Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Optimización de la selección de horneado de discos de zirconia para la manufactura de piezas dentales

Tesis de Licenciatura en Ciencias de la Computación

Gonzalo Raposo
gonzalo.a.r@gmail.com
LU 144/03

Director: Pablo Factorovich (factorovich@gmail.com)
Codirector: Brian Curcio (bcurcio@gmail.com)
Buenos Aires, 2024

OPTIMIZACIÓN DE LA SELECCIÓN DE HORNEADO DE DISCOS DE ZIRCONIA PARA LA MANUFACTURA DE PIEZAS DENTALES

La manufactura de discos de zirconia involucra un proceso de calentamiento y enfriado que dura entre 3 y 5 días hábiles. La cantidad de hornos disponibles en una fábrica estándar, y consecuentemente la capacidad para hornear/producir discos, puede ser considerablemente menor que la cantidad de discos necesarios para suplir la demanda de ordenes de compra si no se la gestiona adecuadamente. En esta tesis estudiamos el uso de una metaheurística como solución a la toma de decisión: qué discos deberían ser seleccionados para el siguiente horneado.

Palabras claves: Optimización Combinatoria, Zirconia, Manufactura, Horno, Fabricación, Orden de compra.

Índice general

1. INTRODUCCIÓN

1.1. Uso de discos de zirconia

En la fabricación de piezas dentales, un material ampliamente utilizado es la zirconia [1], reconocida por su excepcional resistencia y durabilidad. A partir de este material se forman discos de determinado diámetro y diferentes espesores, a partir del cual se muelen las piezas dentales capa tras capa, de una forma similar a la de un escultor removiendo mármol para dar lugar a la escultura, aunque de una forma automática, mediante una máquina que recibe como entrada el disco de zirconia y un plano 3D de cómo deberían ser las piezas finales.
Estamos en contacto con una empresa que no solo fabrica las piezas dentales finales, sino también los discos de zirconia usados para tal fin, según la demanda de órdenes de compra que la misma empresa recibe. Considerando que el proceso desde que se recibe una orden de compra a través del sitio web, hasta que la pieza dental final se envía por correo está ampliamente digitalizado[1], resulta de especial importancia el software que se usa para decidir qué discos se deberían hornear, según la cantidad y características de las órdenes de compra al momento de la toma de la decisión.

1.2. Fabricación de discos de zirconia

Más allá de la resistencia y durabilidad que ofrece la zirconia, otros factores de relevancia por los cuales es ampliamente usado son su biocompatibilidad (se integra bien con tejidos humanos sin causar reacciones adversas), resistencia a la corrosión (como pieza dental, está continuamente expuesto a humedad, bacteria y diferentes niveles de pH), es hipoalergénico, tiene baja conducción térmica y, además, por sus cualidades estéticas, ya que su proceso de fabricación, sombreado y coloración le da un aspecto similar a un diente real [2].
El disco de zirconia se presenta en estado sólido como resultado del tratamiento térmico en el horno y anterior a iniciar su proceso de moler. Previo al tratamiento térmico, la zirconia se presenta en forma de polvo. Para obtener el disco de zirconia en estado sólido a partir de su estado en polvo, la zirconia se mezcla con un material aglutinante (en inglés *binder*), cuyo propósito es proveer una estructura rígida en forma de disco que actúe como molde de la zirconia, previo a su tratamiento térmico que finalmente proveerá la firmeza y el resto de los atributos previamente descritos.
La estructura rígida previo al tratamiento térmico se logra mediante un proceso de prensado isostático del polvo de zirconia y el material aglutinante. Durante el proceso térmico, este material se evapora, y, junto a los efectos obtenidos producto de la exposición de zirconia a altas temperaturas (entre 900 y 1000 grados centígrados), se logra obtener el disco de zirconia puro en su versión final. Considerando que demasiado aglutinante evaporado aumenta la presión interna dentro del horno y produce quebramientos y otros efectos no deseados en los discos, es de vital importancia controlar la cantidad de aglutinante presente en el horno al momento de iniciar la cocción.

[1] El término específico se conoce como *Fabricación asistida por computadora*, o **CAM**, por su nombre en inglés *Computer Aided Manufacturing*.

Por este motivo, más allá de la importancia del material aglutinante en lo que se refiere a su función de mantener temporalmente el polvo en forma de disco, tiene un rol preponderante en la toma de decisión sobre qué discos hornear, puesto que la cantidad de aglutinante en el horno, determinada por la cantidad de discos y su respectivo espesor, es un factor limitante en la toma de decisión.

1.3. Descripción del horno

Previo a explorar los factores de relevancia en la toma de decisión sobre qué orden de compra incluir, o no, en un lote de discos en el siguiente programa de horneado, es importante describir algunas características de los hornos, como contexto del cual surgen algunas limitantes para la toma de decisiones.
La principal característica del horno que se utiliza para aplicar el proceso térmico[2] a los discos de zirconia es la cantidad de espacio que tiene para ubicar los discos en su interior. El total de espacio se determina por sus dimensiones físicas. Cada horno se puede pensar como una matriz con filas y columnas, que determinan una ranura con profundidad equivalente a la profundidad del horno. En cada ranura es donde efectivamente se ubican los discos, de modo que el máximo volumen de discos que se pueden cargar en un horno equivale a la sumatoria de volumen de cada ranura, habiendo una cantidad total de ranuras equivalente a $cantidad\_filas \times cantidad\_columnas$.
Por otro lado, los discos contienen un material químico que se evapora durante el proceso de horneado, y siendo que los hornos en consideración no cuentan con un proceso de extracción de gases, hay un limitante de cuántos discos se pueden cargar en el horno considerando el porcentaje de este material (llamado *aglutinante*, se explica en detalle en la sección *Factores físicos y químicos*) por disco, y su densidad.

1.3.1. Plan de horneado

El plan de horneado se determina por dos atributos principales:

- Discos: cuáles se ubican dentro de cada ranura.
- Programa: refiere al programa de PLC[3] va a controlar el plan. Dicho control establece cómo varía la temperatura durante el tiempo en el que los discos están en el horno. Siendo que el foco principal de este trabajo son los discos del horno, el programa no es parte del alcance de nuestro análisis.

Finalmente, cabe destacar que debido a las altas temperaturas que el horno va a alcanzar (1000 grados centígrados), no hay etiquetas que identifiquen las ranuras dentro del mismo, lo cual supone un nuevo desafío en cuanto a cómo identificar a qué orden de compra pertenece cada disco. Como vemos más adelante, tanto el sistema como el operario siguen una convención en donde discos de distinto espesor no pueden ser ubicados en la misma ranura.

[2] El nombre técnico en inglés es *sintering*.
[3] Por sus siglas en inglés *Programable Logic Controller*. Dispositivo que permite controlar mediante una lógica programable qué conectores recibirán una señal eléctrica y por cuánto tiempo.

1.4. Factores de relevancia

Nos enfocamos ahora en cuáles son los factores de relevancia que influyen en la decisión de qué discos incluir en la siguiente ronda de horneado. Es importante destacar diferentes categorías de factores, dependiendo de la naturaleza de la cual se desprende la restricción que determina el factor de relevancia.

1.4.1. Factores físicos y químicos

Como ya hemos mencionado, la cantidad de aglutinante presente en el horno está restringida, y es determinado por la cantidad y espesor de los discos, la densidad del material y el porcentaje de aglutinante combinado con cada disco de zirconia, que varía según el tipo de material usado. En la sección de experimentación se detallan las instancias particulares empleadas.
Si bien es cierto que hay una relación entre el aglutinante y el volumen disponible en el horno (un mayor volumen disponible permite mayor cantidad de evaporación de aglutinante), no se puede reducir uno en función del otro, puesto que hay un tercer atributo que contribuye al uso del volumen dentro del horno. Estos son los separadores.
Los separadores son elementos de material no evaporable cuya función es separar los discos ubicados en forma consecutiva, uno detrás del otro en cada ranura. A mayor espesor del disco, mayor número de separadores se necesita. Cada separador tiene un diámetro y un espesor predeterminado. De modo que un disco de 30 mm de espesor necesita más separadores que un disco de 10 mm. Los separadores lógicamente ocupan un volumen no despreciable dentro del horno, y no contribuyen al total de aglutinante presente. Por este motivo, el total de aglutinante y el volumen físico disponible son variables que se considerarán en forma separada.
Lógicamente, el volumen del horno es otro factor importante, puesto que no se pueden ubicar más discos que el espacio físico que provee dicho horno. La forma específica de considerar el volumen como un factor determinante para la elección de las órdenes de compra es comparando la sumatoria del espesor de los discos que componen cada orden de compra (y sus respectivos separadores) con el volumen disponible en el horno. Dicho volumen lo calculamos según cada ranura en donde se pueden ubicar discos. De modo que si tenemos un horno de 4 filas, 6 columnas, una profundidad de 240 mm y un diámetro de disco de 98 mm, el volumen total para ubicar discos dentro del horno es de $\pi \times 49^2 \times 240 \times 4 \times 6$ mm^3
Considerando que la estética es un factor de relevancia y que existen ligeras diferencias entre dos lotes de discos horneados, se deduce que los discos correspondientes a una orden de compra no pueden ser ubicados en dos lotes distintos. Dicho de otra forma, una orden de compra se incluye en su totalidad en el plan del horno, o no se incluye.
Finalmente, y considerando que no hay identificadores dentro del horno para cada ranura donde se ubiquen los discos, en el proceso de fabricación se suele acordar una convención de ubicación de discos para facilitar su identificación, que tanto el sistema tiene en cuenta a la hora de tomar decisiones, como el operario también tiene en cuenta a la hora de ubicar los discos y retirarlos del horno. En nuestro caso en particular, dicha convención establece que en una misma ranura todos los discos tienen que poseer el mismo espesor, lo cual deriva en otro factor de relevancia para la toma de decisión sobre el plan a crear. A este factor de relevancia lo llamamos *compatibilidad de discos*, puesto que el conjunto de órdenes de compra podría ser válido en cuanto al contenido de aglutinante permitido

y volumen disponible en el horno, pero no existe una forma válida de ubicar los discos según esta restricción particular.

1.4.2. Factores de negocio

En cuanto a restricciones de lógica de negocio a la hora de armar un plan de horneado, el factor más relevante es la prioridad de la orden de compra.
Las órdenes de compra se determinan que son de prioridad o bien por ser asignadas de dicha forma cuando se crean (debido al volumen de compra, el cliente, etc.) o bien por la cantidad de días transcurridos desde que se creó la orden hasta el día de la fecha, a saber, mayor la espera, mayor la necesidad de resolver esa orden. Para simplificar el proceso en cuestión, abstraemos todas estas condiciones en simplemente un atributo *prioridad* para la orden de compra. De modo que, a efectos de toma de decisión, las órdenes de prioridad deberían ser consideradas primero en la ubicación dentro del siguiente plan de horneado. Sin embargo, es importante destacar que no es condición necesaria que todas las órdenes de compra de prioridad al momento de la toma de decisión sean parte del plan resultante, aun si estas fueran compatibles. Podría darse el caso que se cree un plan de horneado que no incluya todas las órdenes de prioridad, porque permite hacer un mejor uso de la capacidad del aglutinante y volumen del horno.

1.5. Objetivo de la tesis

El objetivo de la tesis consiste en generar una solución computacional que determine planes de horneado válidos (según los factores de relevancia enumerados) que permita a los operarios usarlos para cargar el horno con las órdenes de compra seleccionadas. Los resultados se tienen que obtener de una forma eficiente tanto en materia de calidad de solución como en tiempo de ejecución.
Para dar con una solución factible, incluimos como parte de los objetivos de la tesis hacer una revisión del estado del arte de esta problemática (o similares), así como también su definición precisa.
Puesto que para dar con dicha solución se necesitan hacer experimentos computacionales, incluimos como parte de los objetivos generar instancias de pruebas y proveer un entorno de ejecución en donde dichas pruebas puedan ser ejecutadas y validadas.
Finalmente, pretendemos que el generador de instancias, los algoritmos de la solución y el simulador de pruebas sean los suficientemente parametrizables para ser usado por tomadores de decisiones como una herramienta para decidir, por ejemplo, qué características de horno serían adecuadas considerar cuando se piensa adquirir uno nuevo.

2. REVISIÓN BIBLIOGRÁFICA

Como primer paso en la elaboración de una posible solución a esta problemática de selección de órdenes de compra, recurrimos a la revisión bibliográfica, con el objetivo de entender qué trabajos ya se han realizado, sea para la misma problemática en cuestión, o para alguna similar sobre la cual podamos extrapolar conclusiones, ideas y lecciones aprendidas, que nos permitan una mejor solución.

2.1. Búsqueda de trabajos relacionados

Una primera búsqueda en *Google Scholar* y *PubMed* mediante el término *zirconia disc manufacturing algorithm* nos permite observar que, en su amplia mayoría, los resultados refieren al proceso de fabricación, en categorías como:

- Impresión 3D para la fabricación de implantes dentales [3].
- Análisis espacial de la superficie de discos de zirconia expuestos a distintos tratamientos por láser [4].
- Diferencias entre piezas a base de zirconia creadas con impresoras 3D o *milling machines* [5].
- Explicación del proceso de fabricación asistido por computadora y su relación con aspectos clínicos [6].
- Uso de zirconia y tecnología en la fabricación de restauraciones dentales [7].
- Metaheurística para la optimización de discos de zirconia de capas múltiples con graduación [8].
  Si bien este trabajo está más relacionado que los anteriores a la problemática en cuestión, lo cierto es que el foco aquí está en cómo determinar las capas de los discos de zirconia, junto a su efecto degradé, para obtener el producto final con la coloración buscada. No es relevante a nuestros efectos porque, si bien el diseño de las capas de los discos (en los casos de discos multicapas) es parte del proceso de fabricación, no es parte del alcance de nuestro estudio porque ese diseño ya estaba preestablecido en la empresa sobre la cual nos inspiramos en esta tesis.
- Análisis de modelos de regresión para encontrar parámetros de cortado de las piezas finales y obtener una textura final ideal [9].
- Materiales cerámicos y tecnologías aplicadas a odontología restaurativa basada en implantes [10].

Si bien ese tipo de trabajo nos ayuda a comprender el contexto, no se aplica directamente a la problemática en cuestión, dado que, como se puede observar, el foco no está relacionado con la selección de discos para los lotes de horneado. Por tal motivo, cambiamos el criterio de selección a *manufacturing furnace planning algorithm*, donde obtenemos mejores resultados.

2.2. Refinamiento del criterio de búsqueda

El primer resultado que obtenemos es un trabajo de planeamiento de orden para la fabricación de planchas de acero [11]. Si bien el foco de estudio es completamente distinto al de esta tesis (planchas de acero vs. piezas dentales), hay algunos aspectos importantes que se pueden extrapolar de este trabajo. El primero de ellos es el hecho de que el objetivo es minimizar una función de costo multi-objetivo (costo de demora, costo de inventario y utilidad de las capacidades). Para tal fin, se determina una suma con pesos de cada coste para así transformar un problema de optimización multidimensional en uno de una sola dimensión. Las restricciones inherentes al problema se modelan mediante penalidades de la función objetivo. La solución computacional es una metaheurística de tipo *optimización de enjambre de partículas*[1].
Se encuentra entre los resultados otro trabajo similar en la industria del acero, donde se crea un sistema de optimización multicriterio para tratamientos de calor en paralelo, basado en simulaciones [12]. En este estudio el objetivo principal es la optimización del uso de energía mediante un mejor planeamiento de la secuenciación o disposición de órdenes en hornos paralelos, para su correspondiente tratamiento por calor. Una observación importante que podemos dilucidar de la revisión de este trabajo es el alcance de la toma de decisión sobre qué disco incluir en el horno o no. Si el alcance involucra solo un horno disponible al momento de crear el plan, el objetivo entonces es determinar cuál es la mejor aplicación posible en ese momento. Si, en cambio, el alcance involucra máquinas (hornos) paralelas, el objetivo se centra entonces en el orden de ubicación de órdenes en el conjunto de hornos que pueden funcionar en paralelo, tal cual lo estudia el trabajo en cuestión. El algoritmo de búsqueda de una solución cercana al óptimo está basado en metaheurísticas, particularmente *algoritmos genéticos*[2].
Otro resultado de la búsqueda es un caso de estudio en procesos de fabricación que involucran el uso de hornos para tratamientos de calor [13]. Este trabajo también trata de máquinas (hornos) paralelas y la creación de un planeamiento dinámico de órdenes que pueden llegar en un momento dado y cambiar un cronograma de órdenes existente. Más allá del aspecto dinámico, otro aspecto destacable de este estudio es el *set up time* de cada máquina previo a dar lugar al tipo tratamiento de calor para el lote de órdenes seleccionado. Este es un punto importante porque en nuestra problemática de selección para los hornos no contamos con *set up time*, más allá del tiempo que le toma al operario ubicar los discos seleccionados en el horno, que a fines prácticos se desprecia. Para atender la necesidad de tiempos de preparación antes mencionado, la solución está basada en metaheurísticas también, particularmente **GRASP**[3], por sus siglas en inglés.
Otro trabajo de interés que forma parte de los resultados de la búsqueda está relacionado también con tratamientos de calor en fabricación de materiales, siendo aluminio en este caso [14]. Este es también un caso de optimización multiobjetivo, ya que, por un lado, se busca minimizar el número de planes para cargar el horno, y por el otro, minimizar el material sobrante. Lo interesante de este problema es que hay dos instancias de agrupamiento: decidir en qué horno agrupar las órdenes de entrada, y en segundo lugar, decidir cómo agrupar el material derretido para ubicarlo en los moldes finales. La solución también está basada en metaheurísticas, particularmente *Specialized Multi-Objective Ant Bee Co-*

[1] Nombre en inglés es *Particle Swarm Optimization*, **PSO**.
[2] *Genetic Algorithms* o simplemente **GA**, por sus siglas en inglés.
[3] El nombre completo es *Greedy Randomized Adaptative Search Procedure*.

lony and Decomposition[4]. Más allá de las particularidades de este problema que lo hacen distinto al que se intenta resolver en nuestra tesis, es interesante el análisis de literatura que se hace previo a determinar cuál es el método propuesto. En particular, plantean que el primer problema de optimización de agrupamiento lo ven como un problema que bien podría ser modelado como el problema de la mochila[5] o el problema de empaquetamiento[6]. Esto es de particular importancia, puesto que el problema que estudiamos en esta tesis es similar a ese primer agrupamiento, con la diferencia de que no consideramos múltiples hornos, sino uno solo.
En líneas similares a este último trabajo revisado, encontramos este otro estudio de planeamiento y organización de moldes para minimizar el stock, demoras y tiempos de preparación de los hornos[15]. Los autores dividen el problema en dos etapas: en primer lugar, la carga del horno, que lo modelan como un problema de la mochila. En segundo lugar, el planeamiento de carga de los hornos, para lo cual usan un algoritmo genético. A esta combinación la llaman **GAKP**, por sus siglas en inglés *Genetic Algorithm Knapsack Problem*.
En este caso, en particular, el problema de la mochila es de una sola dimensión: la utilidad que se obtiene al incluir el ítem en consideración. Los autores resuelven esta selección mediante un algoritmo exacto, en particular, programación dinámica.
Siguiendo esta línea de especialización de la revisión, enfocamos la búsqueda en problemas de la mochila, considerando dos aspectos importantes para el objetivo de esta tesis:

- Optimización multiobjetivo: queremos maximizar el uso del horno, al igual que el aglutinante y la cantidad de órdenes de compra de prioridad.
- No incluimos fracciones de ítems (órdenes de compra y discos) en los hornos.

En el libro[16] se estudia específicamente el problema de la mochila. En particular, en el capítulo 9 se estudia el problema de la mochila de más de una dimensión (*MKP* de ahora en más). De nuestra revisión del capítulo destacamos algunos aspectos:

- MKP es un caso especial de programación entera, con la restricción de que todos los coeficientes son positivos y las variables 0 o 1.
- MKP es un problema complejo con sustancial énfasis en el uso de metaheurísticas.
- Se menciona un repaso de literatura notable presentado por Chu y Beasly[17].

2.3. Enfoques basados en problemas de mochila multidimensional

Continuamos entonces la revisión bibliográfica con el trabajo de Chu y Beasly en particular, con foco en el problema de la mochila multidimensional entero[7].
Si bien en esta revisión no se estudia una problemática en particular, sino la esencia del problema y posibles soluciones, resulta de importancia su lectura para identificar dos aspectos fundamentales:

[4] **SMOABC/D** por sus siglas en inglés.
[5] *Knapsack problem* en inglés.
[6] *Bin packing problem* en inglés
[7] **01MKP**, por sus siglas en inglés *0/1 Multidimensional Knapsack Problem*.

- Si bien hay muchos estudios en el uso de algoritmos exactos (siendo *Branch and Bound* y *Dynamic Programming* los más representativos), al tratarse de un problema NP hard, el uso de heurísticas y metaheurísticas pareciera ser más adecuado. En particular, los autores comparan tiempos de ejecución de su algoritmo genético con el software comercial *CPLEX* para instancias de problemas chica ($n <= 100$), quedando en evidencia la inviabilidad computacional de una solución exacta para instancias grandes.
- En la utilización de metaheurísticas, surge la pregunta "qué opción es la más adecuada para este tipo de modelos?". En dicha revisión no solo se repasan heurísticas tales como *búsqueda Tabú*, *métodos golosos*, *heurísticas basadas en cotas*, *recocido simulado* y *algoritmos genéticos*, entre otros, sino que los resultados del algoritmo genético creado por los autores es comparado con dichas metaheurísticas, lo cual provee una base para entender que heurísticas ofrecen buenos resultados para problemas de tipo MKP.

De este repaso en particular, los autores concluyen que su algoritmo genético tiene un mayor tiempo de ejecución que los restantes, pero que dicho tiempo de ejecución no es sustancial y representa un buen *tradeoff*, considerando que la calidad de soluciones de su implementación es mejor que la de los otros algoritmos con menor tiempo de ejecución.
A partir de este último repaso, el objetivo es ahora enfocar aún más el proceso de revisión, en soluciones a problemas del tipo 0/1MKP, con especial énfasis en la comparación de soluciones heurísticas en cuanto a tiempo de ejecución y calidad de los resultados obtenidos. En este sentido, encontramos un repaso más moderno sobre la problemática planteada [18], donde además de explicar algunas variaciones del problema (no es de interés en este punto) y de ejemplificar casos reales donde se encuentran este tipo de problemáticas, finalmente se explican algunos enfoques de heurísticas (goloso, relajación de restricciones), metaheurísticas (búsqueda tabú, búsqueda de vecindario variable, recocido simulado) y un apartado especial dentro de las metaheurísticas: las soluciones basadas en poblaciones, como ser algoritmos genéticos, colonia de hormigas y enjambre de partículas. En este sentido, las heurísticas más populares en los trabajos revisados son las golosas, y en cuanto a metaheurísticas, son los algoritmos genéticos[8]. Comparando ambos (contando las metaheurísticas basadas en poblaciones dentro del conjunto genérico de metaheurísticas), el resultado es una relación de 20 % y 80 %, respectivamente. O sea, los autores encontraron que las metaheuristicas son considerablemente más populares que las heurísticas, midiendo popularidad según cantidad de trabajos publicados.
Desafortunadamente, este repaso solo contempla y compara la popularidad de los enfoques mencionados, no así los resultados obtenidos en sí mismos para cada enfoque. Al no encontrar un repaso que dé respuesta a esa comparativa específicamente, como parte del alcance de esta tesis, creamos nuestro propio repaso de resultados comparativos entre distintos enfoques para la resolución de 01MKP.

2.3.1. Comparativa de resultados de distintos enfoques para resolver 0/1MKP

Considerando que este no es el objetivo principal de la tesis, sino generar comparativas suficientes para decidir bajo qué modelo de metaheurística enfocar el problema de

[8] En realidad, el porcentaje mayor para metaheurísticas es *Other Metaheuristics*, que refiere a combinaciones de metaheurísticas y enfoques ad hoc.

fabricación antes expuesto, este repaso no es exhaustivo.
Previo a explicar el método utilizado para hacer la comparativa, es de vital importancia mencionar que una comparativa no sería posible si las instancias de prueba sobre las que se publicaron resultados no fueran las mismas. Considerando que los resultados pertenecen a distintos trabajos escritos por distintas personas, la única forma en la cual los resultados sean comparables es que dichos artículos hayan usado algún tipo de biblioteca de instancias de referencia. En nuestro caso, las soluciones dentro del alcance de la comparativa usan una biblioteca de instancias de referencia [19].
Ahora sí, el protocolo de revisión y comparativa es el siguiente:

- tomamos las referencias a soluciones específicas del último repaso analizado [18].
- identificamos las instancias de referencia sobre las que se obtuvieron los resultados.
- considerando que para instancias de $n > 100$ no se conocen soluciones exactas, identificamos la *mejor solución conocida* como referencia.
- de los resultados reportados, colectamos el tiempo de ejecución y la calidad de la solución.

Es importante destacar que al no haber un estándar de reporte de resultados de calidad de soluciones, de todas las formas de reportar resultados (algunos lo hacen según porcentaje de brecha respecto a la mejor solución [20], otros reportan el valor absoluto de la mejor solución [21]), establecemos una relación de orden, a efectos de encontrar la solución con mejor *tradeoff* entre calidad de resultado y tiempo de ejecución. No listamos todas las soluciones relevantes, sino las mejores de cada categoría y las mejores categorías en cuanto a calidad de solución.
En primer lugar, los mejores resultados reportados que se usan como base comparativa son los reportados por Chu y Beasly [17], que corresponde a una metaheurística basada en *algoritmos genéticos*. Además de proveer los mejores resultados encontrados, también publican su biblioteca de instancias de pruebas, a efectos de que otros autores puedan usarlo para comparar bajo el mismo conjunto de pruebas.
A continuación, listamos los trabajos que se incluyeron en esta revisión comparativa no exhaustiva.

Categoría de solución (nombre en inglés)	**Calidad de solución (orden)**	**Tiempo de convergencia**	**Referencia**
Genetic algorithm	Brecha[9] menor al 1 %[10]	Bajo (máximo de 1900 segundos)	Chu y Beasley [17], 1998
Tabu-Search	Cercano al mejor conocido, incluso superándose para instancias grandes	Pobre (máximo de 33 horas)	Vasquez & Vimont [21], 2005
ACO Hybrid (Nested Partition + BAS + LP)	Obtiene (y supera en ocasiones) la mejor solución conocida	Comparable al del benchmark	Al-Shihabi y Ólafsson [22], 2010
Parallel ACO	Encuentra óptimo mayoría de las veces	Versión paralela reporta 1/3 de tiempo de ejecución comparado con GA[11]	Leguizamon y Michalewicz [23], 1999
Relaxation Guided VNS	0.5 % brecha	Restringido a 500 segundos	Puchinger J. y Raidl G [20], 2008
Genetic Algorithm (sexual selection)	Brecha ∼1 %	Promedio entre 300 y 1400 segundos	VARNAMKHASTI y LEE [24], 2012

Tab. 2.1: Comparativa de resultados

2.4. Conclusiones de la revisión bibliográfica

2.4.1. Alcance de decisión

A la hora de tomar una decisión para la creación del plan del horno, solo consideramos un horno y el conjunto de órdenes de compra al momento. Futuros hornos disponibles, u otros hornos disponibles al momento de la creación del plan, o futuras órdenes de compra que puedan llegar, no son parte del alcance de trabajo de la tesis.

2.4.2. Tiempo de preparación

Distintos tipos de órdenes relacionados con productos podrían requerir distinto tipo de preparación del horno, en cuyo caso deberíamos tener en cuenta en el proceso de selección

[9] Refiere a la diferencia de calidad de solución.

[10] Con relación a la solución óptima calculada por CPLEX. Para instancias con n>100, la ejecución es interrumpida al llegar a 42MB de memoria o 1800 segundos de ejecución.

[11] El algoritmo genético donde se compara tiempo de ejecución es una implementación propia, sin optimización. No corresponde a resultados de Chu y Beasley [17].

a efectos de minimizar el tiempo total del proceso. Siendo que esta característica de la problemática no aplica a nuestro tema en cuestión, no la incluimos como parte de nuestra tesis.

2.4.3. Tipo de modelo

Después de evaluar distintos problemas relacionados, concluimos que la selección de órdenes de compra a ser parte del plan del horno se modela como un problema 0/1MKP, representando el horno el concepto de la mochila, y el aglutinante total, el volumen total acumulado y la cantidad de órdenes de prioridad las dimensiones de la mochila. Además, considerando que no se pueden fraccionar las órdenes de compra por horno, el modelo es binario (0/1), en el sentido que dada una cierta orden de compra, o bien es parte del plan de horneado en su totalidad o no lo es.
Dicho problema de la mochila multidimensional cuenta además con la restricción de cómo se ubican los discos dentro del horno (*compatibilidad de discos*), que no nos permite usar una implementación estándar de MKP.

2.4.4. Tipo de solución

Luego de haber revisado cuáles son los distintos enfoques estudiados en la literatura para resolver este tipo de problemáticas, y haber hecho una comparativa informal sobre los resultados reportados en cada uno de ellos, podemos decidir cuál consideramos es el mejor enfoque para el caso de estudio de nuestra tesis en particular.
Considerando que la solución de Chu y Beasly:

- provee la mejor calidad[12].
- tiene un tiempo de ejecución bajo.
- se considera en la literatura el estándar de referencia, del cual se consideran las mejores soluciones conocidas (para aquellas instancias sin solución exacta).

concluimos que una solución basada en algoritmos genéticos resulta la más apropiada para resolver el problema de esta tesis basado en 0/1MKP.

[12] Excepto en algunos casos particulares ($n = 500$), donde Vasques y Vimont ofrecen una solución ligeramente superior, a un coste de ejecución de alrededor de 33 horas.

3. MODELO DE LA SOLUCIÓN

Habiendo revisado la literatura y decidido que enfoque de metaheurística usar para implementar la solución, en este capítulo el foco es enunciar un modelo que especifique:

- parámetros de entrada.
  - parámetros de la solución.
  - parámetros de la orden de compra.
  - parámetros del horno.
  - parámetros del disco y separador.
- variables de decisión.
- función a optimizar (función objetivo).
- restricciones.

3.1. Definición del modelo

3.1.1. Parámetros de la solución

- Cantidad de órdenes de compra.
- Máximo número de discos por orden de compra.

3.1.2. Parámetros de las órdenes de trabajo

- El atributo de ser prioritaria o no.

3.1.3. Parámetros de un horno

- El máximo aglutinante permitido para el horno.
- La profundidad de cada ranura.
- La cantidad de filas y columnas.

3.1.4. Parámetros de un disco y separadores

- El número de separadores para el grosor de un disco dado, perteneciente a una orden de trabajo.
- El grosor de un separador.
- La densidad del aglutinante utilizado.
- El porcentaje de aglutinante por disco.
- El diámetro de disco y separador.

3.1.5. Variables de decisión

- La decisión de incluir una orden de trabajo.
- La decisión de dónde incluir cada disco de la orden de trabajo en el mencionado horno.

3.1.6. Función objetivo

Maximizar:

$$f_{objetivo}(solucion) = \alpha\theta + \beta\theta' + \delta\theta'' - \epsilon\theta''' + \gamma\theta'''' \tag{1}$$

Donde:

- α es la normalización del total de aglutinante en *solucion*, con respecto al máximo permitido en el horno. Si, por ejemplo, el máximo aglutinante permitido en el horno es de 6 kg, y la sumatoria de aglutinante de los discos que se consideran ubicar en el horno es 5.4 kg, entonces la normalización resulta ser 0.9.
- β es la normalización del volumen total de las órdenes de trabajo en *solucion*, con respecto al máximo disponible en el horno.
- δ es la normalización del número de órdenes de compra de prioridad en *solucion*, con respecto al total de órdenes de compra de prioridad disponibles.
- ϵ es la normalización de las órdenes de compra prioritaria no presentes en *solucion*.
- siendo n la cantidad de órdenes de compra en consideración, definimos γ como la normalización de la cantidad de órdenes de compra en *solucion*, con respecto a n.
- $\theta, \theta', \theta'', \theta''', \theta''''$ son los coeficientes de peso para el aglutinante, volumen, prioridad, penalidad de no incluir una orden de prioridad y rendimiento.

Desde un punto de vista intuitivo, la ecuación (1) se explica cómo:

- Primer, segundo y tercer término: Beneficio total de incluir las órdenes de compra seleccionadas, según su importancia definida con los pesos asignados al total de aglutinante, volumen y cantidad de órdenes incluidas. Se pretende maximizar esta combinación de sumas.
- Cuarto término: Penalización total por no incluir órdenes de trabajo prioritarias en la solución considerada. Se pretende minimizar esta penalización.
- Quinto término: El número total de órdenes de trabajo incluidas en la solución beneficia positivamente el resultado de la función objetivo. Si bien su influencia en el resultado total es menor que sus contrapartes (aglutinamiento, volumen y prioridad), a igualdad de condiciones se pretende elegir la solución que mayor cantidad de órdenes incluya.

3.1.7. Restricciones

1. El total de aglutinante de las órdenes de trabajo incluidas no debe exceder el límite de aglutinante permitido en el horno.
2. El volumen total de las órdenes de trabajo incluidas (incluyendo sus correspondientes separadores) no debe exceder el límite de volumen del horno.
3. La profundidad total del horno no debe ser excedida al sumar el número de discos en un nivel y columna, junto con los separadores.
4. Si una orden de trabajo se incluye, todos los discos de dicha orden deben ser cargados.
5. No se pueden colocar discos de diferentes grosores en el mismo nivel y columna del horno.

3.2. Algoritmos de la solución

En esta sección explicamos de forma intuitiva los algoritmos que se usaron para implementar el modelo antes mencionado, junto a un pseudocódigo que contribuya a la explicación.
A pesar de que realizamos un esfuerzo considerable en la implementación del código para generar una biblioteca de instancias de pruebas y un testador automático, no explicaremos en detalle dichos algoritmos en esta sección, sino que daremos una idea de su implementación en la sección de resultados, ya que nuestro objetivo no es la forma de generación de las instancias, sino su aprovechamiento para la realización de experimentos computacionales.

3.2.1. Esquema general de la solución

Si bien la problemática a resolver se basa en un modelo de metaheurísticas enfocado en el problema de la mochila multidimensional, eso representa sólo una parte del total de la solución. Dicho de otra forma, identificamos dos problemáticas a resolver

- Decidir qué órdenes de compra se cargarán en el horno. Este es el principal problema a resolver, y en el cual se implementa un algoritmo genético para explorar posibles soluciones.
- Habiendo decidido cuáles son las órdenes de compra que serán parte de la siguiente carga de un horno, se necesita determinar cómo se ubicarán los discos de dichas órdenes dentro del horno, respetando las restricciones arriba mencionadas.

En un principio buscamos una solución que pudiera dar respuesta a ambos subproblemas simultáneamente, a saber, buscar las órdenes a cargar en el horno y su respectiva ubicación dentro del mismo. La complejidad inherente producto de mezclar el *qué* (qué órdenes de compra) y el *cómo* (cómo se ubican esas órdenes de compra) nos llevó a elegir un enfoque alternativo. Fue así que cambiamos a un esquema basado en los subproblemas arriba mencionados.
El segundo subproblema no es parte del alcance de la tesis, puesto que no representa un problema de optimización, ya que para el conjunto de órdenes de compra seleccionado sabemos que existe al menos un plan de horneado válido y que es lo más cercano al óptimo que pudimos encontrar según nuestro modelo de priorización.

3.2.2. Decidiendo qué órdenes de compra incluir

Como se dijo antes, implementaremos un algoritmo genético, inspirado en la solución de Chu y Beasley[17]. Como primer aspecto, y en el mismo sentido que el mencionado paper, una solución la representamos mediante un arreglo de índices de órdenes de compra.

$$\text{órdenes de compra} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & ... & n \end{bmatrix}$$
$$\text{j-ésima solución } \mathbf{s_j} = \begin{bmatrix} 7 & 9 & 1 & 4 & 0 & ... & 2 \end{bmatrix}$$

En primer lugar, a diferencia del trabajo de referencia, el conjunto de soluciones no es una representación binaria, sino que cada celda del arreglo de soluciones contiene un índice. Dicho índice hace referencia a la clave de un diccionario donde se encuentran todas las órdenes de compra, junto a sus atributos. La razón por la cual decidimos implementarlo de esta forma es para facilitar el cálculo de la función objetivo, ya que cada índice del vector distinto a 0 no solo nos dice que hay una orden de compra seleccionada, sino también especifica cuál es esa orden de compra exactamente, para luego buscar sus atributos y hacer los cálculos pertinentes para corroborar factibilidad en cuanto a las restricciones, que se explica más adelante.
Si bien podríamos haber codificado el id de la orden de compra usando el índice del vector de solución, eso fuerza una rigidez del vector de solución que complicaría el algoritmo de entrecruzamiento. Expandiendo el vector binario a un vector de ids de órdenes de compra, logramos un desacoplamiento que brinda mayor flexibilidad en la implementación.
Es importante destacar que el invariante de representación de la población para la solución s_{j_i} de la generación i establece que todas las soluciones consideradas por el algoritmo son válidas[1]:

$$\neg\exists j \in [1, num\_soluciones] \wedge s_{j_i} \text{ es inválida}$$

Considerando esta representación de una solución, describimos la función objetivo antes descrita con el siguiente pseudocódigo:

[1] Entiéndase por solución válida aquella que cumple con todas las restricciones enumeradas en la sección 3.1.7, y por inválida, una solución que no cumple con al menos una restricción.

Algorithm 1 Función objetivo

Entrada: Arreglo s de tamaño n
Entrada: Diccionario $work\_orders$ de tipo <id, <aglutinante_total, volumen_total, prioridad_total, discos>>
Entrada: Horno h
Entrada: $peso\_aglutinante$
Entrada: $peso\_volumen$
Entrada: $peso\_prioridad$
Entrada: $peso\_prioridad\_descartada$
Entrada: $peso\_cantidad\_ordenes$
Salida:: Función objetivo de s

$aglutinante_{solucion} \leftarrow \sum_{s[j]>0} work\_orders[s[j]].aglutinante\_total$
$volumen_{solucion} \leftarrow \sum_{s[j]>0} work\_orders[s[j]].volumen\_total$
$prioridad_{total} \leftarrow \sum_{j} work\_orders[j].prioridad$
$prioridad_{solucion} \leftarrow \sum_{s[j]>0} work\_orders[s[j]].prioridad$
$ordenes_{solucion} \leftarrow \sum_{s[j]>0} 1$
$prioridad\_no\_incluidas \leftarrow n - prioridad_{solucion}$
$aglutinante\_normalizado \leftarrow aglutinante_{solucion}/aglutinante(h)$
$volumen\_normalizado \leftarrow volumen_{solucion}/volumen(h)$
$cantidad\_ordenes\_normalizado \leftarrow ordenes_{solucion}/n$
$prioridad\_normalizado \leftarrow prioridad_{solucion}/prioridad_{total}$
$prioridad\_descartadas\_normalizado \leftarrow prioridad\_no\_incluidas/prioridad_{total}$
if *aglutinante* o *volumen* excedido **then**
    return 0 // *Una forma de penalizar soluciones invalidas*
end if
$fitness \leftarrow (aglutinante\_normalizado \times peso\_aglutinante +$
$volumen\_normalizado \times peso\_volumen +$
$prioridad\_normalizado \times peso\_prioridad -$
$prioridad\_descartadas\_normalizado \times peso\_prioridad\_descartada +$
$cantidad\_ordenes\_normalizado \times peso\_cantidad\_ordenes)$
return $fitness$

Como puede observarse, la multidimensionalidad del problema se reduce a una sola variable $fitness$, mediante una ecuación lineal que combina el peso de cada atributo en un único valor. Dicho peso es determinado por expertos de dominio según su escalafón de importancia en cuanto a qué tipo de soluciones son las que el algoritmo debería tratar de favorecer en cuanto a su ubicación en el horno.
El objetivo de normalizar cada dimensión del problema, y por consiguiente, la función objetivo, es dar al lector una rápida idea intuitiva de qué tan bueno es el resultado, puesto que siempre se encuentra entre 0 y 1.

3.2.3. Población inicial

La creación de la población inicial sobre la que se irán produciendo nuevas generaciones influye considerablemente en la diversidad de futuras generaciones y la calidad de las mismas. Por este motivo, elegimos una variante de algoritmos golosos aleatorios, para generar el total de la cantidad de soluciones por población. Dicha variante predetermina un máximo sobre el cual buscar la mejor opción en cada paso.
A continuación se detallan los dos métodos para generar las soluciones iniciales.

3.2.3.1. Algoritmo goloso por cada dimensión

Si bien no generamos soluciones producto del uso de este algoritmo, lo incluimos como parte de la explicación, ya que es un componente vital del algoritmo goloso restringido, que es el que finalmente usamos para generar las soluciones.
Ahora bien, yendo particularmente a este algoritmo, por cada una de las dimensiones del problema (aglutinamiento, volumen y prioridad), implementamos un algoritmo goloso sencillo y rápido, para generar soluciones que elijan órdenes de compra a incluir en el horno que seleccionen en cada iteración la orden que provee el mejor valor de la componente sobre la que se basa el algoritmo.

Algorithm 2 *goloso_aleatorio*: Algoritmo goloso_aleatorio

Entrada: n tamaño de la solución
Entrada: Diccionario *work_orders* de tipo <id, <id, aglutinante_total, volumen_total, prioridad_total, discos>>
Entrada: Enumeración *componente* para $\{aglutinante, volumen, prioridad\}$
Entrada: *max_comp* es el número normalizado máximo que *componente* puede tomar
Salida:: Arreglo s de tamaño n con los ids de las órdenes incluidas

$s \leftarrow$ arreglo de tamaño n inicializado a 0
discos_incluidos $\leftarrow$ arreglo de tamaño *cantidad_espesores* inicializado a 0
$i = 0$
ordenes_ordenadas $\leftarrow$ ordenar *work_orders* (descendente) según *componente*
while $i < n$ y se respeten restricciones de aglutinante y volumen **do**
  $i' \leftarrow$ número entre (i,$i + tam\_mejores\_soluciones\_a\_elegir$) sin exceder n
  orden_actual $\leftarrow$ *ordenes_ordenadas*$[i']$
  if (*orden_actual.discos* es compatible con *discos_incluidos*) y (suma *componente* para *discos_incluidos* y *orden_actual.componente*) $<$ *max_comp* **then**
    $s[i] \leftarrow$ *orden_actual.id*
    agregar *orden_actual.discos* a *discos_incluidos*
  end if
  $i \leftarrow i + 1$
end while
return s

Como mencionamos previamente, una solución se considera como tal siempre y cuando cumpla con todas las restricciones. Este algoritmo refuerza ese sentido en dos aspectos.
En primer lugar, cuando detecta que no hay más lugar en alguna componente restrictiva (aglutinamiento o volumen), termina la iteración y devuelve las órdenes seleccionadas hasta el momento. Podría ocurrir que en el instante k, la mejor orden disponible en ese momento provea más aglutinante (en caso de estar buscando una solución golosa por aglutinante) del que hay disponible en el horno, en cuyo caso ignora dicha orden y procede con la siguiente, con la esperanza de encontrar una cuyo aglutinante no exceda lo que queda disponible.
En segundo lugar, aún cuando hubiere suficiente espacio en el horno para la orden preseleccionada, podría ocurrir que no haya una ubicación compatible entre los discos de las órdenes ya incluidas en la solución en construcción y los discos de la orden en consideración. Si este fuera el caso, ignora la orden y procede con la siguiente.

3.2.3.2. Algoritmo goloso aleatorio y restringido

La mayor diferencia entre un algoritmo goloso regular y aleatorio es que el segundo no necesariamente elige siempre la mejor opción, sino que elige una de las mejores k opciones (siendo k un parámetro del algoritmo). Esta variante remueve determinismo para una misma entrada provista en dos ejecuciones distintas, lo cual aporta diversidad, que es una cualidad deseada en la población de la metaheurística.
La variante implementada que se ha denominado *restringido* refiere al aporte de aleatoriedad en cuanto al máximo valor al que se quiere llegar. Si, por ejemplo, se buscará un algoritmo goloso aleatorio y restringido para un horno con un volumen de 1000 cm^3, con

un valor de *restricción* = 0.8, significa que el algoritmo goloso aleatorio intentará ocupar un volumen de 800 cm^3, aun cuando haya disponibilidad en el horno para otros 200 cm^2. Si *restricción = 1*, el algoritmo se comporta como un goloso aleatorio regular.
El objetivo de este segundo nivel de aleatoriedad es agregar un factor más de diversidad a las soluciones que se vayan generando.
Detallamos a continuación la forma de uso de esta variante de algoritmo goloso aleatorio para la generación de las restantes soluciones de la población inicial.

Algorithm 3 *generar_soluciones_golosas*: generación de soluciones golosas aleatorias restringidas

Entrada: Diccionario *work_orders* de tipo <id, <id, aglutinante_total, volumen_total, prioridad_total, discos>>
Entrada: m número de soluciones a generar
Entrada: n tamaño de cada solución
Salida:: c conjunto de tamaño m con las soluciones generadas

dimension ← elegir aleatoriamente *aglutinante*, *volumen* ó *prioridad*
restriccion ← elegir aleatoriamente entre 0 y 1
$c \leftarrow \emptyset$
for $i = 1$ to m **do**
  $c \leftarrow c \cup goloso\_aleatorio\_restringido(n, work\_orders, dimension, restriccion)$
end for
return c

Un aspecto importante a destacar en cuanto al rango de *restricción*[2] es el mínimo valor que puede tomar (cercano a 0). El objetivo de darle al algoritmo la posibilidad de restringirlo casi completamente es darle lugar a que esa solución (si se elige como padre) pueda combinarse con otras sin incurrir en el incumplimiento de algunas de las restricciones.
La capacidad del algoritmo de generar soluciones que puedan combinarse con otras, da lugar a que, por entrecruzamiento y a través de las distintas generaciones, se pueda llegar a una mejor solución que aquellas de las que se partió. Si, en cambio, cada solución inicial se generase mediante un algoritmo goloso regular (aleatorio o no), las posibilidades de entrecruzamiento con otras soluciones (también generadas en forma golosa habitual) sería menor, reduciendo la posibilidad de generar mejores soluciones por vías de entrecruzamiento.

3.2.4. Regeneración de la población

Hay dos aspectos importantes a decidir en materia de regenerar la población o parte de esta. En primer lugar, hay que decidir si se quiere regenerar o no, mediante algún tipo de operador, que puede ser estático o dinámico, en donde el primero no está sujeto a variabilidad de progreso de la población, sino que la regeneración toma lugar en intervalos regulares. Un operador dinámico, en cambio, monitorea el progreso de la población y solo toma lugar al observar que alguna condición se cumple. Por ejemplo, no hay progreso de calidad de solución en n generaciones.

[2] Experimentación alrededor del rango de restricción para el algoritmo goloso restringido no es parte del alcance de la tesis y lo listamos como trabajo a futuro en la sección 6.3

Inicialmente, implementamos una estrategia donde el operador de decidir cuando regenerar se basaba en monitoreo de calidad de soluciones de la población y era dinámico, en el sentido de que se activaba cuando no se veía un progreso en n (configurable) número de generaciones, comparando mejor solución actual con mejor solución histórica. En cuanto al operador de elección de candidatos de la población a regenerar, en un intento de preservar la evolución lograda hasta la generación actual, implementamos una regeneración que preserva las mejores x soluciones de la población actual (no histórica) y regenera las restantes. El motivo de mantener las mejores soluciones históricas fuera del proceso de selección para el operador de regeneración es el de fomentar exploración.
Luego de correr las primeras pruebas del algoritmo observamos que la calidad de soluciones por entrecruzamiento de las generaciones posteriores decaía en cada nueva generación, dejando entrever que, o bien los nuevos miembros de la población no eran compatibles con los miembros preservados, o que el algoritmo de entrecruzamiento no eran eficaz.
Otro aspecto que observamos durante las pruebas es que, una vez corregido y mejorado el algoritmo de entrecruzamiento, las subsiguientes generaciones evidenciaban leves mejorías de calidad de soluciones, pero en la mayoría de las ocasiones, dicho progreso era interrumpido por una nueva regeneración tomando lugar. Esto se daba porque el operador de decisión comparaba la mejor solución actual con mejor solución histórica, independientemente de si la población actual mantenía una tendencia de mejora. Por tal motivo, introdujimos un segundo contador al operador de decisión, para comparar no solo contra la mejor solución histórica, sino también contra las x mejores soluciones anteriores. El pseudocódigo del esquema general de la búsqueda de soluciones (explicado en detalle en la sección *Algoritmo general*) con énfasis en la estrategia de regeneración lo detallamos a continuación.

Algorithm 4 Estrategia de regeneración de la población

Entrada: n tamaño población
Entrada: x_1 máximo número de generaciones sin mejoría con respecto al mejor historico
Entrada: x_2 máximo número de generaciones sin mejoría con respecto al mejor reciente
Entrada: y número de mejores soluciones recientes a conservar
Salida:: s_{mejor} mejor solución encontrada

$poblacion \leftarrow$ generar población inicial de tamaño n
$mejor_{historico}$ = obtener mejor solución de $poblacion$
$i_{historico} \leftarrow 0$ // cantidad generaciones sin mejora historica
$i_{reciente} \leftarrow 0$ // cantidad generaciones sin mejora reciente
$mejor_{reciente} \leftarrow \emptyset$
while no se llegó al máximo permitido de generaciones **do**
    $mejor_{actual}$ = obtener mejor solución de $poblacion$
    if $mejor_{actual}$ es mejor que $mejor_{historico}$ **then**
        $mejor_{historico} = mejor_{actual}$
        $i_{historico} \leftarrow 0$
    else
        $i_{historico} \leftarrow historico + 1$
        if $mejor_{actual}$ es mejor que todos los recientes **then**
            incluir $mejor_{actual}$ en $mejor_{reciente}$
            if tamaño $mejor_{reciente} > y$ **then**
                remover peor solución de $mejor_{reciente}$
            end if
            $i_{reciente} \leftarrow 0$
        end if
    end if
    if $i_{historico} > x_1$ o $i_{reciente} > x_2$ **then**
        $poblacion \leftarrow regenerar\_poblacion()$
    end if
    $seleccion\_poblacion \leftarrow seleccion\_padres()$
    $poblacion \leftarrow entrecruzar(seleccion\_poblacion)$
end while
return $mejor_{historico}$

A continuación, se encuentra el pseudocódigo para el operador de regeneración de soluciones per se, donde, como mencionamos anteriormente, preserva un conjunto de soluciones y regenera las restantes. Al igual que en algoritmos anteriores, para eliminar determinismo en la toma de decisiones, introdujimos un elemento aleatorio al operador, mediante la definición de dos variables: $min_{regeneracion}$ y $max_{regeneracion}$, representando respectivamente los límites de un rango cerrado sobre el cual elegir un número aleatorio que determina el número de las mejores soluciones que se preservan, regenerando las restantes.
Por último, cabe destacar que las nuevas soluciones se crean usando el algoritmo goloso restringido mencionado anteriormente.

Algorithm 5 *regenerar_poblacion*: operador de regeneración de la población

Entrada: *poblacion* arreglo de tamaño n
Entrada: *min* minimo de soluciones a regenerar
Entrada: *max* máximo de soluciones a regenerar
Entrada: *poblacion* población con soluciones regeneradas

$i \leftarrow$ número aleatorio entre *min* y *max*
$nuevas_{soluciones} \leftarrow generar\_soluciones\_golosas(n, i)$
asignar $nuevas_{soluciones}$ a *poblacion* entre índices i y n
return *poblacion*

3.2.5. Selección de padres

Durante la fase de generación de soluciones para la siguiente generación, hay que elegir cuál es la población sobre la que se elegirán los padres de las soluciones hijas. Una consideración importante en esta decisión es la de mantener un balance saludable entre *exploration* (diversidad) y *exploitation* (exploración alrededor de instancias de calidad), lo cual se traduce en la siguiente pregunta: ¿cuántas soluciones de las mejores históricas hasta el momento debemos considerar y cuántas de la población actual?
Para esto definimos un nuevo parámetro del algoritmo llamado *corte_seleccion*, que determina cuántas soluciones de las mejores históricas se consideran en el proceso de selección de padres (el resto se toman de la población actual)[3].

3.2.6. Mutación y entrecruzamiento

Considerando que la representación de una solución es una variante de la representación *binary string* usado por los autores en el trabajo de referencia [17], el concepto de mutación que implementamos se asemeja al de *bit-flip mutation*, ya que el flip de un bit (orden de compra) consiste únicamente en silenciar la orden de compra (asignar valor 0 en vez del id) en el arreglo de la solución siendo generada. Notar que de esta forma el efecto de una mutación aplicada a una solución es siempre destructivo, puesto que remueve órdenes y nunca agrega. La responsabilidad de agregar radica en la lógica de entrecruzamiento. Una de las razones de ser de esta estrategia es que remover órdenes de compra es trivial, sin embargo, agregar una orden de compra producto de una mutación que sea distinta a la que se encuentre en los padres y que respete las restricciones (recordar que solo se consideran

[3] El análisis de distintos resultados para este parámetro en particular es también considerado en el apartado de trabajo a futuro, en la seccion 6.3

Algorithm 6 *seleccion_padres*: efectúa una selección de soluciones a combinar

Entrada: *poblacion* arreglo de tamaño $tam_{poblacion}$
Entrada: $mejores_{historico}$ arreglo de tamaño $tam_{poblacion}$
Entrada: *corte_seleccion* valor decimal entre 0 y 1
Salida:: *seleccion* arreglo de tamaño $tam_{poblacion}$

$punto_{corte} \leftarrow tam_{poblacion}$. *corte_seleccion*
seleccion ← arreglo de tamaño $tam_{poblacion}$
$subarreglo_{historicos}$ ← subarreglo de $mejores_{historico}$ entre índices 1 y $punto_{corte}$
subarreglo de *seleccion* entre 1 y $punto_{corte} \leftarrow subarreglo_{historicos}$
$subarreglo_{poblacion}$ ← subarreglo de *poblacion* entre índices 1 y $tam_{poblacion} - punto_{corte}$
subarreglo de *seleccion* entre $punto_{corte}$ y $tam_{poblacion} \leftarrow subarreglo_{poblacion}$
return *poblacion_seleccionada*

soluciones válidas en el proceso de exploración) es considerablemente más complejo, pues involucra no solo buscar que orden disponible se puede asignar en cuando a aglutinante y volumen, sino también en cuanto a compatibilidad de discos de cada orden de compra. Con relación al entrecruzamiento, implementamos tres estrategias, explicadas con mayor detalle en las siguientes subsecciones.

3.2.6.1. Single-point crossover

Es la más sencilla de implementar. Se toma un punto fijo arbitrario y se forma la solución hija con las órdenes de compra a la izquierda de un padre y las de la derecha del otro padre.

Algorithm 7 Single-point crossover

Entrada: $padre1$ arreglo de órdenes de tamaño n
Entrada: $padre2$ arreglo de órdenes de tamaño n
Entrada: $umbral_{entrecruzamiento}$ valor decimal
Salida:: $hijo$ arreglo de órdenes de tamaño n

mezclar $padre1$
mezclar $padre2$
if $umbral_{entrecruzamiento}$ < número aleatorio generado **then**
  return $padre1$
end if
$punto_{separacion}$ ← número aleatorio entre 1 y n
$subarreglo_{padre1}$ ← subarreglo de $padre1$ entre 1 y $punto_{separacion}$
$subarreglo_{padre2}$ ← subarreglo de $padre2$ entre $punto_{separacion}$ y n
$hijo$ ← concatenar $subarreglo_{padre1}$ y $subarreglo_{padre2}$
return $hijo$

Cabe destacar que para incrementar diversidad y compatibilidad de los padres, dando como resultado una solución hija factible que no se deseche, es necesario mezclar los elementos de los respectivos padres. El motivo está relacionado con la forma en la que creamos las soluciones iniciales o regeneradas: el algoritmo *goloso aleatorio restringido*.

Dicho algoritmo agrega órdenes de compra en el arreglo hasta que ya no puede hacerlo más, produciendo normalmente arreglos degenerados a izquierda, o sea, arreglos con valores distintos a 0 mayormente en los índices más bajos, dejando 0 en todo el resto.
Con este tipo de distribución de valores en cada padre, y si no se hiciera una mezcla, el algoritmo *single-point crossover* eligiría habitualmente mayoría de soluciones de $padre1$ y algunas pocas (o ninguna) de $padre2$, afectando seriamente la diversidad de soluciones.

3.2.6.2. Bit-selection crossover

Este es el mismo enfoque que el implementado en el trabajo de referencia [17], pero con una ligera variante. La solución hija se crea eligiendo aleatoriamente la orden de compra de un padre o del otro. En ocasiones no se elige ninguno y se decide silenciar (poner a 0) el gen (la orden de compra). Este último atributo representa un mecanismo de mutación embebido en la lógica de entrecruzamiento, por lo cual cuando se usa esta versión de entrecruzamiento, se desactiva la lógica de mutación antes descrita.
Los resultados obtenidos que se analizan en el apartado de experimentos computacionales se obtuvieron usando esta implementación de entrecruzamiento y mutación.

Algorithm 8 Bit-selection crossover

Entrada: $padre1$ arreglo de órdenes de tamaño n
Entrada: $padre2$ arreglo de órdenes de tamaño n
Entrada: $prob_{mutacion}$ número decimal
Salida:: $hijo$ arreglo de órdenes de tamaño n

$hijo \leftarrow$ arreglo de tamaño n
for i en [1,n] **do**
  $distribucion_{prob}$ = [(1-$prob_{mutacion}$)/2, (1-$prob_{mutacion}$)/2, $prob_{mutacion}$]
  $opcion$ = generar número aleatorio entre 1 y 3 con probabilidades $distribucion_{prob}$
  if $opcion$ es 1 **then**
    $child[i] \leftarrow padre1[i]$
  else
    if $opcion$ es 2 **then**
      $child[i] \leftarrow padre2[i]$
    else
      $child[i] \leftarrow 0$
    end if
  end if
end for
return $hijo$

3.2.6.3. Domain-knowledge crossover

Finalmente, decidimos implementar una tercera variante que sea exclusiva a nuestra problemática de estudio de la tesis. En este caso, en la solución hija se incluyen tantas órdenes de prioridad provenientes de los padres como sea posible, y las disponibles restantes se ocupan seleccionando órdenes no prioritarias de los padres. Esta es una heurística golosa en cuanto al número de órdenes de prioridad que, a diferencia de la heurística golosa por

prioridad explicada anteriormente (ver *Algoritmo 2*), con el correr de las generaciones debería generar mejores soluciones, puesto que continúa iterando y combinando alternativas de solución.

Algorithm 9 Domain-knowledge crossover

Entrada: $padre1$ arreglo de órdenes de tamaño n
Entrada: $padre2$ arreglo de órdenes de tamaño n
Salida:: $hijo$ arreglo de órdenes de tamaño n

$ordenes_{prioridad} \leftarrow \emptyset$
$ordenes_{no\_prioridad} \leftarrow \emptyset$
for i en [1,n] **do**
  if $padre1[i]$ es orden de prioridad **then**
    agregar $padre1[i]$ a $ordenes_{prioridad}$
  else
    agregar $padre1[i]$ a $ordenes_{no\_prioridad}$
  end if
  if $padre2[i]$ es orden de prioridad **then**
    agregar $padre2[i]$ a $ordenes_{prioridad}$
  else
    agregar $padre2[i]$ a $ordenes_{no\_prioridad}$
  end if
end for
$hijo \leftarrow$ arreglo de tamaño inicializado con 0s
$hijo \leftarrow$ tomar a lo sumo n elementos de $ordenes_{prioridad}$
$hijo \leftarrow$ tomar a lo sumo n - $tamano(hijo)$ elementos de $ordenes_{no\_prioridad}$
return $hijo$

Si bien hicimos algunos experimentos con esta implementación durante la fase de desarrollo, no se generaron resultados de pruebas formales para esta implementación.

3.2.7. Operador de reparación

Durante la exploración de soluciones, sea mediante entrecruzamiento o regeneración de nuevas soluciones, y considerando la forma en la que decidimos modelar una solución (vector de ids), puede darse el caso en el que una solución candidata contenga órdenes de compra duplicadas. Por tal motivo, y considerando que en todo momento las soluciones en consideración tienen que ser factibles en cuanto a sus restricciones, aplicamos un operador de reparación que básicamente ignora soluciones inválidas.
Si bien consideramos en primera instancia un operador de reparación que no descarte una solución, sino que la modifique en algún sentido para hacerla factible, lo cierto es que corregir una solución que sea incompatible en cuanto a asignación de discos es considerablemente complejo y atenta contra la velocidad y simplicidad de generar múltiples soluciones en cada nueva generación de la metaheurística.
Para evitar que el algoritmo continúe indefinidamente, generando y descartando soluciones, implementamos condiciones de parada para tal fin, que se muestran en la sección siguiente, donde explicamos el algoritmo general.

3.2.8. Algoritmo general

Finalmente, y para cerrar con este apartado de la explicación de la solución implementada, detallamos a continuación el esquema del algoritmo general. En parte este algoritmo ya se listó en el *Algoritmo 4*, aunque en aquella sección el foco fue la explicación de la regeneración de la población, motivo por el cual se omiten detalles en referencia a dicha regeneración.

Algorithm 10 Algoritmo general

Entrada: n tamaño población
Entrada: $max_{generaciones}$ valor entero
Salida:: s_{mejor} mejor solución encontrada

$poblacion \leftarrow$ generar población inicial de tamaño n
$mejores_{historico} \leftarrow$ arreglo tamaño n
$mejores_{actual} \leftarrow$ arreglo tamaño n
$padre1_i \leftarrow 0$
$padre1_j \leftarrow 0$
$prox_{generacion} \leftarrow$ arreglo tamaño n
while no se llegó al máximo permitido de generaciones **do**
  $mejores_{actual} \leftarrow$ ordenar $poblacion$ según la función objetivo
  if hay que regenerar $poblacion$ **then** // ver Algoritmo 4
    $poblacion \leftarrow regenerar\_poblacion()$
  end if
  $mejores_{historico} \leftarrow$ combinar $mejores_{historico}$ con $mejores_{actual}$
  $seleccion \leftarrow seleccion\_padres($ combinación $mejores_{historico}$ y $poblacion)$
  while no haya generado la nueva generación y no exceda el límite de intentos **do**
    $hijo_{creado} \leftarrow False$
    while $\neg hijo_{creado} \wedge$ no exceda intentos **do**
      $max_{indice} \leftarrow padre1_i + padre_{size\_elite}$
      $padre1_{indice} \leftarrow$ elegir aleatoriamente entre índices $padre1_i$ y max_{indice}
      $padre2_{indice} \leftarrow$ elegir aleatoriamente entre índices 1 y n
      $hijo \leftarrow entrecruzar(seleccion[padre1_{indice}], seleccion[padre2_{indice}])$
      if $hijo$ es válida **then** // ver *algoritmo 11*
        agregar $hijo$ a $prox_{generacion}$
        $hijo_{creado} \leftarrow True$
      end if
      incrementar número de intentos
    end while
    $padre1_i \leftarrow padre1_i + 1$
  end while
end while
return $mejores_{historico}$

Un punto importante del algoritmo general es el de decidir si una solución es válida. A saber, hay 3 condiciones generales que hay que determinar:

- La solución es válida en cuanto a cantidad de aglutinante y volumen de sus discos (con respectivos separadores). Esto hace referencia a las restricciones 1, 2, 3 y 4 del modelo antes descrito en la sección 3.1.5.
- La solución no contiene órdenes duplicadas.
- La solución es compatible a nivel discos en cuanto a disposición dentro del horno. Refiere a la restricción 5 del mencionado modelo.

Puesto que las soluciones generadas por las variantes de algoritmos golosos explicados en *Algoritmo 2* y *Algoritmo 3* cumplen con las tres condiciones enumeradas, es necesario garantizar que las soluciones producto de un entrecruzamiento también sean compatibles con mencionadas condiciones. Detallamos a continuación con el pseudocódigo para tal fin.

Algorithm 11 Es solución válida

Entrada: *horno*
Entrada: *solucion* arreglo de tamaño n
Entrada: h altura de un separador
Entrada: $disco_{radio}$ decimal representando radio del disco
Entrada: m tamaño de la población
Salida: valor booleano

$z \leftarrow 30$ // cantidad de grosores
$discos_{acumulados} \leftarrow$ arreglo de tamaño z inicializado a 0
$separadores \leftarrow$ arreglo tamaño z inicializado con cantidad de separadores por grosor
$separadores_{altura} = separadores \times h$
$grosores \leftarrow$ arreglo tamaño z con valor los respectivos índices
$no_{repetidos} \leftarrow$ asegurarse que *solucion* no tenga repetidos
$aglutinante_{valido} \leftarrow \sum$ aglutinante en *solucion* $<$ límite aglutinante de *horno*
$j \leftarrow 0$
$es_{compatible} \leftarrow True$
while $j < m \wedge es_{compatible}$ **do**
    // todas las operaciones son entre vectores
    $discos_{candidato} \leftarrow solucion[j]$.discos
    $discos_{acumulados} \leftarrow discos_{acumulados} + discos_{candidato}$
    $discos_{altura} \leftarrow discos_{acumulados} \times (grosores + separadores_{altura})$
    $discos_{volumen} \leftarrow \pi \times radio^2_{disco} \times discos_{altura}$
    $cant_{ranuras\_por\_grosor} \leftarrow discos_{volumen}$ / volumen por ranura de *horno*
    $es_{compatible} \leftarrow es_{compatible}$ & $(\sum cant_{ranuras\_por\_grosor}) <$ cantidad ranuras del *horno*
    $j \leftarrow j + 1$
end while
return $no_{repetidos} \wedge aglutinante_{valido} \wedge es_{compatible}$

Por otro lado, cabe destacar la elección de $padre1_{indice}$ y $padre2_{indice}$. El objetivo es generar la población de la siguiente generación mediante el entrecruzamiento de buenas soluciones seleccionadas (mediante una combinación de mejores soluciones históricas y actuales) y soluciones compatibles con la generación actual. Para eso, determinamos una iteración semiestructurada, donde $padre1_{indice}$ representa la i_{esima} mejor solución considerada, y $padre2_{indice}$ la j_{esima} solución compatible considerada.

Por otro lado, para promover diversidad, la elección de $padre1_{indice}$ no es determinística, sino que tiene una componente aleatoria, donde en realidad elegimos el padre dentro de un conjunto de mejores soluciones de tamaño $padre_{size\_elite}$.
Considerando que puede haber soluciones incompatibles para un $padre1_{indice}$ dado, creamos el ciclo interno, que tiene por objetivo (sin incurrir en un ciclo infinito) encontrar un par ($padre1_{indice}$, $padre2_{indice}$) cuyo entrecruzamiento produzca una solución hija válida que pueda ser incluída en la siguiente generación.
Finalmente, con relación a complejidad temporal, podemos observar que en el peor caso, los tres ciclos de la solución general están acotados por constantes, y dentro de cada ciclo se podría tener que regenerar la población (orden lineal para cantidad de órdenes de compra), sumado a elegir padres (también lineal) y dentro del ciclo más interno, corroborar que la solución es válida (complejidad lineal respecto a la cantidad órdenes y discos) y hacer el entrecruzamiento (orden lineal respecto a cantidad de órdenes), lo cual nos lleva a que el orden de complejidad del algoritmo pertenece a

$$O(n^2 m)$$

con n la cantidad de órdenes de compra, m la cantidad máxima de discos por orden de compra.

4. EXPERIMENTOS COMPUTACIONALES

En este capítulo compartimos los experimentos que hicimos para el algoritmo genético (el cual es el objetivo de la tesis) y los comparamos contra implementaciones golosas, con el objetivo de tener una base de comparativa de resultados. Para eso creamos un generador de instancias que no solo nos permite replicar escenarios con características similares a las observadas en el entorno real del cual surgió esta problemática, sino también escenarios hipotéticos, con el objetivo de evaluar los resultados de los algoritmos en estudio ante dichas condiciones de estrés. Siendo que la implementación del generador de la biblioteca de instancias no es el objeto de estudio de nuestra tesis, no ahondamos en detalles de implementación, sin embargo, presentamos una descripción detallada de las instancias de prueba por cada experimento, así como el motivo por el cual las incluimos en el conjunto de experimentación.

4.1. Parámetros de los algoritmos

Contamos con dos tipos de parametrización: por un lado, la de los algoritmos de la solución y la instancia del horno que se usa para hacer los experimentos. Por otro lado, los parámetros relacionados con la generación de las instancias de prueba para las órdenes de compra específicas. En esta sección enumeramos la primera categoría.

4.1.1. Parametrización del algoritmo genético

Nuestra implementación del algoritmo está sujeta a un número considerable de parámetros de entrada. Cabe destacar que las variables aquí explicadas están ancladas al valor que se especifica (producto de una breve experimentación manual durante la fase de desarrollo y valores de referencia de la literatura) y no es parte del alcance de la tesis explorar una combinación ideal de los mismos. Por otro lado, no es parte del alcance de la tesis expresar cada parámetro aquí enunciado en función de los parámetros de entrada de la soluciónón (sección 3.1.1).

- Tamaño de la población: 50.

- Número de generaciones: 2000.
  Esta es la condición de parada del algoritmo, o sea, devuelve la mejor solución encontrada después de haber generado y evaluado soluciones en 2000 generaciones poblacionales.

- Probabilidad de entrecruzamiento: 0.8.
  Esto refiere a que, dados dos padres de una generación, cuál es la probabilidad de producir un hijo producto de su entrecruzamiento. Si no hay entrecruzamiento, el hijo toma el valor del padre, o sea, uno de los padres pasa de generación en generación.

- Probabilidad de mutación: 0.1.
  Este valor representa la probabilidad de que un hijo sufra los efectos de una mutación.

- Tamaño de elección para el algoritmo goloso aleatorio: $tamano_{poblacion} \times 0.2$.
  En la implementación de algoritmo goloso aleatorio para la generación de soluciones iniciales, este valor representa el conjunto de mejores soluciones sobre el cual tomar la decisión golosa.

- Mínimo umbral limitante para goloso restringido: 0.1.
  Representa el rango aleatorio sobre el cual tomar el valor que determinará la restricción impuesta al algoritmo goloso restringido.

- Máximo umbral limitante para goloso restringido: 1.
  Idem anterior, pero representando el rango superior.

- Máxima cantidad de generaciones sin mejoras: 200.
  Representa la cantidad máxima permitida de generaciones sin lograr una mejora de calidad de solución (respecto a la función objetivo) con respecto a la mejor solución encontrada hasta el momento. Pasado este umbral, se regenera un porcentaje de la población.

- Máxima cantidad de generaciones sin mejoras recientes: $tamano_{poblacion}$.
  Esta variable representa el mismo concepto que la anterior, pero comparando la mejor solución de la población actual con respecto a la mejor solución de la población anterior, a diferencia de la anterior que comparaba con la mejor solución histórica.

- Mínimo y máximo valor de rango para la regeneración de soluciones: [0.3, 1].
  Cuando decidimos regenerar la población después de una serie de generaciones sin mejoras, se escoge un número aleatorio en este rango, que determina el número de soluciones de la población actual que será reemplazado por nuevas soluciones regeneradas.

- Cantidad de mejores históricos en selección: 0.1.
  Determina cuántas de las mejores soluciones históricas se consideran como padre en el proceso de selección.

- Cantidad de mejores padres seleccionados: 0.2.
  En el proceso de selección, para quitar determinismo en la selección de padres, se utiliza este parámetro para elegir el próximo mejor padre dentro de un subconjunto de mejores padres (sin reposición). En concreto, este parámetro nos dice que el siguiente mejor padre a ser seleccionado se elige al azar dentro del 20 % de los mejores padres.

- Límite de intentos de generar una generación: $tamano^2_{poblacion}$.
  Este valor es simplemente una condición de parada límite para evitar que el algoritmo siga ejecutándose indefinidamente en el caso extremo de que no haya forma de combinar padres de la generación actual para generar soluciones hijas válidas. No se observó ninguna ocurrencia para el conjunto de instancias generado.

4.1.2. Parametrización de la función objetivo

En esta sección detallamos el peso asignado a cada variable que es parte de la función objetivo.

- Aglutinante: 0.3.
- Volumen: 0.1.
- Prioridad 0.6.
- Penalidad de no incluir una orden de prioridad: 0.1.
- Rendimiento: 0.05.

Como podemos observar, a la hora de evaluar y determinar cómo se priorizan las diferentes dimensiones del problema, le damos mayor importancia a la **prioridad** de las órdenes de compra.
Cabe destacar que esta asignación de pesos a las variables no surge producto de un proceso de exploración, sino que intenta replicar la experiencia y la recomendación de los expertos de negocio en el ámbito real de donde surgió este problema. Por ejemplo, una recomendación fue la de no priorizar una asignación cercana al óptimo del aglutinante, puesto que cualquiera que sea la estrategia de carga del horno, de seguro se llegue al límite de aglutinante. Observamos esto mismo en los resultados de la sección 4.3.

4.1.3. Instanciación de variables de generación de instancias

Si bien la implementación del algoritmo genético es independiente del conjunto de instancias con el que se lo ejecute, para nuestra experimentación creamos instancias específicas que detallamos a continuación.
En primer lugar, se utilizan dos tipos de hornos: *grande* y *chico*, cada uno con las siguientes características:

- Chico
  - Profundidad: 580 mm.
  - Columnas: 3.
  - Niveles (filas, si se lo piensa como una matriz): 5.
  - Máximo aglutinante permitido: 3.1 kg.
- Grande
  - Profundidad: 740 mm.
  - Columnas: 4.
  - Niveles: 6.
  - Máximo aglutinante permitido: 6.15 kg.

El objetivo de tener dos hornos es evaluar cómo se comportan los algoritmos en condiciones donde los recursos (aglutinante, volumen) están muy limitados (horno chico), y un horno donde los recursos son considerablemente más abundantes (horno grande). Ejecutando los algoritmos con las mismas instancias de prueba en los dos tipos de hornos, nos permite evaluar y comparar la capacidad de administración de recursos de las soluciones, así como su calidad.
Por otro lado, se toman algunos valores estándares como base para la generación de nuestras instancias:

- diámetro de los discos: 98 mm.
- se toma como valor de densidad del aglutinante de 5.6 g/cm^3 por disco, que corresponde a aglutinantes como *Polyvinyl Alcohol (PVA)* o *Polyethylene Glycol (PEG)*.
- se toma un porcentaje de aglutinante por disco equivalente al 4 %, que corresponde a un valor intermedio para un rango estándar en la industria de entre 3 y 5 %.
- se toma un espesor 3 mm para los separadores.
- se toma una cantidad de separadores de disco por grosor correspondiente al 10 % del grosor de disco, redondeado hacia arriba. Por ejemplo, un disco de 30mm de espesor necesita 3 separadores, de 3 mm cada uno.
- los posibles espesores por disco que consideramos son (expresados en mm): {10, 12, 14, 16, 18, 20, 22, 25, 30}.

4.2. Parametrización de instancias generadas

A continuación detallamos cómo parametrizamos y generamos las instancias de pruebas para los algoritmos. El objetivo es generar instancias y agruparlas en categorías, sobre las cuales hacer un recorrido exploratorio que nos permita identificar las categorías en las cuales los algoritmos ofrecen mejores o peores resultados, y extraer conclusiones sobre los patrones observados.

4.2.1. Generación de discos

Para los discos por cada orden de compra, los parametrizamos tanto en lo referente al grosor de los discos como a su cantidad.
Para la cantidad de discos, definimos dos variables min_{discos} y max_{discos} y tomamos un valor aleatorio en ese rango.
En cuanto al grosor de los mismos, utilizamos un generador de números aleatorios en el rango de los grosores permitidos (enumerados en la sección anterior) con distribución normal parametrizada en *esperanza* y *desvio_estandar*. La cantidad generada de estos números aleatorios equivale a la cantidad de discos definida en el párrafo anterior.

4.2.2. Generación de órdenes de compra

La cantidad de órdenes de compra generadas es también parametrizable mediante la variable $cant_{ordenes}$.
En cuanto al atributo de prioridad de las órdenes de compra, utilizamos una variable $prob_{prioridad}$ que define la probabilidad que la orden sea de alta prioridad. De esta forma, cuando generamos la orden, se crea como orden de prioridad con una probabilidad $prob_{prioridad}$. Por ejemplo, si generamos instancias con $cant_{ordenes} = 100$ y $prob_{prioridad} = 0.1$, se espera que haya alrededor de 10 órdenes de compra de prioridad.

4.2.3. Repeticicidad de instancias por categoría

Finalmente, es importante destacar que, considerando los aspectos aleatorios antes descritos en la generación de las instancias, consideramos una variable extra llamada

repeticicidad_categoria, la cual determina cuántas instancias en particular son generadas por cada categoría de instancia, entendiéndose por categoría al resultado de fijar las variables de generación antes descritas.
Por ejemplo, tomamos $repeticicidad\_categoria = 20$ y la categoría determinada por:

- $cant_{ordenes} = 100$.
- $min_{discos} = 5$.
- $max_{discos} = 30$.
- $esperanza = 18$.
- $desvio\_estandar = 1$.
- $prob_{prioridad} = 0.2$.

El algoritmo de generación de instancias se ejecuta 20 veces con dichas variables de entrada. El objetivo de la repetición por categoría es evitar generar instancias dentro de los rangos permitidos que pudiera exhibir características particulares pero no comunes, que puedan llevar a inferir conclusiones erróneas. Mediante la repetición de la generación de las mismas, obtenemos un muestreo que reduce la probabilidad de generar un *outlier*.

4.3. Método de experimentación

Habiendo definido los parámetros para la generación de instancias de prueba, pasamos a explicar ahora cómo se utilizaron para obtener los resultados que compartimos en la siguiente sección.
A efectos de generar una diversidad de categorías de instancias para estudiar los algoritmos y los resultados que estos producen bajo diferentes circunstancias, definimos un método de prueba de la siguiente forma:

- Fijamos las variables correspondientes al algoritmo genético, la función objetivo, el horno y el aglutinante (con los valores mencionados en las secciones 4.0.1, 4.0.2 y 4.0.3).
- Para las variables de generación de instancias (sección 4.1 y subsecciones), fijamos valores para un subconjunto ($repeticicidad\_categoria$, min_{discos}, $esperanza$ y $desvio_{estandar}$) y definimos rangos de valores para el resto. En nuestro caso, esta es la configuración elegida:
  - $horno = \{chico, grande\}$.
  - $cant_{ordenes} = \{50, 100\}$.
  - $repeticicidad\_categoria = 20$.
  - $min_{discos} = 5$.
  - $max_{discos} = \{20, 50\}$.
  - $esperanza = 16$.
  - $desvio\_estandar = 1$.
  - $prob_{prioridad} = \{0.1, 0.5, 0.9\}$.

- Generamos instancias parametrizando variables de entrada según el producto cartesiano de las variables mencionadas, con sus respectivos posibles valores.
- Por cada categoría de instancia, calculamos el promedio de las métricas de interés.

Siguiendo este protocolo, creamos, ejecutamos y evaluamos $2 \times 2 \times 20 \times 2 \times 3 = 480$ instancias, correspondientes a 12 categorías, 20 instancias por categoría y 2 hornos para ubicar las órdenes[1].
Sería ideal generar un mayor número de categorías, pero considerando que en promedio una ejecución de del algoritmo genético para 50 órdenes de compra toma 30 segundos, y 1 minuto para 100 órdenes, el tiempo total de ejecución para el producto cartesiano de categorías rápidamente escala y se cuenta por miles de horas.

4.3.1. Entorno de ejecución

Escribimos el código de los algoritmos en Python, versión 3.11.9 y lo ejecutamos en una laptop MacBook Pro, con chip M3 Max (de 14 núcleos CPU[2], de los cuales 6 núcleos son de alta velocidad con frecuencia 4.06 GHz y los restantes con velocidad de 2.8 GHz) y memoria de 36 GB.

4.4. Resultados de las pruebas

Como primer enfoque, sería ideal tener una visión global producto de analizar la diferencia[3] de resultados obtenidos entre nuestro algoritmo genético GA y los algoritmos golosos de comparación, para cada una de las categorías de instancias antes descritas y por cada dimensión del problema[4]. Dicha tabla tendría un encabezado del estilo (*Categoría de instancia, Horno, Métrica, GA, Brecha goloso por aglutinante, Brecha goloso por volumen, Brecha goloso por prioridad*). Sin embargo, no es posible incluir dicha tabla por limitaciones de espacio[5], de modo que mostraremos fragmentos de interés del mismo.

4.4.1. Resultados comparativos de alto nivel

La primera vista que evaluamos filtra aquellos resultados donde la brecha del algoritmo goloso comparado con nuestro algoritmo genético es mayor al 10 %[6], y resultados en donde los algoritmos golosos exhiben una brecha negativa (es decir, su solución fue mejor que la de nuestro GA). Dichas columnas, que llamamos *Tasa positiva* y *Tasa negativa*, las normalizamos dividiendo por el total de filas correspondientes: $cantidad\_filas\_filtradas/cantidad\_filas$.

[1] A efectos prácticos, contamos el tipo de horno como elemento de una categoría, de modo que el total de categorías es 24. El total de instancias se mantiene el mismo.
[2] No usamos los núcleos GPU disponibles.
[3] A esta diferencia nos referiremos como la brecha.
[4] A las dimensiones del problema nos referiremos como *métrica*.
[5] El lector puede consultar el archivo *resultados_vista_global.csv* con todos los resultados pertinentes a este experimento en particular.
[6] Una brecha superior al 10 % es equivalente a decir que el resultado para la métrica en cuestión es menor al 90 %, en comparación con el resultado del GA para la misma métrica.

Algoritmo	Métrica	Tasa positiva	Tasa negativa	#TN[7]
greedy_binder_gap	FITNESS	1.0	0.0	0
	BINDER	0.0	0.35	7
	VOLUME	0.8	0.0	0
	PRIORITY	1.0	0.0	0
	THROUGHPUT	1.0	0.0	0
greedy_volume_gap	FITNESS	1.0	0.0	0
	BINDER	0.0	0.35	7
	VOLUME	0.8	0.0	0
	PRIORITY	1.0	0.0	0
	THROUGHPUT	1.0	0.0	0
greedy_priority_gap	FITNESS	0.5	0.05	1
	BINDER	0.0	0.0	0
	VOLUME	0.05	0.0	0
	PRIORITY	0.7	0.05	1
	THROUGHPUT	0.95	0.0	0

El primer elemento de interés es que la tasa positiva para la métrica de la función objetivo (*fitness*) es 1 para los golosos por aglutinante (*greedy_binder_gap*) y volumen (*greedy_volume_gap*), lo cual interpretamos como que en todas las categoría de instancias, estos algoritmos encuentran una solución con un resultado menor al 90 % comparado con los resultados del GA. Para el caso del goloso por prioridad (*greedy_priority_gap*), la tasa positiva respecto a la misma métrica es del 50 %. De estos datos deducimos que la calidad de las soluciones de GA es considerablemente superior a los algoritmos golosos, lo cual es esperado. Es cierto que en esta deducción no hay discriminación por categoría de instancia, ya que no es esperable que la brecha se mantenga estable independientemente de la categoría, pero ese tipo de análisis es parte del análisis más detallado que se presentará en la siguiente sección.

Otro elemento de interés es que la tasa positiva para el aglutinante (en inglés *binder*) es 0 para todos los algoritmos golosos, es decir, no hay ninguna categoría de instancia en donde un algoritmo goloso exhiba resultados con una calidad de solución menor al 90 %. En otras palabras, todos los algoritmos golosos demuestran encontrar planes de horneado que hacen buen uso del aglutinante disponible. Esto se condice con lo expresado en la sección *4.0.2*, en referencia al aglutinante como primer factor limitante en un plan estándar.

Por otro lado, comparando resultados entre los distintos algoritmos golosos, pareciera ser que quien exhibe menor cantidad de instancias con gap mayor al 10 % es el algoritmo goloso por prioridad. Como observamos en el apartado anterior, la tasa positiva de la métrica *FITNESS* para el algoritmo goloso por prioridad representa la mitad que para los otros algoritmos golosos. Dicha comparativa se interpreta como que la mitad de las instancias devuelven una solución con una calidad menor al 90 % comparado con el GA, mientras que en los otros algoritmos golosos, el total de las instancias exhibe esa brecha.

Siguiendo el análisis comparativo entre algoritmos golosos, observamos que la tasa positiva

[7] Representa la cantidad de instancias cuya función objetivo obtuvo un mejor resultado que el GA.

de los algoritmos golosos por aglutinante y volumen para la métrica *VOLUME* es 0.8, mientras que para el algoritmo goloso por prioridad dicha métrica es 0.05. O sea, las soluciones del algoritmo goloso por prioridad con un resultado menor al 90 % respecto al GA para la métrica *VOLUME* son escasas.
En el caso de la tasa negativa, observamos algunas instancias[8] en donde un algoritmo goloso obtuvo una mejor solución que el GA. Notar que esta situación de un algoritmo goloso obteniendo mejores resultados que el GA la podríamos solucionar muy fácilmente si los resultados de los tres golosos se incluirían como parte de la población inicial del GA. En ese caso, el GA los consideraría como mejor solución de la generación 0 y automáticamente los promovería a mejor histórico y no se reemplazarían a no ser que se encuentre una mejor solución. Si bien es un enfoque válido (de hecho así lo hacíamos en las primeras implementaciones), consideramos que no incluir dichas soluciones en la población inicial nos daría mejor visibilidad en cuanto a la capacidad exploratoria del GA, y su comparación con los algoritmos golosos tendría más valor.
En el caso de la tasa negativa del algoritmo goloso por prioridad, tiene sentido que tanto la métrica *FITNESS* como *PRIORITY* sean negativa, puesto que justamente *PRIORITY* es lo que el algoritmo goloso intenta optimizar, y considerando que la dimensión *prioridad* es la que tiene mayor peso en nuestra función objetivo (ver sección 4.0.2), tiene sentido que la función objetivo del algoritmo goloso por prioridad sea mayor que la del GA.
Ahora bien, con una primera vista de alto nivel de los datos y las comparaciones de resultados, podemos enfocarnos en cuantificar las diferencias observadas.

4.4.2. Resultados cuantificando brechas

Expandiendo los resultados de la tabla anterior, con foco en cuantificar las brechas observadas de los algoritmos golosos comparados con el GA, mostramos a continuación otra tabla producto del mismo conjunto de datos, pero con énfasis en cuantificar las brechas, calculando el promedio por categoría.

[8] Notar que la cuantificación de *Tasa negativa* se hace sobre la cantidad de instancias, no de categorías de instancias.

Algoritmo	Métrica	Cuantificación TP	Cuantificación TN[9]
greedy_binder_gap	FITNESS	0.57	-
	BINDER	-	1.0016
	VOLUME	0.88	-
	PRIORITY	0.26	-
	THROUGHPUT	0.47	-
greedy_volume_gap	FITNESS	0.58	-
	BINDER	-	1.0016
	VOLUME	0.88	-
	PRIORITY	0.26	-
	THROUGHPUT	0.47	-
greedy_priority_gap	FITNESS	0.86	1.0009
	BINDER	-	-
	VOLUME	0.89	-
	PRIORITY	0.75	1.0071
	THROUGHPUT	0.77	-

Cuantificar las brechas entre los algoritmos golosos y el GA nos permite dimensionar qué tan bueno o malo son los resultados en cuanto a la calidad de soluciones. En primer lugar, podemos observar que los resultados del GA son considerablemente mejores que los golosos en cuanto a la función objetivo. Para esa métrica en particular, existe una única instancia de prueba donde el GA no devuelve el mejor resultado, y es en la comparación con el algoritmo goloso por prioridad, como habíamos observado en la tabla anterior. En esta tabla, al cuantificar la diferencia, observamos que dicha diferencia entre ambos es inferior al 0.1 %.
Incluso comparando todas las instancias y métricas donde un algoritmo goloso haya obtenido una solución de mejor calidad que el GA, se puede observar que el promedio de dicha mejoría en ningún caso supera el 0.7 %. Esto nos dice que el GA obtiene soluciones considerablemente mejores que los algoritmos golosos, y que ocasionalmente encuentra soluciones ligeramente peores que las heurísticas golosas, con una diferencia promedio de calidad de solución imperceptible. Necesitaríamos hacer un análisis más detallado, considerando máximos de valores de las diferencias, para concluir que tan mejores son las soluciones de un goloso en esas instancias en particular. No incluimos dicho análisis en el alcance de la tesis.

4.4.3. Resultados de brecha para la función objetivo por categoría de instancia

A continuación, observamos un gráfico donde nos enfocamos en la función objetivo y la brecha con el algoritmo genético, discriminado por categoría de instancia. Del total de categorías de la ejecución, se muestran sólo aquellas donde la brecha entre el algoritmo goloso por prioridad y el algoritmo genético fue menor o igual al 90 %, que corresponde al

[9] Utilizamos "−"para denotar la ausencia de instancias que cumplan esa condición.

40 % del total de categorías para la métrica de función objetivo[10].

[10] Si consideramos el tipo de horno un elemento más de la categoría de instancia, entonces vemos 10 categorías de instancias, de un total de 24 categorías.

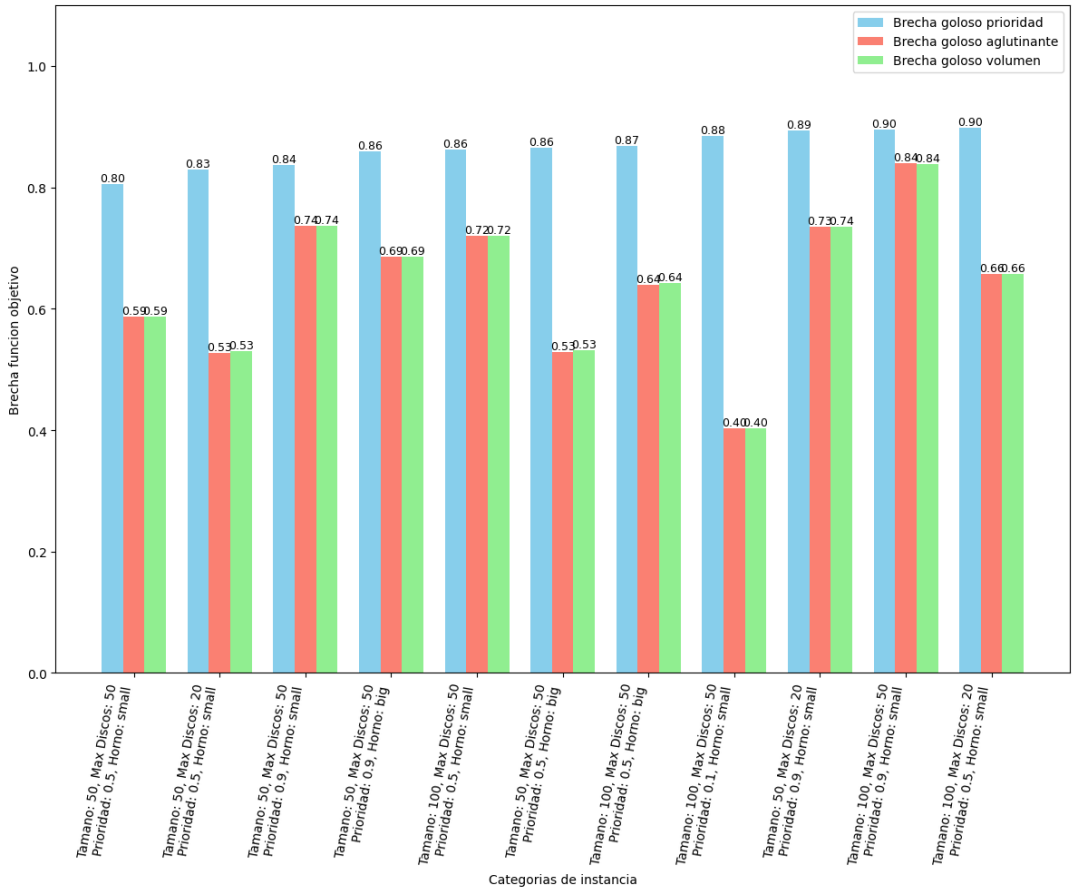

Fig. 4.1: Categorías de instancia con peor función objetivo.

Podemos observar que el 70 % de las categorías de las instancias incluidas en este filtro por resultados corresponden a hornos pequeños, lo cual podría indicar que en situaciones en donde se necesita hacer una mejor administración de los recursos y probar distintas combinaciones de ubicación de órdenes en el horno (junto a sus discos), los algoritmos golosos sufren considerables penalidades por decisiones tempranas que no pueden revertir, algo que sí puede hacer el algoritmo genético en su constante proceso de combinación de soluciones, por alguno de los métodos nombrados.

4.4.4. Resultados de tiempo de ejecución

Mostramos a continuación un gráfico de tiempo de ejecución por categoría de instancia.

Algoritmo	Cantidad de órdenes	Max discos	Promedio tiempo ejecución[11]
GA	100	20	04:15.521
	100	50	01:54.178
	50	20	01:17.743
	50	50	01:07.806
GREEDY_BINDER	100	20	00:00.002
	100	50	$< 1ms$
	50	20	$< 1ms$
	50	50	$< 1ms$
GREEDY_PRIORITY	100	20	00:00.001
	100	50	$< 1ms$
	50	20	$< 1ms$
	50	50	$< 1ms$
GREEDY_VOLUME	100	20	00:00.001
	100	50	$< 1ms$
	50	20	$< 1ms$
	50	50	$< 1ms$

Observamos una diferencia sustancial en cuanto a los algoritmos golosos y el algoritmo genético. Esto es esperable, puesto que los golosos solo prueban una combinación de órdenes, mientras que el genético prueba tantas combinaciones como sea posible en la cantidad de generaciones que se configuró como máximo número permitido (2000 para este caso).

4.4.5. Diferencia de tiempos entre categorías de instancias

Es llamativo el tiempo de ejecución que observamos para el primer tipo de categoría de instancia del algoritmo genético, que representa el doble que la siguiente categoría. Es importante destacar que, si bien las características de cada categoría dan una idea de la complejidad que la misma pudiera tener, puede no transferirse en la práctica cuando se crean las instancias. La razón de ser de esta posible discrepancia entre la categoría de instancia y la instancia en sí misma creada es que los parámetros de entrada para la creación de instancias se usan como parte de un rango de elección de un número aleatorio, de modo que una combinación de *Cantidad de órdenes*=100 y *Max discos*=20 pudiera generar instancias de mayor complejidad que aquellas cuya configuración es *Cantidad de órdenes*=100 y *Max discos*=50.
Un análisis más detallado de tiempos de ejecución y casos borde de instancias creadas que puedan diferir en complejidad de su configuración están fuera del alcance de esta tesis, puesto que nuestro foco radica en la calidad de las soluciones.

4.4.6. Métricas del algoritmo genético

A continuación, exploramos algunos resultados particulares al algoritmo genético. El gráfico de la figura 4.2 representa el crecimiento de la función objetivo para todas las

[11] Expresado en *minutos*:*segundos*:*milisegundos*.

instancias de ejecución, agrupadas por categoría y tomando el valor medio de cada una. El eje Y está normalizado al valor final que toma la función objetivo. El eje X representa el número de generación, que está fijado a 2000 en la ejecución de las pruebas, pero aquí está recortado, puesto que para la generación 1500, todas las categorías de instancias habían alcanzado lo que sería su máximo valor de función objetivo. La línea horizontal punteada representa el 90 % del valor máximo de la función objetivo, en un intento de capturar visualmente la velocidad de convergencia del algoritmo. Como podemos observar, en la generación 200 todas las categorías convergen a un valor de su función objetivo superior al 90 %, obteniendo el máximo valor en algunas de ellas.

La velocidad de convergencia se da producto de unas pocas mejoras, tal como podemos apreciar en el gráfico, y las primeras mejoras son las que evidencian una magnitud mayor, comparadas con las últimas (se evidencia en la longitud de los incrementos por escalón). Conforme el valor de la función objetivo se acerca al mejor valor obtenido de la instancia (valor 1 en el eje Y), las mejoras se vuelven menos perceptibles.

Considerando que no hay mejoría de solución sustancial después de la generación 1200, cabe preguntarnos si se justifica continuar con la ejecución, ya que todas las categorías de instancia alcanzaron su máximo para ese entonces.

También podríamos usar este resultado como elemento de diseño para una refactorización del algoritmo, donde el criterio de parada incluya una condición extra, referente a la cantidad máxima de generaciones sin evidenciar mejoría de la mejor solución obtenida hasta el momento.

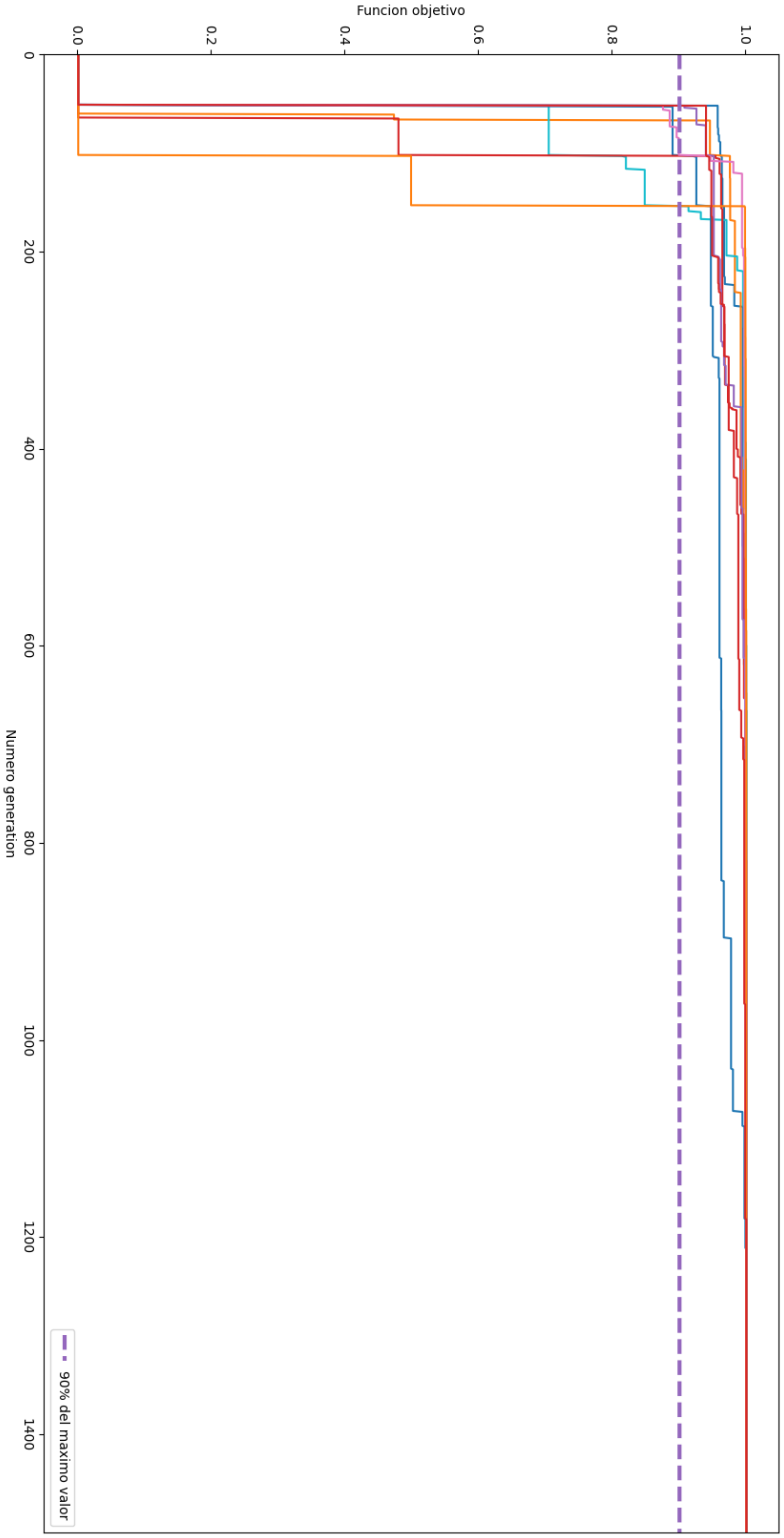

Fig. 4.2: Crecimiento función objetivo

Si bien el gráfico anterior nos da un indicio de la magnitud de las mejoras encontradas por el algoritmo genético, es difícil observar el número total de mejoras por categoría. Tampoco hay una discriminación por tipo de mejora. A saber, una mejora en la calidad de solución se pudo haber dado o bien por un entrecruzamiento, o bien producto de una regeneración (total o parcial) de la población existente. Observamos estos dos aspectos en el siguiente gráfico.

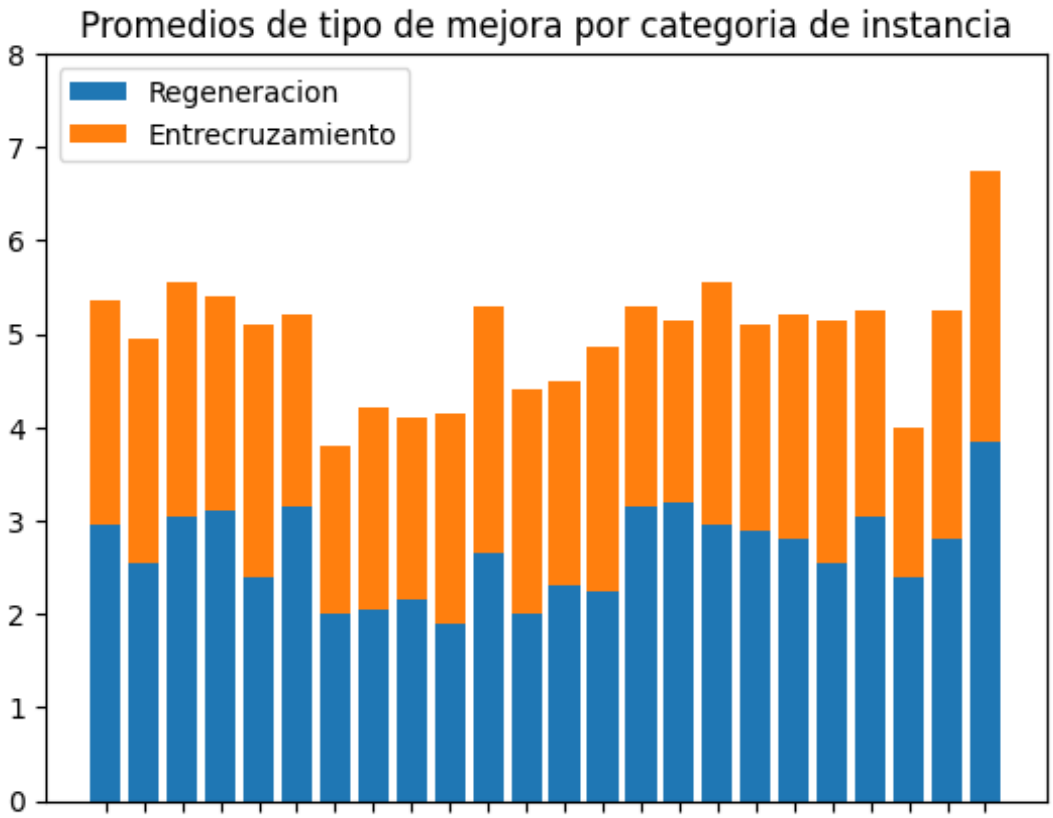

Fig. 4.3: Tipos de mejoras.

Como podemos ver, se evidencia una ligera mayor cantidad promedio de mejoras por regeneración que por entrecruzamiento. Considerando el funcionamiento de los algoritmos explicados anteriormente, una mejora por regeneración se da producto de no haber mejoras por entrecruzamiento durante X cantidad de generaciones, de modo que se podría decir que un algoritmo de entrecruzamiento eficaz que frecuentemente encuentra nuevas mejoras no daría lugar a potenciales mejoras por regeneración. El hecho de que en este gráfico observamos incluso mayores mejoras por regeneración que por entrecruzamiento podría ser un indicio de que el algoritmo de entrecruzamiento usado (bit-selection) no es el más apropiado para este tipo de problema. Quizás un algoritmo que incorpore mayor conocimiento de dominio en la forma que se entrecruzan soluciones (especialmente considerando la restricción de ubicación de discos dentro del horno) podría ofrecer mejores resultados. Más experimentación alrededor de ese aspecto particular pareciera necesaria.

5. CONCLUSIONES

Podemos extraer algunas observaciones a partir de los resultados compartidos. En primer lugar, utilizando datos de complejidad equivalente y superior a las que nos encontramos en el entorno de producción donde surgió este problema, el algoritmo genético encuentra soluciones de calidad superior a heurísticas simples, en un tiempo de ejecución tolerable, de modo que bien podría ser usado en dichas situaciones reales.
Por otro lado, sea cual fuere el algoritmo utilizado, el aglutinante es siempre el factor de mayor limitante, puesto que todos los algoritmos ofrecen soluciones que alcanzan el máximo posible de aglutinante cargado en el horno.
El algoritmo goloso por prioridad es el que mejores soluciones obtiene de los golosos, y en ocasiones dichas soluciones son equiparables al algoritmo genético. Es importante esta aclaración porque se condice con lo observado empíricamente, donde los expertos de dominio siempre recomendaban maximizar la ubicación de las órdenes de prioridad en el horno.
Del análisis de soluciones por categoría de instancia, notamos cierto grupo de soluciones donde el algoritmo goloso por prioridad (el mejor de los golosos) obtiene soluciones de mala calidad, respecto al GA.
Dichas soluciones parecen darse cuando los recursos son escasos y se hace necesario una mejor administración de los mismos, probando distintas combinaciones de órdenes, evitando decisiones tempranas que se penalicen con una baja calidad de la solución.
Según el tiempo de convergencia observado, podemos concluir que el criterio de parada del algoritmo genético puede modificarse. O bien mantener la condición de número de generaciones, pero reduciendo el número máximo, o bien agregando algún aspecto dinámico de corte del estilo *terminar ejecución si no hay mejoría del mejor valor histórico en X número de generaciones*. Sea cual sea la elección, dicho cambio reducirá considerablemente el tiempo de ejecución total del algoritmo.
El algoritmo de entrecruzamiento usado en la experimentación no es tan eficaz como se esperaría, puesto que se encuentran más mejoras por regeneración que por entrecruzamiento, producto de que el algoritmo de entrecruzamiento no produce mejores soluciones por largos periodos de generaciones.
Finalmente, es importante destacar que la solución genética propuesta, junto con el generador de instancias y el simulador de pruebas, mediante su parametrización, podrían utilizarse en conjunto como herramienta de toma de decisiones. Siguiendo el ejemplo dado en la introducción, si se evaluara adquirir un nuevo horno, se podrían usar el generador y el simulador para entender cómo se comportará dicho horno y qué beneficios podría tener adquirirlo.

6. TRABAJO A FUTURO

Identificamos algunos aspectos en donde podríamos expandir el presente trabajo en pos de obtener mejores resultados, ya sea en materia de calidad de soluciones o en materia de tiempo de ejecución. Si bien el tiempo de ejecución actual es aceptable para su uso dentro de la industria, es importante destacar que un menor tiempo de ejecución daría lugar a decisiones más sofisticadas (que requieren mayor tiempo de ejecución) en los algoritmos, durante la exploración de nuevas soluciones, que podría llevar a mejores soluciones en cuanto a calidad de las mismas.
Complementando las dos categorías arriba mencionadas, identificamos una tercera categoría de mejoras, relacionada con la elección de los parámetros de ejecución del algoritmo. Finalmente, consideramos una categoría independiente, no relacionada con la calidad de soluciones obtenidas, pero sí relacionada con la definición de la metaheurística y sus parámetros. Comentamos brevemente a continuación algunas mejoras identificadas por cada categoría.

6.1. Mejoras en la calidad de la solución

- Algoritmo de selección de padres: en vez de usar una selección golosa aleatoria, podríamos incorporar conocimiento de dominio para seleccionar soluciones que sean compatibles entre sí (con relación a restricciones de la ubicación de discos en las ranuras), de modo que su entrecruzamiento produzca no solo soluciones factibles, sino soluciones cuyo plan de ubicación de discos reduzca espacios libres dentro del horno, y por ende mejore la calidad de la solución. Por ejemplo, si combinamos soluciones con órdenes de compra cuyos discos son de los mismos espesores, esto minimiza la fragmentación interna dentro del horno y produce una solución de mejor calidad que otra solución cuyas órdenes de compra poseen una mayor diversidad de grosores de sus discos. Cabe recordar que discos de distinto grosor no se pueden ubicar en la misma ranura dentro del horno, lo cual puede causar fragmentación interna en el mismo, dejando espacios sin usar.

- Algoritmo de entrecruzamiento: relacionado con el ítem anterior, si incorporamos conocimiento de dominio en el entrecruzamiento de dos soluciones, podríamos generar soluciones con mayor compatibilidad de discos.

- Permitir cierto margen de soluciones no factibles: actualmente el algoritmo solo explora soluciones factibles, independientemente de que tan inválidas sean. Si cuantificamos el grado de invalidez y permitiéramos soluciones cuyo margen cuantificado sea menor a α, eso aumentaría la diversidad de soluciones, lo cual podría llevarnos a mejores soluciones.

6.2. Mejoras en cuanto al tiempo de ejecución

- Omitir soluciones repetidas: durante la creación de una nueva generación descartamos soluciones que no sean factibles. Si además descartamos soluciones que ya hayan sido generadas previamente, esto reduciría el tiempo de ejecución, ya que no

revisaríamos soluciones que ya se evaluaron. Es cierto también que, considerando los aspectos aleatorios del algoritmo, podría ocurrir que terminemos combinando la solución de una forma distinta a la que lo hicimos en su primera consideración, generando nuevas soluciones que hubiéramos descartado al ignorar duplicados.

- Criterio de parada: como observamos, podríamos reevaluar el criterio de parada actual y acortarlo sin temor a perder calidad de soluciones.
- Si incrementamos el uso de memoria para almacenar cálculos que luego se puedan reusar, estaríamos mejorando ligeramente el tiempo de ejecución. Dado que no cambiaría el orden de complejidad temporal, esta mejora se puede categorizar como una mejora netamente de implementación. Podemos ver un ejemplo de esta mejora en el *algoritmo 11*, cuando en un intento de validar que la solución por entrecruzamiento sea válida, calculamos el volumen que ocupa la solución candidata, según las órdenes de compra ya consideradas y la nueva que se pretende agregar. Por cada orden de compra, calculamos el volumen total de la solución nuevamente. Si mantuviéramos el volumen calculado, sólo necesitaríamos calcular el volumen de la nueva orden en consideración, reduciendo así el número de cálculos.

6.3. Elección de parámetros

El algoritmo cuenta con 18 parámetros fijos, cuyo análisis de impacto en la calidad de soluciones no estudiamos como parte del alcance de esta tesis. Algunos de ellos son:

- tamaño de población 50.
- tasa de mutación 0.1.
- umbral máximo de no mejoría respecto a mejor histórico 200.
- umbral máximo de no mejoría respecto a mejor histórico 50.
- tasa de entrecruzamiento 0.8.

Podríamos buscar de forma programática la mejor combinación de parámetros con los cuales obtengamos una mejoría en la calidad de soluciones. Considerando una instancia de ejecución por cada elección de parámetros, una búsqueda de mejor combinación de parámetros por fuerza bruta (producto cartesiano) no sería computacionalmente factible[1], de modo que necesitaríamos implementar un algoritmo aproximado.

6.4. Definición de metaheurística

Al comienzo del capítulo 4 (*Experimentos computacionales*) enumeramos todos los parámetros del algoritmo, así como el valor fijo que toman. Observamos una mejora a futuro donde expresamos todos los parámetros mencionados (no solo unos pocos) en función del tamaño de la entrada, puesto que facilita la descripción de la complejidad de las instancias de prueba y su comparación.

[1] Considerando que cada parámetro podría tener al menos 5 opciones de valores y cada ejecución toma 60 segundos en promedio, eso nos lleva a un tiempo de ejecución para una única ejecución de instancia del problema de 60×5^{18} segundos.

Por otro lado, como mencionamos brevemente, debido a las decisiones aleatorias que tomamos en la generación de instancias de prueba por categoría, los parámetros mencionados, si bien arrojan una idea de la complejidad, no determinan la complejidad de una instancia. Sería interesante proveer algún mecanismo o *puntaje* que nos devuelva la complejidad concreta de cada instancia, luego de haberla generado con sus respectivos valores aleatorios. De esa forma, los datos que obtendríamos serían más precisos y declarativos en cuanto a su complejidad inherente.
Relacionado con este último punto, si conociéramos precisamente cuál es la complejidad pertinente a cada instancia y categoría de instancia, podríamos hacer un análisis más exhaustivo de la complejidad de convergencia por categoría y dificultad de instancia.

Bibliografía

[1] Abhishek Apratim, Prashanti Eachempati, Kiran Kumar, Krishnappa Salian, Vijendra Singh, Saurabh Chhabra, and Sanket Shah. Zirconia in dental implantology: A review. *J Int Soc Prev Community Dent*, 5(3):147–156, 2015.

[2] Yasser Alfawaz. Zirconia Crown as Single Unit Tooth Restoration: A Literature Review. *J Contemp Dent Pract*, 17(5):418–422, 2016.

[3] Yung-Chang Cheng, Deng-Huei Lin, Cho-Pei Jiang, and Yuan-Min Lin. Dental implant customization using numerical optimization design and 3-dimensional printing fabrication of zirconia ceramic. *International Journal for Numerical Methods in Biomedical Engineering*, 33(5):e2820, 2017. e2820 CNM-Dec-15-0254.R2.

[4] Erveton Pinheiro Pinto, Robert S. Matos, Marcelo A. Pires, Lucas dos Santos Lima, Ştefan Ţălu, Henrique Duarte da Fonseca Filho, Shikhgasan Ramazanov, Shahram Solaymani, and Claudio Larosa. Nanoscale 3d spatial analysis of zirconia disc surfaces subjected to different laser treatments. *Fractal and Fractional*, 7(2), 2023.

[5] Henriette Lerner, Katalin Nagy, Nicola Pranno, Fernando Zarone, Oleg Admakin, and Francesco Mangano. Trueness and precision of 3d-printed versus milled monolithic zirconia crowns: An in vitro study. *Journal of Dentistry*, 113:103792, 2021.

[6] Irina Grădinaru1, Alice Arina Ciocan Pendefunda, Loredana Liliana Hurjui, Balcoș Carina, Mihaela Mitrea, Oana Adina Armencia, and Magda Ecaterina Antohe. Practical aspects involved in technological algorithm for making fixed protheses using digital methods. *Romanian Journal of Oral Rehabilitation*, 13(3):271–280, 2021.

[7] Maria-Antonela Beldiman, Anca Vițalariu, Cristina Vasilache, Georgiana Macovei, Denisa Polocoșeriu, Liana Aminov, and Nicoleta Ioanid. Zirconia. characteristics and technology for fixed restorations. *Romanian Journal of Oral Rehabilitation*, 13(3):140–149, 2021.

[8] Ehsan Askari, Paulo Flores, and Filipe Silva. A particle swarm-based algorithm for optimization of multi-layered and graded dental ceramics. *Journal of the Mechanical Behavior of Biomedical Materials*, 77:461–469, 2018.

[9] Ján Duplák, Samuel Mikuláško, Darina Dupláková, Maryna Yeromina, and Rastislav Kaščák. Analysis of a Regression Model for Creating Surface Microgeometry after Machining Zirconia YML Used for Dental Application. *Biomimetics (Basel, Switzerland)*, 9(8):473, 2024.

[10] Se-Wook Pyo, Dae-Joon Kim, Jung-Suk Han, and In-Sung Luke Yeo. Ceramic Materials and Technologies Applied to Digital Works in Implant-Supported Restorative Dentistry. *Materials (Basel, Switzerland)*, 13(8):1964, 2020.

[11] Shixin Liu, Jiafu Tang, and Jianhai Song. Order-planning model and algorithm for manufacturing steel sheets. *International Journal of Production Economics*, 100(1):30–43, 2006.

[12] Thomas Sobottka, Felix Kamhuber, and Bernhard Heinzl. Simulation-based multi-criteria optimization of parallel heat treatment furnaces at a casting manufacturer. *Journal of Manufacturing and Materials Processing*, 4(3), 2020.

[13] Adil Baykasoğlu and Fehmi B. Ozsoydan. Dynamic scheduling of parallel heat treatment furnaces: A case study at a manufacturing system. *Journal of Manufacturing Systems*, 46:152–162, 2018.

[14] Hao Zhang, Lianbo Ma, Junyi Wang, and Liang Wang. Furnace-grouping problem modeling and multi-objective optimization for special aluminum. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 6(3):544–555, 2022.

[15] Victor C.B. Camargo, Leandro Mattiolli, and Franklina M.B. Toledo. A knapsack problem as a tool to solve the production planning problem in small foundries. *Computers & Operations Research*, 39(1):86–92, 2012. Special Issue on Knapsack Problems and Applications.

[16] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Multidimensional Knapsack Problems*, pages 235–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[17] P. Chu and J Beasley. A Genetic Algorithm for the Multidimensional Knapsack Problem. *Journal of Heuristics*, 4(3):63–86, 1998.

[18] Naimi M. El Amri H. Laabadi, S. and B. Achchab. The 0/1 Multidimensional Knapsack Problem and Its Variants: A Survey of Practical Models and Heuristic Approaches. *American Journal of Operations Research*, 8:395–439, 2018.

[19] J. E. Beasley. Or-library: Distributing test problems by electronic mail. *The Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[20] Jakob Puchinger and Günther Raidl. Bringing order into the neighborhoods: Relaxation guided variable neighborhood search. *J. Heuristics*, 14:457–472, 10 2008.

[21] Michel Vasquez and Yannick Vimont. Improved results on the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, 165(1):70–81, 2005.

[22] S. Al-Shihabi and S. Ólafsson. A hybrid of nested partition, binary ant system, and linear programming for the multidimensional knapsack problem. *Computers & Operations Research*, 37(2):247–255, 2010.

[23] G. Leguizamon and Z. Michalewicz. A new version of ant system for subset problems. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1459–1464 Vol. 2, 1999.

[24] MOHAMMAD JALALI VARNAMKHASTI and LAI SOON LEE. A genetic algorithm based on sexual selection for the multidimensional 0/1 knapsack problems. *International Journal of Modern Physics: Conference Series*, 09:422–431, 2012.

Esta tesis posee un archivo anexo, disponible en:

http://hdl.handle.net/20.500.12110/seminario_nCOM000819_Raposo_anexo