj} &= \begin{cases} 1, & \text{Si el número de autos que requieren la opción } k \text{ en la secuencia de } s_k \text{ autos que comienzan en la posición } j \text{ del período anterior exceden } r_k \\ 0, & \text{En otro caso} \end{cases}
\end{aligned} \tag{2}
$$

Función objetivo

$$\min F = \sum_{k=1}^{\text{opt}} \sum_{j=1}^{nc-s_k+1} Y_{kj} + \sum_{k=1}^{\text{opt}} \sum_{j=1}^{s_k k-1} Z_{kj} \tag{3}$$

Restricciones

- Solo una clase de autos puede ser asignadas a cada posición de la secuencia:

$$\sum_{i=1}^{\text{cl}} C_{ij} = 1 \quad j = 1, \ldots, \text{nc} \tag{4}$$

- Todos los autos de cada clase deben ser incluidos en la secuencia:

$$\sum_{j=1}^{\text{nc}} C_{ij} = n_i \quad i = 1, \ldots, \text{cl} \tag{5}$$

- Restricciones de proporción para las opciones del período actual:

$$
\begin{gathered}
\sum_{i=1}^{\text{cl}} \sum_{l=j}^{j+s_k-1} o_{ik} * C_{il} \leq r_k + M * Y_{kj} \\
k = 1, \ldots, \text{opt} \quad y \quad j = 1, \ldots, \text{nc} - s_k + 1
\end{gathered} \tag{6}
$$

- Restricciones de proporción para la transición entre un período anterior y el actual:

$$
\begin{gathered}
\sum_{i=1}^{\text{cl}} \sum_{l=j}^{j} o_{ik} * C_{il} + \sum_{\text{m=pp}-s_k+j+1}^{\text{pp}} cpp_{mk} \leq r_k + M * Z_{kj} \\
k = 1, \ldots, \text{opt} \quad y \quad j = 1, \ldots, \text{nc} - s_k - 1
\end{gathered} \tag{7}
$$

- Naturaleza de las variables:

$$
\begin{aligned}
&C_{ij} \in \{0, 1\}; \quad i = 1, \ldots, \text{cl} \quad y \quad j = 1, \ldots, \text{nc} \\
&Y_{kj} \in \{0, 1\}; \quad k = 1, \ldots, \text{opt} \quad y \quad j = 1, \ldots, \text{nc} - s_k - 1 \\
&Z_{kj} \in \{0, 1\}; \quad k = 1, \ldots, \text{opt} \quad y \quad j = 1, \ldots, s_k - 1
\end{aligned} \tag{8}
$$

El objetivo de este modelo es minimizar la cantidad de veces que la demanda puede exceder la capacidad de las estaciones de ensamblaje. La función objetivo es representada en la expresión (3); el lado izquierdo de la suma (Y_{kj}) trata sobre los autos del período actual, mientras que el el lado derecho (Z_{kj}) traba sobre la transición entre dos períodos. Un detalle importante es que la restricción de la expresión (6) formaliza los márgenes de las opciones para cada período, lo que permite incumplir la restricción en caso de que no haya soluciones factibles. Lo mismo ocurre para la restricción de la expresión (7) para la transición entre un período anterior y actual.

La función objetivo permite contar la cantidad de restricciones infringidas con el fin de forzar al algoritmo dispersar los problemas de vehículos entre la secuencia, lo que reduce la cantidad total de infracciones a las restricciones y simplificar la gestión de producción en la fábrica.

Las variables descritas son de tipo binarias. C_{ij} indica si los autos de alguna clase están en la secuencia a producir; Y_{kj} indica si se excede la cantidad máxima de autos que pueden tener la misma opción en la secuencia del período actual y Z_{kj} indica lo mismo, pero para la secuencia del período anterior. Por otra parte, la variable cpp_{mk} indica si un auto en la secuencia del período anterior tenía la opción dada, donde pp se refiere al número de autos del período anterior que permite establecer la transición.

El espacio de bśqueda de cada variable es:

$$C_{ij} : 2^{cl*nc}; \quad Y_{kj} : 2^{opt*(nc-s_k-1)}; \quad Z_{kj} : 2^{opt*(s_k-1)} \tag{9}$$

Luego, combinando todas las variables, el tamaño del espacio de búsqueda es:

$$T.E.B = 2^{(cl*nc)*2^{opt*(nc-s_k-1)}*2^{opt*(s_k-1)}} \tag{10}$$

Sin embargo, el tamaño de búsqueda anterior es solamente para el caso en que se instancian las variables en orden. El verdadero tamańo es la combinatoria de los espacios de búsqueda de las tres variables. Como existen muchas variables se produce una explosión combinatoria, por lo que, es necesario recurrir al uso de heurísticas para encontrar una solución.

Finalmente, La solución de este modelo entregará una secuencia de autos que minimizará el número de restricciones infringidas debido a las prioridades de las opciones en el ensamblado. Una desventaja de este modelo es que no considera el color de los autos, sin embargo, puede ser adaptado agregando ciertos parámetros y variables que permitan trabajar con ellos. Véase el *modelo 2* de [8] para esta adaptación.

5 Representación

Resolver el Car Sequencing Problem mediante técnicas de búsqueda completa implica realizar una detallada representación acerca de las soluciones, puesto que se utilizó el lenguaje C++. Éste tipo de técnicas significa tener que utilizar múltiples tipos de soluciones, tales como, variables de tipo entera, vectores y matrices debido a la cantidad de opciones que se deben abarcar. Por lo mismo, hay que considerar que el proceso de backtracking puede significar un alto coste de memoria.

A continuación se explicarán las distintas soluciones utilizadas para la resolución del CarSP mediante FC+GBJ.

5.1 Variables Enteras

Las principales variables tipo entera se asocian a los valores que entrega el input y son:

- **cant_autos**: Variable que almacena la cantidad de autos que deben ser fabricados y puesto en la secuencia. Este valor determina el tamaño que tendrá la solución.
- **cant_opciones**: Variable que almacena la cantidad de opciones existentes en la planta de fabricación para ser instalados en los vehículos, tales como, aire acondicionado, sun-roofs, GPS, etc.
- **cant_clases**: Variable que almacena la cantidad de clases existentes, las cuales diferencian a los vehículos según las cantidades y tipos de opciones asociados a cada uno.
- **cant_min_rest_violadas**: Variable que almacena la cantidad mínima de restricciones de capacidad violadas. Es una variable que se va actualizando conforme avanza el algoritmo y está relacionada con la restricción (6) del modelo matemático.

5.2 Vectores de una dimensión

Se utilizaron vectores de una dimensión para relacionar las distintas características que entrega el input.

Anteriormente se definió la relación p_i/q_i para trabajar la restricción de no exceder la cantidad máxima de autos que pueden tener la misma opción en un mismo bloque; p_i especifica cuántas veces puede aparecer tal opción como máximo en cualquier secuencia q_i de vehículos. Para ello, se utilizan las siguientes representaciones:

- **maxCantAutosBloq**: Vector de enteros que representa los máximos de autos por bloque (p_i), donde el índice es la opción. ($i = 0, \ldots, \mathrm{n}$).

maxCantAutosBloq

Máx. de autos en bloque 0 (p_0)	Máx. de autos en bloque 1 (p_1)	...

- **tamBloq**: Vector de enteros que representa el tamaño del bloque (q_i), donde el índice es la opción. ($i = 0, \ldots, \mathrm{m}$).

tamBloq

Tam. del bloque 0 (q_0)	Tam. del bloque 1 (q_1)	...

- **autos_por_clase**: Vector de enteros que almacena la cantidad de autos de cada clase, donde el índice es el número de la clase.

autos_por_clase

Cant. autos de clase 0	Cant. autos de clase 1	...

- **Solucion**: Vector de enteros que almacena la secuencia de vehículos, donde las celdas, cuyos índices marcan las posiciones, contienen un número que representa a la clase del auto. Ésta solución tiene características muy importantes, ya que, (1) cada celda es una **variable del modelo** (véase la sección de variables del modelo matemático), (2) permite

el cumplimiento de la restricción 4 de asignar una única unidad de producto en cada instante de secuenciación y (3) es un vector dinámico único utilizado para guardar todas las soluciones, es decir, se va sobreescribiendo a medida que se ejecuta el algoritmo.

Solucion

Auto de clase C_i	Auto de clase C_i	...

5.3 Vectores de dos dimensiones

5.4 Dominios

Para trabajar con la técnica de FC+GBJ se utilizan vectores de dos dimensiones llamados **Dominios**. Cada uno consiste en una matriz que representan los **dominios** de cada variable, o bien, los posibles valores que pueden ser asignados a las celdas del vector solución. Éstas matrices son elementos esenciales, puesto que, en cada instanciación de una variable se filtran los dominios de las otras eliminando las clases de autos que puedan provocar soluciones infactibles, como por ejemplo, asignar más vehículos de los disponibles para una clase.

La estructura de este vector consiste en filas que representan las variables del modelo (las celdas del vector solución) y las columnas los posibles valores que puede tomar cada una, por tanto, la clase de auto. A continuación se muestra una matriz de ejemplo al momento de iniciar el programa y asignar valores, esto es, no realizar ninguna instanciación aún. Para este caso, se ingresan 3 autos de clases distintas (0, 1 y 2).

Variable C_i	**Clases**		
0	0	1	2
1	0	1	2
2	0	1	2

Table 1: Matriz dominio de ejemplo para el caso en donde se ingresan 3 autos de 3 clases distintas (0, 1, 2).

Luego, si se asigna el auto de clase 1 en la primera variable, la matriz se modifica a consecuencia de filtrar el dominio con Forward Checking. Por tanto, luego de instanciar la primera variable, los dominios para la segunda instanciación queda así:

Variable C_i	**Clases**	
1	0	2
2	0	2

Para este ejemplo se asume que no hay violación de restricciones en el caso de las opciones; simplemente se elimina de los dominios la clase del vehículo que se asignó y quedó sin stock, de modo que no puedan ser instanciados, ya que producirián una solución infactible.

5.5 Vectores auxiliares

Estos vectores consisten simplemente en relacionar los valores entregados en el input en una sola estructura para organizar los datos y facilitar las consultas.

Uno de ellos es el vector de dos dimensiones de enteros **opciones_por_clase**, el cual almacena las opciones de cada clase de autos. Por ejemplo, para el caso de que existan 4 clases de autos con 3 opciones, cada clase tendrá asociado valores binarios que índican 1 si a tal vehículo se le debe instalar la opción y 0 en caso contrario.

	Opciones		
Clases	**0**	**1**	**2**
0	0	1	1
1	1	1	0
2	0	0	1
3	1	0	0

Por otra parte, un último vector de dos dimensiones de enteros es **opciones_por_secuencia**, el cual es creado para poder calcular la cantidad de restricciones violadas a medida que se va construyendo una solución. Ésta matriz es muy importante, debido a que da consistencia a las restricción (6) del modelo.

Al igual que el vector anterior, relaciona las clases con sus opciones. Sin embargo, ahora las filas son las clases que están asignadas en la solución. Se define a partir del vector de enteros Solucion y opciones_por_clases.

A continuación se muestra el caso en que se han asignado 4 autos en la solución, utilizando la misma relación de opciones por clase del ejemplo anterior. Nuevamente se asume que la secuencia mostrada no viola la restricción (6).

	Opciones		
Vehiculo	**0**	**1**	**2**
2	0	0	1
1	1	1	0
2	0	0	1
3	1	0	0

6 Descripción del algoritmo

Antes de describir el algoritmo es importante detallar sobre algunas características del problema y la técnica a implementar. Es importante recordar que el CarSP se modeló como programación en enteros, donde su función objetivo es minimizar la cantidad de restricciones violadas, es decir, la solución óptima se encuentra cuando la cantidad de restricciones infringidas es cero. Lo anterior implica lo complejo que puede ser aplicar técnicas de búsqueda completa a este problema.

La técnica implementada es *Forward Checking* con *Retorno Guiado por el Grafo de Restricciones*, o más conocido como FC+GBJ. El Fordward Checking es una técnica de búsqueda completa que se basa en un algoritmo de look-forward, donde en cada etapa de la búsqueda del backtracking, se comprueba hacia adelante la asignación de la variable actual con todos los valores de las futuras variables que están restringidas con la variable actual. Los valores de las futuras variables que son inconsistentes con la asignación actual son filtrados temporalmente de

sus dominios. Si el dominio de una variable futura se queda vacío, la instanciación actual se deshace y se prueba con un nuevo valor. Si ningún valor es consistentes, entonces se lleva a cabo el backtracking cronológico. Sin embargo, si se combina con una técnica de retorno dirigido como GBJ, en caso de error, es decir, cuando todas las instanciaciones de una variable fallan, se puede regresar a la variable conectada por una restricción más recientemente instanciada.

La idea principal del algoritmo es, una vez que se obtienen los datos y se crean los elementos descritos en la sección anterior, utilizar *recursivamente* FC+GBJ para búscar soluciones. La estructura general del algoritmo es la siguiente:

Algorithm 1 Forward Checking para CarSP

1. Seleccionar C_i
2. Instanciar $C_i \leftarrow a_i : a_i \in D_i$
3. Razonar hacia adelante (forward-check):
   (a) Eliminar de los dominios de las variables $(C_{i+1}, \ldots, C_n)$ aún no instanciadas, aquellas clases de autos que queden sin stock y/o puedan superar la cantidad de violaciones mínimas si se asigna.
4. Si quedan clases disponibles en los dominios de todas las variables por instanciar, entonces:
   (a) Si i < Cantidad de Autos, incrementar i, e ir al paso (1).
   (b) i = Cantidad de Autos, salir con la solución.
5. Si existe una variable por instanciar, sin valores posibles en su dominio, entonces retractar los efectos de la asignación $C_i \leftarrow a_i$:
   (a) Si quedan valores por intentar en D_i, ir al paso (2).
   (b) Si no quedan valores:
      i. Si i > 1, decrementar i y volver al paso (2).
      ii. Si i = 1, salir sin solución

Donde:

- C_i: Slot i de la solución o variable del modelo.
- D_i: Dominio del Slot i de la solución o variable del modelo.
- a_i: Clase perteneciente al Dominio de i (D_i).
- $i = 1, \ldots,$ CantAutos.

El pseudocódigo anterior representa básicamente como se va creando el árbol de todas las posibilidades mientras realiza Forward Checking. Es importante destacar que el objetivo del algoritmo es ir creando soluciones que vayan minimizando la cantidad de restricciones de capacidad violadas, esto es, cuantas veces se infringe la restricción (6) del modelo matemático, cual índica que la solución tiene sub-secuencias de vehículos que exceden el tamaño máximo de opciones por bloque.

Inicialmente, la cantidad de restricciones de capacidad violadas comienza con un valor negativo y cambia su valor una vez que se encuentra la *primera* solución completa, es decir, con todas las variables instanciadas y que tenga todas las clases asignadas consistentemente; el valor de la cantidad de restricciones violadas por exceder de los tamaños de bloques para cada opción pasa a ser ese valor mínimo, cuyo valor es positivo). Luego, este valor se modifica cada vez que encuentre una solución completa que evaúe una menor función objetivo.

A grandes rasgos, lo que hace el algoritmo es imprimir soluciones cada vez que se encuentre una que minimice ĺas restricciones violadas, por tanto, con el paso del tiempo cuesta más encontrar las mejores soluciones. Finalmente, el óptimo global se encuentra una vez que ésta cantidad es nula.

Continuando con la implementación de las partes, para realizar la selección de variables, ya sea, escoger una celda del vector de soluciones, inicialmente se instancia el primer valor, con i igual a 1. Del mismo modo, la clase que se asigna a la variable se escoge de manera iterativa, esto es, iterar el dominio de la siguiente variable y escoger de menor a mayor.

Por otra parte, para realizar el filtro de los dominios siguientes, se utilizan dos criterios. El primero es eliminar aquellas clases que queden sin stock por haberlas asignado en las instanciaciones. Segundo, se eliminan las clases que asignarlas pueda significar superar el mínimo de violaciones de restricciones de capacidad que hay hasta el momento. De este modo, las restricciones se revisan siempre en el filtro.

Para eliminar las clases que queden sin stock se utiliza un vector de enteros dinámico, esto es, al igual que el vector de solución hay solo uno y se va modificando mediante la ejecución. Cuando se asigna la clase a una variable, se busca en el vector y se decrementa; luego, al momento de realizar el filtrado se revisa y si su stock es cero, se elimina.

En el caso de filtrar la cantidad mínima de restricciones de capacidad violadas se utiliza la variable tipo entera y dinámica descrita en la sección anterior. La idea es que cada vez que se realiza el proceso de filtrado, si una clase no es eliminada por stock, se agrega *parcialmente* a la solución y se evalúa si agregarla podría perjudicar la función objetivo (superar el mínimo de restricciones violadas). En caso de que asignarla no supere el valor mínimo, se deja; en caso contrario, se elimina.

Para controlar los retornos tipo GBJ se utiliza un vector de enteros dínamicos del mismo tamaño de la solución. La idea es que cada vez que se realice el filtro del FC, a la variable

que se le elimine una clase de su dominio, se le guarda en el vector la *última* variable que le provocó una eliminación. De este modo, se tiene un registro de todas las variables conectadas a otras por restricciones, y por tanto, al momento de que FC+GBJ reconozca un fallo, se realice backtracking hasta llegar a la variable que provocó el fallo (la última que le borró un elemento en su dominio). Así, el algoritmo registra conflictos periódicamente.

Finalmente, combinando todas las implementaciones, el valor de la cantidad de restricciones violadas que imprime va disminuyendo hasta un punto en que la ejecución recursiva del FC-GBJ encuentre óptimos globales, es decir, que la cantidad sea nula, o hasta que no pueda encontrar mejores soluciones, lo que quiere decir que la instancia utilizada es infactible.

7 Experimentos

La codificación del algoritmo y la experimentación de las pruebas fueron realizadas por un computador con las siguientes características:

Procesador	Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHZ 2.30GHZ
RAM	12.0 GB
Sistema Operativo	Windows 10 Education

Table 2: Características del computador utilizado para las pruebas del algoritmo

Como las instancias del problema están en archivos de texto, lo primero que se realizó fue una lectura de éstos, almacenar los datos en sus respectivas variables y posteriormente ejecutar el algoritmo. El programa tiene la opción de especificar que archivo se quiere abrir, por lo que, para modificar los parámetros simplemente se cambiaban los valores desde el archivo antes de ejecutarlo. Este era un ejercicio necesario, debido a que trabajar con técnicas de búsqueda completa implicaba perder eficiencia para instancias muy grandes.

Para la construcción del algoritmo era necesario trabajar con parámetros pequeños que asegurasen encontrar soluciones óptimas en cortos periodos de tiempo, o en su defecto, determinar que el problema era infactible. Además, estas técnicas son deterministas, por lo que, obtener soluciones óptimas distintas en cada ejecución indicaría errores o fallas del algoritmo. Por tanto, se experimentó principalmente con la instancia de prueba *Problem Test*, la cual representa un caso para 10 vehículos, hasta lograr que el algoritmo retornara siempre las mismas soluciones óptimas. También se modificó para encontrar otros tipos de errores.

Más tarde se intentaría con las instancias grandes, cuales representaban casos de hasta 100 y 200 vehículos. El objetivo era experimentar si el programa tenía problemas de asignación de memoria a causa de la excesiva recursión que podría significar una técnica como ésta. Finalmente, se buscó analizar que resultados dejarían las instancias mayores, puesto que se sabía de antemano cuales eran satisfacibles o inviables.

	Valores		
Instancia	**Vehículos**	**Opciones**	**Clases**
Problem Test	10	5	6
Problem 4/72	100	5	22
Problem 6/76	100	5	22
Problem 10/93	100	5	25
Problem 16/81	100	5	26
Problem 19/71	100	5	23
Problem 21/90	100	5	23
Problem 36/92	100	5	22
Problem 41/66	100	5	19
Problem 26/82	100	5	24

Table 3: Instancias utilizadas para experimentación en base a sus características.

8 Resultados

Con el fin de poder analizar las variaciones de los resultados en base a los distintos experimentos con las instancias, se registraron distintos datos dependiendo de las instancias.

Para el caso de los experimentos con la instancia pequeña *Problem Test* se obtuvieron resultados positivos debido a que se encontraron soluciones óptimas que minimizaban totalmente la cantidad de restricciones violadas. Las siguientes tablas muestran algunos resultados generales de su ejecución:

Instancia	**Tiempo**	**Sol. Óptimas Encontradas**
Problem Test	0.03125	6

Table 4: Resultados generales para la experimentación con la instancia Problem Test.

Se observa que el tiempo que tardó en entregar todas las soluciones factibles es considerablemente bajo. La cantidad de vehículos que se ingresan influye en estos resultados, sin embargo, gran parte se debe al uso de la técnica FC+GBJ. Si se vuelve a analizar el tamaño de búsqueda planteado en la sección del modelo matemático, es posible notar e inferir la cantidad de posibilidades que debe recorrer el algoritmo. Más aún, es sorprendente notar que solo existe 6 soluciones óptimas dentro del inmenso total de combinaciones posibles para tal instancia.

Solución 1							Solución 2							Solución 3						
0		1	0	1	1	0		0		1	0	1	1	0		0		1	0	1	1	0
1		0	0	0	1	0		2		0	1	0	0	1		2		0	1	0	0	1
5		1	1	0	0	0		5		1	1	0	0	0		5		1	1	0	0	0
2		0	1	0	0	1		1		0	0	0	1	0		1		0	0	0	1	0
4		1	0	1	0	0		4		1	0	1	0	0		5		1	1	0	0	0
3		0	1	0	1	0		3		0	1	0	1	0		3		0	1	0	1	0
3		0	1	0	1	0		2		0	1	0	0	1		4		1	0	1	0	0
4		1	0	1	0	0		4		1	0	1	0	0		2		0	1	0	0	1
2		0	1	0	0	1		3		0	1	0	1	0		3		0	1	0	1	0
5		1	1	0	0	0		5		1	1	0	0	0		4		1	0	1	0	0

Table 5: Soluciones óptimas (cantidad de restricciones violadas nula) encontradas en la experimentación con la instancia Problem Test. La columna izquierda representa la secuencia óptima según sus clases y las columnas derechas las opciones de cada clase.

Solución 4							Solución 5							Solución 6						
4		1	0	1	0	0		5		1	1	0	0	0		5		1	1	0	0	0
3		0	1	0	1	0		2		0	1	0	0	1		3		0	1	0	1	0
2		0	1	0	0	1		4		1	0	1	0	0		4		1	0	1	0	0
4		1	0	1	0	0		3		0	1	0	1	0		2		0	1	0	0	1
3		0	1	0	1	0		3		0	1	0	1	0		3		0	1	0	1	0
5		1	1	0	0	0		4		1	0	1	0	0		4		1	0	1	0	0
1		0	0	0	1	0		2		0	1	0	0	1		1		0	0	0	1	0
5		1	1	0	0	0		5		1	1	0	0	0		5		1	1	0	0	0
2		0	1	0	0	1		1		0	0	0	1	0		2		0	1	0	0	1
0		1	0	1	1	0		0		1	0	1	1	0		0		1	0	1	1	0

Table 6: Soluciones óptimas (cantidad de restricciones violadas nula) encontradas en la experimentación con la instancia Problem Test. La columna izquierda representa la secuencia óptima según sus clases y las columnas derechas las opciones de cada clase.

Un detalle interesante en las soluciones mostradas en las Tablas 5 y 6 es la dispersión de la ubicación de estas. Las soluciones 1, 2 y 3 están relativamente cercanas entre si, ya que, sus secuencias son parecidas, las 3 parten con el auto clase 0; mientras que las otras soluciones se crean teóricamente al otro extremo del árbol de combinaciones. Esto quiere decir que, para instancias más grandes, la forma en que eligen las variables o valores de los dominios pueden influir en la rápidez en que se encuentran soluciones. El algoritmo desarrollado para este documento realiza las elecciones de los valores de los dominios de manera secuencial (menor a mayor); sin embargo, usar técnicas como de *ordenamiento de variables o valores* podrían mejorar considerablemente los resultados, sobretodo para las instancias de a continuación.

La siguiente tabla muestra los resultados obtenidos para instancias grandes, esto es, de 100 vehículos con 20 opciones en promedio. Todas las instancias fueron ejecutadas con 5 minutos de límite; al pasar ese tiempo, se detuvo el algoritmo y se entregó como solución parcial la última que minimizaba la cantidad de restricciones violadas.

	Valores			Resultados Solución encontrada	
Instancia	**Vehículos**	**Opciones**	**Clases**	**N° Rest. Violadas**	**Tiempo [s]**
Problem 4/72	100	5	22	171	453.00
Problem 6/76	100	5	22	151	461.00
Problem 10/93	100	5	25	173	300.74
Problem 16/81	100	5	26	173	329.09
Problem 19/71	100	5	23	173	317.27
Problem 21/90	100	5	23	173	355.02
Problem 36/92	100	5	22	163	303.76
Problem 41/66	100	5	19	152	366.29
Problem 26/82	100	5	24	167	301.06

Table 7: Resultados obtenidos en la experimentación con instancias de 100 vehículos con un límite de 300 $[s]$ de ejecución.

Un detalle importante es que todas terminaron con cantidades de restricciones violadas altas, las cuales después del primer minuto ya dejaban de disminuir, es decir, la rápidez con que se encontraban soluciones mejores disminuía con el tiempo. A los 5 minutos ya no disminuían.

Son diversos los fáctores que influyen estos resultados. Primero, la técnica de búsqueda completa intenta buscar el óptimo global, por lo que, busca en una infinidad de opciones, a pesar de que se filtren los dominios. Segundo, depende de la técnica que acompañe al FC+GBJ; anteriormente se dijo que el algoritmo implementado elegía los valores de menor a mayor, por lo que aquí hubiese sido útil experimentar con heurísticas de ordenamiento de valores. Esto quiere decir que las soluciones encontradas pertenecen a las primeras combinaciones de secuencias, mientras que la solución óptima global podría estar incluso en el otro extremo (aquellas que parten con vehículos de clase 5).

Finalmente, la última tabla muestra los resultados obtenidos para un tiempo de ejecución de 3 horas aproximandamente para la instancia *Problem 4/72*.

Instancia	**Tiempo [s]**	**N° Rest. Violadas**
Problem 4/72	10951	160

Table 8: Resultados de la ejecución de la instancia 4/72 por 3 horas.

Como se esperaba, no fue mucho lo que varió con 3 horas más de ejecución, donde solo se disminuyó en 21 cantidad de restricciones violadas para la misma instancia. Esto quiere decir, al menos con el algoritmo implementado en éste estudio, que resolver instancias grandes podría tomar una cantidad indeterminada de tiempo, incluso para resolver que no existe una solución óptima.

El siguiente gráfico muestra la relación entre la cantidad de restricciones infringidas vs el tiempo para la ejecución de la instancia Problem 4/72.

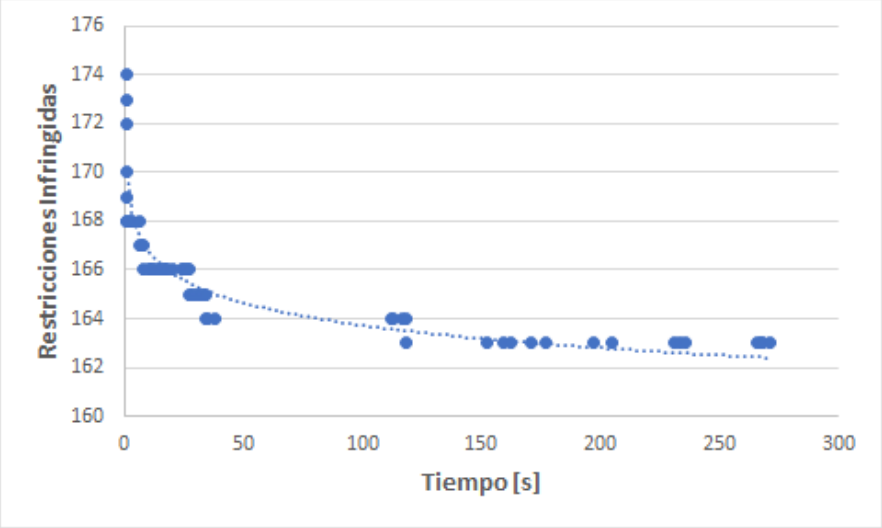

Figure 1: Gráfico de Cantidad de Restricciones Infringidas vs Tiempo para la ejecución de la instancia Problem 4/72 en un rango de 5 minutos.

Se puede observar que a medida que pasa el tiempo, decrece la rapidez con la que se encuentran soluciones, llegando a un punto que deja de disminuir por largos periodos de tiempos. La recursión llega a un cierto punto en que se generan muchas combinaciones a evaluar y las nuevas llamadas tardan mucho en evaluarse por ser demasiadas. En otras palabras, el gráfico representa como se baja de nivel en el árbol de búsqueda, donde al principio es fácil instanciar y luego se vuelve más difícil debido a la complejidad de las restricciones en tales puntos.

Lo anterior se debe de igual manera a la forma en que se asignan las clases de autos de los dominios, de modo que las soluciones se van encontrando secuencialmente. Además, la recursión llega a un cierto punto en que se generan más combinaciones a evaluar. En otras palabras, el gráfico representa

Por tanto, las técnicas de búsqueda completa son efectivas para resolver instancias fáciles o pequeãs del CarSP y pierden competitividad con instancias grandes, en el sentido de encontrar la solución que nulifique la cantidad de restricciones violadas.

9 Conclusiones

En este documento se estudiaron distintos enfoques y técnicas para abarcar el Car Sequencing Problem, como también se desarrolló una implementación para resolverla mediante la utilización de la técnica de búsqueda completa Forward Checking con Retorno Guida por el Grafo de Restricciones (FC+GBJ).

Los enfoques principales se centran en modelar el problema como CSP o ILP. Las distintas heurísticas analizadas, tales como, los enfoques greedy, métodos de búsqueda local, algoritmos genéticos y colonia de hormigas se adaptan correctamente al CarSP, por lo que, todas pueden resolver el mismo problema. En base al modelo descrito en la Sección 4 y a la competencia ROADEF'2005, la estrategía más prometedora es formular el CarSP como un problema de programación lineal entera porque permite encontrar una secuencia óptima mediante la minimización de algún critero. Cualquiera de las heurísticas que se utilice debiese construir una

solución inicial rápidamente si el objetivo principal es minimizar y el problema es polinomial. Sin embargo, los resultados muestran que cuando se busca minimizar la cantidad de veces que se infringe una restricción de capacidad es mejor utilizar heurísticas tipo greedy. Finalmente, para encontrar mejores resultados conviene utilizar búsqueda local con los criterios descritos en la Sección 3.

En el contexto de la implementación del algoritmo para resolver el CarSP, primero se modeló el problema como ILP, cuyo objetivo se basaba en encontrar soluciones óptimas recursivamente, minimizando la cantidad de restricciones infringidas debido a las prioridades de las opciones en el ensamblado. No obstante, la relación del CarSP como problema de tipo CSP es fuerte, ya que para encontrar el óptimo global es necesario encontrar la solución minimice tal cantidad a nula. De todas las pruebas realizadas, se concluyó que la técnica asignada es efectiva con instancias pequeñas del CarSP; mientras que para instancias grandes, pierde competitividad.

Algunos de puntos importantes que se pueden rescatar de la experimentación son:

- Modelar el problema como tipo CSP y utilizar técnicas de búsqueda completa es efectiva con instancias pequeñas del CarSP; mientras que para instancias grandes, pierde competitividad. Esto se debe que al tener un espacio de búsqueda grande le toma mucho tiempo al algoritmo recorrer todas las posibles soluciones para encontrar el óptimo global.
- Lo anterior se puede mejorar si se utilizan distintas heurísticas para ordenar variables o valores. Usar el algoritmo recursivamente eligiendo valores de menor a mayor puede afectar considerablemente en la búsqueda de óptimos, ya que las mejores soluciones están dispersas.
- Considerando el Estado del Arte realizado, los mejores resultados se logran usando útilizando técnicas de búsqueda incompletas, de modo que la búsqueda local tiene los mejores resultados hasta ahora. Quizás no logren encuentrar los óptimos globales pero si pueden minimizar en mejor tiempo la cantidad de restricciones violadas.
- Se demostró que las técnicas de búsqueda completa son deterministas. Para una instancia que entrega soluciones óptimas globales (o que no tiene solución), ejecutar el algoritmo debe dar siempre los mismos resultados.
- A mayor cantidad de vehículos entregados, se deben realizar más iteraciones para disminuir la cantidad de restricciones violadas que entregan las soluciones.
- Si hay una cantidad de autos constantes pero la cantidad de clases aumentan, se realizan más iteraciones debido a que se complica el problema.

Finalmente, una idea interesante para continuar con investigaciones futuras es la posibilidad de integrar heurísticas que tengan capacidades complementarias; enfoques como la programación con restricciones y la programación en enteros pueden ser utilizados para proveer acercamientos con una solución inicial que permita podar eficientemente el espacio de búsqueda. En el sentido de la implementación realizada, mantener la recursión del FC+GBJ pero cambiar la forma en que se instancias las variables.

10 Bibliografía

References

[1] *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2002. ACM. 548021.

[2] Joaquín Bautista. Modelos y métricas para la versión robusta del car sequencing problem con flotas de vehículos especiales. *Direccion y Organizacion*, 60:57–65, 2016.

[3] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *The Journal of Logic Programming*, 27(3):185 – 226, 1996.

[4] Andrew J. Davenport and Edward P. K. Tsang. Solving constraint satisfaction sequencing problems by iterative repair. pages 345–357, 1999.

[5] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings of the 8th European Conference on Artificial Intelligence*, ECAI'88, pages 290–295, Marshfield, MA, USA, 1988. Pitman Publishing, Inc.

[6] Andreas Drexl and Alf Kimms. Sequencing jit mixed-model assembly lines under station-load and part-usage constraints. *Management Science*, 47(3):480–491, 2001.

[7] B Estellon and F Gardi. Car sequencing is np-hard: a short proof. *Journal of the Operational Research Society*, 64(10):1503–1504, 2013.

[8] Caroline Gagné, Marc Gravel, and Wilson L. Price. Solving real car sequencing problems with ant colony optimization. *European Journal of Operational Research*, 174(3):1427 – 1448, 2006.

[9] Jens Gottlieb, Markus Puchta, and Christine Solnon. A study of greedy, local search, and ant colony optimization approaches for car sequencing problems. pages 246–257, 2003.

[10] M. Gravel, C. Gagné, and W. L. Price. Review and comparison of three methods for the solution of the car sequencing problem. *The Journal of the Operational Research Society*, 56(11):1287–1295, 2005.

[11] Tamás Kis. On the complexity of the car sequencing problem. 32, 07 2004.

[12] J. H. M. Lee, H. F. Leung, and H. W. Won. Performance of a comprehensive and efficient constraint library based on local search. pages 191–202, 1998.

[13] Bruce D. Parrello, Waldo C. Kabat, and L. Wos. Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *Journal of Automated Reasoning*, 2(1):1–42, Mar 1986.

[14] Christine Solnon. Solving permutation constraint satisfaction problems with artificial ants. In *Proceedings of the 14th European Conference on Artificial Intelligence*, ECAI'00, pages 118–122, Amsterdam, The Netherlands, The Netherlands, 2000. IOS Press.

[15] Christine Solnon, Van Dat Cung, Alain Nguyen, and Christian Artigues. The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the roadefâ2005 challenge problem. *European Journal of Operational Research*, 191(3):912 – 927, 2008.

[16] Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic programming. *Artif. Intell.*, 58(1-3):113–159, December 1992.