UNIVERSIDAD AUTÓNOMA DEL ESTADO DE HIDALGO

INSTITUTO DE CIENCIAS BÁSICAS E INGENIERÍA

DOCTORADO EN CIENCIAS EN INGENIERÍA CON ÉNFASIS EN ANÁLISIS Y MODELACIÓN DE SISTEMAS

TESIS DOCTORAL

UNA BÚSQUEDA HÍBRIDA UTILIZANDO ALGORITMOS GENÉTICOS Y ESCALADA DE COLINAS CON REINICIO MÚLTIPLE PARA EL FLEXIBLE JOB SHOP SCHEDULING PROBLEM

Para obtener el grado de
Doctora en Ciencias en Ingeniería con Énfasis en Análisis y Modelación de Sistemas

PRESENTA

M. en C. Nayeli Jazmín Escamilla Serna

Director
Dr. Juan Carlos Seck Tuoh Mora

Pachuca de Soto, Hgo., México , febrero 2024

UNIVERSIDAD AUTÓNOMA DEL ESTADO DE HIDALGO
Instituto de Ciencias Básicas e Ingeniería
School of Engineering and Basic Sciences
Área Académica de Ingeniería y Arquitectura
Department of Engineering and Architecture

Número de control: ICBI-AAIyA/236/2024
Asunto: Autorización de impresión

MTRA. OJUKY DEL ROCÍO ISLAS MALDONADO
DIRECTORA DE ADMINISTRACIÓN ESCOLAR
PRESENTE.

El Comité Tutorial de **Tesis** del programa educativo de posgrado titulado **"UNA BÚSQUEDA HÍBRIDA UTILIZANDO ALGORITMOS GENÉTICOS Y ESCALADA DE COLINAS CON REINICIO MÚLTIPLE PARA EL FLEXIBLE JOB SHOP SCHEDULING PROBLEM"**, realizado por la sustentante **Nayeli Jazmín Escamilla Serna** con **número de cuenta 281355** perteneciente al programa de **Doctorado en Ciencias en Ingeniería, con Énfasis en Análisis y Modelación de Sistemas**; una vez que ha revisado, analizado y evaluado el documento recepcional de acuerdo a lo estipulado en el Artículo 110 del Reglamento de Estudios de Posgrado, tiene a bien extender la presente:

AUTORIZACIÓN DE IMPRESIÓN

Por lo que la sustentante deberá cumplir los requisitos del Reglamento de Estudios de Posgrado y con lo establecido en el proceso de grado vigente.

Atentamente
"Amor, Orden y Progreso"
Pachuca, Hidalgo a 16 de febrero de 2024

El Comité Tutorial

Dr. Juan Carlos Seck Tuoh Mora
Director

Dr. Joselito Medina Marín
Miembro del comité

Dr. Irving Barragán Vite
Miembro del comité

Dr. José Ramón Corona Armenta
Miembro del comité

UNIVERSIDAD AUTÓNOMA DEL ESTADO DE HIDALGO ICBI ÁREA ACADÉMICA INGENIERÍA Y ARQUITECTURA

Ciudad del Conocimiento
Carretera Pachuca-Tulancingo km 4.5 Colonia Carboneras, Mineral de la Reforma, Hidalgo, México. C.P. 42184
Teléfono: +52 (771) 71 720 00 ext. 4000, 4001
Fax 2109
aai_icbi@uaeh.edu.mx

Dedicatoria

Este trabajo de tesis se lo dedico a mi director de tesis, a el Dr. Juan Carlos Seck Tuoh Mora. Agradezco profundamente su valiosa colaboración, dirección, dedicación, paciencia y sobre todo su tiempo invertido para dirigirme en la realización de este proyecto de investigación. No hay palabras suficientes para expresarle mi más sincero agradecimiento.

Agradecimientos

A Dios antes que nada, por permitirme culminar esta etapa de mi vida.

A mi director de tesis, el Dr. Juan Carlos Seck Tuoh Mora por el tiempo dedicado, por su guía y por su dirección académica para la elaboración y culminación de este trabajo de investigación.

A mis asesores al Dr. Joselito Medina Marín, al Dr. Irving Barragan Vite y al Dr. José Ramón Corona Armenta por su valiosa colaboración para el mejoramiento de sintaxis de este trabajo de tesis.

A el Consejo Nacional de Ciencia y Tecnología (Conacyt) por la beca otorgada durante este trabajo Doctoral.

A mi mamá que sin ella y su valioso apoyo no estaría donde estoy actualmente, ya que por su esfuerzo, dedicación y sacrificio soy la persona que soy, y toda la vida le estaré eternamente agradecida y la amaré por siempre.

A la vida y el universo, por permitirme estar en el momento y tiempo indicado, para ingresar a este programa de estudio de posgrado.

La vida no es fácil, para ninguno de nosotros. Pero... ¡qué importa! Hay que perseverar y, sobre todo, tener confianza en uno mismo. Hay que sentirse dotado para realizar alguna cosa y que esa cosa hay que alcanzarla, cueste lo que cueste.

Madame Curie

Resumen

Este estudio se enfoca en el problema de programación de tareas para el Flexible Job Shop Scheduling Problem (FJSSP), que por sus características se asemeja a los sistemas de manufactura actuales con alta flexibilidad, donde una operación puede ser realizada por varias máquinas. Se presenta un novedoso algoritmo híbrido llamado GA-RRHC. Se propone un método de optimización híbrida con un enfoque jerárquico, aplicando Algoritmos Genéticos (GA) como método de búsqueda global, implementando diferentes operadores, de mutación y cruce, aplicando una vecindad inspirada en los autómatas celulares (CA), seleccionando la mejor solución. Se refina la solución con una búsqueda local implementando la escalada de colinas con reinicio (RRHC) donde se explota la información de las celdas inteligentes (smart_cells) para minimizar el makespan, mediante pequeños cambios, perfeccionando la nueva mejor solución. El punto novedoso es la hibridación del algoritmo GA y el RRHC en el FJSSP, conjuntamente con la aplicación de la vecindad tipo CA en un GA para un problema FJSSP ya que hasta el momento no se han aplicado juntos. Los algoritmos propuestos se implementaron en Matlab. Para comprobar la eficiencia del algoritmo se comparó el GA-RRHC con otros métodos propuestos, se prueba tomando los experimentos de 4 bancos de prueba. Se aplicó una prueba estadística utilizando la desviación porcentual (RDP) y la prueba de Friedman como método de validación, obteniendo resultados satisfactorios, demostrando que el GA-RRHC es un método competitivo en comparación de otros algoritmos de la literatura para instancias FJSSP con alta flexibilidad.

Abstract

This study focuses on the task scheduling problem for the Flexible Job Shop Scheduling Problem (FJSSP), which due to its characteristics is similar to current manufacturing systems with high flexibility, where an operation can be carried out by several machines. A novel hybrid algorithm called GA-RRHC is presented. A hybrid optimization method with a hierarchical approach is proposed, applying Genetic Algorithms (GA) as a global search method, implementing different mutation and crossover operators, applying a neighborhood inspired by cellular automata (CA), selecting the best solution. The solution is refined with a local search implementing restart hill climbing (RRHC) where the information from the smart cells (smart_cells) is exploited to minimize the makespan, through small changes, perfecting the new best solution. The novel point is the hybridization of the GA algorithm and the RRHC in the FJSSP, together with the application of the CA type neighborhood in a GA for a FJSSP problem since so far they have not been applied together. The proposed algorithms were implemented in Matlab. To check the efficiency of the algorithm, GA-RRHC was compared with other proposed methods, it is tested by taking the experiments of 4 test benches. A statistical test was applied using the percentage deviation (RDP) and the Friedman test as a validation method, obtaining satisfactory results, demonstrating that GA-RRHC is a competitive method compared to other algorithms in the literature for FJSSP instances with high flexibility.

Contenido

Lista de figuras

Lista de tablas

Introducción y propósito de la investigación

En este trabajo se aborda el tema de programación de tareas con máquinas flexibles, también conocido como Flexible Job Shop Scheduling Problem-(FJSSP). La importancia de la programación radica en que debido a los constantes cambios en un mercado muy volátil, se está incrementado a pasos agigantados la demanda de productos. A raíz de esto se han estado desarrollando nuevos procesos tecnológicos para cubrir las demandas del consumidor.

Con el incremento de la demanda de productos en el mercado, las industrias buscan diversas maneras de expandir sus ventajas competitivas, y una manera es mediante la optimización en su producción [77], donde se tiene que encontrar una mejor solución a sus diversas actividades de programación.

El área denominada Scheduling abarca problemas que tienen en común la necesidad de planificar la ejecución de un conjunto de operaciones o tareas durante un periodo de tiempo. La dificultad de encontrar un programa óptimo va a depender primordialmente del ambiente de producción, las restricciones del proceso y el indicador de desempeño [76].

Para solucionar este tipo de problemas se debe tener en cuenta tanto las restricciones de precedencia como las de los recursos, para dar lugar así a un plan de producción viable. La mayor parte de los estudios en el campo de aplicación de ingeniería para la optimización de problemas de programación han sido estudiados en el ámbito de asignación de tareas o trabajos en el cual son procesados en recursos productivos (máquinas). Debido a esto, la secuenciación de una serie de operaciones y ubicarlas en espacio de tiempo sin violar ninguna de las restricciones impuestas por el problema se le conoce en inglés como Scheduling; este tipo de programación caen en los problemas de optimización combinatoria (OC), para la asignación de recursos limitados a lo largo del tiempo, donde el objetivo es minimizar los tiempos de finalización

de la última tarea, o maximizar el número de productos elaborados en una fecha de entrega acordada, por mencionar algunos.

Para los problemas de secuenciación de operaciones se han desarrollado diversos métodos a partir de técnicas matemáticas tradicionales que están limitadas por el número de elementos de los problemas. Por otro lado, están las técnicas heurísticas, que abordan problemas de gran tamaño en un tiempo computacional adecuado, por lo que estas últimas son las elegidas en este trabajo para la optimización de este problema de programación.

En esta tesis se propone el uso de dos nuevas técnicas heurísticas híbridas, esta propuesta se basa en el uso del enfoque jerárquico donde se hace una combinación de 2 técnicas metaheurísticas, es decir, primero para la búsqueda global se resuelve el subproblema de enrutamiento, y como segundo paso para la búsqueda local se aborda el subproblema de programación de tareas. Como técnicas de búsqueda global se utilizan operadores genéticos. Esta técnica de búsqueda global se utiliza para evitar el estancamiento prematuro de soluciones y evitar que queden atrapadas en un óptimo local. Como herramienta de búsqueda local se aplica la búsqueda metaheurística de escalada de colinas con reinicio múltiple. La aportación de este trabajo es que al utilizar un método híbrido se extraen las fortalezas de cada uno de los algoritmos, para que sea resuelto de manera efectiva y simple el problema del FJSSP con el fin de minimizar el makespan, es decir el tiempo máximo o total en que se tarda en finalizar todas las tareas.

Las técnicas heurísticas híbridas propuestas en este trabajo se implementaron en Matlab, con estos métodos se optimizaron los problemas de Hurik [49], las instancias mt06, mt10 y mt20 de Fisher[34] y las 40 instancias de la01-la40 de Lawrence [55].

Propósito de la investigación

Hoy en día con la creciente demanda en el mercado, los sistemas actuales de producción se enfrentan a constantes problemas para poder solventar esa demanda, y uno de los principales retos a enfrentar es, desarrollar una programación eficiente para la planificación y la gestión de los procesos de fabricación, implementando soluciones en los sistemas flexibles, principalmente en la programación de tareas para poder disminuir los tiempos de producción y así atender esa demanda que es cada vez mayor debido al incremento en la diversidad de productos requeridos por los clientes.

El propósito de este estudio es crear un modelo innovador para abordar un problema de programación de planificación de tareas (scheduling), que, debido a su dificultad de abordaje, y a la naturaleza del problema en si, es clasificado en la categoría de problemas NP-hard, donde no se puede hallar una solución efectiva en un tiempo razonable mediante modelos matemáticos tradicionales. El principal problema para encontrar buenas soluciones, es la dimensión de los problemas, debido a que al existir variaciones en cuanto al número de operaciones, trabajos y máquinas, es necesario tener una flexibilidad de máquinas, donde cada máquina sea capaz de realizar varias tareas, teniendo que establecer el mejor orden de entrada de operaciones para ahorrar tiempos de procesamiento. Este tipo de problemas son conocidos como de flexibilidad. Por lo que esta investigación se centra en el estudio de problemas flexibles como lo es el FJSSP.

Este tipo de problemas que son conocidos como combinatorios, donde su principal objetivo es encontrar buenas soluciones y minimizar cierta función dada. En este trabajo para lograr este objetivo, es necesario un buen algoritmo que pueda encontrar la mejor solución óptima, y minimizar los tiempos de producción. Para resolver esta problemática se ha estudiado desde décadas atrás y desde entonces ha sido un problema abierto, ya que siempre va a haber variaciones en los problemas, debido a la demanda es necesario aumentar el número de operaciones en la producción, por lo que siempre habrá más tareas que programar.

Para resolver los problemas de programación de tareas, debido a su dificultad para abordarlos, la mayoría de los estudios de la literatura han aplicado algún tipo de metaheurística. Las técnicas metaheurísticas utilizan reglas de prioridades clasificadas como procedimientos heurísticos convencionales y los más usados son los que utilizan un método de búsqueda y de población.

De aquí surge la necesidad de seguir innovando en algoritmos metaheurís-

ticos que permitan una planeación de operaciones cercanas al óptimo en un tiempo razonable.

Planteamiento del problema

De acuerdo con [77] el aumento de las industrias ha crecido exponencialmente hoy en día debido a la demanda de productos en el mercado, por lo que la industria manufacturera ha tenido que enfrentarse a numerosos problemas, y su principal interés es buscar la forma de cubrir esa demanda del mercado y expandir sus ventajas competitivas. Para resolver los inconvenientes y lograr sus objetivos es tener una buena planeación y mejorar la optimización de su producción.

Para cubrir la demanda que se enfrenta la industria manufacturera es mediante la reducción de tiempos de procesamiento para entregar la producción a tiempo. Pero al aumentar la producción, la industria se enfrenta a otros problemas tanto de programación como de recursos. Para obtener una buena programación [81] que es la asignación de recursos para realizar un conjunto de tareas en un periodo de tiempo se necesita una programación eficiente, mediante la programación de tareas para reducir alguna función objetivo.

El problema de programación de tareas en un ambiente flexible (FJSSP), es porque se tienen varias máquinas, y trabajos, pero no una buena asignación de tareas, existiendo muchos tiempos muertos, es necesario una buena planeación de la producción para generar soluciones que minimicen el tiempo de procesamiento de las secuencias de operaciones. El problema es determinar cuál es esa secuencia óptima en la que se obtenga una buena programación de operaciones y de asignación de máquinas que minimice el makespan, es decir, el tiempo necesario para completar todos los trabajos.

Objetivos

Los objetivos a cumplir en esta investigación son los siguientes.

Objetivo General

Desarrollar un nuevo método para optimizar el Flexible Job Shop Scheduling conjuntando dos técnicas metaheurísticas híbridas, la primera de ellas se basa en una búsqueda global utilizando los algoritmos genéticos y la segunda, en una búsqueda local usando la escalada de colinas, con el propósito de hallar buenas soluciones que minimicen el makespan en un tiempo aceptable.

Objetivos Específicos

1. Definir el tipo del modelo de programación de tareas para lograr una codificación simplificada y efectiva de utilizar.

2. Diseñar en Matlab el algoritmo que genere el modelo de programación para el FJSSP.

3. Desarrollar un método de optimización integrando los algoritmos genéticos con la escalada de colina, para minimizar el tiempo de ejecución de las tareas programadas.

4. Validar el método a través de pruebas estadísticas comparativas con los resultados obtenidos en investigaciones recientes publicadas sobre el problema.

Justificación

El problema actual que enfrentan las industrias del sector manufacturero es la planificación de la producción, ya que debido al incremento de los procesos industriales para solventar la demanda del mercado se han incrementado el número de operaciones, y ahí es cuando surgen los distintos problemas de fabricación, debido a que es necesario desarrollar una buena planificación en los procesos de producción y para esto, cada industria debe evaluar sus necesidades y plantear soluciones de manera rápida, ya que el tiempo es un factor primordial para la toma de decisiones. Para solucionar los problemas de planificación de tareas es necesario desarrollar una programación eficiente y poder desarrollar un buen algoritmo de optimización que solvente los problemas que se estén presentando.

La programación de la producción es uno de los asuntos más trascendentales en la planificación y programación de los sistemas de manufactura modernos [18], esto implica que la programación de las operaciones sea uno de los aspectos más críticos inmersos en dicha planificación y gestión de los procesos de fabricación [74].

Un sistema de fabricación necesita tener una programación de tareas eficiente y óptima, a lo que se le denominó scheduling [4]. El scheduling es un proceso de toma de decisiones que se usa principalmente en las industrias manufactureras y de servicios y se refiere a la asignación de recursos limitados entre las tareas respecto a periodos de tiempo determinados con el objetivo de optimizar uno o más objetivos [75]. Entre los más ampliamente utilizados son en el proceso de llegada del trabajo, la política de inventario, diversos problemas shop, así como su configuración y sus atributos de trabajo [99], en donde sus objetivos son minimizar el número de tareas, disminuir el tiempo máximo de tareas de inicio a fin, tiempos grises, tiempos de transporte, tiempos de entrega, por mencionar algunas.

La dificultad para encontrar una programación óptima va a depender principalmente del entorno del taller, las limitaciones del proceso y del indicador de rendimiento [76]. Para solucionar este tipo de problemas se debe tener en cuenta tanto las restricciones de precedencia como las de recursos, para dar lugar a un plan de ejecución viable.

La mayor parte de los estudios en el campo de aplicación de la ingeniería para la optimización de problemas de programación [74], [58], [60], [30] han analizado la asignación de tareas o trabajos procesados por recursos productivos (máquinas), para encontrar la secuenciación de una serie de operaciones

y ubicarlas en un espacio de tiempo sin violar ninguna de las restricciones impuestas por el problema abordado.

Alcance y delimitación

Los alcances y delimitaciones de este trabajo de investigación se abordan a continuación

Alcance

- El alcance de esta investigación es desarrollar un método de optimización para el problema del FJSSP.
- Se puede seguir realizando investigación sobre el problema del FJSSP de pasar de un objetivo a abordar casos multiobjetivo.

Delimitación

- Esta investigación solo realiza el modelado y experimentación computacional, comparando los resultados con aquellos publicados en la literatura especializada.
- El modelo de optimización desarrollado no será implementado en casos reales de la industria.

Hipótesis

Es factible optimizar el problema del FJSSP mediante la aplicación de dos técnicas metaheurísticas en el mismo algoritmo, una aplicada a la búsqueda global usando una vecindad de tipo autómata celular, y otra para la búsqueda local, obteniendo una programación de tareas satisfactoria.

Organización de la Tesis

El presente trabajo de tesis se ha organizado de la siguiente manera:

Capítulo I. Introducción y propósito de la investigación. Se presenta la introducción a la problemática, abordando el propósito de la investigación, el planteamiento del problema, los objetivos, la justificación, el alcance y delimitación y la hipótesis.

Capítulo II. Revisión de la literatura. Se hace una revisión de los trabajos recientes sobre la optimización del FJSSP utilizando metaheurísticas híbridas y se discute el área de oportunidad para seguir haciendo investigación en este contexto.

Capítulo III. Descripción del FJSSP. Se presenta la descripción formal del problema a optimizar.

Capítulo IV. Algoritmo híbrido basado en algoritmos genéticos y escalada de colinas para el FJSSP. Se describen los dos algoritmos metaheurísticos utilizados en este trabajo de investigación.

Capítulo V. Metodología y comparación. Se describe la metodología utilizada para la aplicación del algoritmo propuesto, así como los experimentos con los que se comparó el algoritmo para comprobar su eficacia.

Capítulo VI. Resultados experimentales. Los resultados obtenidos con el GA-RRHC son reportados así como el análisis estadístico obtenido mediante la comparación con los otros algoritmos de la literatura.

Capítulo VII. Conclusiones. Las conclusiones y el trabajo a futuro son discutidas en el capítulo, se hace mención de los trabajos desarrollados a lo largo de este estudio doctoral .

Capítulo I

Revisión de la literatura

Los problemas de scheduling son una subclase de problemas conocida por sus siglas en inglés como CSP (constraint satisfaction problem) o problemas de satisfacción de restricciones. Este tipo de problemas son más familiares de lo que se piensa, ya que se pueden encontrar en distintas áreas, principalmente en la industria, en la ciencia y la administración. Los problemas de scheluding son conocidos en general como NP-hard, a lo que se refiere que son de gran complejidad y no pueden ser resueltos mediante métodos matemáticos en un tiempo satisfactorio.

El problema de scheduling también conocido como programación de tareas, es el problema de secuenciar una serie de operaciones y ubicarlas en el espacio y tiempo adecuados sin violar ninguna de las restricciones impuestas. Los problemas de scheduling son problemas de optimización combinatoria, es decir, la asignación de recursos limitados a tareas a lo largo del tiempo. Este tipo de problemas tienen en común la planificación y ejecución de un conjunto de operaciones o tareas en el tiempo de inicio a fin. Un problema de scheduling tiene como finalidad la optimización de uno o más objetivos, como minimizar el número de tareas, disminuir el tiempo máximo de tareas de inicio a fin, tiempos grises, tiempos de transporte, tiempos de entrega, por mencionar algunas [75].

Los problemas de programación es el resultado de asignar las tareas o trabajos para que sean procesados en los recursos o máquinas. Este tipo de programación es característico de los problemas Job Shop Scheduling Problem (JSSP), conocido como uno de los problemas combinatorios más complejos de los problemas de optimización, y ha sido de gran interés de estudio en las áreas de producción. Para ser mas cercano a un ambiente de produc-

ción real, se implemento la posibilidad de que una operación de un trabajo sea procesado en dos o más máquinas, resultando así el Flexible Job Shop Scheduling Problem (FJSSP) [77].

Para solucionar este tipo de problemas se debe tener en cuenta tanto las restricciones de precedencia como las de recursos, para dar lugar así a un plan de ejecución viable. Durante las últimas tres décadas, este tipo de problema ha captado el interés de un gran número de investigadores.

El problema FJSSP, es una generalización del JSSP clásico, en términos generales se denomina como un procedimiento para organizar operaciones en un conjunto de máquinas, este conjunto consiste en máquinas independientes que trabajan en paralelo con características y capacidades operativas que en la mayoría de los casos puede diferir de máquina en máquina [10]. Para resolver el problema FJSSP, hay que considerar dos subproblemas, asignación de máquina y secuenciación de operaciones. La asignación de la máquina, es asignar una operación a una sola máquina para su procesamiento, mientras que la secuenciación de operaciones es programar el orden de todas las operaciones para obtener soluciones viables y de calidad [77].

La primera vez que fue introducido como tal el término de FJSSP fue por Brucker y Schlie [11], posteriormente por Brandimarte [10], siendo este último, uno de los primeros en abordar la instancia del FJSSP con enfoque heurístico. A partir de entonces hasta nuestros días, el FJSSP ha sido estudiado por diversos investigadores, aplicando distintos tipos de heurísticas y metaheurísticas para resolverlo.

El trabajo de Bruker y Schlie [11] es de gran importancia para la historia de programación flexible de tareas, ellos abordaron uno de los problemas más difíciles de resolver, un problema de taller multi propósito (MPM), inspirado en aquellos sistemas de fabricación flexible donde las máquinas están equipadas con diferentes herramientas, este trabajo propuesto a diferencia de su predecesor el JSSP clásico, que solo se encarga del problema de la secuencia de operaciones, en donde se programa un conjunto de trabajos a través de un conjunto de máquinas definidas con el fin de minimizar la función objetivo, el trabajo MPM dio paso al FJSSP, donde se permite que una operación pueda ser procesada por cualquier máquina que este disponible de un conjunto de ellas. En el FJSSP se resuelven dos subproblemas, el de enrutamiento y el de programación, y ellos fueron los primeros en presentar el FJSSP y proponer un gráfico polinomial. Su trabajo se ha tomado como referencia para investigaciones futuras, siendo ellos quien sentaron las bases de la formulación matemática del FJSSP.

Durante los últimos 30 años se han desarrollado gran variedad de métodos para resolver el FJSSP debido a su complejidad, que van desde métodos matemáticos tradicionales como técnicas enumerativas, técnicas de Branch y Bround, reglas de despacho, de planificación, hasta diversos tipos de optimización. A lo largo de los años los problemas scheduling han aplicado diferentes algoritmos con distintos enfoques de optimización inteligentes, aplicando diversos tipos de algoritmos de optimización como las técnicas metaheurísticas, entre los principales se encuentran los algoritmos evolutivos (AE) basados en la supervivencia del más apto en el que se encuentran los algoritmos genéticos (GA) y cuyos principios básicos fueron establecidos por Holland en 1975 [46], y posteriormente descritos por Goldberg en 1989 [41], la optimización de colonia de hormigas(ACO) introducida por Dorigo, M. en 1992 [27], la optimización de enjambre de partículas(PSO) desarrollado por Kennedy, J. y Eberhart, R. en 1995 [53], la Búsqueda Tabú (TS) atribuida a Glover en 1989 [39], solo por mencionar algunos algoritmos de optimización inteligentes, del cual se adaptan a distintos problemas de programación, y han sido abordados en distintas investigaciones, encontrado soluciones de alta calidad en un tiempo computacional razonable.

Es por eso que un gran número de investigadores han decidido encontrar una mejor solución o cercana a esta y han optado encaminarse por las técnicas con enfoque de optimización heurísticas y metaheurísticas, así como en aquellas búsquedas híbridas inteligentes.

Cuando se empezó a estudiar el FJSSP hace tres décadas, una de las primeras propuestas de solución fue el trabajo de investigación de Brandimarte [10]. Este abordó el problema del FJSSP utilizando un enfoque heurístico, del cual usó el método de reglas de despacho y un algoritmo de búsqueda tabú jerárquico para resolver el problema, e introdujo 15 instancias a resolver, con diferentes números de trabajos y máquinas.

El FJSSP al ser un problema complejo se basa en dos enfoques de modelado, en enfoque concurrente y en enfoque jerárquico. El enfoque concurrente, se basa en resolver aquellos problemas de enrutamiento y programación juntos, resolviendo el problema como un todo. El enfoque jerárquico, se basa en fragmentar el problema original para reducir su complejidad. Es así que para modelar el FJSSP es más sencillo mediante la aplicación de un enfoque jerárquico [10].

Hubo una preocupación por parte de Brandimarte en desarrollar este problema, y es que sea un método de solución fácil de implementar; lo suficientemente simple para ser entendido y capaz de permitir alguna forma

de interacción con el usuario; pero a la vez general para permitir ajustes y abierto a futuras mejoras; y sobre todo que sea capaz de producir soluciones razonablemente buenas con un esfuerzo computacional razonable. Desde entonces se ha estudiado el FJSSP en busca de nuevas propuestas de solución.

Otro de los trabajos iniciales y más citados dentro del FJSSP es el de Mastrolilli y Gambardella [67], que presentaron dos funciones basadas en el vecindario, utilizaron técnicas de búsqueda local proponiendo un procedimiento de búsqueda tabú, y encontraron una mejor solución a 38 de 43 problemas de referencias bien conocidos.

El trabajo de Xia y Wu [88] propusieron un enfoque de solución jerárquica para resolver objetivos múltiples en el FJSSP con el fin de acercar el problema a un escenario más cercano a la realidad. El enfoque propuesto hace uso del PSO para asignar las operaciones en máquinas y el algoritmo SA para programar operaciones en cada máquina, considerado minimizar el makespan, el total de carga de trabajo de máquinas y la carga de trabajo de la máquina crítica.

Pezzella, et al. [74] utilizaron un algoritmo genético para el FJSSP, este algoritmo integró distintas estrategias de selección y reproducción, demuestra que mezclando diferentes reglas para encontrar la población inicial, selección de individuos y operadores de reproducción, obtuvieron la mejor solución conocida a varios problemas hasta ese momento.

También se encuentra el trabajo de Gao, Sun, y Gen [35] que abordaron el FJSSP utilizando un algoritmo genético híbrido, fortaleciendo la búsqueda de los individuos mejorados con la variable de descenso de vecindad variable (VND). Ellos trataron un problema multiobjetivo en donde buscan encontrar el makespan mínimo, la carga de trabajo máxima de cada máquina y la carga de trabajo total mínima. Este método utiliza dos procedimientos de búsqueda local, el primero para una operación en movimiento y el segundo para dos operaciones en movimiento. Para corroborar su rendimiento, tomaron como referencia 181 problemas, de los cuales en 119 instancias obtuvieron la misma solución y en 38 encontraron mejores soluciones.

Por último, el trabajo de Zhang, Shao, Li, y Gao [97], el cual es de los más citados, aborda el problema del FJSSP optimizándolo a través de la hibridación de dos algoritmos, el PSO y TS, combinándolos para resolver un problema de varios objetivos en conflicto, para problemas de gran escala. Se utiliza el PSO por su alta eficiencia de búsqueda combinando búsqueda local y búsqueda global y la TS se usa para encontrar una solución casi óptima. Se probó la eficiencia del método con 4 instancias, demostrando resultados

satisfactorios.

De los trabajos más recientes de esta última década se encuentra los siguientes, el trabajo de Amiri, et al. [3], que proponen un algoritmo de búsqueda de vecindad variable (VNS) aplicado al FJSSP y su función objetivo es el makespan. Se presentan varias vecindades, donde utilizan operaciones de asignación para generar soluciones vecinas, y comparan la eficacia de su algoritmo con 181 problemas de referencia conocidos; Kacem data [51], BRdata [10], BCdata [6], DPdata [22] y HUdata [49].

Li, Pan, y Liang [56] proponen un algoritmo híbrido utilizando la TS y la VNS, consideran tres objetivos de minimización, produciendo soluciones vecinas para la asignación de máquinas, y realizando búsquedas locales en la programación de operaciones. Su modelo obtiene buenos resultados en 4 de 5 problemas de prueba, pero sin encontrar una mejor solución.

Dalfard y Mohammadi [21] estudian el FJSSP considerando máquinas paralelas y costos de mantenimiento, proponen un modelo matemático aplicando un algoritmo genético híbrido (HGA) y el algoritmo SA, y fue probado con 12 experimentos utilizando múltiples trabajos.

Por su parte, Yuan, Xu, y Yang, [93] utilizan como criterio de optimización el makespan para adaptar el algoritmo de búsqueda de armonía (HS) en el problema de FJSSP. Se desarrollaron técnicas para convertir el vector de armonía continua en dos vectores y así reducir el espacio de búsqueda, y finalmente introducen un esquema de inicialización mediante la combinación de técnicas heurísticas y aleatorias, incorporando la búsqueda local en el HS, todo esto con el fin de acelerar el proceso de búsqueda local en la vecindad. Se comparan su algoritmo con 201 problemas de referencia conocidos. Para el manejo de datos emplearon varios problemas de distintas referencias bien conocidas como las de Kacem data, Fdata [32], BRdata, BCdata, DPdata y HUdata; representando sus resultados con diagramas de Gantt .

Shahsavari-Pour y Ghasemishabankareh [79] aplican una combinación de GA y SA, denominándolo como NHGSA, para optimizar tres funciones objetivo. Sus resultados demuestran que superan a otros métodos resolviendo los mismos puntos, pero no especifican con qué métodos comparan sus resultados.

En la investigación de Li, Pan, y Tasgetiren [57] se aplica un algoritmo discreto de abejas artificiales, denominado DABC, con tres objetivos como criterio de optimización. Se adopta una estrategia autoadaptativa, representada para el análisis de datos por dos vectores discretos y una TS. Se utilizan los problemas BRdata y sus resultados los analizan con tres métricas princi-

pales, para evaluar la calidad y diversidad. Es decir, el número de soluciones no dominadas obtenidas, la distancia media del frente no dominado obtenido al frente de Pareto y la relación de las soluciones no dominadas obtenidas.

En el trabajo de Chaudhry y Khan [16] se concentró en estudiar 191 artículos relacionados al FJSSP. Se encontró que la función objetivo más investigada fue el makespan y se refiere al tiempo de tarda en completarse un trabajo, seguido de encontrar el makespan mínimo, de igual forma también se ha estado investigando la función objetivo múltiple. Este trabajo concluye que se han utilizado 55 funciones objetivo diferentes en varias publicaciones, y el makespan resultó ser la medida de rendimiento más utilizada, ya que fue aplicada en 88 trabajos de investigación (44,67%), mientras que en otros 78 documentos (39.59%) el makespan se usa en combinación con otra función objetivo, y se observó que el 59% de los documentos utilizaron técnicas híbridas o EAs.

En el trabajo de Chang, Chen, Liu, y Chou [15] la función objetivo a minimizar es nuevamente el makespan, y se propone un GA con la codificación de soluciones factibles, incorporando el método Taguchi para aumentar la efectividad del GA. Se evaluó el rendimiento del algoritmo propuesto tomando los problemas de [10] y la diferencia entre las velocidades de convergencia obtenidas aplicando los algoritmos GA y HTGA. Se presentan gráficas para mostrar la comparación de la velocidad de convergencia entre HTGA, TGA y eGA mejorados.

En el artículo de Zambrano, Bekrar, Trentesaux, y Zhou [95] se estudió el FJSSP para calcular los tiempos de liberación de trabajos adecuados con el fin de satisfacer las demandas de producción en relación con fechas de vencimiento específicas. Se propone la optimización con dos metaheurísticas: GA y PSO, así como dos enfoques diferentes para manejar los tiempos de liberación de trabajos. En el GA, los tiempos de liberación se tratan como variables dependientes, mientras que el PSO permite la integración de los tiempos de liberación del trabajo como variables independientes dentro de la codificación de partículas. Estos enfoques metaheurísticos se compararon utilizando tres puntos de referencia, dos adaptados de la literatura y uno inspirado en una celda de fabricación real. Sus resultados mostraron que el GA y el PSO obtienen desempeños similares para problemas restringidos y no restringidos.

En el trabajo de Gao, et. al [36] se aborda el problema de programación en la ingeniería de remanufactura aplicado a un FJSSP, dividiéndolo en dos etapas, la programación y la reprogramación cuando llega un nuevo trabajo.

La incertidumbre en el momento de los retornos en la remanufactura se modela como una nueva restricción de inserción de trabajo en FJSSP. Se propone un algoritmo de colonia de abejas artificiales de dos etapas (TABC) para programar y reprogramar con la inserción de nuevos trabajos. También se plantea una nueva regla para inicializar la población de colonias de abejas y una búsqueda local en conjunto para mejorar el rendimiento del algoritmo, y se comparan tres estrategias de reprogramación. Sus resultados y las comparaciones muestran que TABC fue eficaz tanto en la etapa de programación como en la de reprogramación.

En artículo de Zheng y Wang [98], aborda el FJSSP proponiendo la restricción por recursos dobles (DRC-FJSSP) siendo estas las variables dependientes. Se utiliza un algoritmo de optimización de la mosca de la fruta guiado por el conocimiento (KGFOA), con dos tipos de operadores de búsqueda basados en permutación para realizar la búsqueda basada en el olfato para la secuencia de trabajo y la asignación de recursos máquina y trabajador. Se incorpora también una etapa de búsqueda guiada con dos nuevos operadores de búsqueda para ajustar la secuencia de operaciones y la asignación de recursos. Debido a la combinación de la búsqueda guiada por el conocimiento y la búsqueda basada en el olfato, la exploración global y la explotación local puede equilibrarse. Los resultados comparativos muestran que el KGFOA fue más eficaz que los algoritmos existentes para resolver el DRC-FJSSP.

El estudio de Ahmadi, et al. [1] propone una metodología multiobjetivo para el FJSSP en el caso específico de situaciones de avería de máquinas. El algoritmo utiliza dos versiones del algoritmo genético de ordenación no dominante (non-dominated sorting genetic algorithm) NSGA y NSGA-II que normalmente son usados para resolver problemas multiobjetivo muy grandes. También se hace uso del frente de Pareto figurando un entorno de fabricación más realista. El algoritmo se comparan en función a cuatro criterios.

En su trabajo de investigación Li y Gao [58] proponen un algoritmo híbrido utilizando el GA y la TS con el objetivo de minimizar el makespan. El algoritmo propuesto tiene buena capacidad de búsqueda y equilibra muy bien la intensificación y diversificación; se compara la eficiencia del algoritmo con 6 instancias de referencia y 201 problemas abiertos. Se concluye que el algoritmo tiene buena capacidad de búsqueda con poco tiempo computacional y puede mantener un buen equilibrio entre la explotación y la exploración de soluciones.

Un estudio de caso fue abordado por Li, et al. [60] utilizando el algoritmo HABC, que es un algoritmo híbrido de colonia de abejas artificiales (ABC) y

el algoritmo TS mejorado para resolver el FJSSP en una empresa de máquina textil. Se introdujeron tres estrategias de reprogramación, que fueron reensamblar la programación, la intersección de la programación y la inserción de la programación, para abordar eventos dinámicos como nuevos trabajos insertados y averías en máquinas, donde la función objetivo es minimizar el makespan. Se muestra que el algoritmo HABC tiene una buena capacidad de explotación, exploración y fiabilidad de rendimiento para resolver el FJSSP.

Un problema multiobjetivo fue abordado por Deng et al. [24], donde trabajan el FJSSP utilizando el NSGA-II en combinación con la guía evolutiva de abejas (BEG-NSGA-II), denominándolo (MO-FJSSP). Dado que una desventaja de muchos algoritmos evolutivos multiobjetivo es su convergencia prematura, este trabajo hace uso de una optimización de dos etapas para evitar esta situación y así cumplir con sus objetivos que son minimizar el tiempo de finalización máximo, la carga de trabajo de la máquina más utilizada y la carga de trabajo total de todas las máquinas.

En el trabajo de Wu, Wu y Wang [86] se utiliza la optimización de colonia de hormigas para resolver el FJSSP, con base en el modelo de grafo disyuntivo 3D, tendiendo cuatro objetivos que son minimizar el tiempo de finalización, el costo de penalización por demora o anticipación, tiempo por inactividad promedio de la máquina y el costo de producción.

Una revisión de la literatura especializada se realiza por Amjad M., et al. [4], estudiando la aplicación de los algoritmos genéticos en el FJSSP. Se lleva a cabo una revisión exhaustiva tomando los últimos 20 años, analizando un total de 190 artículos publicados desde 2001 hasta diciembre de 2017. En su revisión, encuentran que los enfoques híbridos han ido utilizándose más y más a lo largo de los años, muestran que los GA es la técnica más utilizada para resolver un problema de FJSSP, tanto de manera individual como conjuntamente de forma híbrida con otras técnicas para la solución a estos problemas.

En el escrito de Shen, Dauzère-Pérès, y Neufeld [81], se aborda el FJSSP utilizando tiempos de configuración dependientes de la secuencia (sequence-dependent setup time, SDST) y un modelo de programación lineal entero mixto (MILP) para minimizar el makespan, mediante el uso del TS como algoritmo de optimización. Se aplican funciones específicas y una estructura de diversificación, el proceso se compara con instancias de referencias bien conocidas y dos metaheurísticas de la literatura, donde se obtienen buenos resultados.

El trabajo de Rodríguez, de Aguilar, e Hideaki [77] utiliza nuevas es-

trategias en la inicialización de la población, desplazamiento de partículas, asignación estocástica de operaciones y gestión parcial y totalmente flexible de escenarios para implementar un algoritmo híbrido, utilizando el PSO para el subproblema de enrutamiento en las máquinas y explorar el espacio de solución, mientras que una escalada de colinas con reinicio aleatorio (RRHC) se aplica al subproblema de programación de búsqueda local. Se concluye que la búsqueda estocástica tienen un buen rendimiento y gran variedad de soluciones en el espacio de programación.

El artículo de Gong, Deng, Gong, Liu, y Ren [43] considera dos factores (el humano y la máquina) para establecer un nuevo problema de doble FJSSP (double flexible job-shop scheduling problem, DFJSP) en el que no solo la flexibilidad de las máquinas es considerada sino también el aspecto humano. Se consideran simultáneamente dos indicadores, el tiempo de procesamiento y factores relacionados con la protección del medio ambiente. Se presentan y resuelven diez puntos de referencia para la aplicación del sistema de producción, utilizando un nuevo HGA como algoritmo de optimización. Se aplica además el método Taguchi para la evaluación del algoritmo propuesto.

En el trabajo de Xie y Chen [89] se analiza el FJSSP utilizando tiempos inciertos con el objetivo de minimizar el intervalo gris y el makespan. Se diseña un GA utilizando información gris basada en memoria externa con estrategia de elitismo, para reducir lo más posible los tiempos inciertos y así minimizar el makespan.

El artículo de Sreekara, et al. [82] resuelve el FJSSP para la minimización del makespan y la carga de trabajo de las máquinas, utilizando un modelo de programación no lineal (MINLP) e incorporando diferentes perturbaciones en tiempo real para la optimización con un nuevo algoritmo híbrido entre el PSO y el GA, obteniendo soluciones de alta calidad.

El escrito de Meng, Pan, y Sang [69] aborda el FJSSP para minimizar el flujo total de trabajo y costos de inventario, mediante la implementación de un algoritmo híbrido entre la colonia de abejas artificiales (hyABC) y el algoritmo de aves migratorias modificadas (MMBO), obteniendo una capacidad de búsqueda satisfactoria. Se compara la eficiencia de su algoritmo con instancias de diferentes escalas y otros algoritmos recientes. El muestreo en este problema son los costos de inventario, con este análisis se determina cuáles son los valores de costos de inventario, para así minimizar el flujo total de trabajo.

Tang et al. [84] aplican dos métodos de optimización, de búsqueda global y búsqueda local, para el FJSSP abordando dos características significativas

en la producción de fundición práctica, en donde consideran dos variantes, el intervalo de tiempo tolerado y el intervalo de tiempo de inicio limitado. En este caso, la población representa la producción de fundición, su unidad de análisis es la calidad para así generar una población inicial y con esos datos obtener un nuevo enfoque de selección y proponer un algoritmo con el fin de minimizar la tardanza total, el tiempo extra, y el tiempo máximo de finalización.

En el trabajo de Wu, Shen, y Li [87] se presenta un modelo para calcular el consumo de energía de la máquina en diferentes estados y el efecto de deterioro para determinar el tiempo real de procesamiento. Se aplica una optimización híbrida utilizando el algoritmo SA y se propone una heurística de programación de ahorro de energía.

Un algoritmo novedoso fue realizado por Lin, Zhu, y Wang [62] quienes proponen un método denominado de optimización híbrida de multiversos (hybrid multi-verse optimization, HMVO). Se aborda una variante difusa del FJSSP utilizando la técnica de vinculación de ruta e incorporando una fase mixta basada en inserción para ampliar el espacio de búsqueda y la búsqueda local para mejorar la solución. Utilizan tres conjuntos de referencia, demostrando la eficiencia del algoritmo propuesto.

La propuesta de Li, et al. [61] presenta un algoritmo híbrido de clasificación no dominado elitista (ENSHA) para un FJSSP multiobjetivo. Se desea minimizar el makespan y los costos totales de la instalación proponiendo una estrategia de aprendizaje basada en un algoritmo de distribución (EDA), comprobando su efectividad al analizar 39 instancias de referencia y un caso de estudio real.

En el trabajo de Gao, et al. [37] se analizan los métodos de optimización basados en inteligencia de enjambre (SI) y algoritmos evolutivos (EA) así como sus mejoras para resolver el FJSSP. Se menciona que más del 60 % de las publicaciones están relacionadas con SI y EA. Primero, presentan el modelo matemático de FJSSP y las restricciones en aplicaciones. Luego, la codificación y decodificación, las estrategias para conectar el problema y los algoritmos y las estrategias para inicializar algoritmos utilizando operadores de búsqueda locales para mejorar el rendimiento y la convergencia. Por último, se usa un algoritmo genético híbrido clásico (GA) y un algoritmo competitivo imperialista (ICA) con la búsqueda de vecindad variable (VNS) para resolver el FJSSP.

En el trabajo de Dai, Tang, Giret, y Salido [20] se menciona que la flexibilidad de los recursos y las restricciones complejas en un sistema de fabrica-

ción flexible hacen que la programación de la producción sea un problema complicado de programación no lineal. Con este fin, formulan un modelo de optimización multiobjetivo con el objetivo de minimizar el consumo de energía y la capacidad de producción con limitaciones de transporte. Luego, se desarrolla un GA mejorado para resolver el problema. Finalmente, se llevan a cabo experimentos integrales para evaluar el rendimiento del modelo y algoritmo propuestos. Los resultados experimentales revelaron que el modelo y el algoritmo propuestos pueden resolver el problema de manera efectiva y eficiente. Esto puede proporcionar una base para que los tomadores de decisiones obtengan una programación energéticamente eficiente en un sistema de fabricación flexible.

De los trabajos más recientes que han abordado el FJSSP se encuentra el estudio de Yang, Huang, Yu, y Bing [91], que maneja simultáneamente tres objetivos, maximizar la carga de trabajo, minimizar el tiempo de espera de cada operación y las averías de las máquinas. Se propone una medida llamada RMc con el fin de determinar el impacto del tiempo de espera en la robustez por la probabilidad de la avería de la máquina. Se obtienen resultados computacionales que muestran que el RMc evalúa con precisión la solidez de los programas con una pequeña cantidad de costo computacional.

En el trabajo de Goerler, Lalla-Ruiz, y VoB [40] se aborda el problema de programación en general, a pesar de no ser específicamente el tipo flexible, considerando el dimensionamiento de lotes con restricciones ejemplificadas por medio de reprocesos y restricciones de vida útil para artículos defectuosos, del cual se encuentran numerosas aplicaciones en entornos industriales. Se propone un algoritmo que usa la escalada de aceptación tardía como una solución novedosa que explota e integra el algoritmo de escalada y los enfoques exactos para acelerar el proceso de solución en comparación con la resolución del problema por medio de un método tradicional. Los resultados computacionales muestran los beneficios de incorporar enfoques exactos dentro de la escalada de colinas conducen a soluciones de alta calidad en tiempos computacionales breves.

En el artículo de Defersha y Rooyani [23] se trata el FJSSP para determinar simultáneamente tanto la asignación de operaciones a las máquinas como sus secuencias. Se desarrolla un GA de dos etapas (2SGA). La primera etapa del 2SGA tiene una codificación de solución que solo dicta la secuencia en la que se consideran las operaciones para su asignación. La segunda etapa, a partir de las soluciones de la primera etapa, permite que el algoritmo busque en todo el espacio de soluciones al incluir soluciones que podrían haber sido

excluidas debido a la naturaleza codiciosa de la primera etapa. El resultado más interesante que encontraron fue que la versión secuencial del algoritmo propuesto (usando un solo CPU) superó una implementación paralela del GA regular que usa muchos CPUs. También notaron que la superioridad del algoritmo propuesto es mucho mayor al resolver problemas de gran tamaño, lo que hace que sea una opción viable para resolver problemas prácticos que normalmente se encuentran en las industrias.

Otro algoritmo inteligente se propone en Alzaqebah, et al. [2] donde se utiliza un algoritmo de enjambre denominado Brain Storming Optimization (BSO) que simula el proceso de lluvia de ideas de los humanos. Con el fin de mejorar la búsqueda global del BSO, se propone una nueva estrategia de actualización para realizar de forma adaptativa varios métodos de selección y estructuras de vecindad. Además, el algoritmo BSO tiene una buena capacidad para explorar el espacio de búsqueda agrupando las soluciones y buscando en cada grupo de forma independiente, lo que conduce a una velocidad de convergencia lenta. Para mejorar la capacidad de intensificación local y para superar la convergencia lenta se aplica el Late Aceptación Hill Climbing (LAHC) con tres vecindades. Se llevaron a cabo extensos experimentos computacionales en cuatro puntos de referencia bien conocidos para FJSSP, y el rendimiento del algoritmo BSO se comparó con el del algoritmo propuesto. Los resultados demuestran que el algoritmo propuesto supera al algoritmo BSO. Además, el enfoque propuesto supera los algoritmos más conocidos en algunos casos y es comparable con estos métodos en otros conjuntos de datos.

El trabajo de Ding y Gu [26] afirma que estudiar el FJSSP es esencial para promover la eficacia de la producción, también menciona que diferentes tipos de algoritmos mejorados de optimización PSO han producido resultados superiores para el FJSSP en las últimas décadas. Mientras tanto, se combina el PSO con el algoritmo de optimización del aprendizaje humano (HLO), que es un algoritmo de aprendizaje simple y adaptativo. Las comparaciones de resultados con otros algoritmos relacionados revelan que HLO-PSO puede resolver de manera eficiente la mayoría de FJSSP de objetivo único.

Mihoubi, Bouzouia y Gaham [70] proponen un enfoque de simulación y optimización asistida por sustituto basado en reglas de programación para resolver un FJSSP realista. El enfoque propuesto equilibra la reactividad a corto plazo frente a perturbaciones repetitivas, así como el rendimiento general de los sistemas de fabricación. Aplicaron los GA haciendo uso de un sustituto neuronal híbrido y un modelo DES (simulación de eventos discretos) para el cálculo de la función de aptitud, considerando el criterio de mini-

mización de makespan y pedidos urgentes. Su modelo de resolución propuesto proporciona herramientas técnicas para futuros sistemas de control, permitiendo la implementación práctica de sistemas de ensamblaje personalizados en la Industria 4.0, apoyándose en tecnologías emergentes innovadoras.

Gong, et al. [42] proponen un modelo matemático de un FJSSP extendido con flexibilidad del trabajador (FJSPW) y presentan un algoritmo híbrido de colonia artificial de abejas (HABCA) para resolver el FJSPW propuesto. En el HABCA se implementa un nuevo método de búsqueda local efectivo para mejorar la velocidad y la capacidad de explotación del algoritmo. Se aplica además el método Taguchi de Diseño de Experimentos para obtener una mejor combinación de parámetros clave del HABCA. Los resultados del HABCA son muy efectivos en instancias de FJSPW a gran escala.

Fan, et al. [31] estudian un problema FJSSP del mundo real a gran escala para probar su algoritmo propuesto. Desarrollaron un algoritmo híbrido Jaya, que es capaz de resolver algunos problemas complicados de ingeniería. El algoritmo Jaya se combina con la búsqueda Tabu, en el cual consideraron múltiples rutas críticas. El algoritmo propuesto intenta continuamente mejorar las soluciones, que se representan en forma de dos vectores, moviéndolos hacia la mejor solución y alejándolos de la peor solución en la población actual.

Awad y Abd-Elaziz [5] mencionan que para lograr una alta productividad en la Industria 4.0, es necesario la reconfiguración de la fabricación tradicional. En su trabajo realizan un estudio de caso, donde introducen un nuevo método de optimización que toma datos adicionales disponibles de una máquina, para resolver el problema de programación de un FJSSP. Su solución busca la optimización desde una visión integral con respecto a los datos interdependientes, diseñando un enfoque dividido en dos etapas para mejorar la búsqueda.

A pesar de la efectividad de cada una de las técnicas heurísticas y metaheurísticas aplicadas en los trabajos antes mencionados, no existe una técnica en especial que pueda resolver de manera efectiva aquellos problemas flexibles de manera óptima, ya que siempre van a existir variaciones de variables entre problemas, y no se puede generalizar que una metaheurística sea mejor que otra por la forma en que funcionan y por la naturaleza del problema FJSSP. Siguiendo esta tendencia de investigación, esta tesis propone la hibridación de dos técnicas metaheurísticas para resolver instancias del FJSSP y obtener valores óptimos que sean equiparables o mejores que los reportados en la literatura.

La Figura I.1 muestra los principales autores clásicos que dieron origen al desarrollo del FJSSP, así como aquellos que desarrollaron por primera vez las heurísticas utilizadas de referencia a este trabajo de tesis. La Tabla I.1 muestra un resumen con los aspectos más importantes de algunos de los trabajos mostrados anteriormente, mencionando la función objetivo , su propuesta y método utilizado en cada trabajo.

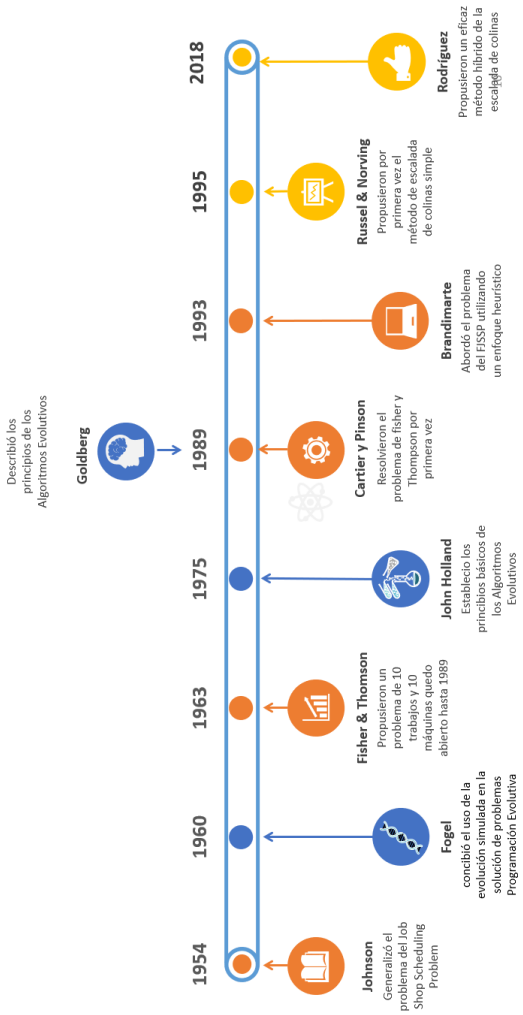

Figura I.1: Linea de tiempo con los principales exponentes clásicos en este trabajo

blue _____

	Función objetivo	Algoritmo				
Artículo	Mono-objetivo/ Multi-objetivo	Técnica Híbrida	Búsqueda Global Exploración	Búsqueda Local Explotación	Propuesta	Método
Mastrolilly y Gar-bardella [67]	* Minimizar makespan	−−	VNS	TS	Función basada en vecindario.	Utilizan técnicas de búsqueda local y presentan dos funciones de vecin-dad Nopt1 y Nopt2.
Xia y Wu [88]	* Minimizar el makes-pan (tiempo de final-ización máximo). * Total de carga de tra-bajo de máquinas. * Carga de trabajo de la máquina crítica.	Si	PSO	SA	Solución jerárquica para objetivos múltiples	PSO proporciona soluciones ini-ciales para SA. El valor de apti-tud de cada partícula se calcula mediante el algoritmo SA. El algo-ritmo SA es solo un subalgoritmo para todo el proceso de búsqueda. El PSO utiliza las soluciones eval-uadas por SA para seguir evolucio-nando.
Pezzella, Morgan-tia, y Ciaschettib [74]	minimizar makespan	−−	GA	−−	Nuevas estrategias selección y reproducción para generar la población inicial, seleccionar los individuos para reproducir nuevos individuos.	Adoptan enfoque de [51] mejorán-dolo, reordenan trabajos y máquinas, buscan el mínimo global
Gao, Sun y Gen [35]	* Makespan mínimo. * Carga de trabajo máx-ima de la máquina. * Carga de trabajo total mínima.	Si	GA	Descenso de Vecindad Variable (VND)	Encontrar intervalos de tiempo asignables para las operaciones eliminadas , de acuerdo el concepto de tiempo de evento más antiguo y más reciente	Esquema de representación de dos vectores y un método de decodifi-cación que interpreta cada cromo-soma en un programa activo
Zhang y otros [97]	* Minimizar el tiempo máximo de finalización * Minimizar la carga de trabajo de tiempo máximo * Minimizar la carga de trabajo total de las máquinas simultáneamente.	Si	PSO	TS	Omitir el método de actualización de desplazamiento de velocidad concreto del PSO tradicional para el FJSSP multiobjetivo.	el PSO asigna las operaciones a las máquinas y para la programación de las operaciones en cada máquina se aplica la BT

Amiri y otros [3]	Minimizar el makespan	– –	VNS	– –	Emplear dos estructuras vecinas para el problema de secuenciación y tres estructuras vecinas para el problema de asignación para generar soluciones vecinas.	Utiliza codificación lineal conocida como lista de secuenciación de tareas para representar soluciones.
Li, Pan y Lian [56]	* El tiempo máximo de finalización (makespan). * La carga de trabajo total de las máquinas. * La carga de trabajo de la máquina crítica.	HTSA	– –	TS & VNS	Presentan dos reglas de búsqueda de vecindad adaptables para realizar la búsqueda local en el modulo de asignación de máquinas con la TS y la VNS.	Integran tres estructuras de vecindad de inserción e intercambio para la programación de operaciones, y una función de desplazamiento para decodificar una solución a un horario activo.
Dalfard, y Mohammadi [21]	* Restricciones de mantenimiento . * Entorno de fabricación dinámico. * Flexibilidad operativa debido a las máquinas paralelas.	Algoritmo Genético Híbrido (HGA)	GA	SA	Modelar un método para el problema del taller de trabajo dinámico flexible multiobjetivo con máquinas paralelas (MO-FDJSPM) con restricciones de mantenimiento.	modelo matemático, programación no lineal entera mixta.
Yuan, Xu y Yang [93]	* Minimizar las ganancias.	Búsqueda de armonía híbrida (HHS)	Búsqueda de Armonía HS	Búsqueda Local	Incorporar la HS y una búsqueda local, se desarrollan técnicas de conversión para hacer que el HS continuo se adapte para resolver el FJSSP discreto.	A través del código de dos vectores para el FJSSP y decodificación activa; Mapear el vector armonía; Introducir un esquema de inicialización.
Shahsavari-Pour y Ghasemisha-bankareh [79]	* minimizar el tiempo máximo de finalización de todas las operaciones (makespan), * minimizar la carga de trabajo de la máquina más cargada, * minimizar la carga de trabajo total de todas las máquinas	Algoritmo genético híbrido y recocido simulado (NHGASA)	GA	SA	Utilizan los enfoques de Pareto (algoritmo genético multiobjetivo (MOGA)) para obtener la solución óptima aplicando sus tres objetivos desde un inicio.	El GA genera la primera población inicial y luego el SA mejora los individuos en cada población. Si no hay mejora en el algoritmo SA, se cambia un método de alteración en cada individuo
Li, Pan, y Tasgetiren [57]	* Tiempo máximo de finalización, * la carga de trabajo total de las máquinas * la carga de trabajo de la máquina crítica	Algoritmo de colonias de abejas artificiales discretas (DABC)	–	TS & Artificial Bee Colony(ABC)	Utilizaron la estrategia de generación de fuentes de alimentos vecinas autoadaptable para llevar a cabo la exploración y explotación de soluciones dentro de la solución completa.	En la primer etapa utilizaron la TS y el ABC lo integraron en el algoritmo DABC basado en enfoque de Pareto
Ahmadi, et al.[1]	* Optimizar el makespan. * Avería de máquina	2 Algoritmos Evolutivos (EA)	NSGA II & NRGA	–	Aplican una combinación de mejora de duración y de estabilidad simultanea en la programación	Utilizan un enfoque de simulación para evaluar el estado y la condición de las averías de la máquina.

Li y Gao [58]	Minimizar el makespan	Algoritmo híbrido efectivo (HA)	GA	TS	Resolver problemas de programación en el campo de la fabricación, como el JSP, el problema de planificación y la programación de procesos integrados, etc	Diseñan un método de codificación efectivo, con operadores genéticos y una estructura de vecindad, y utilizan la TS para un ajuste adaptativo en el proceso de evolución de HA
Li, et al. [60]	minimizar el tiempo máximo de finalización (makespan)	Algoritmo Híbrido de colonia artificial de abejas (HABC)	ABC	TS	Mediante una estrategia de reprogramación aplicando tres tipos de programación: programación de reensamblaje, de intersección y de inserción	Aplicando la ruleta de agrupación por conglomerados para inicializar mejor la población e introducen un operador cruzado para las ABC
Deng et al. [24]	* Minimizar el tiempo máximo de finalización, * Minimizar la carga de trabajo de los mas saturados, * Minimizar la carga de trabajo total de todas las máquinas.	Algoritmo genético de clasificación no dominada guía evolutiva de abejas II (BEG-NSGA-II)	Abeja evolutiva guiada (BEG) & Algoritmo genético de clasificación no dominada II (NSGA-II)	–	Una BEG para una optimización de dos etapas para resolver el multiobjetivo (MO)	Primero aplican el algoritmo NSGA-II con tiempos de iteración para obtener la población inicial, y segundo obtienen las soluciones óptimas de Pareto para mejorar la capacidad de búsqueda
Shen, Dauzère-Pérès, y Neufeld [81]	Minimizar los tiempos de producción.	–	–	TS	Proponen resolver un problema del FJSSP con tiempos de preparación dependientes de la secuencia.	Presentan un modelo matemático que resuelven instancias pequeñas de manera óptima. Utilizan un modelo de gráfico disyuntivo. Aplican la TS con nuevas funciones de vecindad y una estructura de diversificación específica.
Rodríguez, de Aguilar, e Hideaki [77]	* Minimizar el makepan * La carga de trabajo de la máquina crítica, *La carga de trabajo total de las máquinas.	PSO+RRHC	PSO	RRHC	Proponen una mejor manera de generar nuevas estrategias en la inicialización de población, desplazamiento de partículas, asignación estocástica de operaciones, gestión de escenarios parcial y totalmente flexible	Utilizan el PSO para gestionar el enrutamiento de la máquina y el RRHC para encontrar el mejor horario para completar la ruta de todos los trabajo mediante una descomposición jerárquica del problema.
Xie y Chen [89]	* Minimizar el tiempo de procesamiento de intervalos grises.	–	GA	–	Proponen un modelo de programación job shop scheduling de incertidumbre para minimizar el intervalo de tiempos de grises, aplicando un GA de elitismo (EGA), acoplando su estrategia en la memoria externa.	Diseñaron nueve tamaños diferentes de cajas G-FJSP, adoptaron un diseño de experimento ortogonal para seleccionar los parámetros clave del algoritmo genético.

Tang et al. [84]	* Minimizar el tiempo extra total de TTI. * La tardanza total. * el tiempo máximo de finalización	Algoritmo híbrido de PSO discretas integrada con un algoritmo SA (HDPSO-SA)	PSO	SA	Descomponer en fases la búsqueda global y local. Emplearon tres enfoques metaheurísticos multiobjetivo	Con la búsqueda global mejoran la calidad de la población inicial con un nuevo enfoque de selección para acelerar la convergencia del algoritmo. Diseñan cuatro estructuras vecinales bajo dos estrategias de búsqueda local para saltar la trampa de la solución óptima local.
Lin, Zhu, y Wang [62]	* Minimización del tiempo de procesamiento (LS)	Optimización híbrida multiverso (HMVO)	–	Reenlace de ruta (Path relinking); & Heurística de inserción; & Basada en pares	Integran estrategias de búsqueda local eficientes y tres conjuntos de referencia	Intercambio de objetos a través de los agujeros negros/blancos y los agujeros de gusano integrados en el Path relinking, y una heurística basada en inserción , y con la búsqueda local basada en pares (Pairwise-based) para mejorar la solución.
Li, et al. [61]	* Minimizar el tiempo de finalización máximo (tiempos de ejecución) * Los costos totales de instalación (TSC)	Algoritmo híbrido de clasificación no dominado elitista (ENSHA)	Operaciones genéticas	–	Implementar reglas de asignación de tareas dependientes del problema con una novedosa evolución	Aplican tres reglas de asignación de trabajos y después utilizan un modelo evolutivo bipoblacional mejorado implementando dos poblaciones, una polación principal (MP) y una auxiliar (AP)
Dai, Tang, Giret, y Salido [20]	* Minimizar el consumo de energía y el rendimiento	Programación de un FJSP eficiente en energía (EFJSP)	GA	–	Desarrollar un algoritmo genético mejorado para resolver el FJSSP.	Formulan un modelo de optimización multiobjetivo para minimizar el consumo de energía y el rendimiento para un FJSP con restricciones de transporte.
Goerler, Lalla-Ruiz, y VoB [40]	* Minimizar el consumo de energía y el rendimiento	Escalada de colinas de aceptación tardía (LAHCM)	—	Escalada de colinas (HC)	Solucionar el dimensionamiento y programación de lotes por medio de la reelaboración y restricciones de vida útil de artículos defectuosos (GLSP)	Con el LAHCM explota e integra el algoritmo para acelerar el proceso de solución para resolver el problema por medio de una solución general
Defersha y Rooyani [23]	* Configuración dependiente de la secuencia. * Fecha de lanzamiento de la máquina. * Tiempo de retraso.	Algoritmo genético de dos etapas , (2SGA)	GA	–	Desarrollar un algoritmo genético de dos etapas (2SGA)	En la primera etapa se codifican la asignación de operaciones; y en la segunda etapa, el algoritmo realiza una búsqueda en todo el espacio de soluciones, e incluye soluciones que pueden ser excluidas.
Alzaqebah, et al. [2]	* Minimizar el makespan	Algoritmo de optimización de llu-vía de ideas (BSO)	BSO	Escalada de colinas de aceptación tardía (LAHC)	Retoman la lluvia de ideas en humanos para recopilar nuevas ideas y mejorarlas iterativamente	Con la búsqueda global mejoran la adaptación de diferentes métodos de selección y de vecindad. Con la búsqueda local hacen que el algoritmo converja rápidamente.

Ding y Gu [26]	* Minimizar el makespan	algoritmo híbrido (HLO-PSO)	PSO	Optimización del aprendizaje humano (HLO)	Promover la eficiencia y eficacia de la producción.	Utiliza varias combinaciones del PSO mejorado, utilizando estrategias de programación bajo la arquitectura de algoritmo de HLO.

Tabla I.1: Revisión de la literatura

Inspirándose en estos hallazgos, este trabajo de investigación presenta un nuevo algoritmo para optimizar el FJSSP, motivada por el hecho que en diferentes investigaciones han aplicado algoritmos genéticos o escalada de colinas, han obtenido buenos resultados de acuerdo a la revisión de la literatura, se eligen estas técnicas metaheurísticas para aplicar en este trabajo de investigación. En particular, este trabajo toma como base los resultados en [77], [56] y [2]. El objetivo de esta investigación es proponer un algoritmo aplicando dos técnicas metaheurísticas híbridas que minimicen el tiempo de procesamiento de todas las tareas (o makespan) en problemas FJSSP con alta flexibilidad, es decir, donde un mayor número de máquinas puedan hacer la misma tarea. Esto se realiza aplicando una búsqueda global con operadores combinatorios genéticos, y después una búsqueda local aplicando la escalada de colinas con reinicio múltiple. La primera metaheurística explora el conjunto de soluciones para resolver el subproblema de enrutamiento para realizar la asignación de máquinas, resolviendo el problema de selección de máquina, asignando las operaciones a las máquinas disponibles; mientras que la segunda metaheurística realiza la explotación de estas soluciones para encontrar el mejor orden de tareas en las máquinas disponibles para la resolución del subproblema de programación, resolviendo el problema de secuenciar las operaciones que son asignadas en todas las máquinas.

Capítulo II

Descripción del FJSSP

En este capítulo se da la definición y fundamentos del FJSSP, aplicando este tipo de programación para problemas de asignación de máquina, o también conocido como programación de tareas. Este tipo de programación es utilizado en este trabajo para implementar una estrategia de optimización.

II.0.1 Fundamentos del Flexible Job shop Scheduling Problem

La programación (scheduling), es la asignación de tareas a recursos, de manera que puedan procesarse y/o fabricarse de manera óptima, en un lapso de tiempo adecuado para realizar un conjunto de trabajos. Un problema de programación siempre tendrá tres elementos característicos, primero las tareas u operaciones (trabajos) a realizar, segundo, los recursos que se tienen disponibles (máquinas) para la elaboración del primer elemento, y el tercer elemento es la finalidad o el objetivo (función objetivo) que se pretende optimizar [75], Figura II.1.

Para representar a los primeros dos elementos se tiene m máquinas M_j cuando $(j = 1, \ldots, m)$, teniendo que procesar n trabajos J_i cuando $(i = 1, ..., n)$. Para una programación schedule se tiene que para cada trabajo hay una asignación de uno o más intervalos de tiempo procesada en una o más máquinas. La programación puede representarse mediante diagramas de Gantt, como se observa en la Figura II.4.

El tercer elemento en un problema de planificación o programación, la función objetivo, es establecer un orden de procesamiento, en el que se pueda alcanzar un objetivo deseado de manera óptima. Entre los principales ob-

Figura II.1: Programación de fabricación

jetivos de estudio que se desea encontrar es, el tiempo total requerido para completar todas las operaciones, la máxima demora, la máxima anticipación, por mencionar algunos. Por lo tanto, para alcanzar estos objetivos se diseñan programas para alcanzar varias medidas de desempeño.

Este tipo de problemas de programación son conocidos en general como NP-hard debido a su gran complejidad. Se pueden encontrar gran número de enfoques en la literatura, unos abordan el problema como una búsqueda de solución exacta, usando programación entera lineal. Para los métodos exactos, se encuentra la programación matemática como el método, Branch and bound (ramificación y acotamiento) [72], la programación lineal [64], y la relajación lagrangiana [17]. Estos métodos matemáticos aseguran una convergencia global y han funcionado muy bien en la solución de instancias pequeñas, pero en aquellos problemas que van aumentando de tamaño, requieren un tiempo computacional muy alto [76]. Es por eso que un gran número de investigadores han decidido encontrar una mejor solución o cercana a esta y han optado encaminarse por las técnicas con enfoque de optimización heurísticas y metaheurísticas, así como en aquellas búsquedas híbridas inteligentes. [92]

Para realizar un diseño de un problema de programación de la producción, va en función de varias fases o aspectos, de acuerdo con [99] los más utilizados son:

1. el proceso de llegada de trabajos,
2. la política de inventario,
3. los atributos shop y de trabajo, y

4. la configuración shop.

La primera fase a considerar es el proceso de llegada del trabajo, donde los problemas de programación de la producción se clasifican en dos tipos de programaciones, la dinámica y la estática; en la primera, los trabajos llegan a la máquina de manera intermitente o aleatoria, donde se puede llegar a suscitar cualquier tipo de interrupción en el entorno de fabricación. En la segunda, los trabajos llegan a una máquina al mismo tiempo que se encuentra inactiva después de un intervalo de tiempo fijo. [4]. Es así que la importancia de programación va a depender siempre del entorno de fabricación.

La programación dinámica, que es la que se aplica en este trabajo, es una de las técnicas más utilizadas para tratar problemas de optimización combinatoria. La programación dinámica se puede aplicar tanto a problemas que se pueden o no resolver en un tiempo polinomial. Esta programación dinámica ha demostrado ser muy útil para problemas estocásticos.

La programación dinámica en términos sencillos es un esquema de enumeración completo, donde utiliza un enfoque de divide y vencerás, con el fin de minimizar la cantidad de cómputo a realizar. El enfoque general consiste en una serie de subproblemas donde con cada uno se va obteniendo una solución, y esto se realiza uno a uno hasta encontrar la solución del problema original. De manera particular, su forma de trabajo es que, primero determina una solución óptima para cada subproblema, obteniendo una contribución a la función objetivo, es decir, que en cada iteración determina la solución óptima para un subproblema. Finalmente, se encuentra una solución para el subproblema actual utilizando toda la información obtenida antes en las soluciones de todos los subproblemas anteriores, para encontrar una solución al problema general [75].

La segunda fase del diseño a tomar en cuenta en algunos problemas va de acuerdo a la política de inventario, en el cual puede ser abierto y cerrado, donde un problema puede ser abierto si todos los productos se fabrican bajo pedido o puede ser cerrado si todos los productos se fabrican según el stock. También existen los sistemas híbridos donde se combinan los sistemas abiertos y cerrados, pero este tipo sistemas se encuentran comúnmente en problemas de la vida real.

La tercera fase son los atributos shop, aquí los problemas de programación de la producción también se pueden clasificar como deterministas (deterministic machine scheduling) si los tiempos de procesamiento del trabajo y la disponibilidad de la máquina y todos aquellos datos del problema de planifi-

cación se conocen a priori, y sino, estos se conocen como probabilísticos. Los modelos determinísticos son modelos estudiados por la optimización combinatoria.

Dentro de la planificación determinística, se realiza la planificación y secuenciación de tareas. Debido a la complejidad de este tipo de problemas, es importante aplicar los algoritmos más adecuados, cuyo tiempo de ejecución sea adecuado para el tamaño del problema.

Un gran número de investigadores que ha desarrollado programación toma a consideración lo siguiente, primero se considera el número de trabajos y el número de máquinas, tomando en cuenta que son finitos. El número de trabajos se denota por n y el número de máquinas por m. Comúnmente el subíndice j indica un trabajo, y el subíndice i indica la máquina. Cuando el trabajo demanda varios pasos de procesamiento u operaciones, entonces se denota por *(i,j)*, lo que significa que es el paso de procesamiento u operación del trabajo j en la máquina i. Los datos que comúnmente se necesitan para un trabajo j son, el tiempo de procesamiento *($p_{i,j}$)*, la fecha de publicación *(rj)* la fecha de vencimiento *(dj)*, y el peso *(wj)* [75].

De acuerdo con [75] define los datos como a continuación se explican:

- Tiempo de procesamiento (Processing time, $p_{i,j}$). El $p_{i,j}$ representa el tiempo de procesamiento del trabajo j en la máquina i. El subíndice i se omite si el tiempo de procesamiento del trabajo j no depende de la máquina o si el trabajo j solo debe procesarse en una máquina determinada.

- Fecha de publicación (Release date, r_j). La fecha de publicación *(r_j)* del trabajo j también puede denominarse fecha de preparación, y es el momento en que el trabajo llega al sistema, es decir, es la primera vez en que se libera el trabajo j y puede comenzar su procesamiento.

- Fecha de vencimiento (Due date, d_j). La fecha de vencimiento *(d_j)* del trabajo j, representa la fecha de finalización del trabajo comprometida o la fecha límite, es decir, la fecha que se promete terminar el trabajo al cliente. Llega a existir la posibilidad que se permite la finalización de un trabajo después de su fecha de vencimiento, pero esto conlleva una consecuencia, la cual es que se incurre en una penalización.

- Peso (Weight, w_j). El peso *(w_j)* del trabajo j significa que es esencialmente un factor de prioridad, donde denota la importancia del trabajo j

en relación con los otros trabajos en el sistema. Por ejemplo, este peso puede representar el costo real de mantener el trabajo en el sistema. Este costo podría ser un costo de mantenimiento o inventario; también podría representar la cantidad de valor ya agregado al trabajo.

Un problema de programación se describe mediante un triplete $\alpha \mid \beta \mid \gamma$. El campo α describe el entorno de la máquina y contiene solo una entrada. El campo β proporciona detalles de las características y restricciones del procesamiento y puede no contener ninguna entrada, una sola entrada o varias entradas. El campo γ describe el objetivo a minimizar y, a menudo, contiene una sola entrada.

Una vez conocidos todos los datos que se necesitan para una programación lo siguiente es saber el tipo de diseño donde se va a aplicar la programación, y de acuerdo con [4] en los diseños de máquinas existe una clasificación, en la cual, va a depender de acuerdo a los requisitos, tanto para el proceso de fabricación como del producto, y entonces los talleres (shop) siendo aquellos que están en función de sus múltiples trabajos y recursos, se han clasificado en varios diseños como se muestra en la Figura II.2, donde la presente clasificación da una importancia mayor al caso job shop.

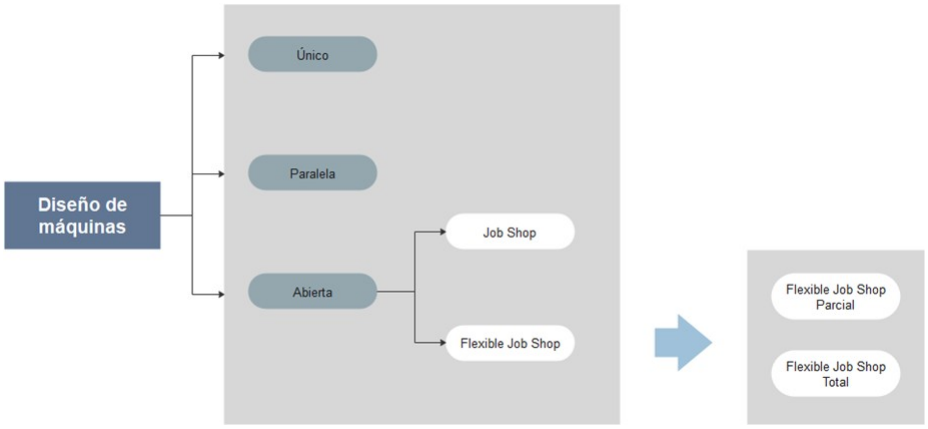

Figura II.2: Clasificación del diseño de máquinas [4]

De acuerdo con [75] y [99], definen a los diseños de máquina único, paralelo y abierta como sigue:

1. Campo α

1.1 **Único:** El entorno de producción de una sola máquina, es el más simple de todos los entornos de las máquinas posibles y es un caso especial en comparación con los demás entornos de máquinas que son más complicados. Los modelos de una sola máquina a menudo tienen propiedades que no tienen ni las máquinas en paralelo ni las máquinas en serie. Los resultados que se pueden obtener para los modelos de una sola máquina no solo brindan información sobre el entorno de una sola máquina, sino que también brindan una base para las heurísticas que son aplicables a entornos de máquinas más complicados.Este tipo de entorno, fue el primero en abordarse académicamente y sus características y hallazgos se han aplicado a problemas más complejos. En la práctica, los problemas de programación en entornos de máquinas más complicados, a menudo se descomponen en subproblemas que se ocupan de máquinas individuales. Son muy útiles para estudiar estructuras seriales más complejas. Por ejemplo, un entorno de máquina complicado con un solo cuello de botella puede dar lugar a un solo modelo de máquina, donde, la máquina es el cuello de botella de todo el proceso, y para generar un buen programa para este tipo de problema es muy importante obtener un buen rendimiento. Su tiempo de flujo de un trabajo se calcula tomando el tiempo de finalización menos el tiempo de liberación, esto en otras palabras, es el tiempo total que permanece el trabajo en el sistema, y el orden de importancia de los trabajos se introducen ponderaciones para que se preste de mayor a menor importancia los trabajos. Otro ejemplo importante cuando existe una variante de este tipo de entorno es el problema de programación de una sola máquina con configuraciones dependientes de la secuencia, este problema es muy conocido con el nombre de, problema de agente viajero (travelling salesman problem, TSP), que por su complejidad cae en el tipo de problema NP-Hard. Este problema se puede resolver de manera eficiente solo si se aborda en instancias pequeñas, pero tanto este problema de agente viajero, como el de cuello de botella, se consideran como problemas difíciles computacionalmente cuando sus instancias son de tamaño de mediano a grandes, por lo que se tiene que hacer uso de alguna técnica heurística.

1.2 **Paralela:** Este tipo de entorno, la mayoría de las veces se aplica

en los problemas de programación de la vida real, y a diferencia del entorno único, aquí ya se consideran varias máquinas, donde, múltiples máquinas pueden ocurrir en paralelo o en serie o en ambos. Para este tipo de programación de máquinas en paralelo, se considera que cada trabajo se puede procesar en cualquiera de las máquinas y los tiempos de procesamientos son independientes de la máquina siendo las máquinas idénticas. El problema en este entorno, es identificar que máquina va a procesar cada trabajo y en que orden van a entrar. Al igual que en el entorno de una sola máquina, aquí también algunos problemas se pueden resolver de manera óptima (problemas de instancias pequeñas), por ejemplo el problema de flujo mínimo se puede utilizar la regla del despacho para encontrar el trabajo con el menor tiempo de procesamiento asignado a la máquina menos cargada; mientras que otros problemas solo se podrán resolver por aproximación (problemas de instancias grandes), por ejemplo, para encontrar el makespan mínimo, no podrá resolverse aplicando algoritmos exactos, sino también haciendo uso de alguna heurística.

En la programación de las máquinas existen 3 tipos de máquinas paralelas:

1.2.1 Máquinas idénticas en paralelo *(P_m)*. Cuando se tiene un número m máquinas idénticas en paralelo *(P_m)*, entonces el trabajo j requiere una sola operación donde pueda procesarse ya sea en cualquiera de las m máquinas, o si es que pertenece a un subconjunto dado. Pero si el trabajo j no pudiera procesarse en cualquier máquina, solo se procesará si pertenece a un subconjunto M_j específico, entonces la entrada M_j va a aparecer en el campo β.

1.2.2 Máquinas en paralelo con diferentes velocidades *(Q_m)*. Se tiene un número m máquinas en paralelo con diferentes velocidades. Por lo que la velocidad de la máquina i se denota por v_i, y se tendrá un tiempo p_{ij}; tiempo que el trabajo j dedicará a la máquina i y que es igual p_j/v_i suponiendo que el trabajo j recibe todo el procesamiento de la máquina i. Este tipo de entorno se conoce como máquinas uniformes. Por ejemplo, si todas las máquinas tienen la misma velocidad teniendo $v_i = 2$ para todo i y $p_{ij} = p_j$, entonces el entorno será idéntico al

anterior.

1.2.3 Máquinas no relacionadas en pa-ralelo *(R_m)*. Este entorno es una generalización del caso anterior de las máquinas en paralelo con diferentes velocidades; aquí también en este entorno hay m máquinas diferentes en paralelo, y la máquina puede procesar el trabajo j a la velocidad v_{ij} y con un tiempo p_{ij}; tiempo que el trabajo j dedicará a la máquina i, y que es igual a p_j/v_{ij} (al igual que el entorno anterior también suponiendo que el trabajo j recibe todo el procesamiento de la máquina i). Pero si las velocidades de las máquinas son independientes de los trabajos, donde, $v_{ij} = vi$ para todo i y j, entonces, se concluye que el entorno será idéntico al anterior.

1.3 **Abierta:** El diseño del entorno de los problemas de máquinas individuales y paralelas, no es común ver que este tipo de problemas se desarrollen para entornos de producción reales, en cambio, en los últimos años la investigación académica se ha enfocado en problemas más complejos. Por tal motivo, los problemas de diseño de entorno abierto se utilizan ampliamente para modelar procesos de producción industrial.

Estos problemas se identifican por tener múltiples trabajos y recursos. Todos estos problemas son aquellos con ambiente a problemas general shop. Una característica de los problemas shop, y lo cual se va a resaltar entre los distintos tipos de taller (shop), es que van a estar en función siempre de acuerdo al patrón de flujo de trabajos y de recursos. Entre los más importantes y que han sido abordados en la literatura se encuentran el problema de programación de taller abierto (Open Shop Scheduling Pro-blem, OSSP), el problema de programación de taller de flujo (Flow Shop Scheduling Problem, FSSP), el problema de programación de taller de trabajo flexible (Flexible Job Shop Scheduling Problem, FJSSP), el problema taller de programación (Job Shop Schedu-ling Problem, JSSP), y por otro lado y no menos importantes pero que no han sido tan abordados en la literatura, se encuentran los problemas de programación de talleres mixtos (mixed shop scheduling problems, MSSP) y los problemas de programación de talleres grupales (Group Shop Scheduling Problems, GSSP). Donde a continuación se explican los tipos de taller shop.

Tipos de problemas Shop Scheduling

Todos los problemas de ambiente Shop tienen una manera general de como trabajan. El problema de planicación general (general shop scheduling problem, GSSP), se puede describir de la siguiente manera. Se tiene un conjunto de n trabajos $\{J_1, \ldots, J_n\}$ donde cada trabajo $i = 1,\ldots,n$, es procesado sobre un conjunto de m máquinas Mk, $\{M_1, \ldots, M_m\}$. Cada trabajo J_i consta de un conjunto de operaciones o tareas $O_{ij}(j = 1, \ldots, n_i)$ donde n_i es el número de operaciones del trabajo J_i, con tiempos de procesamiento p_{ij}. Lo que significa que cada operación O_{ij} debe ser procesada en una máquina $\mu_{ij} \in \{M_1, \ldots, M_m\}$. Para cada una de las operaciones O_{ij} se tiene un tiempo de procesamiento $t_{O_{ijk}}$, y solo puede ser procesada en una sola máquina del conjunto M_{ij}.

Hay que tomar en cuenta que dependiendo el problema, pueden o no, existir relaciones de precedencia entre las operaciones de todos los trabajos. Una regla general, en los problemas GSSP cada trabajo solo puede ser procesado sobre una máquina, y cada máquina solo puede procesar un trabajo en un instante. El objetivo de estos problemas es encontrar un cronograma factible que minimice alguna función objetivo de C_i (tiempo de finalización) de los trabajos $i = 1, \ldots, n$, y normalmente es la que se asume en los trabajos de investigación, ya que la meta es optimizar una o varias funciones objetivo [12].

Los problemas que derivan del GSSP se denominan casos especiales, a partir de éste es que añaden alguna restricción dependiendo el problema. Los problemas derivados del GSSP y más referenciados, y de acuerdo con [12] se definen como:

1.3.1 El problema **Open Shop Scheduling Problem (OSSP)**. Es un caso especial del GSSP en el que cada trabajo i consta de n_i operaciones $O_{ij}(j = 1, \ldots, n_i)$ donde O_{ij} debe procesarse en la máquina M_j, cada una de las operaciones del trabajo solo se procesará sobre una máquina diferente, no existiendo alguna relación de precedencia entre las operaciones. Así, el problema es encontrar el orden de procesamiento de las operaciones que pertenecen a un mismo trabajo, y también se tiene que encontrar el orden de las operaciones que van a ser

procesadas en la misma máquina.

1.3.2 El problema **Flow Shop Scheduling Problem (FSSP)**. Es un problema general del GSSP, en el que, cada trabajo i consta de n_i operaciones O_{ij} con tiempos de procesamiento $p_{ij}(j = 1, \ldots, n_i)$ donde O_{ij} debe ser procesado en la máquina M_j; Sus restricciones de precedencia es de la forma $O_{ij} \rightarrow O_{i,j+1}(i = 1, \ldots, n_i - 1)$ para cada $i = 1, ..., n_i$, es decir, que las operaciones de todos los trabajos deben utilizar un orden secuencial, del trabajo 1 con la máquina 1, hasta el trabajo j_{ni}

1.3.3 El problema **Job Shop Scheduling Problem (JSSP)**. Es un caso especial del problema del taller general GSSP que generaliza el problema del taller de flujo FSSP. Se tienen n trabajos $i = 1, ..., n$ y m máquinas $M_1, \ldots, M_m$. El trabajo i consiste en una secuencia de n_i operaciones $O_{i1}, O_{i2}, \ldots, O_{i,ni}$ que deben procesarse en este orden, es decir, tenemos restricciones de precedencia de la forma $O_{ij} \rightarrow O_{i,j+1}(j = 1, \ldots, n_i - 1)$. Hay una máquina $\mu_{ij} \in \{M_1, \ldots, M_m\}$ y un tiempo de procesamiento p_{ij} asociado a cada operación $O_{ij}.O_{ij}$ debe procesarse para unidades de tiempo p_{ij} en la máquina μ_{ij}. Las operaciones de un determinado trabajo tienen un cierto orden y se deben ejecutar secuencialmente, por lo que el objetivo es encontrar un orden determinado de las operaciones de cada máquina, donde el problema es encontrar un cronograma factible que minimice alguna función objetivo en función de los tiempos de finalización Ci de las últimas operaciones $O_{i,ni}$ de los trabajos. Si no se establece de otra manera, se asume que $\mu_{ij} \neq \mu_{i,j+1}$ para $j = 1, \ldots, n_i - 1$.

1.3.4 El problema **Flexible Job Shop Scheduling Problem (FJSSP)**. Es un caso especial del problema del taller general GSSP que generaliza el problema del JSSP. Además de la secuenciación de las operaciones, también se consideran las asignaciones de tareas de la máquina (routing). El problema es dividido en dos subproblemas en ruta y secuenciación, en el siguiente apartado se explica con mayor profundidad, ya que esta programación es el objeto de estudio de esta investigación.

1.3.5 El problema de talleres mixtos **Mixed Shop Scheduling problems (MSSP)** las rutas que tiene la máquina de los trabajos i se pueden fijar con o sin restricciones. El problema MSSP se puede considerar como una combinación de los tres problemas de programación, el Flow, Job y el open shop scheduling problem respectivamente. Este problema surge debido a que en un sistema de control y planificación de la vida real puede ser una mezcla de los talleres anteriores denominados puros. Por lo tanto, en la literatura reportan [80] la necesidad de introducir un taller mixto, en el que las rutas se den para algunos trabajos, como en el Flow shop o job shop, pero en las rutas de los demás son irrelevantes, como en el problema de open job shop. El término taller mixto Mixed Shop se usó para ejemplificar una forma más general de un sistema de etapas múltiples, es decir, una mezcla entre taller de flujo flow shop y un taller abierto open shop, y por este motivo se le denominó taller mixto. Es definido como sigue, se tiene un conjunto de n trabajos $\{J_1, \ldots, J_n\}$, un conjunto de m máquinas $\{M_1, \ldots, M_m\}$ y un número de operaciones $O_i (i = 1, \ldots, n)$ que indica el número de operaciones del trabajo $J_i \in n$.

1.3.6 Problema de talleres grupales **Group Shop Scheduling Problems, (GSSP)** al igual que el anterior, este problema también comparte características similares al FSSP, JSSP y al OSSP. Se tiene a O_{ij} un conjunto de operaciones que se divide en subconjuntos J = $\{J_1, \ldots, J_n\}$ y subconjuntos $M = \{M_1, \ldots, M_m\}$, donde n es el número de trabajos, m es el número de máquinas. Sea J_i, donde cada trabajo i consta de n_i operaciones $O_{ij} (j = 1, \ldots, n_i)$ donde O_{ij} debe procesarse en la máquina M_j. Las operaciones i de cada trabajo pertenecen a g grupos $G = G_1, \ldots, G_g$. Existen dos tipos de restricciones, primero donde las operaciones no son restringidas si están dentro de cada grupo, por el contrario, si las operaciones pertenecen a diferentes grupos, estas deben cumplir con alguna relación de precedencia entre los grupos, misma que es impuesta por el problema. También puede existir el caso específico, donde cada operación constituya un grupo, es

decir, que el problema GSSP es igual al problema JSSP. En resultado, si para todos los trabajos, todas las operaciones de trabajos i pertenecen al mismo grupo, entonces el problema GSSP es equivalente al OSSP [99].

2. Campo β

   Dentro de este campo incluyen ciertas restricciones de procesamiento y limitaciones, que pueden incluir múltiples entradas. Las posibles entradas en el campo β son:

   2.1 Fecha de lanzamiento, (Release date , r_i). Cuando existe el release date y aparece en el campo β , significa que, el trabajo i no puede iniciar su procesamiento antes de su fecha de lanzamiento r_i. Por lo tanto, si r_i no aparece en el campo β , el procesamiento del trabajo i puede iniciar en cualquier momento. A diferencia del r_i, se encuentran la fecha de vencimiento(due date) y esta no se especifican en el campo β. La función objetivo da suficiente información si hay, o no hay, fechas de vencimiento.

   2.2 Preferencias (*prmp*).

   Desde que inicia hasta que finalice un trabajo, las preferencias significan que no es necesario mantener forzosamente el trabajo en una máquina. Ya que se puede interrumpir el procesamiento del trabajo en cualquier momento y en su lugar colocar un trabajo diferente en la máquina, y todo el procesamiento que se realizó hasta ese momento no se pierde por el nuevo trabajo que sustituye al trabajo anterior. El trabajo que fue sustituido en ocasiones requiere otra vez volver a colocarlo ya sea en la misma máquina o en otra diferente (en el caso de las máquinas paralelas), y terminar el tiempo de procesamiento restante. Existe un condicional de las preferencias, y es que están permitidas si y solo si, está incluido *prmp* en el campo β, por lo tanto, cuando no está incluido *prmp*, no están permitidas las preferencias.

   2.3 Restricciones de precedencia (Precedence constraints, *prec*).

   Pueden aparecer en una sola máquina o en un entorno de máquinas paralelas, y cuando existen restricciones de precedencia, es necesario que uno o más trabajos se concluyan antes de permitir que

otro trabajo comience su procesamiento . Existen distintos tipos de restricciones de precedencia:

- Si cada trabajo tiene un predecesor y un sucesor como máximo,
- A las restricciones se les denomina como cadenas,
- Las restricciones se denominan *outtree*, si cada trabajo tiene máximo un predecesor,
- Las restricciones se denominan *intree*, si cada trabajo tiene máximo un sucesor,
- Si no aparece *prec* en el campo β, los trabajos no están sujetos a restricciones de precedencia.

2.4 Tiempos de configuración dependientes de la secuencia (Sequence dependent setup times, s_ik).

S_{jk} representa el tiempo configuración que corre entre el procesamiento de los trabajos i y k; s_{0k} representa el tiempo de configuración para el trabajo k solo si k es el primero en la secuencia, y s_{i0} representa el tiempo de limpieza después del trabajo i, solo si el trabajo i es el último en la secuencia, tanto s_{0k} y s_{i0} pueden ser cero. Si el tiempo de configuración entre los trabajos i y k depende de la máquina, se debe incluir el subíndice j, quedando como s_{ijk}. Si s_k no aparece en el campo β, entonces se asume que todos los tiempos de configuración son cero o independientes de la secuencia, en este caso solo se van a incorporar en los tiempos de procesamiento.

2.5 Recirculation ($rcrc$).

La recirculación puede ocurrir en un Job shop o en Flexible Job Shop cuando un trabajo puede visitar más de una vez una máquina o un centro de trabajo.

Cualquier otra entrada que pueda aparecer en el campo β se explica por sí misma. Por ejemplo, las fechas de vencimiento, a diferencia de las fechas de lanzamiento, no se especifican de manera clara en este campo; ya que el tipo de función objetivo es el que indica si los trabajos tienen o no fechas de vencimiento.

3. Campo γ

El objetivo a minimizar está en función de los tiempos de finalización de los trabajos y dependen del tipo de programación (schedule). Para el tiempo de finalización que tiene la operación del trabajo i en la máquina j se denota como C_{ij}. Para el tiempo de finalización de la última máquina con el último trabajo i se denota por C_i. La función objetivo puede estar con base a los tiempos de entrega. Por lo tanto, el retraso(lateness) que tiene el trabajo i se define en II.1:

$$L_i = C_i - d_i, \tag{II.1}$$

Lo que indica que, es positivo cuando el trabajo i finaliza tarde, y negativo cuando se completa en forma temprana.

La tardanza (tardiness) del trabajo i se define en la ecuación II.2:

$$T_i = max(C_i - d_i, 0) = max(L_i, 0) \tag{II.2}$$

Cuando se habla de retraso o tardanza (lateness y tardiness), existe una diferencia, y esta radica en que, tardiness nunca va a ser negativa. Existe una penalización del trabajo j, como se muestra en la ecuación II.3:

$$f(x) = \begin{cases} 1 & \text{si } C_j > d_j \\ 0 & \textit{de lo contrario} \end{cases} \tag{II.3}$$

Criterio de Optimalidad

De acuerdo con [12], para indicar el tiempo de terminación del trabajo J_i, este se denota por C_i, y el costo asociado se indica con $f_i(C_i)$. Hay esencialmente dos tipos de función costo total, la ecuación II.4 es utilizada para problemas de cuello de botella y la ecuación II.5 está dirigida para aquellos que se requiere hacer suma de objetivos:

$$f_{max}(C) := max\{f_i(C_i)|i = 1, \ldots, n\} \tag{II.4}$$

$$\sum f_i(C) := \sum_{i=1}^{n} f_i(C_i) \tag{II.5}$$

Un problema de programación scheduling, su fin es hallar un programa con el que obtenga un cronograma factible en el que se minimice la función de costo total.

Cuando no se especifican las funciones f_i, lo que se va a realizar es establecer $\gamma = f_{max}$ o $\gamma = \sum f_i$. Pero en la gran mayoría de los casos se considera como función especial al f_i.

Las funciones objetivo más comunes son:

- El makespan C_{max}.

  El C_{max} es definido como $max(C_1, \ldots, C_n)$, y es el tiempo de finalización del último trabajo una vez que concluyo todos los trabajos (operaciones) en el sistema. Para obtener un makespan mínimo, implica obtener un buen aprovechamiento y programación de las operaciones en las máquinas.

$$\begin{aligned} &max\{C_i | i = 1, \ldots, n\} \\ &\gamma = C_{max} \end{aligned} \tag{II.6}$$

- El tiempo de flujo total (flow time).

  La suma de los tiempos de finalización es conocido en la literatura como, tiempo de flujo.

$$\begin{aligned} \gamma &= \textstyle\sum_{i=1}^{n} C_i \\ &= \textstyle\sum C_i \end{aligned} \tag{II.7}$$

- tiempo de flujo ponderado total (weighted total flow time).

  De acuerdo con [75] la suma de los tiempos de finalización ponderados de los n trabajos, da una referencia de los costos totales de mantenimiento o inventarios incurridos por la programación. El tiempo de finalización ponderado total se denomina tiempo de flujo ponderado.

$$\begin{aligned} \gamma &= \sum_{i=1}^{n} w_i C_i \\ &= \sum w_i C_i \end{aligned} \tag{II.8}$$

Existen otras funciones objetivo, pero a diferencia de las anteriores, estas van a depender de las fechas de vencimiento d_i que a su vez están asociadas con los trabajos J_i. Por lo tanto, para cada trabajo J_i se define:

- retraso-lateness

$$L_i = C_i - d_i \tag{II.9}$$

- anticipación-earliness

$$E_i = max\{0, d_i - C_i\} \tag{II.10}$$

- tardanza-tardiness

$$T_i = max\{0, C_i - d_i\} \tag{II.11}$$

- desviación absoluta-absolute deviation

$$D_i = |C_i - d_i| \tag{II.12}$$

- desviación al cuadrado-squared deviation

$$S_i = (C_i - d_i)^2 \tag{II.13}$$

- penalización unitaria-Unit penalty.

$$U_i = \begin{cases} 0 & \text{si } C_i \leq d_i \\ 1 & \textit{de lo contrario} \end{cases} \tag{II.14}$$

Cuando una función objetivo es regular, se identifica cuando el objetivo no es decreciente con respecto a todas las variables C_i . Mientras que cuando se involucran las funciones E_i, D_i y S_i entonces no son regulares. Y el resto de las funciones son regulares.

Existen dos clasificaciones para determinar si un programa es programable o no, es decir, se le denomina a un programa activo, cuando no es posible programar trabajos(operaciones) no sin antes violar alguna restricción. Cuando a un programa se le denomina semiactivo es, si no se puede procesar ningún trabajo(operación) no sin antes cambiar el orden de procesamiento o violar las restricciones.

II.0.2 Definición: Formulación del problema FJSSP

El problema de programación Flexible Job Shop Scheduling Problem (FJSSP), es una extensión del problema clásico del Job Shop Scheduling Problem (JSSP) pero con la diferencia que en este tipo de problemas, la flexibilidad está más encaminada a un escenario de la vida real donde existe la posibilidad de que una operación o tarea de un trabajo se procese por dos o más máquinas compatibles.

El FJSSP se puede resumir como un plan para organizar operaciones en un conjunto de máquinas, este conjunto consiste en máquinas independientes que trabajan en paralelo con características y capacidades operativas que pueden ser similares o diferentes (Brandimarte, 1993).

Un FJSSP general puede formularse de la siguiente manera [58], [77] :

1. Hay un conjunto de n trabajos $J = \{J_1, J_2, J_3, .., J_n\}$, para procesarse en un conjunto de m máquinas.
2. El conjunto de máquinas m se indica como: $M = \{M_1, M_2, M_3, \ldots, M_m\}$.
3. Cada trabajo J_i, consiste en una secuencia de operaciones n_i como: $\{O_{i1}, O_{i2}, O_{i3}, \ldots, O_{in}\}$ donde n_i es el número de operaciones que abarca.
4. Para completar un trabajo es necesario procesar todas las operaciones, respetando la precedencia de las operaciones, es decir, cada operación $O_{ij}(i = 1, 2, 3, ..., n; j = 1, 2, ..., n_i)$ debe ser procesado por un conjunto de máquina dadas hasta completar el trabajo.
5. La ejecución de cada operación $O_{i,j}$ de un trabajo i requiere una máquina de un conjunto de máquinas dadas M_{ij} y el tiempo de procesamiento de $O_{i,j}$ en la máquina M_m, puede ser denotado como p_{ijm}, donde de $M_{ij} \subset M$ (Flexibilidad parcial) o $M_{ij} \subseteq M$ (Total flexibilidad).
6. El FJSSP necesita determinar dos criterios, una asignación, y una secuencia de las operaciones de las máquinas, y esto con el fin para satisfacer un criterio dado. En estos tipos de problemas existen algunas condiciones y los siguientes supuestos se consideran en un FJSSP general:

   6.1 Todas las máquinas están disponibles en el tiempo t = 0.

6.2 Todos los trabajos están disponibles en el momento t = 0.

6.3 Cada operación puede ser procesada por una sola máquina a la vez.

6.4 No hay restricciones de precedencia entre las operaciones de diferentes trabajos; por lo tanto, trabajos son independientes el uno del otro.

6.5 No se permite la anticipación de operaciones, es decir, una operación iniciada no puede ser interrumpida.

6.6 Tiempo de transporte de trabajos entre las máquinas y tiempo para configurar la máquina para el procesamiento de una operación particular se incluye en el tiempo de procesamiento.

Una comparación del JSSP al FJSSP, es que el clásico requiere secuenciación de operaciones en máquinas fijas, mientras que en el FJSSP la asignación de una operación no se fija de antemano y, por lo tanto, se puede procesar en un conjunto de máquinas.

Por lo tanto, el FJSSP no se trata solo con la secuenciación, sino también con la asignación de operaciones a máquinas adecuadas (enrutamiento). El FJSSP, es, por lo tanto, más complejo que el JSSP, ya que considera la determinación de la asignación de la máquina para cada operación.

La flexibilidad de enrutamiento en la programación de trabajos en FJSSP puede ser categorizado en los siguientes dos subproblemas:

1. **Un subproblema de enrutamiento**, donde tenemos que seleccionar una máquina adecuada entre las disponibles para procesar cada operación.

2. **Un subproblema de programación**, donde las operaciones asignadas se secuencian en todas las máquinas seleccionadas para obtener un cronograma factible que minimice un objetivo predefinido.

En este tipo de configuración del FJSSP se presentan dos tipos:

1. Flexibilidad total. Donde se define que todas las operaciones se pueden ejecutar en todas las máquinas disponibles.

2. Flexibilidad parcial. Las operaciones se pueden realizar solo en algunas máquinas, se tiene que especificar donde serán procesadas las operaciones, así como la secuencia que tendrán en cada una de las máquinas.

Grafo disyuntivo

Los gráficos disyuntivos se pueden usar para representar y asociar a un determinado programa factible para aquellos problemas generales de taller (general shop problems). Cuando se aborda un problema en el cual se tiene un conjunto de programas factibles, identificando a estos como aquellos que cuentan siempre con una solución óptima al problema, entonces estamos hablando de una función regular. Entonces, el modelo de gráfico disyuntivo se puede emplear para elaborar horarios (schedules) óptimos. La manera de describir a un grafo disyuntivo para una instancia determinada de un general shop problem está descrito por $G = (V, C, D)$, donde el grafo G contiene los arcos disyuntivos, y por eso el nombre de grafo disyuntivo y se denota por [12]:

- V representa el conjunto de nodos, donde cada nodo (también vista como los vértices) va a representar cada una de las operaciones de todos los trabajos. Dentro de estos vértices, hay dos nodos especiales denominados como la fuente y la culminación de la programación, a estos nodos son representados virtualmente como 0 y $*$ respectivamente, un inicio($0 \in V$)para ser la fuente de G, y un fin ($* \in V$) para ser la culminación de G.

  Para cada uno de los nodos del grafo disyuntivo (G), se asocia un peso a cada nodo. Los pesos del comienzo 0 y fin $*$ son cero, mientras que los pesos que le corresponde a los otros nodos, serán los tiempos de procesamiento de las operaciones correspondientes.

- C representa el conjunto de arcos conjuntivos dirigidos (es decir, no disyuntivos) que conectan operaciones consecutivas del mismo trabajo. Estos arcos reflejan las relaciones de precedencia entre las operaciones. Además, están los arcos conjuntivos que va en relación entre la fuente y todas las operaciones que no tienen antecesor, también entre todas las operaciones sin sucesor y el fin de la programación.

- D representa el conjunto de arcos disyuntivos no dirigidos que conectan operaciones a ser procesadas por la misma máquina. Estos arcos están presentes tanto para cada par de operaciones que pertenecen al mismo

trabajo y que no están conectados por una cadena de arcos conjuntivos y también para cada par de operaciones a ser procesadas en la misma máquina pero que no están conectadas por una cadena de arcos conjuntivos.

En otras palabras, cuando se realiza la construcción del programa, se establecerá en ese momento las orientaciones que tendrán todos los arcos disyuntivos, con esto se determina la secuencia de las operaciones en la misma máquina. Una vez que ya se ha determinado una secuencia para una máquina, los arcos disyuntivos que conectan las operaciones a ser procesadas por la máquina, estos arcos son reemplazados por la flecha de precedencia orientada o arco conjuntivo. Para los arcos disyuntivos D se particiona en grupos, en la cual una pertenece a cada máquina, y el tiempo de procesamiento para cada operación es el peso asociado a los nodos correspondientes.

El FJSSP equivale a encontrar el orden de las operaciones en cada máquina, es decir, establecer la orientación de los arcos disyuntivos de manera que el grafo solución resultante sea acíclico (no hay conflictos de precedencia entre operaciones), esa longitud de la ruta ponderada más larga entre los nodos inicial y terminal, es la que determina el valor del Makespan.

El problema del FJSSP se representa a través de un grafo disyuntivo como se muestra en la Figura II.3.

Cálculo del makespan

Cada nodo i tiene un peso π, en el que indica el tiempo necesario para completar una operación i, y una vez que ya teniendo la secuenciación, se prosigue al cálculo del makespan, sabiendo el peso de cada rama (i, j), entonces el makespan es igual a la longitud del camino más largo entre el nodo inicial y el final, también conocido como camino máximo o camino crítico (critical path), donde la relación de tiempos y la secuencia de trabajos se va guardando en una matriz. En la literatura el makespan es conocido como II.15:

$$Cmax = \max\{C_1 \dots C_j\} \tag{II.15}$$

Para la construcción de las matrices se tendrá la matriz de las máquinas i y los trabajos j, cada una por su cuenta tendrá el tiempo de tardanza por

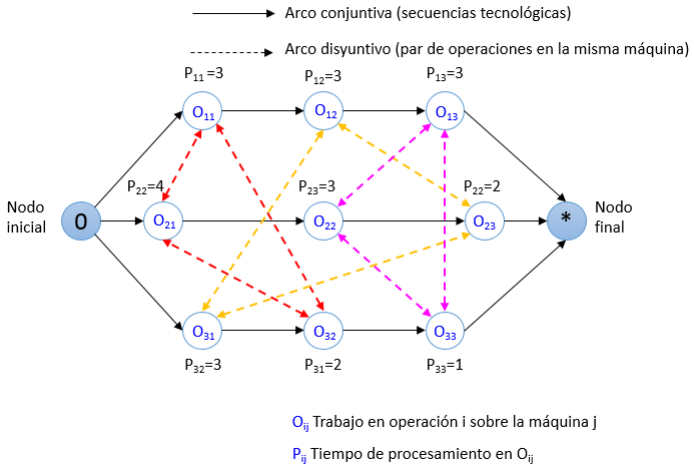

Figura II.3: Ejemplo de grafo disyuntivo de 3x3 [90]

cada operación ejecutada. Entonces el tiempo de inicio de la operación del primer trabajo en la máquina i es igual al tiempo de inicio de la operación del primer trabajo en la máquina $i-1$, donde $1 < i < m$, al ser la primera operación, el tiempo es el que indique del trabajo j de la operación 1, el resto de los trabajos, su valor es 0. El tiempo de inicio de la primera operación del trabajo j en la primera máquina, será el tiempo de inicio de la primera operación del trabajo $j-1$, donde $1 < j < n$. Para la siguiente operación, se toma el máximo valor entre la matriz de las máquinas y los trabajos, dependiendo el trabajo que se realice en esa operación, se toma entonces el valor $i + i - 1$, y lo mismo para la otra matriz, se continúa con el mismo procedimiento, y al finalizar las operaciones, para conocer el makespan, se identifica el valor máximo entre el número de trabajos. La manera de visualizar la representación de la solución de tiempo de un JSSP es el diagrama de Gantt. Como ejemplo, Yamada y Nakano (1997), abordan un problema de 3 máquinas y 3 trabajos, y el diagrama de Gant resultante se observa en la Figura II.4.

En este trabajo el objetivo de programación se toma como criterio el makespan, donde se requiere minimizar el tiempo máximo que tarda en completar todas las operaciones.

El criterio más usado por los autores como Li y Gao [58], Sreekara y otros [82], Xie y Chen [89] , Shen [81], y Lin, Zhu, Wang [62], entre muchos más toman como criterio al makespan.

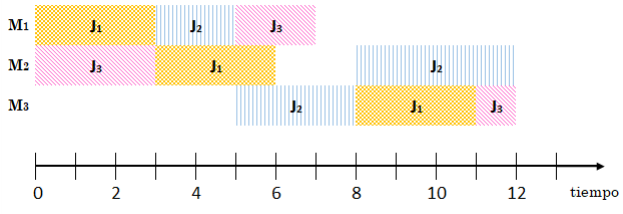

Figura II.4: Ejemplo de un diagrama de Gant de 3x3 [90]

El modelo matemático se toma de referencia del de Li y Gao [58], y Rodríguez y otros [77], aplicando una nueva propuesta matemática.

Capítulo III

Algoritmo híbrido basado en algoritmos genéticos y escalada de colinas para el FJSSP

Existen varios métodos para diseñar el Algoritmo Híbrido, pero de acuerdo a la revisión de la literatura han demostrado que el GA y la escalada de colina son buenos buscadores, para la exploración y la explotación de soluciones. Por esta razón, en este trabajo de tesis se utilizan estas dos técnicas metaheurísticas. Se toma la idea de las cadenas MS y OS, mediante la aplicación de los GA (selección, cruce y mutación), insertando la Escalada de Colinas con reinicio múltiple, como búsqueda local. Pero antes se darán los fundamentos metaheurísticos que dan pauta al uso de las metaheurísticas utilizadas en este trabajo de tesis.

III.1 Fundamentos Metaheurísticos

En la terminología de programación, a menudo se hace una distinción entre una secuencia, un cronograma y una política de programación. Una secuencia suele corresponder a una permutación de los n trabajos o al orden en que se procesarán los trabajos n en una máquina m dada. Un cronograma generalmente se refiere a una asignación de trabajos dentro de un entorno más complicado de máquinas, lo que posiblemente permita la preferencia de trabajos por parte de otros trabajos que se liberan en momentos posteriores [75].

El concepto de una política de programación se usa a menudo en entornos estocásticos: una política prescribe una acción apropiada para cualquiera de los estados en los que puede estar el sistema.

En los modelos deterministas, por lo general, solo las secuencias o los programas tienen importancia. Se deben hacer suposiciones con respecto a lo que el planificador puede y no puede hacer cuando genera un horario. Por ejemplo, puede darse el caso de que un horario no tenga inactividad no forzada en ninguna máquina. Esta clase de programaciones se puede definir de tres maneras:

1. **Horario sin demora (Non delay)**. Un cronograma factible se llama sin demora si ninguna máquina se mantiene inactiva mientras una operación está esperando para ser procesada.

   Exigir que un horario no tenga demoras equivale a prohibir la ociosidad no forzada. Para muchos modelos, incluidos aquellos que permiten apropiaciones y tienen funciones objetivas regulares, existen programaciones óptimas que no tienen demoras. Para muchos modelos considerados en esta parte del libro, el objetivo es encontrar un cronograma óptimo que no presente demoras. Sin embargo, hay modelos en los que puede ser ventajoso tener periodos de ociosidad no forzada.

   Una clase más pequeña de horarios, dentro de la clase de todos los horarios sin demora, es la clase de los horarios sin demora no preventivos. Los horarios sin demora no preventivos pueden conducir a algunas anomalías interesantes e inesperadas.

2. **Horario Activo (Active Schedule)**. Un programa factible no preventivo es llamado activo si no es posible construir otro horario, a través de cambios en el orden de procesamiento en las máquinas, con al menos una operación finalizando antes y ninguna operación finalizando más tarde. En otras palabras, un cronograma está activo si no se puede colocar ninguna operación en un espacio vacío antes en el cronograma mientras se preserva la viabilidad. Un cronograma no preventivo sin demora debe estar activo, pero lo contrario no es necesariamente cierto.

3. **Horario semiactivo (Semi-Active Schedule)**. Un programa factible no preventivo se llama semiactivo si ninguna operación puede completarse antes sin cambiar el orden de procesamiento en cualquiera de las

máquinas. Está claro que un horario activo tiene que ser semiactivo. Sin embargo, lo contrario no es necesariamente cierto.

La figura III.1 muestra un diagrama de Venn de las tres clases de horarios no preventivos: los horarios no preventivos sin demora, los horarios activos y los horarios semiactivos.

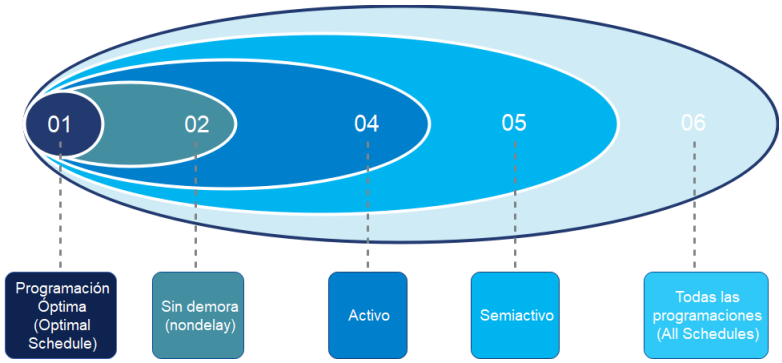

Figura III.1: Diagrama de Venn a de los tipos de planificaciones para el FJSSP

III.1.1 Complejidad computacional de los problemas shop scheduling

Debido a la naturaleza compleja de los problemas de programación existe una gran variedad de problemas de optimización que han tenido gran relevancia, ya sea con enfoque en la práctica o como en la teoría, donde estos problemas están en la búsqueda de la mejor configuración de un conjunto de variables para lograr uno o más objetivos.

De acuerdo con [73], los problemas de optimización se dividen naturalmente en dos categorías:

1. Primero, en aquellas soluciones que están codificadas con variables continuas,

2. Segundo, en las soluciones codificadas con variables discretas.

Dentro de esta última categoría de variables discretas se encuentra una clase de problemas a los que son denominados problemas de optimización combinatoria (OC).

En los problemas continuos, lo que comúnmente se busca es un conjunto de números reales o una función; mientras que en los discretos o combinatorios, se busca un objeto de un conjunto finito, o si es el caso, en un infinito numerable, y estos objetos suelen ser aquellos números enteros, un conjunto, una permutación o un gráfico.

Optimización Combinatoria

Los problemas de programación (shop scheduling problems) son un ejemplo claro de la clase de problemas de optimización Combinatoria (OC), donde las soluciones están codificadas con variables discretas [99].

La OC se puede definir como una instancia de un problema de optimización y es un par (S, f), donde S es cualquier conjunto, o el dominio de los puntos factibles; mientras que f, es la función costo, y esta tomara solo valores enteros no negativos. Según [8];

> *Un problema de Optimización Combinatoria P=(S, f) puede ser definido por:*
> *—un conjunto de variables* $X = x_1, \ldots, x_n$;
> *—dominios de variables* $D_1, \ldots, D_n$;
> *—restricciones entre variables;*
> *—una función objetivo* f *a minimizar, donde* $f : D_1 \times \ldots \times D_n \in \mathbb{R}^+$;
> *El conjunto de todas las posibles asignaciones factibles es:*
> $S = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} | v_i \in D_i, s$ *satisface todas las restricciones*$\}$
> *S se denomina al espacio de búsqueda o solución, ya que cada elemento del conjunto puede verse como una solución candidata. Para resolver un problema de optimización combinatoria, se debe encontrar una solución* $s^* \in S$ *con un valor de función objetivo mínimo, es decir,* $f(s^*) \leq f(s) \forall s \in S$. s^* *se denomina solución globalmente óptima de* (S, f) *y el conjunto* $S^* \subseteq S$ *se denomina conjunto de soluciones globalmente óptimas.*

Todo lo anterior significa que, para encontrar una solución óptima en el espacio de búsqueda, dicho espacio de solución debe explorarse de manera eficiente, y dependiendo el problema, la solución óptima, minimiza o maximiza

la función objetivo, tomando siempre en cuenta las restricciones el problema abordado. A la solución óptima también es conocida como óptimo global. También se considera el espacio de codificación ya que es el que va a representar las posibles asignaciones de las variables del problema una vez que ya se haya codificado, y pueda aplicarse como un método específico en futuros estudios.

Una vez ya tenido claro que la OC su objetivo es encontrar la mejor solución óptima de las soluciones ya existentes donde se minimice una función dada, lo siguiente es determinar la complejidad de problema de programación abordado y a través de qué método se va a resolver.

A lo largo de los años se han ido desarrollando nuevos métodos computacionales que han dado solución a problemas de programación, por lo que algunos de estos problemas pueden resolverse eficientemente, reduciéndolos y aplicándolos a problemas de optimización combinatoria bien conocidos, entre estos se encuentran los problemas con programas lineales, problemas de flujo máximo, o problemas de transporte, también se encuentran los problemas que se pueden resolver con técnicas estándar, como la programación dinámica y los métodos de ramificación y vinculación [12].

También se presentan en aquellas áreas de la informática y en las disciplinas que se aplican métodos computacionales, como la inteligencia artificial, la investigación de operaciones, la bioinformática o el comercio electrónico, entre otros problemas combinatorios bien conocidos también se encuentran en lo concerniente a la planificación, la programación, la calendarización, la asignación de recursos, el diseño de código, el diseño de hardware y la secuenciación del genoma [47].

Cuando se desarrolla un problema de programación o comúnmente un problema de OC, se debe considerar el tipo de complejidad que va a implicar el problema. Lo primero que se busca a resolver un problema nuevo es tratar de encontrar un algoritmo que dé solución al problema y que este se resuelva de manera eficiente, ya que de acuerdo con varios autores, existen problemas computacionales que son más fáciles de resolver que otros. La teoría de la complejidad, proporciona un marco matemático, donde clasifican a los problemas computacionales como fáciles o difíciles [12].

Cuando se dice que, si un problema es fácil o difícil, de acuerdo a la teoría de complejidad, la eficiencia del algoritmo se mide por su tiempo de ejecución, está en función al número de pasos que necesita para una determinada entrada, una función de eficiencia, depende del tamaño de entrada, y esto traducido a un programa de computadora, se define por la longitud de

codificación binaria de la entrada. Por tal motivo, cuando se clasifica que el problema es *fácil* (*easy*) o tratable, este tipo de problema si puede resolverse utilizando un algoritmo de tiempo polinomial. Por el contrario, cuando se dice que un problema es *difícil* (*hard*) o intratable, se refiere que el problema es de dureza NP. [13] . Es decir, que involucra a aquellos problemas de decisión, donde la respuesta deber ser un *si* o un *no*, y traducido a un problema de optimización, la respuesta es una solución factible que minimiza o maximiza una función objetivo dada en C.

Las clases P y NP en un problema computacional, también puede verse como una función h que mapea cada entrada x en algún dominio dado a una salida $h(x)$ en algún rango dado. Dicho algoritmo calcula $h(x)$ para cada entrada x. Ver figura III.2

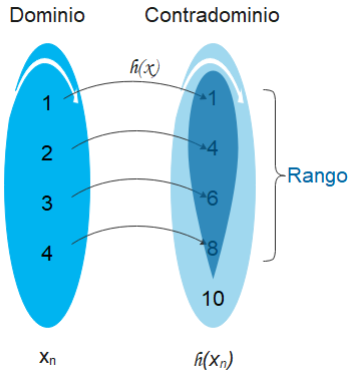

Figura III.2: Ejemplificación de la clase P y NP en una función

Los problemas P son una clase de un conjunto de problemas que pueden ser resueltos en tiempo polinomial por algoritmos determinísticos. Son fáciles de resolver, fácil de verificar, toman un tiempo computacionalmente aceptable para resolver cualquier instancia del problema. En el peor de los casos, buscar un elemento de la lista de tamaño n, requiere n comparaciones. El número de comparaciones aumenta linealmente con respecto al tamaño de entrada. Entonces la búsqueda lineal es un problema P. En la práctica, la mayoría de los problemas son problemas P, buscar un elemento en una matriz $(O(n))$, insertar un elemento al final de una lista enlazada $(O(n))$, ordenar datos usando la selección sort $(O(n)^2))$, encontrar la altura del árbol $(O(log_2 n))$, ordenar los datos usando la ordenación por fusión $(O(nlog_2 n))$. La multiplicación de matrices $O(n^3)$ son algunos de los ejemplos de problemas

P. Los problemas P también se conocen como tratables. Un algoritmo con complejidad $O(2^n)$ toma el doble de tiempo si se prueba en un problema de tamaño $(n+1)$. Tales problemas no pertenecen a la clase P.

Existen problemas que no se pueden resolver en tiempo polinomial, del cual se deben de excluir. Por ejemplo, el problema de la mochila usando el enfoque de fuerza bruta no se puede resolver en tiempo polinomial. Por lo tanto, no es un problema P.

También existen muchos problemas importantes cuya solución no se encuentra hasta el momento en tiempo polinomial, ni se ha probado que tal solución no exista. Algunos ejemplos se encuentran los problemas de: TSP (Trave-lling Sallesman Problem, Agente Viajero), coloración de gráficos, problema de partición, mochila, entre otros más.

Los problemas NP, es un conjunto de problemas que se pueden resolver en tiempo polinomial no determinista (algoritmos polinomiales aún desconocidos). La sigla NP no significa no polinomial, significa tiempo polinomial no determinista. La solución a los problemas de NP, no se pueden obtener en tiempo polinomial, pero dada la solución, se puede verificar en tiempo polinomial.

NP incluye todo el problema de P, es decir, $P \subseteq NP$ El problema de la mochila $(O(2^n))$, el problema del agente viajero $(O(n!))$, la Torre de Hanoi $(O(2^n-1))$, el ciclo hamiltoniano $(O(n!))$ son ejemplos de NP problemas. Los problemas NP se clasifican además en categorías $NP-complete$(*completo*) y *NP-Hard(difícil).*

La figura III.3 muestra la taxonomía de las clases de complejidad. Dentro de los problemas NP se encuentra el subconjunto denominado $NP-complete$, que es una clase de problemas que, dado cualquier problema NP, estos se pueden reducir polinomialmente. Y también se encuentra el conjunto $NP-Hard$, que es una clase de problemas que se clasifican tan difíciles como lo son la clase $NP-Complete$. Como se observa en la figura III.3 se muestra las clases de complejidad, empezando desde la parte inferior aquellos problemas fáciles, hasta llegar a la parte superior encontrando aquellos problemas difíciles.

Cuando se resuelve un problema desde cero, y no es posible encontrar soluciones satisfactorias, y derivado a esto no existe un algoritmo que dé solución al problema a resolver. Entonces se clasifica que dicho problema es de la categoría NP-Hard, lo que significa, con una alta probabilidad, que no hay algún algoritmo eficiente hasta el momento.

En los problemas NP-hard, en general, es muy difícil encontrar soluciones

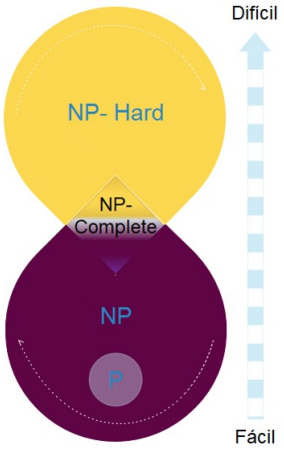

Figura III.3: Clases de complejidad

óptimas. Existen problemas que solo se pueden resolver para instancias pequeñas y con exactitud, ejemplo de estos, son los problemas de brach and bound. Pero cuando el problema ni es pequeño e intervienen instancias más grandes, es cuando se deben utilizar los algoritmos de aproximación. En este caso, se utilizan los bien llamados algoritmos heurísticos, en el cual este tipo de algoritmos, brindan soluciones factibles, pero hay una restricción, ningún algoritmo heurístico, sea cual sea el método que se utilice, ninguno va a proporcionar una solución que se garantice que sea la óptima, pero con una buena programación se puede llegar a alcanzar un valor cercano al óptimo.

Por tal motivo, el algoritmo heurístico es un problema NP-hard. Cuando se aborda un problema para encontrar su minimización o maximización, se puede estimar una posible desviación del valor objetivo óptimo si se dispone de límites inferiores o superiores. Ejemplo de esto son los problemas de optimización heurísticos de búsqueda local y búsqueda global [13].

En conclusión, un FJSSP es un problema de programación que aborda variables discretas de la clase de problema de optimización combinatoria (OC) con dificultad NP porque su solución no se puede resolver en un tiempo polinomial, pero se verifica en un tiempo polinomial, y también es NP porque aborda problemas de gran tamaño, por lo que es difícil encontrar soluciones óptimas, de ahí la necesidad de utilizar algoritmos de aproximación, aplicando 2 algoritmos heurísticos. para alcanzar un valor cercano al más óptimo.

III.1.2 Algoritmos de optimización

Los problemas Scheluding al ser problemas con gran complejidad NP-hard, en la mayoría de los casos no pueden ser resueltos mediante modelos matemáticos tradicionales precisos, por lo que se han propuesto gran cantidad de métodos exactos y algoritmos de aproximación [92].

Los problemas de optimización combinatoria (OC) se pueden resolver mediante dos formas:

1. **Algoritmos completos o exactos**
   Se garantiza que en un problema de OC encuentren para cada instancia de tamaño finito una solución óptima en un tiempo limitado. Sabiendo que los problemas OC como los scheduling problem son NP-Hard y no se pueden resolver en un tiempo polinomial, y para resolver los algoritmos exactos se resuelven mediante un tiempo de cálculo exponencial en la mayoría de lo casos, y sin mencionar que tendría gran peso computacional si se tratara de resolver por este método un problema de aplicación real a gran escala, eso significa que es poco práctico utilizar un método exacto para resolver un problema scheduling. La familia de métodos exactos es suficientemente grande, pero los más utilizados por ser los más completos son los algoritmos de branch and bound, los de programación entera mixta y por los métodos de descomposición [99].

2. **Algoritmos aproximados**
   Se enfoca en problemas complejos o de gran dimensión. Este tipo de método sacrifica la garantía de encontrar soluciones óptimas para obtener soluciones casi óptimas en un tiempo de cómputo razonable y práctico. En este enfoque se dividen en heurísticos y metaheurísticos. [99].

De acuerdo con [4] muestra una clasificación de los algoritmos scheduling, mencionando que esta clasificación no es exhaustiva y solo contiene una visión amplia de las clases de algoritmos, la cual dividen a los algoritmos scheduling en, exactos y aproximados; dentro de los exactos se encuentran los constructivos y los enumerativos; y dentro de los algoritmos aproximados se encuentran, los heurísticos constructivos, los heurísticos de memoria y los metaheurísticos, como se observa en la Figura III.4.

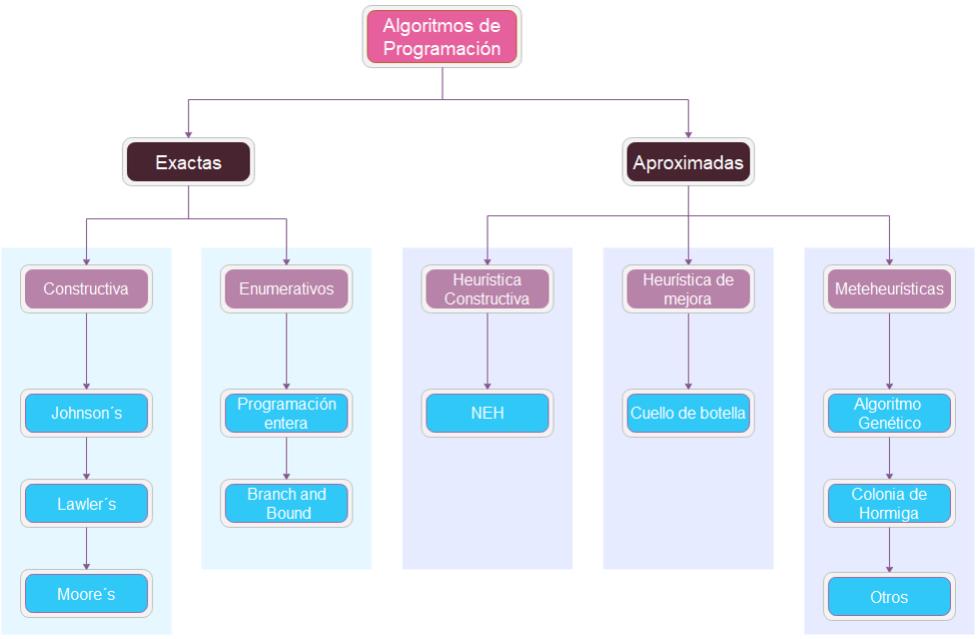

Figura III.4: Algoritmos de programación [4]

En la literatura se pueden encontrar gran número de enfoques para resolver los problemas de optimización combinatoria scheduling, unos abordan el problema como una búsqueda de solución exacta, usando programación entera lineal. Para los métodos exactos, se encuentra la programación matemática como el método, Branch and bound (la ramificación y el límite) [72], la programación lineal [64], la relajación lagrangiana [17], estos métodos matemáticos aseguran una convergencia global y han tenido grandes resultados, y han funcionado muy bien en la solución de instancias pequeñas, pero en aquellos problemas que van aumentando de tamaño, requieren un tiempo computacional muy alto [76].

1. Algoritmos exactos

   1.1 Constructivos

      1.1.1 Algoritmo de Johnson. [50].
      Es un algoritmo que va construyendo paso a paso la secuencia

de tareas que van a ser procesadas y considerando que el orden de tareas va el de menor tiempo en adelante.

1.1.2 Algoritmo de Lawler[54].
Se considera las restricciones de precedencia en los trabajos y trata de evitar retrasos. Su manera de trabajo es programar desde el último trabajo, no se consideran los que tienen precedencia. Se realiza una comparación para encontrar el trabajo que minimice el retraso. Se realiza el mismo procedimiento anterior con el penúltimo trabajo, luego con el antepenúltimo trabajo y así sucesivamente hasta programar todos los trabajos [78].

1.1.3 Algoritmo de Moore [9].
Su objetivo es minimizar la cantidad de los trabajos atrasados. Es de importante utilidad este criterio cuando existen penalizaciones por atraso, cuando no hubo el compromiso de una entrega a tiempo.

1.2 Enumerativos

1.2.1 Programación entera.
Este tipo de programas tienen una solución factible y no ilimitada, y siempre se va a obtener una solución óptima, y se conocen como problemas lineales. Este tipo de problemas tienen a ser ilimitados, es decir, para cada número real K se tiene una solución factible $x$con $z(x) < K$. Un programa lineal entero es un programa lineal en el que todas las variables x_i están restringidas a números enteros [12].

1.2.2 Branch & Bound.
Es una técnica basada en la idea de enumerar inteligentemente todas las soluciones factibles. Se descartan las soluciones no prometedoras, aplicando la estrategia de profundidad. Para descartar cierta solución, se tienen que estimar los límites superior e inferior de la cantidad a optimizar, de manera que no se exploren aquellos nodos cuya función objetivo sea menor o superior a la mejor actual. Este tipo de algoritmo hace uso de la herramienta de ramificación de ahí el nombre de Branch, en el que esta pueda dividir un conjunto determinado de candidatos en dos candidatos más pequeños, y la otra herramienta que usa es la de limitación o bounding y

esta calcula los límites superior e inferior de la función que optimizará dentro de un subconjunto determinado. Cuando se descartan soluciones infructuosas, se denomina como proceso de poda. El algoritmo para cuando los nodos de árbol realiza una poda o se resuelven [48].

2. Algoritmos aproximados

2.1 Heurística Constructiva.

2.1.1 NEM.
Es un tipo de método constructivo de secuenciación, que proporciona secuencias de calidad, en especial para el problema flow shop con el objetivo de minimizar al instante el tiempo máximo de los trabajos (Cmax).

2.2 Heurística de mejora
Es una de las heurísticas más eficientes principalmente para los problemas JSSP. De acuerdo con [99] fue una propuesta de Adam et al., y mejorada por Balas et al. Esta heurística busca minimizar la tardanza total en el job shop y se utiliza para un número finito de máquinas y trabajos, y divide una instancia con m máquinas en M subproblemas de una única máquina. Resuelve los subproblemas de forma individual y de las soluciones obtenidas las combina para obtener la solución de la instancia original.

2.3 Metaheurística
Es encontrar soluciones más allá de una solución óptima. En el siguiente apartado se hablará más a detalle.

III.1.3 Metaheurísticas

Para resolver un problema de optimización combinatoria, no es conveniente utilizar un método exacto, ya que estos se enfocan en instancias pequeñas y con pocos datos con el fin de poderlas resolver en un tiempo razonable y con un resultado óptimo, sin embargo si los datos se van aumentando, el tiempo también aumentará, por lo que para el tiempo de resolución se necesita un tiempo de cálculo exponencial y esto es poco factible debido al tiempo que tardaría en resolverse un problema de instancias grandes, ya

que los problemas reales son a gran escala y esto es la gran limitante de los problemas exactos [99].

El método ideal para aplicar en problemas combinatorios, son las técnicas de aproximación, aplicando una técnica heurística o metaheurística, ya que resuelven problemas difíciles *Hard* en un tiempo razonable, aunque se sacrifica obtener una solución óptima, por una casi óptima, pero se ganaría tiempo computacional, pero no hay garantía de obtener de nueva cuenta una solución de calidad cada vez que se ejecute el algoritmo. La forma básica de los algoritmos de aproximación se llama *heurística*, un nombre derivado del verbo griego que significa *encontrar*.

Los algoritmos heurísticos se basan en la experiencia, no tienen reglas específicas que aplicar, de ahí que estos problemas siguen en constante estudio. Para la resolución del problema se basan en el sentido común, una subclase importante de los algoritmos heurísticos son los metaheurísticos y muchas de estas técnicas intentan imitar fenómenos biológicos, físicos o naturales extraídos del mundo real [9].

La primera vez que fue introducido el término de metaheurística fue por Glover [39] y la definió como un método de mejora local que integra estrategias de alto nivel, capaz de no quedarse atrapado en un espacio de búsqueda y poder salir del óptimo local, también es capaz de explorar exhaustivamente en el espacio de búsqueda.

Para resolver un problema de optimización aplicando un heurístico, la búsqueda para obtener una buena solución se divide en dos fases [48]:

1. Construcción de la solución.
   Es el proceso que se lleva para obtener las soluciones iniciales factibles, y estas sirven de referencia o de punto de partida para continuar con la búsqueda. En esta fase se inicia con una solución vacía, y conforme avanza la búsqueda se van agregando componentes de la solución inicial a la nueva solución que se está construyendo, hasta que se genera una solución factible.

2. Mejora de la solución
   Intenta modificar gradualmente las soluciones iniciales, en función de alguna condición que se cumpla en un tiempo determinado o hasta obtener una solución de calidad.

Durante las últimas tres décadas se han estudiado las técnicas heurísticas y metaheurísticas, ya que se han vuelto más complejos, a partir de entonces se

han ido incorporando nuevos tipos de estrategias para encontrar una solución factible en un espacio de búsqueda complejo y en un tiempo razonable con el fin de evitar la convergencia en los óptimos locales. Una característica muy importante para clasificar las metaheurísticas es el uso que hacen de la historial de búsqueda, es decir, si utilizan memoria o no [8].

Otra forma más intuitiva de clasificar las metaheurísticas se basa en los orígenes del algoritmo. Entre los principales métodos se dintinguen tres tipos de búsqueda, en constructivas, evolutivas y de búsqueda local. Dentro de esta clasificación podemos encontrar los algoritmos que están inspirados en la naturaleza, como los Algoritmos Genéticos o los Algoritmos de hormigas o los de Cúmulo de Partículas; y estan los algoritmos no inspirados en la naturaleza, como la búsqueda Tabú o búsqueda local iterada, global o local, entre otros.
En la Figura III.5 muestra un tipo de clasificación de las metaheurísticas.

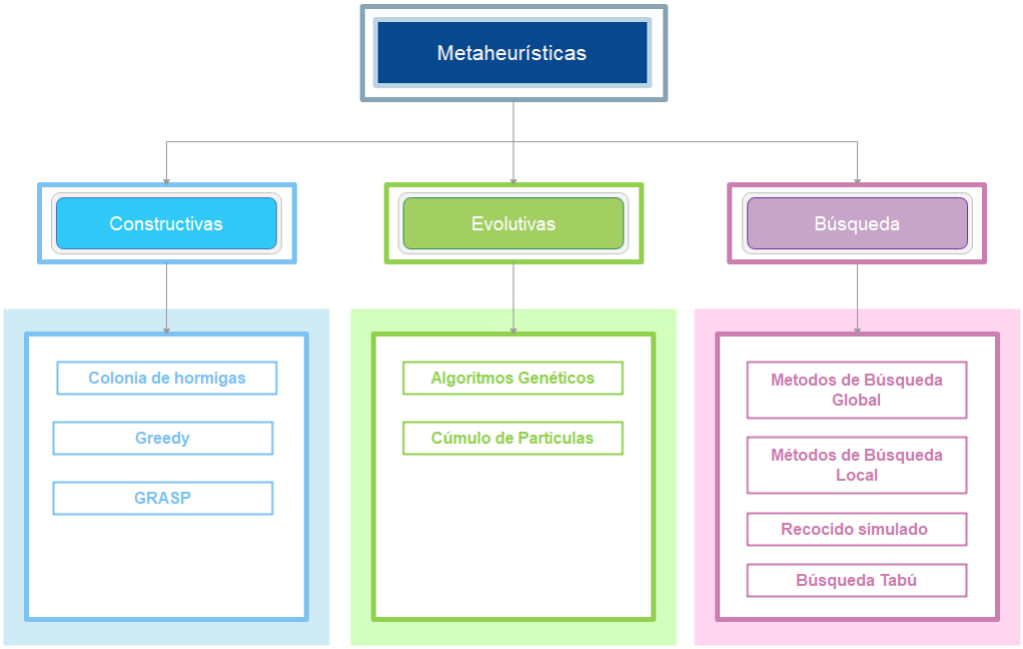

Figura III.5: Clasificación de las metaheurísticas

Basados en procesos naturales o Constructivas

Los algoritmos constructivos se destacan por generar soluciones graduales, donde parten de una solución vacía y se van agregando componentes de la solución hasta formar una solución completa. Este tipo de algoritmos aproximados son rápidos, aunque algunos métodos si generan una alta carga computacional. Este tipo de algoritmos suelen generar soluciones de calidad muy inferior, no producen buenas soluciones en problemas con muchas restricciones [99].

Entre los algoritmos basados en procesos naturales se encuentran, los algoritmos de Colonia de Hormigas, los Greddy, GRASP por mencionar los más importantes de esta categoría.

Colonia de Hormigas

La optimización de colonias de hormigas(ant Colony Optimization, ACO) es una técnica metaheurística inspirada en un proceso natural sobre el comportamiento observado de las colonias de hormigas de la vida real. Sus principios se establecieron por Dorigo et al., en 1991 [27]. El ACO combina técnicas de búsqueda local, reglas de envío (dispatching rules) entre otras técnicas pero dentro del mismo marco de referencia. El ACO es una inspiración en el comportamiento de seguimiento de rastros de las colonias de hormigas. Las hormigas cuando se mueven por un camino hacia un destino, principalmente en búsqueda de recursos alimenticios, dejan en ese mismo camino del recorrido desde el nido hasta la fuente del alimento, una sustancia química en el suelo llamada $feromona$ [65], y la dejan como señal para que otras hormigas también las sigan. Esta feromona influye en el comportamiento de las demás hormigas, ya que dependiendo de la cantidad de feromona en el camino , hay una gran probabilidad que las demás hormigas sigan el camino donde haya una mayor cantidad de feromona. Por lo que las nuevas hormigas al tomar ese camino van a colocar aún más feromonas en ese camino de recorrido, y aquellas rutas que no se visiten regularmente tienden a desaparecer y es por eso que unas rutas son más atractivas que otras por su concentración de feromona. Cuando la ruta a la fuente de destino es más corta, el rastro de feromona es mucho mayor. Por lo tanto, ese rastro de feromona que van depositando, es una forma de comunicación indirecta utilizada por las hormigas llamado *estigmergia* [48], ese sistema puede verse como agentes que interactúan para resolver tareas complejas, de

ahí la inspiración de ese comportamiento para aplicar este mecanismo natural a uno computacional de programación, dando lugar para que se aplicara en una metaheurística, llamado en ese entonces sistema de hormigas, aplicado originalmente en un vendedor ambulante, donde se ha implementado el concepto de retroalimentación positiva del seguimiento de rastros para encontrar el mejor camino posible, a lo largo de los años ha habido variaciones de este algoritmo y fue denominado finalmente como optimización de colonia de hormigas (ACO). Se ha aplicado en distintas ramas principalmente en problemas de optimización continua.

Es así, que en términos de algoritmo de optimización, el ACO considera que una colonia de hormigas va a construir de forma iterativa soluciones. Las soluciones obtenidas por las hormigas pueden no siempre ser óptimas localmente, este tipo de búsqueda se enfoca principalmente en procedimientos de búsqueda local.

GRASP

Otro tipo de metaheurística constructiva es la GRASP (Greedy Randomized Adaptative Search Procedures) o procedimiento de Búsqueda Adaptativa Aleatoria Codiciosa, este tipo de método es un representante de la búsqueda Local Explorativa, y fue propuesta por Feo y Resende en 1995 [33]. Este método es un algoritmo multi-reinicio, en donde cada paso consiste en una fase de construcción y otra de mejora. En la primera fase de se aplica una heurística de construcción y en la segunda fase se aplica una heurística de búsqueda local. En la fase de construcción se generan soluciones factibles de manera iterativa, y en cada iteración de la fase de construcción se mantienen un conjunto de elementos candidatos que se van a ir agregando a la solución parcial que se está construyendo y está determinada por la función greedy (voraz), la cual elige el elemento que da mejor resultado de manera inmediata sin tener en cuenta un panorama más amplio. Todos los elementos candidatos se evalúan aplicando una función la cual mide su mejor atractivo, cuya selección está en función del valor de su aptitud parcial de cada opción disponible. Pero en lugar de seleccionar el mejor de todos, se construye una lista restringida de candidatos RCL(restrinted candidate list), donde están contenidos un cierto número de individuos y esta es la parte greedy del algoritmo. Los candidatos se adaptan porque en cada iteración se actualizan los beneficios cuando se añade el elemento seleccionado a la solución parcial. En lugar de que se seleccione la mejor alternativa greedy, selecciona de manera

aleatoria del conjunto (RCL) el siguiente elemento que se agrega para formar parte de la solución, pero no selecciona al mejor candidato sino que para que haya una diversificación y con el fin de no repetir soluciones se construye una lista con los mejores candidatos, se va asignando una probabilidad en proporción a la función de aptitud, y ahora si de estos se elige uno al azar, y los valores de las alternativas se adaptan a la nueva situación.

En la fase de mejora se vuelve a calcular la lista de los elementos candidatos realizando una nueva iteración, siendo esta parte la adaptativa del método, estos pasos se repiten hasta obtener una solución, y se le aplica posteriormente un método de mejora con una búsqueda heurística local simple. Este proceso de construcción se repite hasta que no pueda mejorar más o hasta que se cumpla el criterio de parada.

Iterated Greedy

Este es otro tipo de heurística de construcción codiciosa conocida como Iterated Greedy. En este método se utilizan para abordar problemas de OC. Este tipo de heurísticas son muy rápidas, y generan soluciones que comúnmente son mejores que las que se generaron uniformemente al azar, o mediante una creación aleatoria pero heurísticamente sesgada. También este tipo de algoritmos se aplican frecuentemente para generar métodos de búsqueda local con perturbación, como son los algoritmos de mejora iterativa, los de búsqueda tabú, y el recocido simulado o los métodos que se basan en poblaciones como los algoritmos meméticos. Aplicar métodos de perturbación y de construcción genera mejores valores en los óptimos locales y un mejor equilibrio entre los tiempos de cálculo y la calidad de la solución.

También se pueden obtener garantías sobre la calidad de las soluciones que se puedan generar en el peor de los casos, llevando a los conocidos algoritmos de aproximación. Para los problemas de solución polinomial, se garantiza que los algoritmos greedy generen soluciones óptimas, pero no aplica en problemas NP-Hard [65].

Basados en poblaciones

Las técnicas metaheurísticas se destacan porque están basadas en poblaciones, trabajan a partir de un conjunto de individuos para representar otras soluciones. A partir de una población aplican un número de iteraciones, la población evoluciona con el fin de obtener una nueva solución con un indi-

viduo mejorado.

Dentro de esta categoría se encuentran los Algoritmos Genéticos , el cúmulo de partículas.

Algoritmos Genéticos

Los Algoritmos Evolutivos tomaron un interés importante por ser una inspiración de la naturaleza, ya que este tipo de sociedades jerárquicamente organizadas de organismos simples como hormigas, abejas y peces, tienen una gama limitada de respuestas individuales pero comportamientos colectivos interesantes, de ahí la importancia de pasar de un sistema natural a uno de programación.

Los fundamentos teóricos de los Algoritmos Genéticos (GA) fueron establecidos por Jhon Holland (1975) [46], y está inspirada en la teoría neodarwiniana de la evolución también conocida como la teoría de la selección natural, donde tiene la regla de *los más aptos sobrevivirán* donde la combinación de cruce entre individuos van a producir individuos superiores y con mejores características. Estos fundamentos se tomaron de base para desarrollarlo en el campo de la optimización, surgiendo así los AG como una técnica de optimización inteligente que puede aplicarse para encontrar soluciones óptimas en gran parte de problemas que son de gran dificultad.

Hablar de un AG es remontar la idea en simular los procesos de la evolución biológica, que es la selección natural y la supervivencia de los más aptos, ya que en la naturaleza los individuos van a competir por los recursos del medio ambiente, y lo mismo sucede en la selección de las parejas para la reproducción. Los individuos que tienen mejores características y son más aptos en rasgos genéticos sobreviven para reproducirse y producir descendencia la siguiente generación. La descendencia que se obtuvo de la reproducción va a portar material genético básico de sus padres, característica importante para garantizar su supervivencia y reproducción en la siguiente generación, conforme se van reproduciendo a lo largo de varias generaciones, el código genético favorable también se va transmitiendo a un número mayor de individuos y esa combinación que se ha dado de los individuos originales, va a generar en las poblaciones nuevas una descendencia super fit que supera hasta sus propios padres. Es así como los individuos evolucionan para adaptarse a su entorno.

Tomando la inspiración del mecanismo de la selección natural, los GA aplican la misma estrategia.

La idea principal en los algoritmos genéticos es generar una población de individuos de forma aleatoria, ya que los individuos son los que van a presentar posibles soluciones a un problema determinado. Los GA están representados por una cadena de genes que están unidos en una solución cromosómica. A cada individuo se le asigna un valor de aptitud representado por su valor de la función objetivo para evaluar su capacidad de adaptabilidad para sobrevivir y reproducirse. Los individuos más aptos tienen la oportunidad de reproducirse al ser seleccionados para la reproducción favoreciendo así a los individuos más aptos o fitness. Entendiendo que el valor de la adaptabilidad de un individuo es la información que el algoritmo utiliza para realizar la búsqueda durante un número de iteraciones (generaciones).

Es así que los mejores individuos pueden ser seleccionados varias veces en una misma iteración y, por el contrario, no se seleccionan aquellos individuos no aptos. Cuando se seleccionan a los mejores individuos, las mejores características van pasando a lo largo de varias generaciones , lo que significa que hay más oportunidad de explorar más áreas del espacio de búsqueda. Es por eso que la evolución de la población se da por medio del auto adaptación y se efectúa mediante la aplicación de tres operadores: selección, recombinación y mutación.

Entendiendo que converger significa que una población evoluciona hacia una uniformidad creciente y la aptitud promedio de la población estará muy cerca de la aptitud más alta. Entonces la población que ya fue obtenida, debe converger para obtener una solución óptima o casi óptima [48].

Particle Swarm

Es un algoritmo bioinspirado en los Algoritmos Evolutivos y es un algoritmo de optimización global. La optimización de enjambre de partículas (PSO) fue propuesto originalmente a mediados de los años 1990 por Russell C. Eberhart y James Kennedy [53] , este modelo se inspiró como una simulación del comportamiento social de organismos sociales como las bandadas de aves y los bancos de peces. El PSO representa los tipos de movimientos físicos de los individuos del enjambre. Utiliza un conjunto de soluciones, donde cada solución es representada como una partícula, y estas soluciones son las que crean el enjambre.

El comportamiento de cada partícula o individuo se ve afectado por el mejor individuo local dentro de un vecindario o por mejor individuo global respecto a todo el enjambre de la población. El PSO permite usar experien-

cias anteriores, y esta característica es única de este algoritmo evolutivo, ya que ningún otro lo maneja, también utiliza estructuras vecinas para regular el comportamiento del algoritmo.

Las soluciones se inicializan aleatoriamente en el espacio de soluciones. Los vectores principales, se consideran dos tipos, el de posición y el de velocidades, del cual estos vectores describen una partícula; el vector de posición se expresa como (x_{ij}, donde i denota la partícula ($iD1;;N, N$ es el tamaño del enjambre) y j denota la dimensión correspondiente de la partícula ($jD1;;d, d$ es la dimensión del problema)); y el vector de velocidades (v_{ij}). El rendimiento de cada partícula se evalúa en la función de aptitud predefinida ($f.x/$).

Por tanto, cada partícula se coloca aleatoriamente en el espacio $d-$ *dimensional* como solución candidata. Siendo una forma muy simple y efectiva de inicializar las partículas [65].

Las simulaciones realizadas con poblaciones modeladas de dichos organismos muestran rasgos de comportamiento inteligente y de capacidades importantes para la resolución de problemas. PSO ha ganado un amplio reconocimiento debido a su eficacia, eficiencia y su fácil implementación [65].

Basados en Búsqueda

Parten de una solución inicial y a partir de esta se realizan iteraciones con el fin de reemplazarla para obtener una mejor. De todas los tipos de metaheurísticas, la más importante es la de metaheurísticas de búsqueda, ya que estas establecen estrategias para recorrer el espacio de soluciones del problema transformando de forma iterativa la solución inicial.

Búsqueda Global

El propósito de este tipo de búsqueda era continuar o extender la búsqueda local, e ir más allá de los óptimos locales, dándose pasó a la búsqueda global. A partir de esto, en las metaheurísticas de búsqueda global surgieron tres formas para escapar de los óptimos locales:

- Volver a iniciar la búsqueda desde otra solución inicial, (Multi start). Realizan búsquedas monótonas que parten de soluciones iniciales, y el método consiste en generar una muestra de soluciones iniciales o también llamada de arranque. Se genera al azar una nueva solución de

inicio cada vez que la búsqueda quede atrapada en una solución óptima local

- Modificar la estructura de entornos que se aplica (metaheurística de entornos variables, VNS). Esta metaheurística fue propuesta por Mladenovic y Hansen (1997) [71], se basa en el cambio en la estructura del entorno y cambiarlas constantemente para escapar de los mínimos locales. El VNS obtiene una solución del entorno de la solución actual, donde ejecuta una búsqueda monótona local hasta alcanzar un óptimo local, que sustituirá a la solución actual si ha encontrado una mejora y modifica la estructura de entorno.

- Permitir movimientos o transformaciones de empeoramiento de la solución actual. Otra forma de evitar quedarse atrapados en el óptimo local es el admitir la posibilidad de pasos de no mejora, es decir, esta metaheurística propone controlar la aceptación de movimientos que no sean de mejora, para que en el transcurso, se vayan mejorando las soluciones encontradas, de todo el proceso de búsqueda se utiliza la información histórica para controlar cuando el recorrido se estuviera estancando en un mínimo local y así evitar la formación de ciclos.

Método de Búsqueda Local

Los métodos de búsqueda local están basados en la exploración de soluciones denominadas soluciones vecinas, es decir, su estrategia está basada en el estudio de soluciones del vecindario o del entorno de la solución que realiza el recorrido, se elige iterativamente la mejor de ciertas soluciones mientras exista alguna mejora posible, se inicia con una solución completa recorriendo una parte del espacio de búsqueda, examina el vecindario y la solución inicial es reemplazada, y así continúa sucesivamente el proceso hasta encontrar una mejor solución óptima local en ese vecindario, hasta que se encuentre un óptimo. Un gran inconveniente de usar este algoritmo es que una vez alcanzando su óptimo local quedan atrapadas y ya no sale de ese espacio de búsqueda.

Recocido simulado (Simulated Annealing)

Este método de optimización de acuerdo y verificable en [94] estuvo inspirado en el algoritmo de Metrópolis y fue introducido por Kirkpareick en

(1983), del cual este método está basado en el proceso físico en la disciplina de la metalurgia que viene de las ideas básicas de la física de la materia sólida. En este algoritmo se genera una solución inicial común x y una temperatura inicial T, el algoritmo empieza con valores largos para una temperatura, seguida de malas soluciones para reemplazar al titular, se examina cada solución vecina, si esta llega a ser mejor que la solución inicial, es tomada como nueva solución, de lo contrario si es peor que la inicial, se acepta como nueva solución solo con la restricción de tomarla con una determinada probabilidad.

Búsqueda Tabú (Tabú search)

Es otra técnica metaheurísticas de las más utilizadas en problemas de optimización. La búsqueda tabú (TS) se ha utilizado ampliamente para la resolución de problemas combinatorios. Este método fue propuesto por Glover (1986) [38] y por Hansen (1986) [45], pero desarrollado más tarde formalmente por Glover (1989) [39]. Su nombre deriva de la palabra Tabú que significa prohibido o restringido. Este algoritmo permite explorar el espacio de búsqueda de manera inteligente, al igual que el algoritmo SA, con el fin de escapar de los óptimos locales.

La TS tiene tres características importantes, y son que solo acepta movimientos cercanos a la solución que mejora la función objetivo. La TS antes de aplicar el criterio de reemplazo, busca encontrar la mejor solución en el vecindario actual. La última característica y la más importante es el uso de memoria a corto plazo llamada lista tabú, en la que se van registrando los movimientos de las soluciones que se han visitado recientemente durante la búsqueda. Los movimientos en la lista tabú se consideran prohibidos por la búsqueda, y esa solución no se pueden volver a visitar durante un cierto número de iteraciones. Ya que lo que se evita es quedar atrapado en una determinada región vecina oscilando entre soluciones que han sido visitando previamente, obligando así a la solución a explorar nuevas áreas en el espacio de búsqueda para escapar de los óptimos locales. El tamaño de la lista tabú normalmente es fijo, para registrar los movimientos nuevos que fueron visitados recientemente algunos movimientos antiguos son eliminados, y el tiempo de duración en la que un movimiento se declara tabú se denomina duración tabú y la estructura de la vecindad que se busca, varia de una iteración a otra.

Un inconveniente de prohibir la búsqueda de movimientos no tabú , es que puede impedir que se puedan explorar áreas de búsqueda prometedoras. Pero

ahí es cuando se implementa una estrategia haciendo uso de criterios de aspiración, donde se permite anular el estado tabú de algunos movimientos que pueden ser atractivos desde una perspectiva de búsqueda. Para determinar en ciertas soluciones el estado tabú, es común identificar alguna característica particular de la solución o movimientos vecinales como indeseables. Y si la solución recién generada contiene los aspectos anteriores, es considerada como tabú.

Es conveniente hacer uso de un mecanismo de intensificación y/o diversificación, ya que la intensificación intenta mejorar la búsqueda en torno a buenas soluciones, mientras que la diversificación trata de obligar al algoritmo a explorar nuevas áreas de búsqueda para escapar de los óptimos locales [48].

Hill Climbing

La escalada de colinas, Hill Climbing (HC), es la forma más sencilla de búsqueda local, donde la nueva solución vecina siempre reemplaza a la solución actual si es de mejor calidad. Por lo tanto, el proceso puede visualizarse como un movimiento paso a paso hacia una solución óptima localmente [48].

III.1.4 Metaheurística elegida para resolver el FJSSP

Para tratar problemas robustos con rendimientos satisfactorios y ofrecer buenos resultados dentro de un cierto tiempo, se eligió a los algoritmos evolutivos, ya que son capaces de dar buenos resultados y proporcionar una respuesta a una infinidad de problemas más complejos y en menos tiempo.

De acuerdo con Eiben y Smith [28], un algoritmo evolutivo (Evolutionary Algorithm,EA), se define como aquellas estrategias de optimización y búsqueda de soluciones que toman como inspiración la evolución en distintos sistemas biológicos. Donde se tiene una población de individuos que tiene recursos limitados, y la competencia por esos recursos provoca la selección natural, provocando el aumento de aptitud poblacional. Teniendo una función para maximizar, se puede crear aleatoriamente un conjunto de soluciones candidatas, y son los elementos del dominio de la función, y es cuando se aplica la función de calidad, como una medida abstracta y cuando mayor sea, mejor será. Y sobre esos valores de aptitud, los mejores candidatos elegidos será la próxima generación, realizado mediante la recombinación y la mutación, donde la combinación es un operador que se aplica a dos o mas candidatos seleccionados llamados padres, produciendo nuevos candidatos

llamados hijos; mientras que la mutación se aplica al candidato obteniendo uno nuevo. Por lo tanto, aplicar la combinación o mutación de un conjunto de candidatos se obtendrá una descendencia, teniendo luego que competir según su estado con candidatos mayores por un lugar para la próxima generación, y esto puede repetirse una y otra vez hasta que un candidato con una buena calidad, de una solución, o simplemente el limite computacional establecido haya alcanzado su límite, pero no necesariamente se obtuvo el mejor candidato.

Búsqueda global

Los Algoritmos Genéticos fueron los elegidos como los encargados de implementar la primera estrategia de optimización que forma parte de la búsqueda global en este trabajo, aplicando las tres funciones mencionadas anteriormente, la selección, el cruce y la mutación.

Pasos del Desarrollo para la búsqueda global:

1. Etapa de selección, será el encargado de la reproducción, se utiliza para seleccionar a los individuos según la aptitud, en este paso se debe garantizar que los mejores individuos tienen una mayor posibilidad de ser padres (reproducirse) frente a los individuos menos buenos.

   Un algoritmo genético puede utilizar distintas técnicas para realizar el proceso de selección de sus individuos, y en este trabajo se utilizan dos variantes en la operación de selección y en esta etapa se aplican dos operadores: elitista y por torneo.

   - Selección elitista. Este tipo de selección los individuos mas aptos pasarán a la siguiente generación, lo que garantiza que los mejores individuos o algunos de los mejores, sobrevivirán. Solo los dos mejores son elegidos.
   - Selección por torneo. En esta etapa se elegirán aleatoriamente un numero n de individuos, de esa selección aleatoria se elige al individuo de mejor fitness(aptitud), y este individuo formará parte de la nueva población y se repetirá el proceso n veces, y en cada repetición se ira eligiendo un nuevo individuo, hasta tener una población depurada

2. Función cruce.Un operador cruzado es seleccionado al azar (50%) para cruzar la cadena del sistema operativo, donde van a intercambiar segmentos de la población, produciendo una descendencia artificial donde los individuos serán la combinación de los padres.

3. Función mutación. Se adoptan dos operadores de mutación para la cadena del sistema operativo. Un operador de mutación es, seleccionado al azar (50%) para mutar la cadena del sistema operativo. Para finalmente tener al mejor descendiente.

Búsqueda local

En esta etapa de búsqueda local, se permite que el proceso de búsqueda explore soluciones y que no disminuya el valor de la función objetivo.

La metaurística a emplear para esta fase de búsqueda local será la escalada de colinas (Hill Climbing,HC), la cual se clasifica como un algoritmo codicioso [77]; porque, después de cada iteración del algoritmo, solo acepta una nueva solución si hay una mejora, lo que genera grandes posibilidades de obtener la solución óptima local. El concepto principal es comenzar con una solución aleatoria (probablemente pobre), y genera iterativamente pequeños cambios en la solución con el objetivo de mejor resultado. Cuando el algoritmo alcanza los criterios de parada, finaliza. Y como eso no se garantiza que el método pueda lograr el mejor resultado, se implementa el reinicio múltiple, para que se reinicie la solución y siga el proceso de búsqueda y no se estanque en un óptimo local. El algoritmo 1 representa el pseudocódigo general de un algoritmo de escalada de colinas HC.

Algoritmo 1: Descripción general de un algoritmo HC

Data: OS, MS y makespan
Result: obtener el mejor smart-cell
inicialización;
cuando después de n iteraciones no encuentra una mejora, se agrega una variable llamada s que es la solución inicial que se recibe ;
for $i = 1; \& j < numero\_iteraciones$ **do**
    for $(i = 1; i \leq n)$ **do**
        Se entra a un condicional if es el encargado de llevar las mejores permutaciones;
        if $tamOldRestart > tamNewRestart$ **then**
            De obtener un mejor makespan toma esa posición y las demás se van recorriendo. Intercambiando la mejor por la peor;
        else
            if $OldRestar > NewRestart$ **then**
                De ser menor se realiza una pequeña perturbación para evitar que se quede atrapado en un óptimo local y entonces se reinicia la búsqueda insertando aleatoriamente una nueva permutación eligiendo otra ruta crítica, generando grandes posibilidades de encontrar una solución óptima;
        Se repite hasta terminar con el ciclo for, que es el encargado del llevar el conteo del número de iteraciones del sistema ;

III.2 Presentación del algoritmo híbrido denominado: Genetic Algorithm and Random-Restart Hill-Climbing (GA-RRHC) para el FJSSP

A lo largo del tiempo se ha estado trabajando en la mejor forma de optimizar una función objetivo para problemas de programación, y esto se ha estado realizando a través de métodos de optimización con la utilización de alguna técnica heurística o metaheurística. En trabajos recientes no solo se esta utilizando un solo método de optimización, sino que se están proponiendo para la resolución de estos problemas es con la combinación con otro método de optimización con el objetivo de aumentar esa capacidad de búsqueda para alcanzar el objetivo deseado. A estas propuestas se les denomina como, algoritmos de optimización híbridos, en el cual, combinan un método de población con otro método de búsqueda local. Estos métodos han dado resultados satisfactorios, por lo que se siguen utilizando para desarrollar nuevas propuestas para disponer de algoritmos eficientes, teniendo una implementación sencilla y con una menor complejidad de ejecución.

Al utilizar métodos poblacionales se implementan distintos tipos de operadores para mejorar el proceso de optimización. En estos métodos, en particular en la parte de la población se aplica una serie de operadores de (selección) mutación e intercambio de información para mejorar el proceso de optimización. En este tipo de métodos se utilizan en serie, es decir, que cada individuo obtiene una nueva posición.

El algoritmo propuesto está basado en la idea de utilizar un vecindario inspirado en los autómatas celulares (CA), utilizados mayormente en la programación del orden de operaciones, para realizar una búsqueda global, donde la solución de un problema complejo se logra mediante la cooperación de operadores simples. Posteriormente se refina a cada solución con la búsqueda local basada en RRHC, lo que facilita la implementación computacional.

Un autómata celular (CA) es un sistema dinámico discreto formado por elementos indivisibles llamados células, donde cada célula cambia de estado a lo largo del tiempo.

Los CA tienen una gran variedad de formas, su principal propiedad es el tipo de cuadricula donde se calcula, siendo la mas simple, el tipo de línea unidimensional, cuando se tienen dos dimensiones, pueden ser las cuadrículas

cuadradas, triangulares o hexagonales. Otra forma de CA , son las cuadrículas cartesianas, teniendo en estas, un número arbitrario de dimensiones, siendo la mas común la denominada red de enteros *d-dimensional* expresada por Z^d. Es así que los CA son matrices de células(1D, 2D y 3D) como se muestra en la Figura III.6, en las que cada célula es un autómata programado de forma idéntica. La característica importante de estos sistemas es que los autómatas de la matriz interactúan con sus vecinos. El cambio de estado puede depender tanto del estado actual de cada celda como de sus celdas vecinas. Con una dinámica tan simple, los autómatas celulares pueden crear un comportamiento global periódico, caótico o complejo.[85] [68][44]

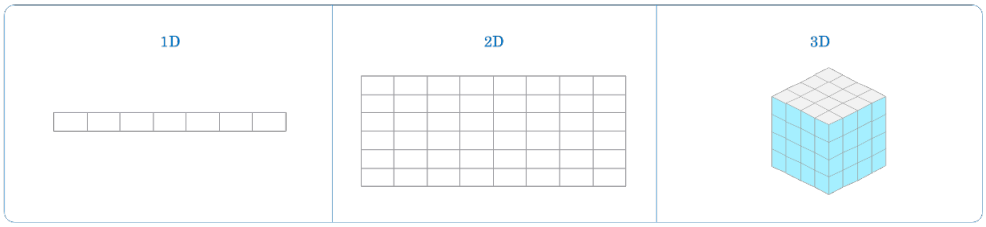

Figura III.6: Tipos de evoluciones de una CA en alguna *d-dimensión*

III.2.1 Propuesta del algoritmo híbrido basado en los GA y HC con reinicio múltiple para resolver el FJSSP

Este trabajo retoma la idea de vecindad tipo CA, como inspiración, ya que este tipo de sistema dinámico realiza cálculos complejos con la información local disponible, con el que se propone un nuevo algoritmo que optimice instancias del problema de FJSSP.

Al retomar la idea de los CA se tienen dos valores posibles para cada celda que son el cero o uno(0 o 1) y las reglas implementadas dependen de los valores del vecino más cercano, en el que retomando la idea de [85] la evolución de un CA será en función del valor de la celda a su izquierda y el valor de la celda a su derecha.

Al estar utilizando algoritmos complejos, lo que se pretende es equilibrar la solución en el espacio de búsqueda, así en cada iteración, realiza dos búsquedas, la global y una local. La búsqueda global se lleva a cabo mediante los algoritmos genéticos, que a su vez son los encargados de realizar

la explotación de soluciones en el espacio de búsqueda; posteriormente la búsqueda local, se utiliza la escalada de colinas, encargada de la exploración de soluciones. Con estas dos estrategias se consigue equilibrar la explotación y exploración para encontrar la mejor solución para el problema del FJSSP.

El algoritmo propuesto tiene un número S_n de soluciones principales o $smart\_cells$, donde cada una de ellas realiza una búsqueda global enfocada principalmente a realizar modificaciones sobre la secuenciación de operaciones , aplicando varios operadores para formar una vecindad y seleccionar la mejor modificación. Después cada $smart\_cell$ ejecuta una búsqueda local enfocada a la asignación de máquinas, y esto se realiza con la aplicación de la escalada de colinas con reinicio, utilizando un número fijo de iteraciones.

Es así que este trabajo es una nueva técnica híbrida denominada GA-RRHC, que combina dos técnicas metaheurísticas: la primera de búsqueda global utilizando algoritmos genéticos(GA) utilizados principalmente en la programación del orden de operaciones. Como segundo paso, cada solución se refina mediante la escalada de colinas con reinicio aleatorio(RRHC). Específicamente para hacer una mejor selección de máquinas de las operaciones críticas, este problema del FJSSP esta dirigido principalmente para problemas que son de alta flexibilidad, es decir, aquellos problemas donde más de la mitad de las máquinas disponibles pueden realizar todas las operaciones.

Estrategia Implementando RRHC

Como ya se menciono anteriormente, una vez que se calculó el tiempo total máximo que tarda una operación en completar su trabajo, para refinar la búsqueda, se realiza una mejor selección de máquinas mediante el uso de la ruta crítica, seleccionando de ellas las operaciones críticas y de esas n rutas, se va a elegir de manera aleatoria un camino, generando a partir de esta una nueva permutación que esta hecha a partir de la ruta crítica seleccionada, este tipo de estrategia es más conveniente para problemas con alta flexibilidad. También como estrategia al aplicar el reinicio, se utiliza para evitar una convergencia prematura de soluciones.

La importancia de la ruta critica para la RRHC, radica en el hecho que una nueva permutación es generada a partir del acomodo de la operaciones que mas se repiten en la ruta crítica, formando así una nueva cadena, del cual este método se utiliza como estrategia de optimización.

Existen una gran cantidad de métodos para implementar en un algoritmo híbrido. Y el fin de este trabajo es encontrar el mejor tiempo de

procesamiento encontrado hasta el momento. Los detalles de la propuesta se muestran a continuación.

La Figura III.7 ilustra el diagrama de flujo del método propuesto. El algoritmo 2 representa el pseudocódigo implementado en el algoritmo híbrido del GA-RRHC.

Algoritmo 2: Descripción general del GA-RRHC

Result: Mejor *smart_cell*
Conjunto de parametros del GA-RRHC;
Inicializa la población de *smart_cells* con S_n soluciones generadas de forma aleatoria;
Se calcula el makespan de cada *smart_cell* ;
Selecciona mejor smart_cell(individuo) en la población;
`/* Inicia ciclo de optimización de exploración y explotación */`
do
  Generar una nueva población seleccionando las mejores *smart_cells* por elitismo y torneo;
  Para cada *smart_cell* seleccionada, usando operadores de cruce y mutación de los GA , generar una vecindad similar al Autómata Celular(CA) para cada solución ;
  El mejor vecino reemplaza al smart-cell;
  Mejora la asignación de máquinas de operaciones críticas en cada celda inteligente aplicando RRHC;
while *(Número de estancamiento* $< G_b$ *ó número de iteraciones* $< G_b$*)*;
Regresar la *smart_cell* con el mínimo makespan;

III.2.2 Codificación y decodificación de las soluciones del algoritmo GA-RRHC

En el algoritmo 3 propuesto para el GA-RRHC, se declaran todo el conjunto de parámetros que intervienen para resolver este problema, se genera una población aleatoria de soluciones S_n denominadas celdas inteligentes (smart-cells). Para representar una solución de una instancia del FJSSP, los pasos a seguir se muestran en el diagrama de flujo, Figura III.8, que corresponde a la resolución de un FJSSP. Este tipo de problemas contienen dos subproblemas,

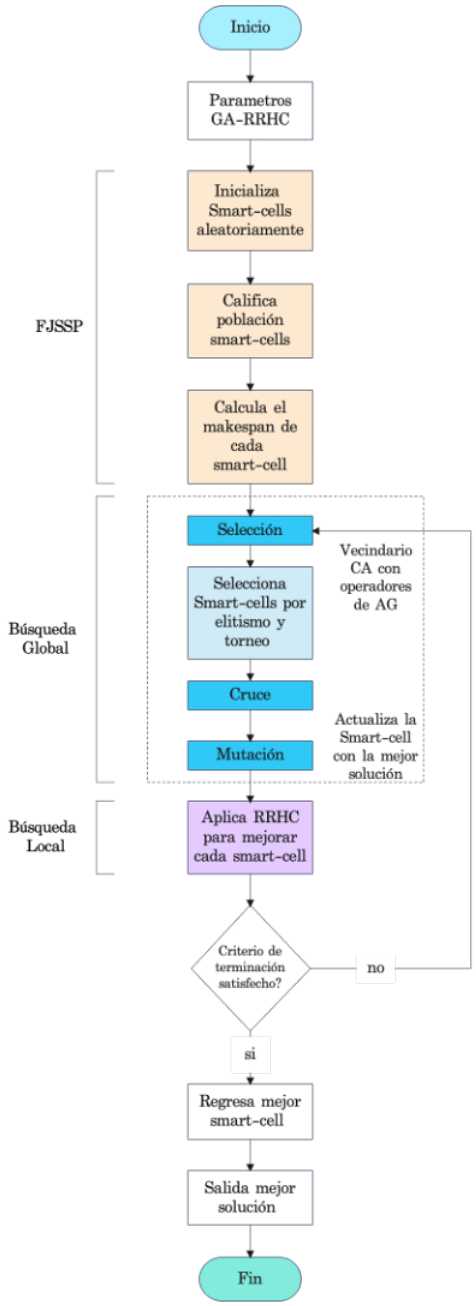

Figura III.7: Diagrama de flujo

y cada smart-cell consta de dos secuencias o cadenas, una cadena será para la selección de operaciones (OS) y otra para la selección de las máquinas (MS). Este algoritmo utiliza la decodificación descrita en [58].

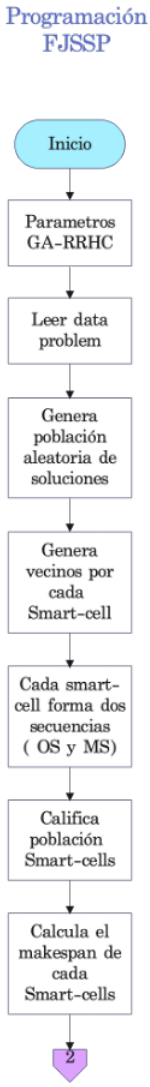

Figura III.8: Diagrama de Flujo de programación de un FJSSP

La cadena OS consiste de una permutación con repeticiones, donde cada trabajo J_i aparece n_i veces. Esta cadena OS se lee de izquierda a derecha, y la $j-ésima$ aparición de J_i indica que debe procesarse la operación O_{ij},

donde n_i es el número de operaciones que contempla el trabajo J_i. Este tipo de codificación tiene una ventaja importante en la secuencia de operaciones, y es que, cualquier permutación de la cadena con repetición OS produzca una secuenciación valida de operaciones, por lo que las operaciones usadas en este trabajo siempre arrojarán una secuenciación factible. La población inicial OS se genera en función del principio de codificación de manera aleatoria.

La cadena MS consiste en las máquinas seleccionadas para las operaciones correspondientes de cada trabajo, esta cadena también tendrá una longitud igual al número de operaciones totales. La cadena se divide en n partes, donde la *i-ésima* parte contiene las máquinas asignadas al trabajo J_i, y tiene tantos elementos como n_i. Para cada *i-ésima* parte de MS, el *j-ésimo* elemento indica la máquina asignada a la operación $O_{i,j}$.

Se resuelve el problema del FJSSP, y como primer instancia se genera la función de leer los datos Bdata del problema.

Con la primera función se inicializa la población de *smart_cells* con S_n soluciones generadas de forma aleatoria, generando la población inicial, obteniendo las primeras smart-cells con sus dos secuencias OS y MS correspondientes.

En la Tabla III.1 se muestra un ejemplo de una instancia FJSSP con tres trabajos J_i, dos operaciones por trabajo Oij, y tres máquinas M_k.

blue

Tabla III.1: Tiempos en una instancia de 3 trabajos, 2 operaciones por trabajo y 3 máquinas.

Trabajo	**Operación**	M_1	M_2	M_3
J_1	$O_{1,1}$	3	4	4
	$O_{1,2}$	1	2	1
J_2	$O_{2,1}$	2	3	3
	$O_{2,2}$	3	3	2
J_3	$O_{3,1}$	3	3	3
	$O_{3,2}$	2	2	1

Con el fin de comprender la forma en como se estructura una cadena OS y MS, para resolver el ejemplo de la instancia FJSSP de la tabla, se procede a la siguiente explicación. De acuerdo con los datos de la Tabla III.1, se observa que se tienen tres trabajos con dos operaciones cada una, a partir de

estos dos datos se parte para iniciar el proceso de optimización. Para ejemplificar la construcción de la cadena antes de su permutación se realiza de la siguiente manera. Primero, se muestra la secuencia de operaciones ordenada sin permutación como se observa en la Figura III.9 , y posteriormente se muestra la secuencia al aplicar la permutación sin repetición, Figura III.10.

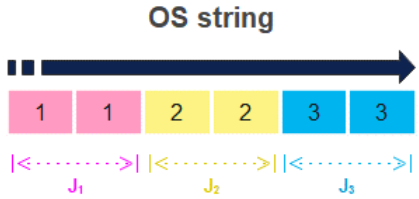

Figura III.9: Ejemplo de codificación de una cadena OS con 3 trabajos y 2 operaciones sin permutar.

Para la población inicial lo que muestra la Figura III.10, cada solución OS se genera aleatoriamente, asegurándose de que cada trabajo J_i aparezca exactamente n_i veces.

Figura III.10: Ejemplo de codificación de una cadena OS con 3 trabajos y 2 operaciones con permutación sin repetición.

Una vez que ya se tiene la secuencia OS, la forma de interpretar estos datos es leyendo la secuencia de izquierda a derecha y la *j-ésima* aparición de J_i corresponde a la operación $O_{i,j}$ del trabajo J_i . Para cada operación, se selecciona de forma aleatoria una de las posibles máquinas que pueda procesar la operación, y esa máquina seleccionada se va a asignar al elemento j de la parte i en la secuencia MS. Entonces la cadena MS inicial, se formó seleccionando una máquina al azar para cada operación de todo el trabajo. El número de operaciones no cambia en todo el proceso de búsqueda. Todo lo anterior explicado, la codificación de una smart_cell se muestra en la Figura III.11.

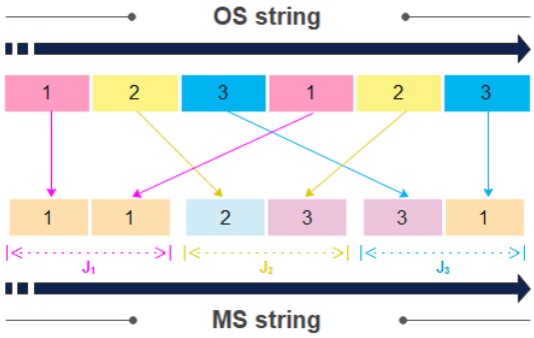

Figura III.11: Codificación de una smart-cell

La solución esta conformada por un par de cadenas OS y MS, estas a su vez forman la denominada smart_cell, del cual representa una solución para una instancia para el FJSSP, en base a la Figura III.11. Estás soluciones de acuerdo a [58] se pueden decodificar en scheduling (cronogramas) activos, semiactivos, sin demora e híbridos. Al ser el makespan un criterio regular, su decodificación en este trabajo se adopta un scheduling activo.

En este ejemplo donde se tienen 3 trabajos , cada trabajo con 2 operaciones a procesar, donde todas ellas pueden ser ejecutadas en las 3 máquinas disponibles, como se puede ver la codificación de esta *smart_cell* consiste en dos cadenas. La primera cadena OS es una permutación con repeticiones , en donde cada trabajo aparece dos veces. La segunda cadena MS contiene las máquinas programadas para cada operación , donde los primeros dos elementos corresponden a las máquinas asignadas a las operaciones $O_{1,1}$ y $O_{1,2}$, el segundo bloque de dos elementos especifican las máquinas asignadas a las operaciones, $O_{2,1}$ y $O_{2,2}$, el tercer bloque de dos elementos especifican las máquinas asignadas a las operaciones, $O_{3,1}$ y $O_{3,2}$, en tal caso de tener mas operaciones o mas trabajos así el proceso a seguir. En la Figura III.12 se muestra de manera clara como es la relación de las cadenas OS y MS en relación a su posición , trabajo y la secuencia de operaciones resultante.

Para la parte de los tiempos, se toma en cada operación $O_{i,j}$ en OS y su máquina asignada k en MS, se toma su tiempo inicial $s(O_{i,j})$ como el mayor entre el tiempo de finalización de la operación anterior $O_{i,j-1}$ y el menor tiempo disponible en la máquina k (no necesariamente después de la última operación programada en esa máquina) donde se tenga un slot o intervalo disponible de tiempo, tal que el tiempo de procesamiento $p_{i,j,k}$ sea

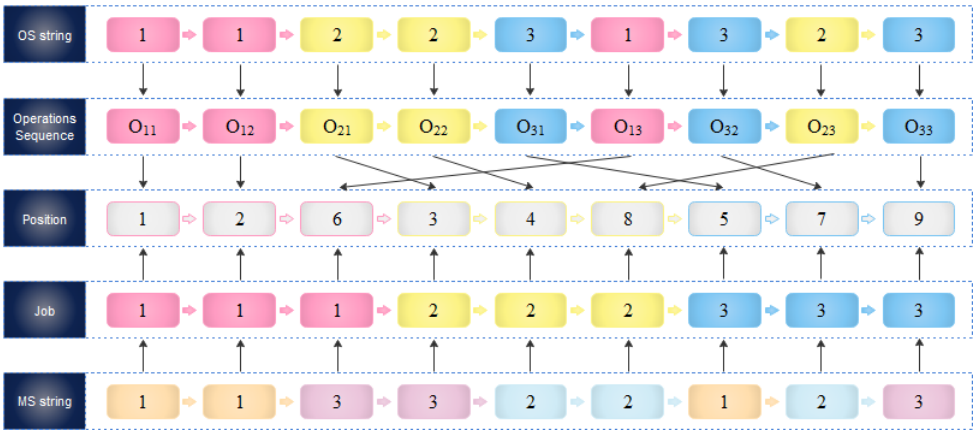

Figura III.12: Relación Cadenas OS y MS

menor o igual al tamaño del slot disponible. El tiempo en que se completa la operación O_{ij} se denomina por $C(O_{ij})$, y para $j = 1$, $s(O_{ij-1}) = 0$ para toda $1 \leq i \leq n$.

Los tiempos de procesamientos van a variar dependiendo donde se realice la operación y de su máquina correspondiente. La Tabla III.1 proporciona un ejemplo de una instancia de FJSSP teniendo tres trabajos, dos operaciones por trabajo y tres máquinas, donde todas las máquinas pueden realizar todas las operaciones.

Para la decodificación de la *smart_cell* al leer la cadena OS de izquierda a derecha al utilizar una programación activa, porque la ejecución de las operaciones no pueden comenzar antes, pero se pueden llegar a cambiar el orden de las operaciones en las máquinas si fuese necesario con el fin de evitar retrasos.

Una posible solución utilizando el orden de operaciones de la *smart_cell* se presenta en la Figura III.11, y el diagrama de Gantt correspondiente a este problema se presenta en la Figura III.13.

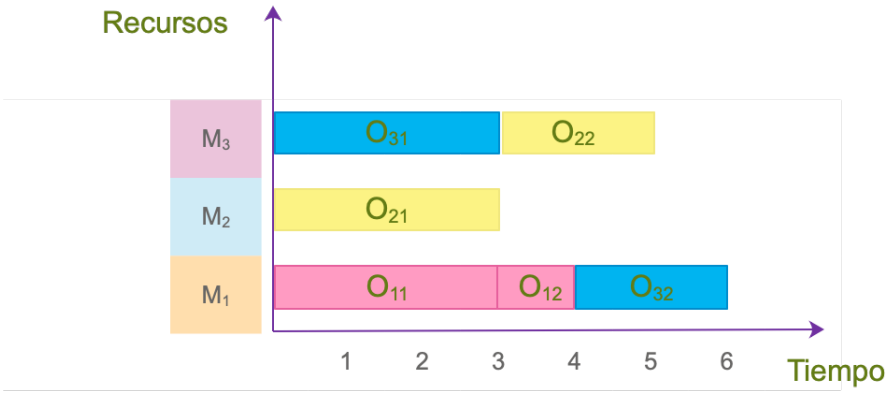

Figura III.13: Diagrama de Gantt problema 3x2

III.2.3 Búsqueda global o de exploración utilizando GA

Cada iteración del GA utilizado en este trabajo implica una etapa de selección para refinar la población de *smart_cells* (celdas inteligentes), favoreciendo a aquellas con menor rendimiento. Inspirado en el concepto de vecindad CA [68] , para cada celda inteligente, se produce una vecindad de nuevas soluciones con diferentes operadores de cruce y mutación. El que tenga el makespan más bajo se elige como la nueva celda inteligente. Para esta etapa su diagrama de flujo se presenta en la Figura III.14Los operadores utilizados para cada parte del GA se describen a continuación.

Método de selección

En el GA-RRHC, en la primera etapa, se emplean dos tipos de selección: elitismo y torneo. Estos tipos de selección, se utilizan con el fin de depurar la población que se va a mejorar, tomando en cuenta el valor del makespan de cada *smart_cells*.

1. Elitismo
   En esta función, se selecciona una proporción de soluciones elitistas denominadas E_p obteniendo las mejores *smart_cells*, es decir, con los mejores valores makespan de la población hasta ese momento, del cual esta información estará disponible para la próxima iteración del cual

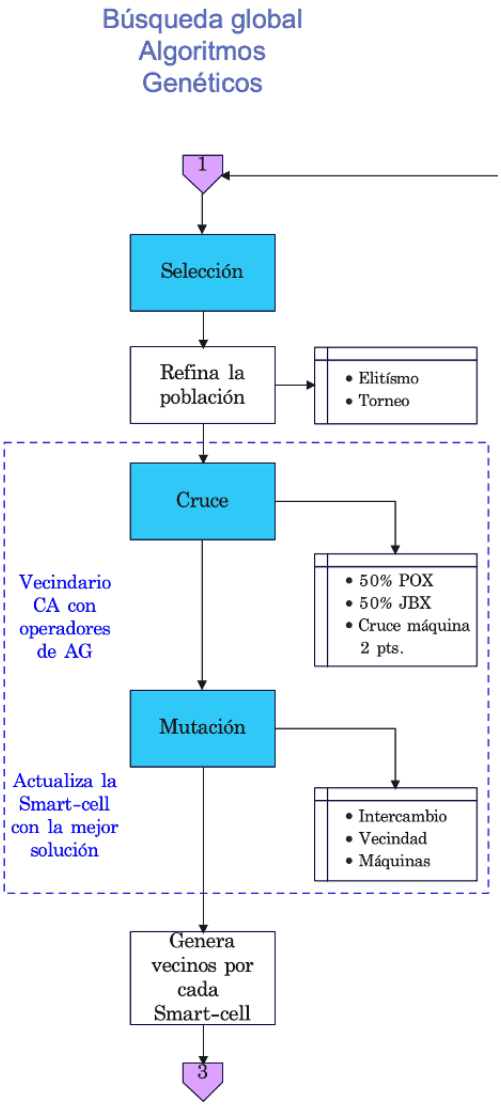

Figura III.14: Diagrama de flujo de los GA

permanecerán sin cambios para la siguiente generación del algoritmo, garantizando que su información permanezca disponible para mejorar al resto de la población.

El proceso empleado es el siguiente, primero se define una variable con el número de soluciones elitistas, en este caso $num_elitistmo = 2$;

Se selecciona sólo a los 2 mejores individuos y se guardan esos individuos en la población nueva con el menor makespan;

Se selecciona el índice de la posición que corresponda al mejor valor makespan;

Asignando esos individuos en las primeras dos posiciones a la población nueva OS , MS y makespan,

Hasta este momento, los operadores genéticos y RRHC no se aplicarán en células inteligentes de élite o elitismo, ya que pasan tal cual las soluciones elitistas para la siguiente generación.

Prosiguiendo así, a la siguiente iteración, la etapa de selección por torneo.

2. Torneo
   En esta función se utiliza una selección de torneo para seleccionar el resto de los miembros de la población $smart\_cells$.

   En esta etapa se eligen aleatoriamente un número n de individuos de un grupo de b smart_cells, se toman pares de forma aleatoria de la población actual, y compiten entre si cada par, seleccionando al $smart\_cell$ con el mejor (fitness o actitud) makespan, seleccionando así al individuo que tenga el valor mas bajo, se repite el proceso n número de veces, y en cada competencia de cada par, se va eligiendo una nueva $smart\_cell$, hasta obtener una población depurada, obteniendo así una nueva población de $smart\_cell$, pasando a formar parte de la población a mejorar en la siguiente iteración, utilizando los operadores de búsqueda global y local que se describen posteriormente.

   En este trabajo, se toma $b = 2$. Esta mezcla de elitismo y torneo permite un balance entre la exploración y la explotación de la información contenida en la población, guardando a las mejores $smart\_cells$ para no perder su información, y permitiendo que $smart\_cells$ con un buen makespan puedan proseguir en el proceso de optimización.

Operador cruce

En esta etapa, se van a adoptar 3 tipos de operadores de cruces, que se denominaron como POX, JBX y MS. El cruce de celdas inteligentes utiliza dos operadores cruzados para las secuencias del OS, cada tipo de cruce se aplica con un 50 % de probabilidad de los individuos, para obtener los mejores descendientes.

1. El primero es el operador de precedencia (POX). El conjunto de trabajos se divide en dos subconjuntos aleatorios J_A y J_B tales que $J_A \cup J_B = J$ y $J_A \cap JB = \varnothing$. Para dos secuencias OS_1 y OS_2, se obtienen dos nuevas secuencias OS_1' y OS_2'. Las operaciones de los trabajos J_A se colocan en OS_1 en el mismo orden que en OS_1. Las operaciones de J_B llenan las posiciones vacías de OS_1, manteniendo el orden de izquierda a derecha (en serie) en que aparecen en OS_2. Se lleva a cabo un proceso análogo para formar OS_2' tomando primero las operaciones de J_A en las mismas posiciones que OS_2. Los espacios vacíos de OS_2 se llenan con las operaciones de J_B en OS_1 en serie. Lo que se traduce como se observa en la Figura III.15.

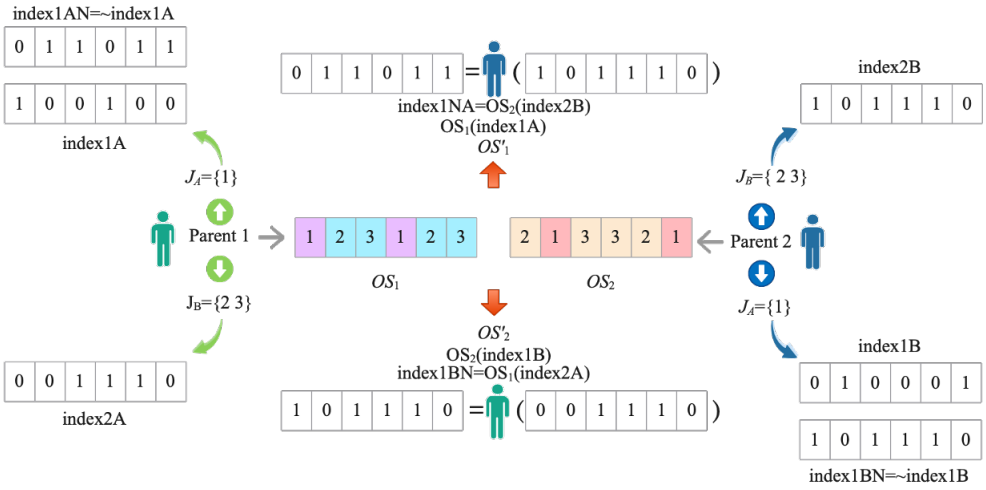

Figura III.15: Ejemplo de index (POX).

Finalmente el resultado final del cruce de operación de presendencia (POX) se muestra en la Figura III.16, donde ejemplifica el cruce de

POX con tres trabajos y seis operaciones.

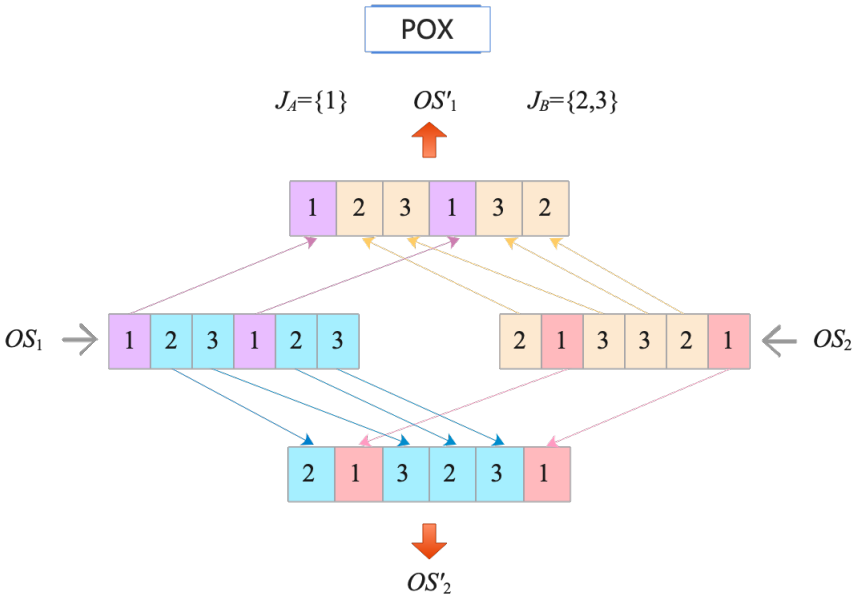

Figura III.16: Ejemplo de cruce de operación de precedencia (POX).

En esta etapa en resumen implica elegir a dos individuos para que intercambien segmentos de su código, produciendo una descendencia cuyos individuos son combinaciones de sus padres.

2. El segundo operador de cruce, es el cruce basado en trabajos (JBX). Al igual que el anterior operador, en este también se define en subconjuntos J_A y J_B. A partir de dos secuencias OS_1 y OS_2, OS_1 es obtenida de la misma forma que POX. La diferencia se encuentra en la especificación de OS_2, primero tomando las operaciones J_B en las mismas posiciones que OS_2. Lo siguiente a realizar, es llenar los espacios vacíos OS_2, y se llenan con las operaciones de J_A en OS_1 consecutivamente. En cuestión de codificación se traduce como se observa en la Figura III.18

   El resultado final del cruce basado en trabajos (JBX) se muestra en la Figura III.18, donde de igual forma se ejemplifica el cruce de JBX con tres trabajos y seis operaciones.

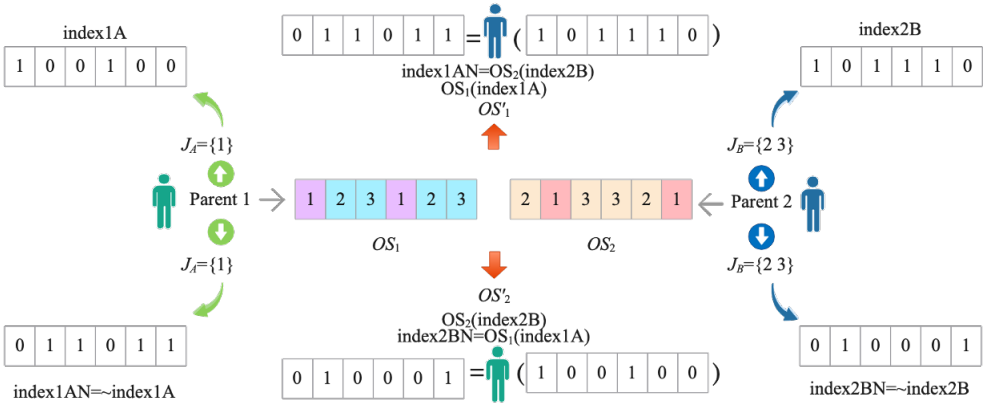

Figura III.17: Ejemplo de index (JBX).

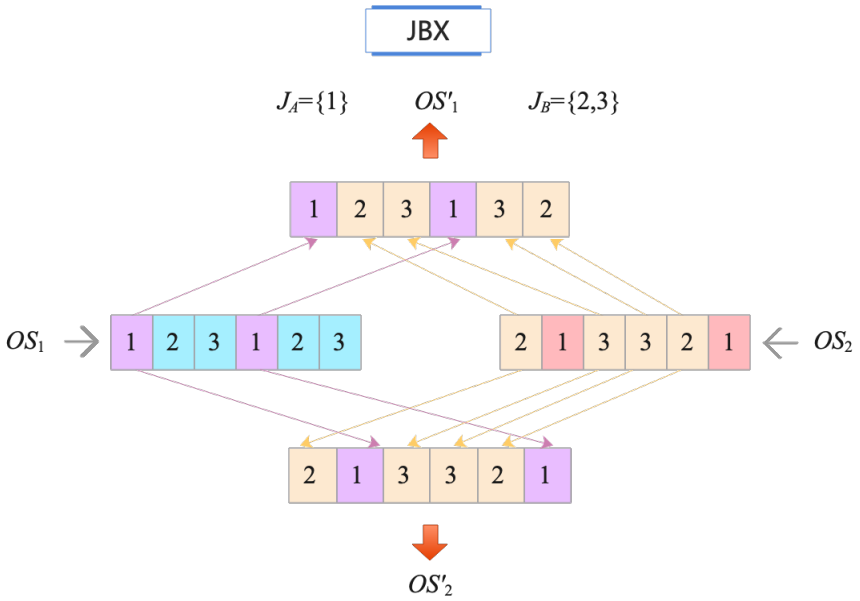

Figura III.18: Ejemplo de cruce de operación de precedencia (JBX).

3. El tercer operador de cruce es, la cadena de secuencia de máquinas MS. En este operador se aplica el cruce en dos puntos. El primer paso es seleccionar dos secuencias de máquinas factibles MS_1 y MS_2, y también se eligen dos posiciones aleatorias P_1 y P_2, donde se toma como condición que $1 < P_1 < P_2 <$ o sea escogido. Se obtiene una nueva secuencia MS_1 tomando los elementos de MS_1 en las posiciones $[1, P_1 - 1]$ y $[P_2 + 1, o]$ y de MS_2 en las posiciones $[P_1, P_2]$. De manera similar, se forma una nueva secuencia MS_2 intercambiando los roles de MS_1 y MS_2, es decir, ahora se van a tomar los elementos MS_2 en las posiciones $[1, P_1 - 1]$ y $[P_2 + 1, o]$ y de MS_1 las posiciones $[P_1, P_2]$.

   Es así como dos cadenas descendientes MS_1 y MS_2 se generaron al intercambiar todos sus elementos entre las posiciones P1 y P2 de sus padres MS_1 y MS_2. Este tipo de cruce asegura que cuando se seleccionen unos padres factibles MS_1 y MS_2, se va a tener como resultado una descendencia factible. La Figura III.19 muestra un ejemplo de cruce de dos puntos para la cadena MS_1 y MS_2 para tres máquinas y seis operaciones. Los puntos elegidos al azar fueron para P(1)=1 y para P(2)=4. Con las cadenas descendientes que se generan con el intercambio de las posiciones P1 y P2 se forma la nueva cadena MS_1 y MS_2.

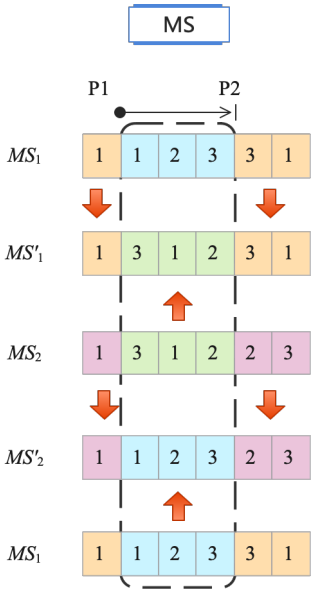

Figura III.19: Operador de cruce para cadena MS

Operador mutación

En la última etapa de la optimización genética mediante GA se encuentra la mutación de las *smart_cells*, en este operador se implementa mediante dos operadores de mutación para la cadena OS, y para cada operador de mutación se utiliza un 50% de probabilidad. En este operador se elige una probabilidad aleatoria, dependiendo el valor resultante, optará por elegir el primer o segundo operador de mutación de intercambio.

- Si resulta la primera mutación de intercambio, se seleccionan dos posiciones aleatorias de la cadena OS, de esas dos posiciones seleccionadas, se intercambian sus elementos para obtener OS.

  Como se muestra en la Figura III.20, de manera aleatoria se eligió la posición P(1)=2 y P(2)=6, del cual simplemente intercambian su valor de una posición por otra, donde la P(1) que es trabajo dos, intercambia ese valor en la P2, y viceversa; la P(2) que es el trabajo tres, intercambia su valor en la otra posición P1.

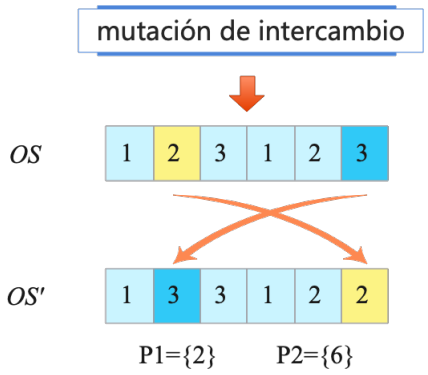

Figura III.20: Operador de mutación intercambio

- Si resulta el segundo operador de mutación, se implementa el método de mutación de intercambio por vecindad, en este operador se realiza el intercambio de posiciones aleatorias y que corresponden a trabajos diferentes.

  Se seleccionan de manera aleatoria tres posiciones pertenecientes a la cadena OS, del cual esos lugares corresponden a tres diferentes trabajos. De las posiciones seleccionadas, se intercambian aleatoriamente, y

se obtiene el nuevo OS'. Como condición al seleccionar tres elementos de OS , deben ser de tres operaciones de distintos trabajos, ya que al intercambiar sus posiciones en la cadena OS no generaría una nueva solución si las operaciones fueran del mismo trabajo. Logrando una perturbación más significativa cuando se seleccionan tres operaciones de distintos trabajos.

Este tipo de perturbación es más relevante para la etapa de exploración para poder escapar de los mínimos locales, esto es mas beneficioso cuando se presentan problemas con un número más significativo de trabajos y operaciones. Este tipo de propuesta fue aplicada por [58], obteniendo un buen resultado en su implementación.

En la Figura III.21, de acuerdo a lo explicado, las posiciones son elegidas de manera aleatoria y sin repetición. Las posiciones elegidas fueron, la P(3),P(4)y P(5). Para la P(1) toma la posición P(5), la P(4) toma la posición P(3) y la P(5) toma la posición P(4), y se debe tomar en cuenta la condición necesaria para la selección e intercambio de operaciones en cada una.

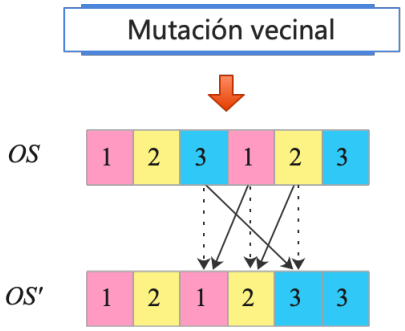

Figura III.21: Operador de mutación intercambio vecindad

El último operador de mutación es para la secuencia de mutación de la cadena MS, y se aplica una mutación a las máquinas asignadas; se elige primero aleatoriamente la mitad de posiciones de la longitud de la cadena expresada como MS $\lfloor o/2 \rfloor$, posteriormente las máquinas deben cumplir con la condición de que sean factibles y que sean diferentes a la selección inicial para cambiar así el valor de esta operación por otro valor, seleccionando del conjunto de máquinas factibles la operación correspondiente.

Un ejemplo de este operador, se tiene una cadena de 6 operaciones, de acuerdo a nuestra expresión $\lfloor o/2 \rfloor$ se redondea al entero más próximo menor que o igual a ese elemento, teniendo entonces $\lfloor 6/2 \rfloor$, en este ejemplo el número de posiciones a mutar son 3 máquinas. Siendo las posiciones aleatorias elegidas la P(2), P(3) y P(6) que mutaron por otra valor máquina. Como se aprecia en la Figura III.22

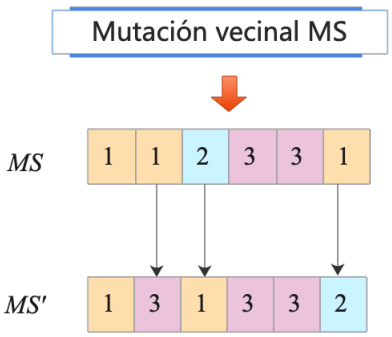

Figura III.22: Operador de mutación vecinal de máquinas

Hasta este momento se ha implementado la búsqueda global con la utilización de búsqueda mediante los algoritmos genéticos, dando paso al siguiente método de optimización.

Inspiración del vecindario tipo CA para aplicar operadores genéticos

Si un sistema se comporta de manera dinámica aplicable a problemas complejos, se pueden llegar a utilizar los AE conjuntamente con los vecindarios CA, ya que los GA son sistemas dinámicos discretos capaces de generar comportamientos globales complejos [68][85].

En diferentes estudios de optimización han aplicado en sus algoritmos el tipo de vecindad CA. De acuerdo a la revisión de la literatura no se encontró que se haya aplicado el tipo de vecindad CA conjuntamente con los GA abordando un problema de FJSSP.

Para la optimización de este problema, en la etapa de optimización, donde se aplica el algoritmo propuesto se implementa la vecindad tipo CA aplicando la técnica heurística con GA.

Como ya se menciono anteriormente el funcionamiento del GA, comienza explorando el espacio de soluciones para la búsqueda global, donde cada

smart_cell generará nuevas soluciones vecinas con el cruce y mutación. Con estas dos soluciones se selecciona la mejor de ellas con la que se haya obtenido el menor makespan, y la que resulte mejor, se actualiza la *smart_cell* original. Con la búsqueda de estos operadores genéticos se mejora la secuencia del OS de cada *smart_cell*. En la Figura III.23 se muestra el algoritmo propuesto implementando la vecindad inspirada en el autómata celular

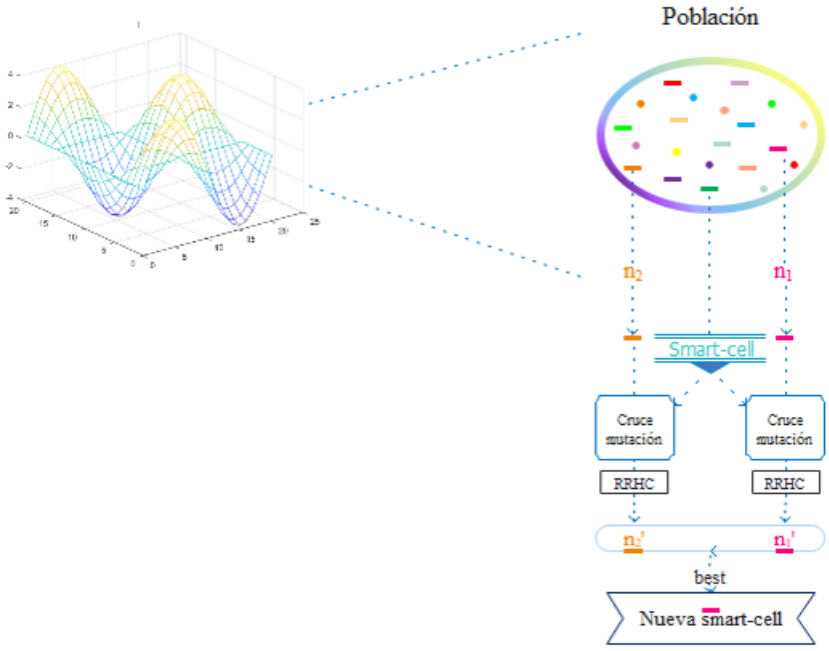

Figura III.23: Vecindad tipo CA que ejemplifica el GA-RRHC

En el siguiente apartado se explica el siguiente método de búsqueda de explotación para optimizar la secuencia MS de cada *smart_cell*.

III.2.4 Búsqueda local de explotación utilizando Escalada de colinas con reinicio aleatorio (RRHC)

La búsqueda local es la encargada de realizar la siguiente etapa de optimización para la explotación de soluciones, y la metaheurística encargada de esta búsqueda es de escalada de colinas con reinicio aleatorio (RRHC)

[77]. Con este método se pretende que con una búsqueda local explote la información de las *smart_cells*, con el fin de obtener un makespan menor. Para esta etapa su diagrama de flujo se presenta en la Figura III.24

El plan es que al utilizar cada *smart_cells* se le va a realizar pequeñas perturbaciones para mejorar el makespan. Con el RRHC se puede reiniciar la búsqueda a partir de una solución en otro punto, y este makespan a reiniciar puede tener un valor peor que el original, para después de un número dado de pasos, poder escapar de los mínimos locales. Este proceso continua después de un número determinado de pasos hasta que se concluye con un valor final de makespan.

El proceso para aplicar el RRHC primero se realiza una detección de cuales son las operaciones críticas de la *smart_cells* para seleccionar la mejor ruta. Esta selección de operaciones críticas definen el valor del makespan, en el cual las operaciones pueden estar vinculadas mediante dos caminos, ya sea por trabajo o por máquina, donde se tiene que realizar una suma de los tiempos de procesamiento, y esta suma comienza desde principio a fin , del cual no se deben contar los tiempos muertos de procesamiento entre operaciones [67], y el resultado obtenido es el makespan.

Para realizar el cálculo de makespan en la ruta crítica se lleva un registro de la tarea previa de cada operación, con el fin de conocer cuales son las operaciones críticas de una *smart_cells* este registro comienza desde el fin al inicio de la ruta, donde las operaciones iniciales de cada trabajo no tienen una operación previa.

Al realizar esta forma de registro, permite un tiempo de cómputo más rápido de las operaciones críticas, ya que se va a tomar una de las últimas operaciones donde el tiempo de finalización tiene que ser idéntico al tiempo que se esta realizando. El siguiente paso es analizar las operaciones anteriores a esa misma máquina y del mismo trabajo, donde se toma el tiempo de finalización igual a la hora de inicio de la presente operación. En caso de que se presente que esas operaciones anteriores mantengan el mismo tiempo de finalización, se procede a realizar una selección aleatoria entre esas dos operaciones previas. Este procedimiento se repite hasta que ya no se cuente con ninguna operación anterior de la última operación analizada.

En la Figura III.25 se muestra la ruta critica en el diagrama de Gantt del problema de $3x2$, es decir, tres trabajos con dos operaciones por trabajo y tres máquinas, mismo ejemplo que se ha estado analizando, y la ruta crítica resultante de esta *smart_cells* es la O_{11}, O_{12}, O_{32}. En la Tabla III.2 se muestra como se fue calculando la ruta crítica de esta *smart_cells* correspondiente.

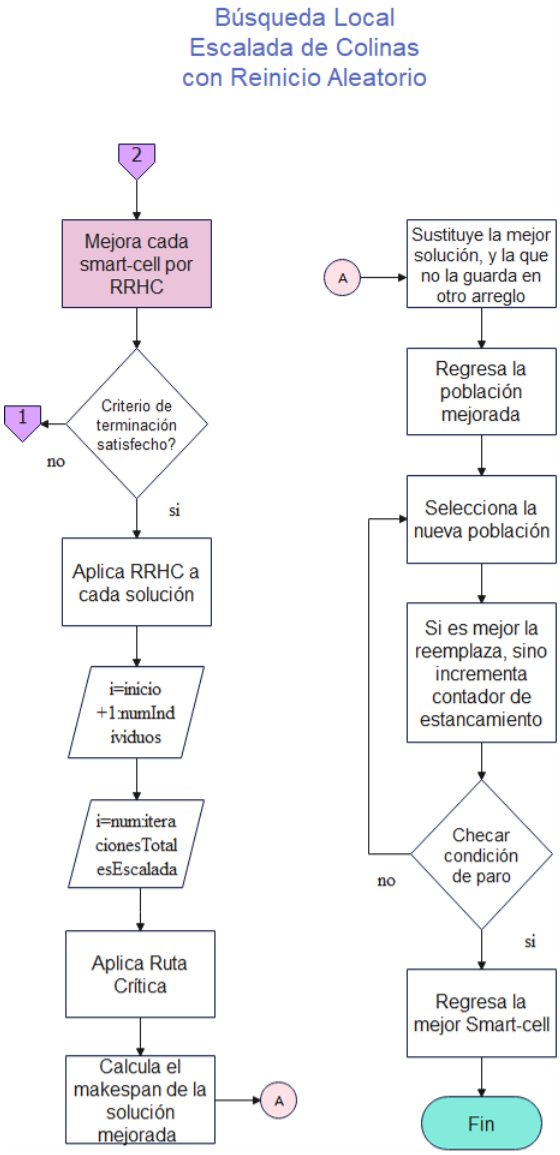

Figura III.24: Diagrama de flujo del RRHC

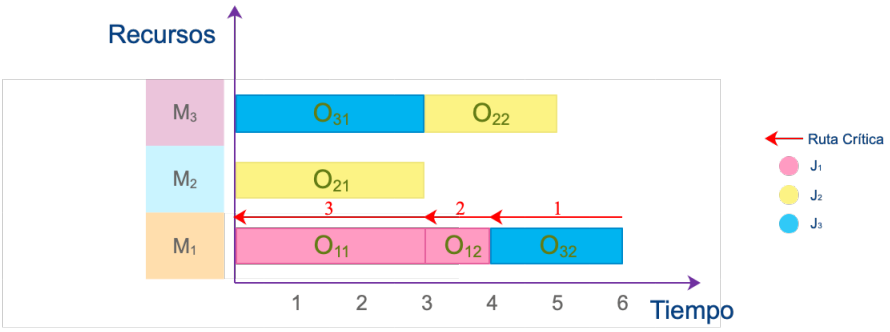

Figura III.25: Ruta crítica problema 3x2

Tabla III.2: Tiempos ruta crítica en una instancia de 3x2

Operación	Tiempo Inicial	Duración	Tiempo Cola
$O_{1,1}$	0	3	3
$O_{1,2}$	3	1	2
$O_{2,1}$	0	3	0
$O_{2,2}$	0	3	0
$O_{3,1}$	0	3	2
$O_{3,2}$	4	2	6

Aplicación de estrategia

La estrategia es implementada desde este punto para producir una mejora, es que al existir un conjunto de operaciones críticas, para refinar la búsqueda, se realiza un intercambio de máquinas haciendo uso de la ruta crítica, es decir, se selecciona de ese conjunto de n operaciones criticas, se toma una operación al azar y se elige una máquina factible diferente para así tener una nueva secuencia de máquinas MS. Pero además de seleccionar una máquina factible, se selecciona otra operación crítica con probabilidad α_c y se intercambia con cualquier otra operación en OS, al realizar este movimiento se obtiene una nueva secuencia OS y se genera una solución diferente

En la Figura III.26 se demuestra la mejora en tiempos, como se observa en la Figura III.25 el tiempo final obtenido es de 6 unidades de tiempo antes de aplicar la estrategia, y al aplicar la mejora, el tiempo final obtenido es de 5 unidades de tiempo Figura III.26 lo cual muestra que hubo una mejora de

la duración en el proceso, reduciendo el tiempo 1 unidad de tiempo, y esto se dio por el cambio de la asignación de máquina de una operación crítica.

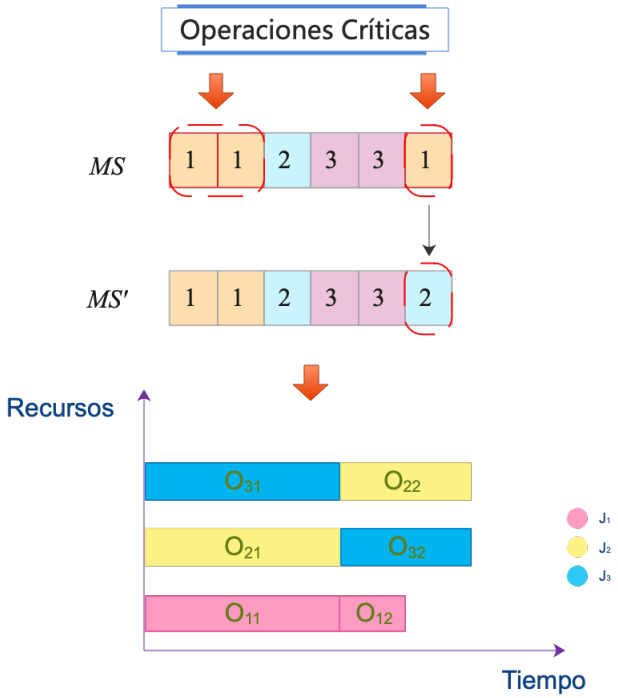

Figura III.26: Estrategia de mejora

En el Algoritmo 3, en la parte de optimización del RRHC, se aplica para iteraciones TIE es decir el total de iteraciones de la escalada, donde se tiene una pila donde se van guardando las nuevas soluciones generadas a través del proceso descrito, mientras que la TIE sea menor al reinicio de escalada TIE $<$ IRE. Si una de las nuevas soluciones mejora el makespan actual, esa solución reemplaza a la *smartcells* vaciando así la pila. Si la duración de la *smartcells* original no se ha mejorado después de la iteraciones de ITE, se toma una nueva solución aleatoria de la pila para reiniciar la escalada.

Este tipo de programación funciona mejor para instancias con alta flexibilidad, ya que el RRHC se enfoca en mejorar la asignación de máquinas.

Integración del algoritmo GA-RRHC

El algoritmo 3 presenta el pseudocódigo del algoritmo GA-RRHC, se explica a grandes rasgos lo que se realizo en cada parte de las etapas de optimización, este algoritmo es la representación del pseudocódigo del diagrama de flujo de la Figura III.7.

Algoritmo 3: Algoritmo utilizado la GA-RRHC

Result: mejor smart_cell
Conjunto de parámetros de la GA-RRHC;
Inicializa la población de *smart_cells* con S_n soluciones generadas de forma aleatoria;
Forma 2 secuencias smart_cells (operaciones OS y máquinas MS);
Califica toda población de smart_cells para obtener su makespan de cada uno ;
Selecciona mejor smart_cell(individuo) en la población;
`/* Inicia ciclo de optimización */`
while *BanderaCiclo* **do**
  `/* Búsqueda de exploración con GA */`
  *Selecciona nueva población para refinar los smart_cells por Elitismo y Torneo ;
  *Se aplica el operador de Cruce, de operaciones y máquinas;
  *Uso del operador de Mutación , de intercambio, de vecindad y máquinas;
  *El mejor vecino reemplaza la smart_cell,actualizando la smart_cell con la mejor solución;
  `/* Con el RRHC se mejora cada célula inteligente */`
  // Continúa ...

```
/* Continuación ...                                              */
while BanderaCiclo (continue ...) do
    /* Aplica búsqueda local con RRHC                            */
    *Se realiza una búsqueda poblacional de Escalada de Colinas a
     cada solución para toda la población, regresando la población
     mejorada;
    for i = num + 1 : numIndividuos do
        *Realiza el HC iniciando ciclo con el número de Iteraciones
         totales;
        for i=1:iteracionesTotalesEscalada do
            *Se entra a un condicional if donde se implementa el uso
             de la ruta crítica(RC), obteniendo las operaciones
             críticas y de máquinas;
            if reinicio==0 then
                *Se obtiene un arreglo de las máquinas factibles para
                 cada operación crítica;
            *Selección aleatoria de la operación crítica y se selecciona
             una nueva máquina factible ;
            *Selección aleatoria de un trabajo de la RC y se asigna a
             una nueva máquina;
            *Se mueve la operación crítica;
            if bandera_oc==1 then
                *Se calcula el makespan de la nueva solución de
                 acuerdo a la codificación HA;
            else
                *Se estima la duración del procesamiento para acelerar
                 el cálculo
```

```
/* Continuación ...                                              */
while BanderaCiclo (continue ...) do
    for i = num + 1 : numIndividuos (continue ...) do
        for i=1:iteracionesTotalesEscalada (continue ...) do
            /* Una mejor solución ha sido encontrada             */
            if makespan < mejorsolucionmk then
                *De obtener un mejor makespan toma esa posición;
                Se entra a un condicional if y se aplica solo si el
                 makespan fue estimado;
                if bandera_oc==0 then
                    Se calcula el nuevo makespan de la solución dada
                     de acuerdo a la codificación HA;
                Se actualiza la mejor solución, y la solución actual;
            else
                Si la duración estimada es peor se aumenta el umbral
                 de reinicio;
                Si es peor el makespan se aplica reinicio+1 y se guarda
                 la asignación de máquinas en un arreglo x
            Se reinicia el Hill-Climbing;
            if reinicio >= iteracionesReinicioEscalada then
                Si el reinicio es mayor, se toma como solución de inicio
                 , alguna solución aleatoria del arreglo x y continua el
                 ciclo ,y sustituyendo la solución actual;
        Regresa la solución mejorada;
    *Selecciona el mejor nuevo individuo de la nueva población;
    *Se actualiza la mejor solución, se entra a un condicional if;
    if nuevoMejorMakespan < mejormakespan then
        Si es mejor la nueva solución comparada con la anterior , se
         procede a actualizar y se reemplaza ;
    else
        contadorEstancamiento = contadorEstancamiento + 1;
        En otro caso incrementa el contador de estancamiento sino se
         mejora el makespan;
    *Se checa condición de paro con otro condicional if;
    *Se imprime cada 20 iteraciones;
    Se obtiene el mejor makespan;
```

Capítulo IV

Metodología y comparación

Este estudio presenta un novedoso algoritmo híbrido nombrado como GA-RRHC, para resolver un problema de taller flexible FJSSP y para problemas con alta flexibilidad. Esta basado en un método híbrido implementando dos metaheurísticas de búsqueda, se aplicó para la búsqueda global los algoritmos genéticos AG y para la búsqueda local la escalada de colinas con reinicio aleatorio RRHC.

Para las pruebas del modelo propuesto GA-RRHC, se realizó la codificación y ejecución del programa en Matlab R2015a$^{(TM)}$. Las características técnicas del equipo donde se realizaron las pruebas son las siguientes: cuenta con un sistema operativo MacOSX Ventura 13.4 de 64 bits, con un procesador Intel Xeon (R), Velocidad de procesador CPU de 2.3 GHz, con una memoria RAM de 128 GB. El codigo fuente esta disponible en Github `https://github.com/juanseck/GA-RRHC`

Para demostrar la eficacia y rendimiento de la propuesta del modelo AG-RRHC, se adoptaron cuatro conjunto de datos con instancias de referencia de problemas que han sido utilizados ampliamente en la literatura especializada. Estos conjuntos de datos tienen instancias con distintos grados de flexibilidad. Es decir, que la tasa de flexibilidad manejada se encuentra entre los valores de 0 y 1, y esto se denota como β. Cuando el resultado de la Ecuación IV.1 es alto, significa que esa misma operación puede ser procesada por más máquinas, y a esto se le llama que son problemas con alta flexibilidad.

$$\beta = \frac{\text{promedio de flexibilidad}}{\text{número de máquinas}} \tag{IV.1}$$

$$Promedio\_de\_flexibilidad = \frac{\sum \text{ máquinas totales}}{\sum \text{ total de operaciones}} \quad \text{(IV.2)}$$

Los experimentos utilizados fueron generados de los problemas *data*, establecidos en la literatura especializada. De los cuatro conjuntos de datos utilizados, se utilizan instancias, del cual se denominan como conocidas, ya que han sido utilizados ampliamente en la literatura FJSSP. Estas instancias son encontradas en la biblioteca de instancias de problemas para Casos de Problema y Estudio Computacional disponible en `https://people.idsia.ch/~monaldo/fjsp.html`.

Estas instancias sirven de base para comparar la eficacia del algoritmo propuesto, ya que al ser instancias denominadas como conocidas ya se conoce la relación del número de trabajos, así como el número de máquinas con su operación a ejecutar y su tiempo de procesamiento para cada operación.

Se toman estas instancias en especial ya que tienen una alta flexibilidad ya que tienen más de 4 máquinas por operación. Las instancias utilizadas se tomaron de los problemas $Kacem, BRdata, Rdata$ y $Vdata$.

Se tomaron las cinco instancias MK data de Kacem [51]; los diez problemas Brdata de Brandimarte [10]; y finalmente las cuarenta y tres instancias Rdata y Vdata de Hurik [49] fueron adaptadas del problema de JSSP clásico que a su vez fueron adaptaciones realizadas por Fisher y Thompson [34] (con sus tres instancias mt), y de Lawrence [55] (con sus cuarenta instancias la), estas adaptaciones las utilizó Hurik para trabajar el problema del FJSSP, en el que presenta tres conjuntos de problemas, y en cada conjunto existe un tipo diferente de flexibilidad, siendo las instancias vdata las que tienen una alta flexibilidad teniendo en promedio 4 máquinas por operación, mientras que las Rdata presentan una mediana flexibilidad, teniendo en promedio 2 máquinas por operación.

- El primer experimento toma el conjunto de datos Kacem [51], se toman cinco instancias MK, que comprende de la MK1-MK5, que van desde los 4 a 15 trabajos y de las 5 a 10 máquinas; de las cuales k1, k3, k4 y k5 tienen flexibilidad total teniendo una valor β=1, a excepción de la instancia k2 con una valor β=0.81,
- El segundo experimento se toma el conjunto de datos BRdata de Brandimarte [10], consta de 10 instancias, siendo los problemas BRdata mk01-mk10, del cual comprenden de los 10 a 20 trabajos y de 4 a 15 máquinas, en cuanto a su flexibilidad es parcial ($\beta \leq 0.35$),

- El tercer experimento se toma el conjunto de datos Rdata de Hurik [49], se toman 43 instancias Rdata, siendo el mt06, mt10, mt20 y del la01-la40, que van de los 6 a 30 trabajos y de 6 a 15 máquinas, su flexibilidad tiene un valor máximo de ($\beta \leq 0.4$),

- Y finalmente el cuarto experimento se toma el conjunto de datos Vdata de Hurik [49], se toman 43 instancias Vdata, siendo también el mt06, mt10, mt20 y del la01-la40, con sus 6 a 30 trabajos y de 6 a 15 máquinas, teniendo una flexibilidad con valor máximo de ($\beta = 0.5$)

Como ejemplo, se toma la instancia Kacem K01 donde se tienen los siguientes datos de la Tabla IV.1, la manera de interpretar estos datos es la siguiente, en la primer linea se muestra que se tienen 4 trabajos, 5 máquinas y se tiene una flexibilidad de 2, lo que significa que al menos 2 máquinas pueden realizar cada trabajo; la segunda linea indica que se tiene en el primer trabajo 3 operaciones, el siguiente valor indica el número de maquinas que va a contener la primer operación , siendo en este caso que la primera operación puede ser procesada por 5 máquinas, y los siguientes datos indican la máquina y tiempo de cada una, es decir, se deben contar en este caso 5 máquinas con su tiempo correspondiente, teniendo que la máquina 1 tiene un tiempo de procesamiento de 2 unidades de tiempo, la máquina 2 tiene un tiempo de procesamiento de 5 unidades, y así sucesivamente hasta completar el número de máquinas que se indico que tiene por operación, llegando así a la siguiente operación y esta segunda operación de este mismo trabajo puede ser procesada por 5 máquinas nuevamente, siendo la primera la máquina 1, con un tiempo de procesamiento de 5 unidades, y esta es la manera de ir leyendo los datos hasta completar todas las operaciones de todos los trabajos. De acuerdo a lo explicado los datos de la instancia MK01 se desglosan como se muestra en la Tabla IV.2.

Tabla IV.1: Conjunto de datos Kacem MK-01 [66]

```
4 5 2
3 5 1 2 2 5 3 4 4 1 5 2 5 1 5 2 4 3 5 4 7 5 5 5 1 4 2 5 3 5 4 4 5 5
3 5 1 2 2 5 3 4 4 7 5 8 5 1 5 2 6 3 9 4 8 5 5 5 1 4 2 5 3 4 4 54 5 5
4 5 1 9 2 8 3 6 4 7 5 9 5 1 6 2 1 3 2 4 5 5 4 5 1 2 2 5 3 4 4 2 5 4 5 1 4 2 5 3 2 4 1 5 5
2 5 1 1 2 5 3 2 4 4 5 12 5 1 5 2 1 3 2 4 1 5 2
```

Para el cálculo del promedio de flexibilidad se aplica la ecuación IV.2 en esta instancia se tienen los siguientes datos, un total de 60 máquinas, el

Tabla IV.2: Conjunto de datos Kacem desglosados instancia MK01

No. Operaciones	No. Máq	m	Tpo	m	Tpo	m	Tpo	m	Tpo	m	Tpo
						$O_{1,1}$					
3	5	1	2	2	5	3	4	4	1	5	2
						$O_{1,2}$					
	5	1	5	2	4	3	5	4	7	5	5
						$O_{2,3}$					
	5	1	4	2	5	3	5	4	4	5	5
						$O_{2,1}$					
3	5	1	2	2	5	3	4	4	7	5	8
						$O_{2,2}$					
	5	1	5	2	6	3	9	4	8	5	5
						$O_{2,3}$					
	5	1	4	2	5	3	4	4	54	5	5
						$O_{3,1}$					
4	5	1	9	2	8	3	6	4	7	5	9
						$O_{3,2}$					
	5	1	6	2	1	3	2	4	5	5	4
						$O_{3,3}$					
	5	1	2	2	5	3	4	4	2	5	4
						$O_{3,4}$					
	5	1	4	2	5	3	2	4	1	5	5
						$O_{4,1}$					
2	5	1	1	2	5	3	2	4	4	5	12
						$O_{4,2}$					
	5	1	5	2	1	3	2	4	1	5	2

número total de operaciones es 12, obteniendo :

$$Promedio\_de\_flexibilidad = \frac{15+15+20+10}{3+3+4+2} = \frac{60}{12} = 5$$

De acuerdo a Ecuación IV.1 se tiene una tasa de flexibilidad de :

$$\beta = \frac{5}{5} = 1$$

El cálculo de β se realiza con cada instancia de todos los conjuntos de datos con el que se obtiene el valor de β ya sea que se obtenga una tasa alta o de baja flexibilidad, y con el valor obtenido, indica qué tasa de máquinas son capaces de realizar la misma operación. Sabiendo de antemano que la tasa de flexibilidad β esta entre 0 y 1, y entre más se acerque a uno o sea igual a uno, significa que casi todas o todas las máquinas disponibles pueden realizar la misma operación; si β=0.5 significa que la mitad de máquinas pueden realizar cada operación y entre más cerca β este a 0(cero), menos máquinas pueden

procesar las operaciones disponibles, a lo que se denominan problemas de baja flexibilidad.

IV.0.1 Parámetros GA-RRHC

El algoritmo GA-RRHC ha sido concebido con nueve parámetros que son los encargados de determinar el funcionamiento del algoritmo. Los parámetros aplicados son: el total de iteraciones para todo el proceso de optimización denominado como G_n; el límite de iteraciones de estancamiento es G_b; el número de $smart\_cells$ generada es S_n; La proporción de $smart\_cells$ de élite es E_p; y l es el número de vecinos de cada $smart\_cells$ generadas por el operador genético. Se consideran también los parámetros de probabilidad dentro del algoritmo, del cual es α_m , que es la encargada de la probabilidad de mutación de una solución; Dentro de la optimización RRHC, los parámetros a considerar son: H_n como el número de iteraciones del RRHC; Las iteraciones para reiniciar el RRHC es H_r, y la probabilidad en esta etapa se considera a α_c.

Dentro de los parámetros contemplados se usaron algunos valores como referencia, obtenidos de [58], [2] y [63], del cual de ellos se tomaron los primeros tres parámetros: el número de generaciones de optimización o iteraciones de todo el proceso G_n, el límite de iteraciones de estancamiento es G_b, y el tamaño de la población o número de $smart\_cells$ es S_n.

Para el número de iteraciones G_n se probaron valores de 200 y 250, quedando como valor este último. Mientras que para S_n [63] tomó valores entre 20 y 100; [58] tomó valor de 400; y [2] tomó valor de 50; aquí se probó con el valor de 100 como nuestro número de tamaño de población $smart\_cells$.

Para G_b mientras [58] tomo como límite de 20, aquí se probó con 50. Entonces se realizó la prueba con los siguientes valores: G_n tiene el valor de 250. G_b toma valor de 50, y S_n con un valor de 100.

Al tomar estos dos trabajos de referencia recientes, se asegura que se obtenga una gran efectividad en el cálculo del makespan, ya que en sus resultados presentados muestran gran efectividad al minimizar el makespan, así como en el tiempo de ejecución de las instancias para el problema del FJSSP. Al utilizar como base los valores de estos tres parámetros ya establecidos, es debido a que se utilizan normalmente en publicaciones especializadas, y son comparables a los utilizados por el método empleado en las siguientes secciones para comparar el rendimiento de GA-RRHC propuesto.

Para el resto de los parámetros, se realiza un ajuste, y se tomaron diferentes escalas de cada parámetro. Para el parámetro l, es el número de vecinos con los que cuenta cada $smart\_cells$, de este algoritmo GA-RRHC que se utiliza para generar la vecindad de búsqueda global generado por los GA, del cual se probó con valores entre 2 y 3, con el fin de preservar una población cercana a las 250 soluciones como máximo, es decir, que es el número de $smart\_cells$ por número de vecinos, y con estos valores se asegura que se mantenga una ejecución computacional cercana a las referencias citadas.

Para la primer etapa de la formación del número de vecinos global, se probaron las combinaciones de probabilidad, tomando para E_p como la proporción de soluciones elitista, se consideran entre los valores de 0.02 y 0.04. Para la probabilidad de mutación α_m se probaron los valores de 0.1 y 0.2. Mientras que para la segunda etapa de optimización se probaron las combinaciones de probabilidad para las operaciones críticas a α_c probando los valores entre 0.025 y 0.05. Para el total de iteraciones de escalada H_n se probó con valores entre 80 y 100. Y para controlar el límite de estancamiento H_r de soluciones, se toma el valor propuesto por [58], para probar H_r como el número de las iteraciones de reinicio de escalada se tomó los valores de 30 y 40.

El método de búsqueda local aplicado, a pesar de ser un método con el que no hay tanto gasto computacional, este proceso de búsqueda en la etapa de ejecución comparado con otros algoritmos es la siguiente, en [63] se prueba una ejecución del TS de hasta 10,000 iteraciones por solución, en [59] su prueba de ejecución aumenta hasta las 80,000 iteraciones por solución. Estos dos trabajos mencionados, cabe mencionar que cada uno aplica una distinta forma de elaboración y creación de sus soluciones así como la estimación de su makespan.

En este algoritmo propuesto, se toma un punto de vista similar a [59], utilizando un número creciente de iteraciones, donde se toma el (número de individuos S_n)(número de iteraciones i) (número de generaciones G_n) para obtener un total de 2,450,000 iteraciones por $smart\_cells$ que se aplicaron en el algoritmo.

Para esta prueba de ajuste preliminar, se probó con un problema de [49], tomando la instancia de mayor flexibilidad, siendo estas instancias parte del conjunto de datos Vdata y descargado de [66], por lo que se seleccionó el problema la31 como prueba para minimizar su makespan, debido a que las instancias la31-la35, son los problemas donde se tiene mayor costo com-

putacional, lo que significa que al menos 5 máquinas en promedio pueden procesar la misma operación, estas instancias vdata constan de 30 trabajos por 10 máquinas, con un total de 300 operaciones. Para realizar el ajuste del resto de los parámetros se va seleccionar la mejor combinación de parámetros con la que se haya obtenido el mejor makespan, y se realizó la prueba de cada parámetro con los siguientes niveles de valores:

1. Para E_p se probaron los valores de 0.02 y 0.04.
2. Para l se probaron los valores de 2 y 3.
3. Para α_m se probaron los valores de 0.1 y 0.2.
4. En la etapa de optimización RRHC
   - 4.1 Para H_n se probaron los valores entre 80 y 100.
   - 4.2 Para H_r se probaron los valores entre 30 y 40.
   - 4.3 Para α_c se probaron los valores entre 0.025 y 0.05.

Como resultado del ajuste de las pruebas de los distintos niveles combinación de cada parámetro, se obtuvieron en total 64 combinaciones diferentes, ver Tabla IV.3. Para cada combinación se realizaron 30 corridas independientes, donde se seleccionó del conjunto de los seis parámetros, con el que se obtuvo el menor makespan en promedio indicado en azul.

En la Tabla IV.4, se muestran los parámetros finales del estudio de sintonización, como los valores elegidos para evaluar el rendimiento del GA-RRHC, con el cual se utilizaron para analizar los resultados para el resto de las instancias.

La Figura IV.1 muestra la convergencia del makespan aplicando el GA-RRHC, con los datos de los parámetros propuestos en la Tabla IV.4 aplicados a la instancia la31 vdata. La implementación se desarrolló en Matlab (TM) con las características técnicas descritas anteriormente.

Los experimentos computacionales se describen a detalle en la siguiente sección.

Tabla IV.3: Combinaciones del ajuste de parámetros probadas en la instancia la31.

E_p	l	α_m	H_n	H_r	α_c
0.02	2	0.1	80	30	0.025
0.02	2	0.1	80	30	0.05
0.02	2	0.1	80	40	0.025
0.02	2	0.1	80	40	0.05
0.02	2	0.1	100	30	0.025
0.02	2	0.1	100	30	0.05
0.02	2	0.1	100	40	0.025
0.02	2	0.1	100	40	0.05
0.02	2	0.2	80	30	0.025
0.02	2	0.2	80	30	0.05
0.02	2	0.2	80	40	0.025
0.02	2	0.2	80	40	0.05
0.02	2	0.2	100	30	0.025
0.02	2	0.2	100	30	0.05
0.02	2	0.2	100	40	0.025
0.02	2	0.2	100	40	0.05
0.02	3	0.1	80	30	0.025
0.02	3	0.1	80	30	0.05
0.02	3	0.1	80	40	0.025
0.02	3	0.1	80	40	0.05
0.02	3	0.1	100	30	0.025
cyan!10!white **0.02**	**3**	**0.1**	**100**	**30**	**0.05**
0.02	3	0.1	100	40	0.025
0.02	3	0.1	100	40	0.05
0.02	3	0.2	80	30	0.025
0.02	3	0.2	80	30	0.05
0.02	3	0.2	80	40	0.025
0.02	3	0.2	80	40	0.05
0.02	3	0.2	100	30	0.025
0.02	3	0.2	100	30	0.05
0.02	3	0.2	100	40	0.025
0.02	3	0.2	100	40	0.05
0.04	2	0.1	80	30	0.025
0.04	2	0.1	80	30	0.05
0.04	2	0.1	80	40	0.025
0.04	2	0.1	80	40	0.05
0.04	2	0.1	100	30	0.025
0.04	2	0.1	100	30	0.05
0.04	2	0.1	100	40	0.025
0.04	2	0.1	100	40	0.05
0.04	2	0.2	80	30	0.025
0.04	2	0.2	80	30	0.05
0.04	2	0.2	80	40	0.025
0.04	2	0.2	80	40	0.05
0.04	2	0.2	100	30	0.025
0.04	2	0.2	100	30	0.05
0.04	2	0.2	100	40	0.025
0.04	2	0.2	100	40	0.05
0.04	3	0.1	80	30	0.025
0.04	3	0.1	80	30	0.05
0.04	3	0.1	80	40	0.025
0.04	3	0.1	80	40	0.05
0.04	3	0.1	100	30	0.025
0.04	3	0.1	100	30	0.05
0.04	3	0.1	100	40	0.025
0.04	3	0.1	100	40	0.05
0.04	3	0.2	80	30	0.025
0.04	3	0.2	80	30	0.05
0.04	3	0.2	80	40	0.025
0.04	3	0.2	80	40	0.05
0.04	3	0.2	100	30	0.025
0.04	3	0.2	100	30	0.05
0.04	3	0.2	100	40	0.025
0.04	3	0.2	100	40	0.05

Tabla IV.4: Parámetros finales para ejecutar el GA-RRHC.

Parametro	Simbolo	Número
Número de iteraciones para todo el proceso de optimización	G_n	250
Límite de iteraciones de estancamiento	G_b	50
Número de smart_cells(celulas-inteligentes)	S_n	100
Proporción de células inteligentes de élite	E_p	0.02
Número de vecinos de cada célula inteligente generada por los operadores genéticos	l	3
Número de iteraciones del RRHC	H_n	100
Iteraciones para reiniciar el RRHC	H_r	30
Probabilidades		
Probabilidad de mutación de una solución	α_m	0.1
Probabilidad de mover una operación crítica en el RRHC	α_c	0.05

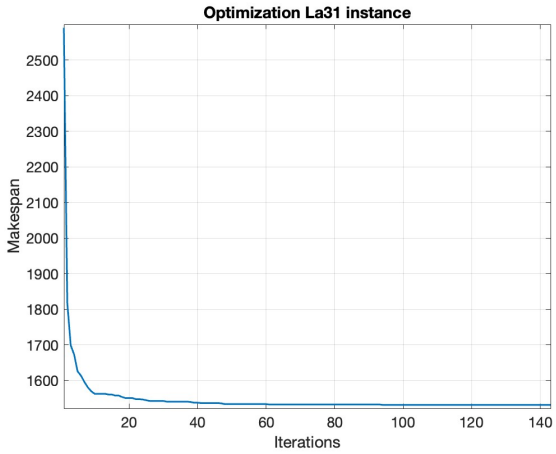

Figura IV.1: Evolución makespan del GA-RRHC aplicados a la instancia la31-vdata

IV.0.2 Análisis comparativo de la complejidad computacional entre algoritmos

Para ejemplificar la efectividad del algoritmo propuesto se comparó con otros seis algoritmos publicados entre 2016 al 2021 para medir el rendimiento de GA-RRHC. Estos algoritmos utilizados fueron:

1. El algoritmo denominado, algoritmo de búsqueda de vecindario global-local (GLNSA), desarrollado en, A global-local neighborhood search algorithm and tabu search for flexible job shop scheduling problem [30];

2. El algoritmo denominado, algoritmo híbrido (HA), del trabajo, An effective hybrid genetic algorithm and tabu search for flexible job shop scheduling problem [58];

3. El algoritmo a comparar fue con el denominado, el procedimiento de búsqueda adaptativo codicioso (GRASP), del trabajo realizado con el nombre de, Greedy randomized adaptive search for dynamic flexible job-shop scheduling (GRASP)[7];

4. Se comparó con el denominado, algoritmo de optimización de lluvia de ideas híbrido y escalada de colina de aceptación tardía denomidado (HBSO-LAHC), en su trabajo nombrado como, Hybrid brain storm optimization algorithm and late acceptance hill climbing to solve the flexible job-shop scheduling problem ([2];

5. El algoritmo con el que se comparó fue con el algoritmo denominado, Algoritmo Jaya mejorado (IJA), del trabajo Greedy randomized adaptive search for dynamic flexible job-shop scheduling [14];

6. El último algoritmo con el cual se comparó fue con el algoritmo denominado, PSO de dos niveles (TIPSO), del trabajo desarrollado como, A two-level particle swarm optimization algorithm for the flexible job shop scheduling problem [96].

Para el análisis comparativo de la complejidad computacional entre algoritmos, al ser cada algoritmo diferente, es difícil comparar entre si los tiempos de ejecución con los diferentes algoritmos, por lo tanto es inadecuado comparar dichos tiempos entre algoritmos, ya que los tiempos de ejecución de

cada uno de estos algoritmos fueron probados en diferentes arquitecturas, lenguajes y habilidades de programación, es por eso que en este trabajo se realiza una comparación de los algoritmos antes mencionados en función de su complejidad computacional ($\mathcal{O}(o)$), con respecto al número total de operaciones (ver Ecuación IV.3), tomando como referencia el número total de operaciones a procesar para una instancia del FJSSP y no por el tiempo de ejecución.

$$\sum_{i=1}^{n} \sum_{j=1}^{n_i} O_{i,j} \tag{IV.3}$$

Para generalizar la forma de medir la complejidad de cada algoritmo, se utiliza una notación estándar para todos los algoritmos a comparar, y esta notación esta definida para los parámetros del algoritmo GA-RRHC.

Para usar esta notación en común para todos los algoritmos, se indica como sigue:

1. El número de iteraciones se denotará por G_n,
2. El número de soluciones que maneja cada algoritmo se define como X,
3. El número de iteraciones del método de búsqueda local se denota por H_n
4. El número de máquinas se denotará m.

Los algoritmos inspirados en una vecindad de tipo autómata celular (GA-RRHC y GLNSA) cumplen que el número de soluciones es $X = S_n l$, donde l es el número de vecinos en la búsqueda global y para la búsqueda local se aplica cada algoritmo a H_n como el número de iteraciones.

De los seis algoritmos abordados, su comparación metodológica es la siguiente:

1. El algoritmo GLNSA, realiza una selección elitista de soluciones, posteriormente una búsqueda de vecinos l, la cual esta basada en tres operaciones clásicas de problemas combinatorios que son, la inserción, intercambio y el path-relinking. Se selecciona el mejor del vecindario y se actualiza el $smart\_cells$. Se aplica la mutación de la máquina en el conjunto de $smart\_cells$ S_n. Y finalmente para la optimización con la

búsqueda local se aplica la TS solo en las *smart_cells* S_n. Dado que $S_n < X$, la GLNSA realiza menos cálculos para la búsqueda local.

2. El algoritmo HA, tiene una arquitectura parecida al algoritmo GLNSA, ya que primero hace uso de los algoritmos genéticos, aplicando 4 operadores X a cada solución, para posteriormente aplicar una optimización con TS para seleccionar diferentes máquinas que se sustenta en el cálculo de las operaciones criticas, así como en la selección de diferentes máquinas para cada operación.

3. El algoritmo GRASP, construye un diagrama de Gantt optimizado, para posteriormente aplicar una búsqueda local greedy H_n, y este producto va a ser cuadrático con respecto al número de operaciones o . Siendo el algoritmo con mayor complejidad computacional.

4. El algoritmo HBSO-LAHC, utiliza un clustering (agrupación) de soluciones, donde se requiere calcular que distancia hay entre las mismas, para posteriormete aplicar 4 estrategias X y 3 vecindades para cada solución, y con una de esas soluciones en operaciones críticas. Para la búsqueda local en cada solución se aplica una escalada de colinas (hill-climbing) con aceptación tardía (Late Acceptance) para H_n pasos

5. El algoritmo IJA, es un modified Java algorithm, que aplica 3 operadores de intercambio a cada solución X y una búsqueda local H_n basada en el intercambio aleatorio de bloques de operaciones críticas

6. El algoritmo TIPSO, utiliza 2 estrategias X aplicando modificaciones al algoritmo PSO para optimizar el problema de enrutamiento, donde la primera etapa de modificación es con respecto a la secuencia de operaciones, con el fin para mejorar dicho orden, y dentro de esa misma etapa, en cada iteración de este primer PSO, se optimiza el problema de asignación de máquina aplicando el otro PSO.

7. Este último algoritmo del presente trabajo GA-RRHC, tiene una arquitectura parecida al GLNSA, haciendo uso de los operadores genéticos para resolver el problema de enrutamiento, generando una población aleatoria S_n produciendo una vecindad de nuevas soluciones X con operadores de cruce y mutación. Posteriormente se utiliza como número de iteraciones H_n a la Escalada de colinas como búsqueda local, para mejorar la asignación de máquinas de operaciones críticas, generando

una nueva población aleatoria de soluciones, y se repite hasta que se cumpla que el número de estancamiento (H_r)< límite iteraciones estancamiento G_b) < Número iteraciones Total (G_n).

La Tabla IV.5 presenta los algoritmos utilizados para la comparación de este trabajo, mostrando su notación simbólica de cada algoritmo de su complejidad computacional antes descrita. Se ordenaron de menor a mayor en función de su complejidad, donde se tomo como referencia a $S_n < X < H_n$, dado que los algoritmos que utilizan una búsqueda local, usualmente H_n tiene un valor alto, y esto es para obtener una mejor búsqueda local y obtener mejores resultados.

Es importante mencionar que el algoritmo GA-RRHC y el GLNSA como se puede observar en la tabla, tiene una complejidad computacional competitiva, en comparación con los algoritmos recientemente propuestos y de última generación mostrados en el estad del arte para el FJSSP.

Tabla IV.5: Algoritmos utilizados en los experimentos y su complejidad computacional.

Algoritmo	Complejidad	Jerarquía
TlPSO	$\mathcal{O}(o(G_n(2X + G_n 2X)))$	1
GA-RRHC	$\mathcal{O}(o(G_n(S_n + X + S_n H_n m)))$	2
GLNSA	$\mathcal{O}(o(G_n(S_n + X + S_n H_n m)))$	2
IJA	$\mathcal{O}(o(G_n(3X + X H_n m))$	3
HA	$\mathcal{O}(o(G_n(4X + X H_n m)))$	4
HBSO-LAHC	$\mathcal{O}(o(G_n(4X + X H_n m)))$	4
GRASP	$\mathcal{O}(o^2(G_n * H_n))$	5

Es relevante destacar que este análisis solo considera la complejidad computacional necesaria para manipular y modificar la programación, como el enrutamiento de las secuencias de operaciones y la asignación de máquinas de una instancia FJSSP. La complejidad computacional para el cálculo para obtener el makespan es $(\mathcal{O}(o)^2)$ y la estimación del makespan considerando sólo a el seguimiento de las operaciones críticas, esta limitada por $(\mathcal{O}(o))$, de la cual, estos dos procesos no están contemplados en el análisis presentado, ya que los algoritmos que se consideraron para la comparación, todos ellos utilizan estas operaciones. Así que, este análisis solo se enfoca en el estudio del proceso computacional, es decir, la forma de qué es lo que hacen diferente en cada método y qué es lo que distingue a cada algoritmo.

Capítulo V

Resultados Experimentales

En este capítulo se presentan los resultados experimentales obtenidos con la aplicación del algoritmo GA-RRHC. El objetivo de este estudio experimental es comparar la eficacia del algoritmo desarrollado para alcanzar la función objetivo de este trabajo que es la optimizacion del FJSSP y alcanzar la minimización del makespan, del cual es comparado con los seis algoritmos mencionados en el capítulo anterior y así probar la eficiencia del algoritmo propuesto.

Para la prueba de los experimentos se utilizaron los conjuntos de datos de referencia, para el primer experimento se aborda las instancias de Kacem [51] [52], para el segundo experimento las instancias de Brandimarte [10], y por último el tercer experimento se probo con las instancias Hurink [49]. Para probar el experimento de Hurink se utilizaron dos conjuntos de datos, siendo uno de baja flexibilidad (rdata), y el otro de alta flexibilidad(vdata). Estos cuatro conjuntos de datos hacen un total de 101 instancias diferentes para probar y comparar la eficiencia del GA-RRHC.

Para los benchmark mostrados para las instancias de Kacem, casi todos los algoritmos referenciados presentan resultados, pero solo el algoritmo GLNSA, IJA presentan todos sus resultados completos en todas las instancias. Los benchmark de las instancias de Brandimarte, todos los algoritmos indicados muestran sus resultados. Por último los benchmark de Hunrik los algoritmos que presentan sus resultados completos son el GLNSA, HA y IJA.

V.0.1 Primer experimento: instancias Kacem dataset

En todos los experimentos desarrollados en este trabajo, en cada uno de los métodos que se seleccionaron para realizar la comparación, fueron probados con los mismos conjuntos de datos que los citados en las referencias. Para cada instancia, se realizaron 30 corridas independientes, y de ese conjunto de ejecuciones se tomo por cada instancia el menor makespan obtenido.

En la Tabla V.1 se muestran los resultados obtenidos del GA-RRHC, donde se indica la instancia, el número de trabajos n, el número de máquinas m, la tasa de flexibilidad β de cada instancia, así la comparación con los algoritmos que presentan resultados para el benchmark Kacem.

En este conjunto de datos Kacem dataset, solo los algoritmos GLNSA e IJA reportan resultados completos, el algoritmo HBSO-LAHC no reporta resultados, por lo tanto no se reporto en la tabla, para el resto de los algoritmos, la mayoría presenta resultados, a excepción de contadas instancias carentes de resultados.

Una característica de este conjunto de datos, es por sus problemas que tienen flexibilidad total, comienzan desde instancias con un bajo número de trabajos y máquinas, hasta llegar a una mayor dimensionalidad, mas trabajos y mas máquinas.

Tabla V.1: Resultados para el Kacem dataset.

Instancia	n × m	β	GA-RRHC	GLNSA	GRASP	HA	IJA	TlPSO
K1	4 × 5	1.	11	11	–	–	11	11
K2	8 × 8	0.81	14	14	14	14	14	14
K3	10 × 7	1	11	11	–	–	11	–
K4	10 × 10	1	7	7	7	7	7	7
K5	15 × 10	1	11	11	11	11	11	–

Los resultados reportados en este experimento Tabla V.1 , muestran en la Figura V.1 su comparación de los resultados obtenidos en cada modelo, donde el GA-RRHC obtiene los mejores resultados conocidos del valor de makespan en todas las instancias, al igual que el GLNSA y el IJA.

Este experimento confirma el rendimiento satisfactorio del GA-RRHC para problemas con alta flexibilidad β . La Figura V.2 muestra el número de los mejores resultados obtenidos por cada algoritmo, obteniendo para el GA-RRHC 5 de 5 de los mejores valores de makespan conocidos .

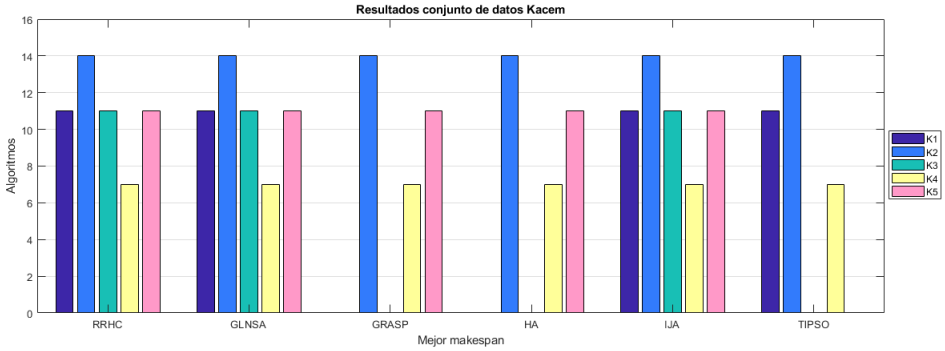

Figura V.1: Resultados Instancias Kacem

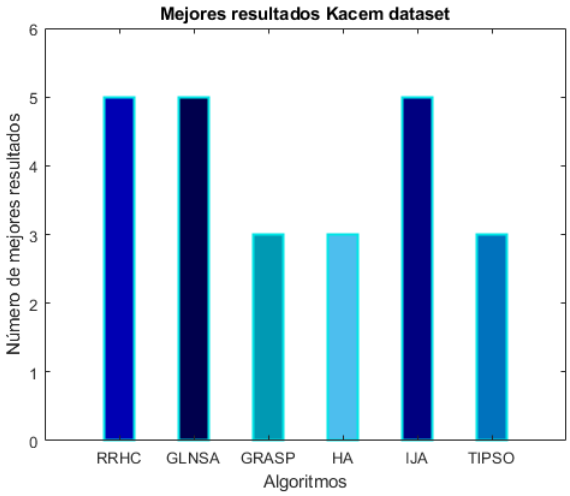

Figura V.2: Número de mejores Instancias Kacem

V.0.2 Segundo experimento: instancias Brandimarte dataset

A partir de este experimento se aplicó una comparación estadística con el algoritmo GA-RRHC con respecto a los demás algoritmos. Para evaluar la diferencia entre dos mediciones se aplicó la desviación porcentual relativa o the relative percentage deviation (RPD) para calcular la estimación de precisión, y con estos datos poder aplicar un diagrama estadístico y visualizar su distribución de la variable del makespan y visualizar su dispersión de datos respecto a su mediana y sus rangos; también se va a aplicar una prueba estadística para realizar el análisis de datos que se calcula a partir de los datos de muestra, y esta prueba es la estadística no-paramétrica de Friedman, utilizada para comparar el rendimiento del GA-RRHC con los otros algoritmos [25].

El RPD es definida en la Ecuación V.1, donde BOV es el menor makespan obtenido por cada algoritmo, y BKV el mejor valor conocido del makespan para cada instancia, y los valores mejor reportados son tomados de [19]

$$RPD = \frac{BOV - BKV}{BOV} \times 100 \tag{V.1}$$

Los resultados del GA-RRHC y su comparación con los otros métodos para el dataset BRdata se muestran en la Tabla V.2, donde el menor makespan obtenido por los algoritmos en cada instancia se marca con un $*$.

En la tabla también se presenta la tasa β de cada instancia que varia de 0.15 a 0.35, lo que significa que varia de entre un 15 a 35% del total de máquinas disponibles que pueden realizar la misma operación. Por lo tanto en este dataset todas las instancias tienen flexibilidad parcial.

Tabla V.2: Resultados para el BRdata dataset.

Instancia	n × m	β	cyan!10!whitewhite BKV	BSO-LAHC	GA-RRHC	GLNSA	GRASP	HA	IJA	TlPSO
MK01	10 × 6	0.2	cyan!10!white36	40∗	40∗	40∗	40∗	40∗	40∗	40∗
MK02	10 × 6	0.35	cyan!10!white24	26∗	26∗	26∗	26∗	26∗	27	26∗
MK03	15 × 8	0.3	cyan!10!white204	204∗	204∗	204∗	204∗	204∗	204∗	204∗
MK04	15 × 8	0.2	cyan!10!white48	60∗	60∗	60∗	60∗	60∗	60∗	60∗
MK05	15 × 4	0.15	cyan!10!white168	173	172∗	173	172∗	172∗	172∗	173
MK06	10 × 15	0.3	cyan!10!white33	61	58	58	64	57∗	57∗	60
MK07	20 × 5	0.3	cyan!10!white133	141	139∗	139∗	139∗	139∗	139∗	139∗
MK08	20 × 10	0.15	cyan!10!white523	523∗	523∗	523∗	523∗	523∗	523∗	523∗
MK09	20 × 10	0.3	cyan!10!white299	307∗	307∗	307∗	307∗	307∗	307∗	307∗
MK10	20 × 15	0.2	cyan!10!white165	204	198	205	205	197∗	197∗	205

blue

En este experimento el rendimiento obtenido del GA-RRHC a sido eficaz para problemas con flexibilidad parcial. De acuerdo con los datos de la Tabla V.2 este algoritmo calculó el mejor makespan en 80% del total de las instancias, al igual que el algoritmo GRASP, obteniendo en ambos casos los mismos resultados óptimos en 8 de 10 de instancias de este dataset, solo por detrás de los algoritmos de IJA y el HA, que alcanzaron 9 y 10 de efectividad respectivamente.

En la Figura V.3 muestra gráficamente el número de los mejores resultados obtenidos para cada algoritmo para este conjunto de datos.

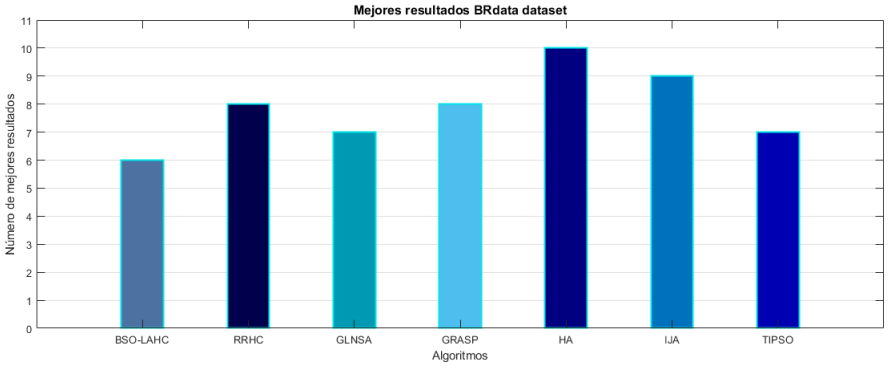

Figura V.3: Número de mejores Instancias BRdata

Para visualizar de una manera simple la distribución de las variables numéricas de cada algoritmo comparado, se utiliza la herramienta estadística boxplot conocida también como diagrama de caja, donde esta herramienta se utiliza para comparar diferentes experimentos y visualizar la dispersión de sus datos entre varios experimentos. En este tipo de diagrama de caja se observan los valores de los cuartiles, su mínimo y máximo, así como su mediana de cada algoritmo.

En la gráfica de la Figura V.4 se observa como se distribuyen los valores de los datos de su función costo de cada algoritmo de sus 10 instancias para este benchmark. Lo que significa que los makespan de las instancias están entre 40 y 300 u. por algoritmo, lo que este diagrama de caja también facilita la comprensión si su distribución es simétrica o asimétrica. En este caso su distribución es asimétrica negativa, ya que sus datos tienden a concentrarse en la parte superior de la distribución. Cuando se tiene que un lado de la caja esta más amplio que el otro significa que los datos del makespan están mas dispersos, y por el contrario, si un lado es menos amplio significa que

los datos de makespan están muy próximos. Lo que también se observa que este algoritmo muestra valores muy próximos, comparados con los otros algoritmos demostrando su competitividad.

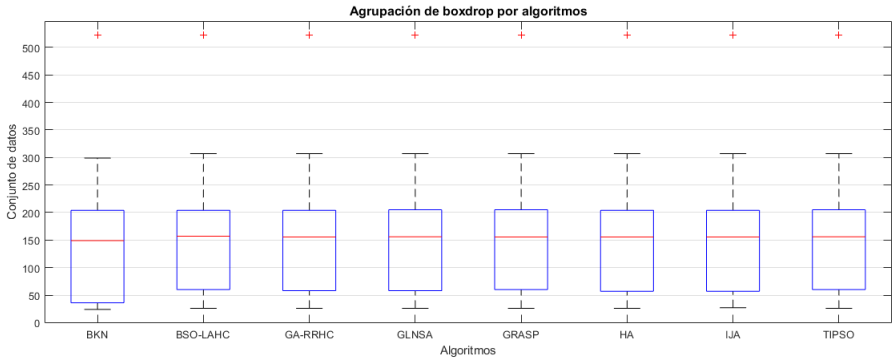

Figura V.4: Agrupación de algoritmos en diagrama de caja instancias BRdata

Aplicando la Ecuación V.1, los valores obtenidos de la desviación porcentual relativa RDP se muestran en la Tabla V.3, y estos datos es la diferencia porcentual entre dos mediciones. Aplicando con el conjunto de datos del RPD se comparan los 7 diferentes algoritmos mediante su mediana de cada uno de los algoritmos, y así visualizar la variable numérica que se esta midiendo de los diferentes grupos, que en este caso es el makespan.

Tabla V.3: Valores RDP de los resultados del experimento de los 7 algoritmos

Algoritmo	BSO-LAHC	GA-RRHC	GLNSA	GRASP	HA	IJA	TlPSO
MK01	10.0000	10.0000	10.0000	10.0000	10.0000	10.0000	10.0000
Mk02	7.6923	7.6923	7.6923	7.6923	7.6923	11.1111	7.6923
MK03	0	0	0	0	0	0	0
MK04	20.0000	20.0000	20.0000	20.0000	20.0000	20.0000	20.0000
MK05	2.8902	2.3256	2.8902	2.3256	2.3256	2.3256	2.8902
MK06	45.9016	43.1034	43.1034	48.4375	42.1053	42.1053	45.0000
MK07	5.6738	4.3165	4.3165	4.3165	4.3165	4.3165	4.3165
MK08	0	0	0	0	0	0	0
MK09	2.6059	2.6059	2.6059	2.6059	2.6059	2.6059	2.6059
MK10	19.1176	16.6667	19.5122	19.5122	16.2437	16.2437	19.5122

blue

Para visualizar la dispersión de los datos se vuelve a utilizar el diagrama de cajas Figura V.5, donde se visualizan los cuartiles, la mediana, los valores

mínimos y máximos, así como sus valores atípicos de cada uno de los algoritmos, con el fin de comparar de distribución de la variable del makespan que se obtuvo con los datos RDP de cada unos de los algoritmos de este conjunto de dataset, observando que los valores son más concentrados en el algoritmo GA-RRHC, el HA y el IJA y su rango de datos son más próximos.

El algoritmo que muestra en su extremo superior o valor máximo más alto o conocido también como bigote, es el algoritmo GLNSA, encontrándose en 43.1034, lo que significa que para que su máximo se encuentre ahí, es porque se toma el valor más próximo al del límite superior, por lo tanto su rango de datos es más amplio o más disperso, mientras que para el resto de los algoritmos muestran sus valores atípicos a ese mismo nivel ya que son puntos que se encuentran fuera del rango de su límite superior.

Finalmente su tipo de distribución se observa que es asimétrica, en este dataset los algoritmos que son asimétricos positivos es porque sus datos se concentraron hacia la parte inferior de la distribución.

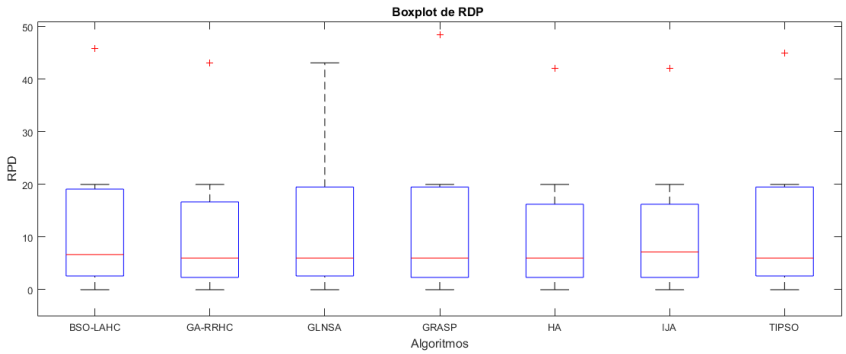

Figura V.5: Diagrama de caja de RPD para 7 algoritmos

El promedio del cálculo del RPD para este dataset se muestra en la Tabla V.4 , y presenta el ranking de cada método aplicado, tomando como referencia el RPD promedio de cada algoritmo de las 10 instancias, también se muestra los resultados del valor p de la prueba no-paramétrica de Friedman por pares, comparando el ranking del GA-RRHC con cada uno de los otros algoritmos. Se puede observar que el algoritmo desarrollado GA-RRHC obtuvo el segundo lugar entre todos los algoritmos comparados, y por su parte el GLNSA obtuvo el tercer lugar entre los algoritmos, demostrando su competitividad con algoritmos de última generación.

La prueba estadística no-paramétrica de Friedman se realizó con el fin de

Tabla V.4: Ranking de algoritmos y prueba de Friedman para datos BRdata dataset.

	Algoritmo	**BSO-LAHC**	**GA-RRHC**	**GLNSA**	**GRASP**	**HA**	**IJA**	**TlPSO**
blue	Average RPD:	11.3881	10.6710	11.0121	11.4890	10.5289	11.2665	11.2017
	Rank:	6	2	3	7	1	5	4
	p-value:	0.0455	-	0.1573	0.0435	0.1643	0.0803	0.0833

corroborar si existe una comparación estadísticamente significativa entre los resultados obtenidos de los algoritmos de los valores obtenidos del RPD [25].

El resultado obtenido con esta prueba de Friedman como se observa en la V.4, donde el valor de $p = 0.05$ es el valor de significancia que delimita, que si $p < 0.05$ no se acepta la hipótesis nula H_o, y caso contrario la hipótesis no se rechaza, lo que significa en esta última condición que los algoritmos se comportan de manera similar, y entonces no existe diferencia entre las variables dependientes.

Por lo tanto se puede observar en la Tabla V.4, para el algoritmo GA-RRHC obtuvo un valor $p < 0.05$ al igual que el algoritmo BSO-LAHC y el GRASP; mostrando que el valor p demuestra diferencias significativas en el desempeño de los 6 algoritmos, por lo tanto al existir variaciones en p en los algoritmos significa que la hipótesis no se acepta, y se concluye que si hay diferencias significativas entre los algoritmos.

En la Figura V.6 se muestra la diferencia con signo entre el RPD promedio del GA-RRHC frente a los otros métodos en pares. Una diferencia negativa significa un rendimiento inferior en comparación con el GA-RRHC.

Con este análisis de este BRdata, se ha demostrando que el GA-RRHC es estadísticamente competitivo con los otros algoritmos para optimizar el dataset BRdata.

V.0.3 Tercer experimento: instancias Hurink-rdata baja flexibilidad

El conjunto de datos Hurink se clasifican en dos tipos, y se diferencia en base a su flexibilidad, donde es baja y alta. En este experimento se aborda el primer conjunto de datos denominado Rdata de Hurink, siendo este benchmak de baja flexibilidad.

En este experimento consta de 43 instancias Rdata y son la, mt06, mt10, mt20 y de la instancia la01-la40; van de los 6 a 30 trabajos y de 6 a 15 máquinas. Su tasa de flexibilidad b oscila entre 0.13 y 0.4, es por eso que

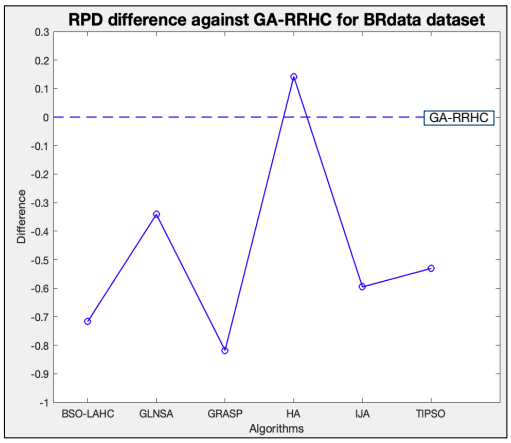

Figura V.6: Comparación de la diferencia RPD por algoritmo para el conjunto BRdata [29]

a este experimento se le conoce como de baja flexibilidad y solo algunas instancias son de media flexibilidad, en este experimento se tienen en promedio 2 máquinas por operación del total de máquinas disponibles que pueden realizar la misma operación.

Los resultados generados aplicando el GA-RRHC y su comparación con los otros algoritmos para este Rdata se muestran en la Tabla V.5, donde solo se incluyen el algoritmo GLNS, HA y el IJA debido a que son los métodos que reportan resultados referentes a este dataset.

El rendimiento en este experimento Rdata aplicando el GA-RRHC se muestra en la Tabla V.5, obteniendo resultados competentes alcanzando el 53.48% de los mejores resultados del total de las instancias para problemas con mediana flexibilidad.

En la Figura V.7 muestra gráficamente el número de los mejores resultados obtenidos en cada algoritmo para este conjunto de datos. Obteniendo con el GA-RRHC 23 mejores instancias solo por detras de IJA con 27 y del HA con 42.

La dispersión de los datos de este experimento se muestra en la Figura V.8, conociendo que el BKV son los valores ideales a alcanzar y que representa el makespan más conocido en cada problema, como se observa en el diagrama de caja, su comparación de la distribución de valores de los algoritmos con respecto al BKV se comportan de manera similar. La mediana $Q2$(linea roja) tomando al BKN como referencia, sus $Q2$ es similar en los

Tabla V.5: Resultados Rdata dataset

blue

Instancia	n × m	β	BKV	GA-RRHC	GLNSA	HA	IJA
mt06	6 × 6	0.33	47	47∗	47∗	47∗	47∗
mt10	10 × 10	0.2	686	686∗	686∗	686∗	686∗
mt20	20 × 5	0.4	1022	1022∗	1022∗	1024	1024
la01	10 × 5	0.4	570	571	571	570∗	571
la02	10 × 5	0.4	529	530∗	530∗	530∗	530∗
la03	10 × 5	0.4	477	477∗	477∗	477∗	477∗
la04	10 × 5	0.4	502	502∗	502∗	502∗	502∗
la05	10 × 5	0.4	457	457∗	457∗	457∗	457∗
la06	15 × 5	0.4	799	799∗	799∗	799∗	799∗
la07	15 × 5	0.4	749	749∗	749∗	749∗	749∗
la08	15 × 5	0.4	765	765∗	765∗	765∗	765∗
la09	15 × 5	0.4	853	853∗	853∗	853∗	853∗
la10	15 × 5	0.4	804	804∗	804∗	804∗	804∗
la11	20 × 5	0.4	1071	1071∗	1071∗	1071∗	1071∗
la12	20 × 5	0.4	936	936∗	936∗	936∗	936∗
la13	20 × 5	0.4	1038	1038∗	1038∗	1038∗	1038∗
la14	20 × 5	0.4	1070	1070∗	1070∗	1070∗	1070∗
la15	20 × 5	0.4	1089	1089∗	1089∗	1090	1090
la16	10 × 10	0.2	717	717∗	717∗	717∗	717∗
la17	10 × 10	0.2	646	646∗	646∗	646∗	646∗
la18	10 × 10	0.2	666	666∗	666∗	666∗	666∗
la19	10 × 10	0.2	647	700∗	700∗	700∗	702
la20	10 × 10	0.2	756	756∗	756∗	756∗	760
la21	15 × 10	0.2	808	850	852	835∗	854
la22	15 × 10	0.2	737	770	774	760∗	760∗
la23	15 × 10	0.2	816	850	854	840∗	852
la24	15 × 10	0.2	775	810	826	806∗	806∗
la25	15 × 10	0.2	752	800	803	789∗	803
la26	20 × 10	0.2	1056	1070	1075	1061∗	1061∗
la27	20 × 10	0.2	1085	1100	1109	1089∗	1109
la28	20 × 10	0.2	1075	1090	1096	1079∗	1081
la29	20 × 10	0.2	993	999	1008	997∗	997∗
la30	20 × 10	0.2	1068	1088	1096	1078∗	1078∗
la31	30 × 10	0.2	1520	1521∗	1527	1521∗	1521∗
la32	30 × 10	0.2	1657	1667	1667	1659∗	1659∗
la33	30 × 10	0.2	1497	1500	1504	1499∗	1499∗
la34	30 × 10	0.2	1535	1539	1540	1536∗	1536∗
la35	30 × 10	0.2	1549	1553	1555	1550∗	1555
la36	15 × 15	0.13	1016	1050	1053	1028∗	1050
la37	15 × 15	0.13	989	1092	1093	1074∗	1092
la38	15 × 15	0.13	943	995	999	960∗	995
la39	15 × 15	0.13	966	1030	1034	1024∗	1031
la40	15 × 15	0.13	955	998	997	970∗	993

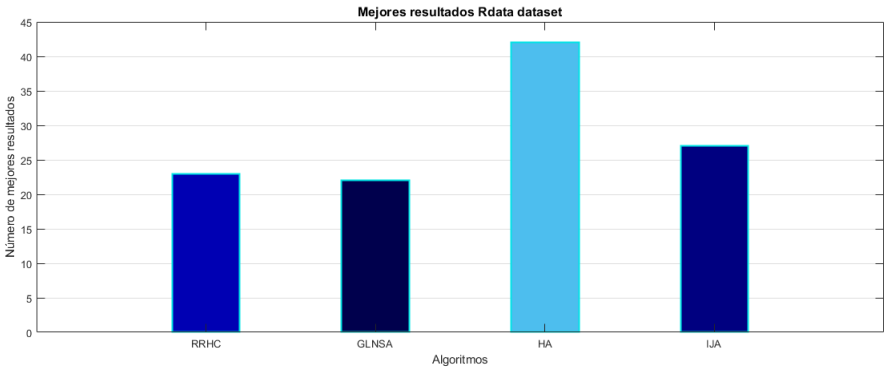

Figura V.7: Número de mejores Instancias Rdata

otros algoritmos. El tipo de distribución en este experimento, se observa que los datos al estar ligeramente sesgados en la parte inferior, indica que su distribución es asimétrica positiva, por lo que la media es mayor que la mediana y los datos se concentran en la parte inferior de la distribución. Tanto sus mínimos y máximos se encuentran en promedio entre 457 y 1065 respectivamente y sus datos atípicos se encuentran entre 47 y 1550. Con lo que el algoritmo GA-RRHC demuestra su competitividad comparado con el BKN.

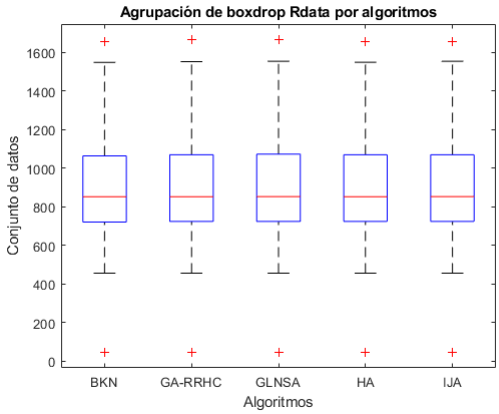

Figura V.8: Agrupación de algoritmos en diagrama de caja Rdata

Se aplica la medida estadística RPD V.1 para evaluar la diferencia entre dos mediciones, a partir del BKN que es el makespan mas conocido por cada problema y el otro es el valor obtenido de cada uno de los cuatro algoritmos a

comparar, con el fin de medir la variación de este conjunto de datos y obtener su diferencia relativa en porcentaje como se observa en la V.6.

Con el conjunto de datos estadísticos de la RDP de las instancias Rdata, la distribución de la variable numérica (makespan) usando cuartiles se muestra en la Figura V.9 algunas cosas como la dispersión, simetría, rango, obteniendo lo siguiente: los datos mas concentrados los tiene el algoritmo HA, seguido del algoritmo GA-RRHC y el IJA y por último el GLNSA. Sus míni-mos se encuentran desde 0, y sus máximos están entre 6 y 8. Las marcas de cruz en HA indican que hay varios datos atípicos y hay varios valores que son exactos al makespan mas conocido. Su distribución de cada algoritmo tiene asimetría positiva por lo que todos sus datos de estimación del valor promedio se encuentran concentrados en la parte inferior de la distribución. Y al estar tan cercana a 0 indica que su estimación al hacer la comparación entre 2 medidas, BKN y el del makespan de cada algoritmo, esa diferencia porcentual relativa da la estimación de la precisión es muy cercana al valor real, por eso se acercan los valores a cero.

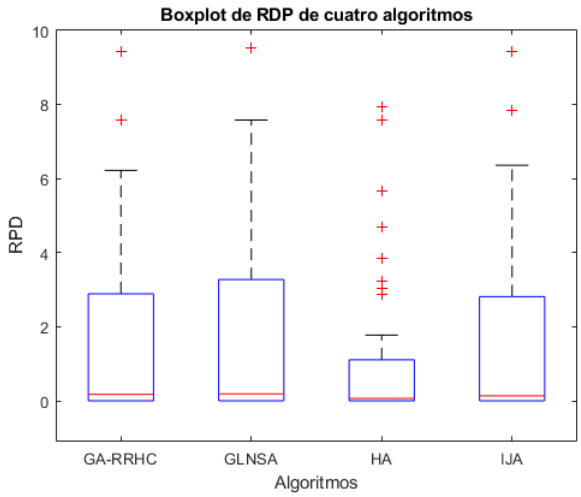

Figura V.9: Diagrama de caja de RPD para 4 algoritmos

Con los datos de la tabla V.6 se obtiene el promedio del cálculo del RPD de cada algoritmo de las 43 instancias, como se muestra en la Tabla V.7, también se muestra los resultados el valor p de la prueba no-paramétrica de Friedman, y tomando como referencia el RPD promedio, se obtiene el ranking del GA-RRHC y cada uno de los otros algoritmos.

Tabla V.6: Valores RDP de los resultados del experimento Rdata con los 4 algoritmos

Algoritmo	GA-RRHC	GLNSA	HA	IJA
mt06	0	0	0	0
mt10	0	0	0	0
mt20	0	0	0.1953	0.1953
la01	0.1751	0.1751	0	0.1751
la02	0.1887	0.1887	0.1887	0.1887
la03	0	0	0	0
la04	0	0	0	0
la05	0	0	0	0
la06	0	0	0	0
la07	0	0	0	0
la08	0	0	0	0
la09	0	0	0	0
la10	0	0	0	0
la11	0	0	0	0
la12	0	0	0	0
la13	0	0	0	0
la14	0	0	0	0
la15	0	0	0.0917	0.0917
la16	0	0	0	0
la17	0	0	0	0
la18	0	0	0	0
la19	7.5714	7.5714	7.5714	7.8348
la20	0	0	0	0.5263
la21	4.9412	5.1643	3.2335	5.3864
la22	4.2857	4.7804	3.0263	3.0263
la23	4	4.4496	2.8571	4.2254
la24	4.3210	6.1743	3.8462	3.8462
la25	6	6.3512	4.6895	6.3512
la26	1.3084	1.7674	0.4713	0.4713
la27	1.3636	2.1641	0.3673	2.1641
la28	1.3761	1.9161	0.3707	0.5550
la29	0.6006	1.4881	0.4012	0.4012
la30	1.8382	2.5547	0.9276	0.9276
la31	0.0657	0.4584	0.0657	0.0657
la32	0.5999	0.5999	0.1206	0.1206
la33	0.2	0.4654	0.1334	0.1334
la34	0.2599	0.3247	0.0651	0.0651
la35	0.2576	0.3859	0.0645	0.3859
la36	3.2381	3.5138	1.1673	3.2381
la37	9.4322	9.5151	7.9143	9.4322
la38	5.2261	5.6056	1.7708	5.2261
la39	6.2136	6.5764	5.6641	6.3046
la40	4.3086	4.2126	1.5464	3.8268

Se puede observar que el algoritmo desarrollado GA-RRHC obtuvo el tercer lugar entre todos los algoritmos comparados, y dicho resultado fue esperado, debido a que la mayoría de los problemas de este dataset son de baja flexibilidad. La pobre competitividad que obtuvo el GA-RRHC respecto con los otros algoritmos, se dio a partir de los resultados de la instancia La21-la40 donde tienen una flexibilidad β baja, que va de 0.2 a 0.13, a pesar de que estas instancias son de mayor magnitud teniendo un mayor número de trabajos y máquinas, pero por su baja flexibilidad no hizo posible obtener buenos resultados aplicando el GA-RRHC, debido a que este algoritmo esta enfocado en la resolución de problemas de instancias del FJSSP con una alta flexibilidad β, sin en cambio de las instancias mt06 - la20, teniendo en algunas instancias hasta 20 trabajos o hasta las 10 máquinas, estas instancias obtienen buenos resultados debido a que su β es media, por lo tanto si obtiene el makespan conocido de estas instancias.

Tabla V.7: Ranking de los algoritmos y prueba de Friedman para el conjunto de datos Rdata.

blue

Algoritmo	GA-RRHC	GLNSA	HA	IJA
Average RPD:	1.5761	1.7768	1.0872	1.5155
Rank:	3	4	1	2
p-value:	-	0.0001	0.0001	0.9195

El resultado obtenido con esta prueba de Friedman, el valor de significancia vale $p = 0.05$, valor que delimita si $p < 0.05$ significando que la hipótesis nula H_o no se acepta , y caso contrario la hipótesis H_o no se rechaza, y cuando no es aceptada es porque si existe una diferencia significativa entre las variables dependientes que son los algoritmos que se están comparando.

Por lo que se observa en la V.7 los datos obtenidos indican que para el algoritmo GA-RRHC obtuvo un valor $p > 0.05$ al igual que IJA, por otro lado el algoritmo GLNSA y el HA obtuvieron $p < 0.05$; mostrando que este experimento p muestra variaciones en sus 4 algoritmos por lo que se concluye que esta la hipótesis no se acepta, debido ha que hay diferencias significativas en el rendimiento de los 4 algoritmos para estas instancias.

El análisis estadístico de la prueba no-paramétrica de Friedman, corrobora si existe una comparación estadísticamente significativa entre los resultados obtenidos de los algoritmos de los valores obtenidos del RPD. Por lo que los resultados obtenidos en este experimento no se muestra una diferencia significativa con el IJA pero si con el GLNSA y el HA.

En la Figura V.10 se muestra la diferencia del algoritmo GA-RRHC con los otros algoritmos para el RPD promedio. Un valor negativo del RPD promedio significa un rendimiento inferior en comparación del GA-RRHC.

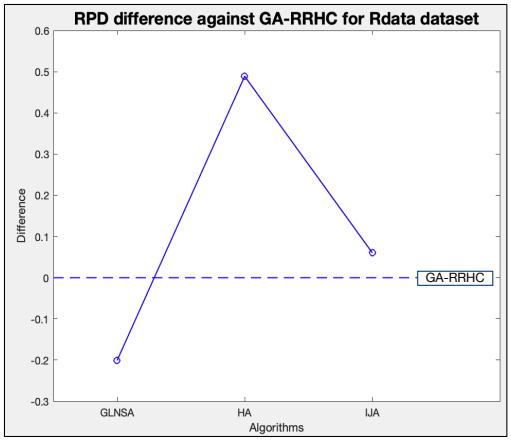

Figura V.10: Comparación de la diferencia RPD por algoritmo para el conjunto Rdata [29]

Con este análisis de este Rdata, se ha demostrando que el GA-RRHC es estadísticamente competitivo para las instancias de media flexibilidad, pero no lo es para aquellas instancias con baja flexibilidad. Esto prueba que el GA-RRHC de acuerdo a su análisis estadístico tiene un performance comparable con respecto al IJA para problemas con baja flexibilidad, superando al GLNSA.

V.0.4 Cuarto experimento: instancias Hurink-vdata alta flexibilidad

El conjunto de datos del benchmark de Hurink [49] se clasifican de acuerdo a su flexibilidad β en dos clases, en el experimento anterior se abordo la primer categoría, que son las instancias Rdata identificadas por ser de baja flexibilidad teniendo un radio $\beta \leq 0.4$; ahora en este experimento se abordan la segunda categoria, que son las instancias Vdata, que se identifican por ser de alta flexibilidad. Como ya se explico, para determinar la flexibilidad, el radio β se va a encontrar entre 0 y 1, y entre más se aproxime a 1 mayor es

su flexibilidad, lo que indica que si un valor β es alto, un mayor número de máquinas pueden realizar más operaciones distintas.

Para este experimento se toman las 43 instancias del banco de pruebas HU-vdata, teniendo en cada instancia un radio $\beta = 0.5$, lo que indica que una operación puede ser procesada en promedio por el 50% de las máquinas, e indicando que es un experimento con alta flexibilidad.

En la Tabla V.8 se muestran los resultados generados por el GA-RRHC y se vuelven a comparar con los tres algoritmos GLNSA, HA e IJA ya que son los métodos que reportan resultados para este dataset. Los resultados señalados con $*$ son los mejores makespan obtenidos entre los 43 algoritmos.

El rendimiento en este experimento Vdata de la V.8 se puede observar que los resultados generados con el GA-RRHC se calculó el 100% de mejores valores obteniendo el primer lugar de todas las instancias con mejores resultados, seguido del HA con el 97.67%, el GLNSA con el 95.34% y por último el que obtuvo un menor de instancias calculadas con el mejor valor fue el IJA con el 72%.

En la Figura V.11 se muestra gráficamente el número de los mejores resultados obtenidos en cada algoritmo para este dataset. Obteniendo por arriba de todos el GA-RRHC obteniendo 43 de 43 instancias de los mejores valores, en segundo lugar el HA con 42, en tercer lugar el GLNSA con 41 y por último el IJA con 31 mejores valores.

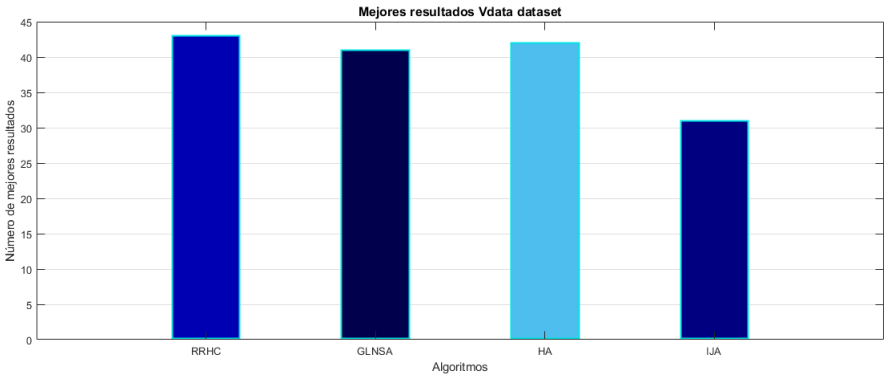

Figura V.11: Número de mejores Instancias Vdata

Con respecto a la dispersión de las datos en este experimento se muestra en la Figura V.12, su análisis de este boxplot conociendo de antemano que el BKV son los valores ideales a alcanzar como el makespan más conocido, esta

Tabla V.8: Resultados experimentales con las instancias HU-Vdata .

Instance	n × m	β	BKV	GA-RRHC	GLNSA	HA	IJA
mt06	6 × 6	0.5	47	47∗	47∗	47∗	47∗
mt10	10 × 10	0.5	655	655∗	655∗	655∗	655∗
mt20	20 × 5	0.5	1022	1022∗	1022∗	1022∗	1024
la01	10 × 5	0.5	570	570∗	570∗	570∗	571
la02	10 × 5	0.5	529	529∗	529∗	529∗	529∗
la03	10 × 5	0.5	477	477∗	477∗	477∗	477∗
la04	10 × 5	0.5	502	502∗	502∗	502∗	502∗
la05	10 × 5	0.5	457	457∗	457∗	457∗	457∗
la06	15 × 5	0.5	799	799∗	799∗	799∗	799∗
la07	15 × 5	0.5	749	749∗	749∗	749∗	749∗
la08	15 × 5	0.5	765	765∗	765∗	765∗	765∗
la09	15 × 5	0.5	853	853∗	853∗	853∗	853∗
la10	15 × 5	0.5	804	804∗	804∗	804∗	804∗
la11	20 × 5	0.5	1071	1071∗	1071∗	1071∗	1071∗
la12	20 × 5	0.5	936	936∗	936∗	936∗	936∗
la13	20 × 5	0.5	1038	1038∗	1038∗	1038∗	1038∗
la14	20 × 5	0.5	1070	1070∗	1070∗	1070∗	1070∗
la15	20 × 5	0.5	1089	1089∗	1089∗	1089∗	1089∗
la16	10 × 10	0.5	717	717∗	717∗	717∗	717∗
la17	10 × 10	0.5	646	646∗	646∗	646∗	646∗
la18	10 × 10	0.5	663	663∗	663∗	663∗	665
la19	10 × 10	0.5	617	617∗	617∗	617∗	618
la20	10 × 10	0.5	756	756∗	756∗	756∗	758
la21	15 × 10	0.5	800	804∗	806	804∗	806
la22	15 × 10	0.5	733	737∗	737∗	738	738
la23	15 × 10	0.5	809	813∗	813∗	813∗	813∗
la24	15 × 10	0.5	773	777∗	777∗	777∗	778
la25	15 × 10	0.5	751	754∗	754∗	754∗	754∗
la26	20 × 10	0.5	1052	1053∗	1054	1053∗	1054
la27	20 × 10	0.5	1084	1085∗	1085∗	1085∗	1085∗
la28	20 × 10	0.5	1069	1070∗	1070∗	1070∗	1070∗
la29	20 × 10	0.5	993	994∗	994∗	994∗	994∗
la30	20 × 10	0.5	1068	1069∗	1069∗	1069∗	1069∗
la31	30 × 10	0.5	1520	1520∗	1520∗	1520∗	1521
la32	30 × 10	0.5	1657	1658∗	1658∗	1658∗	1658∗
la33	30 × 10	0.5	1497	1497∗	1497∗	1497∗	1497∗
la34	30 × 10	0.5	1535	1535∗	1535∗	1535∗	1535∗
la35	30 × 10	0.5	1549	1549∗	1549∗	1549∗	1549∗
la36	15 × 15	0.5	948	948∗	948∗	948∗	950
la37	15 × 15	0.5	986	986∗	986∗	986∗	986∗
la38	15 × 15	0.5	943	943∗	943∗	943∗	943∗
la39	15 × 15	0.5	922	922∗	922∗	922∗	922∗
la40	15 × 15	0.5	955	955∗	955∗	955∗	956

información se toma de referencia para interpretar los datos en los demás algoritmos.

El análisis de esta distribución de datos, es que en todos los algoritmos muestran que en sus cuartiles, su mediana, sus rangos,sus mínimos, máximos y datos atípicos son similares con respecto al BKV, debido que que la mayoría de los makespan en cada una de las instancias son iguales o muy cercanos. El tipo de distribución en este experimento muestra que todos los algoritmos, que sus datos del makespan están sesgados hacia la parte derecha, y su tipo de simetría, es asimétrica positiva, indicando que la mayoría de los datos del makespan se concentran en la parte inferior de la distribución. Por lo tanto la media es mayor que la mediana.

Con lo que el algoritmo GA-RRHC demuestra su competitividad comparado con el BKN.

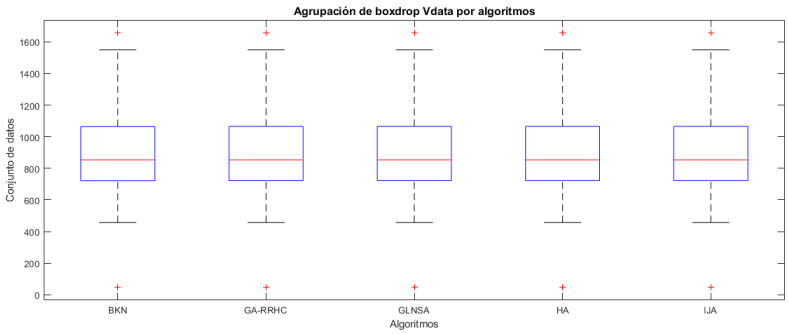

Figura V.12: Agrupación de algoritmos en diagrama de caja Vdata

Los valores obtenidos para medir su variación de este dataset para obtener la diferencia porcentual relativa RPD de cada algoritmo se muestran en la Tabla V.9.

Con los datos estadísticos del RDP, en la Figura V.13 se utilizan para visualizar la distribución del makespan de todas las instancias de cada algoritmo en el que se valida el análisis numérico de los datos de la Tabla V.9.

Se visualiza que la dispersión de los datos en los diagramas de caja cada uno de los cuatro algoritmos, se valida el análisis numérico de los datos RPD. Se observa que los valores obtenidos del RPD en el GA-RRHC tanto su mediana como su rango es muy bajo, lo que significa que los datos están muy concentrados casi cercanos a cero.

Tabla V.9: Valores RDP de los resultados del experimento Vdata con los 4 algoritmos

Algoritmo	GA-RRHC	GLNSA	HA	IJA
mt06	0	0	0	0
mt10	0	0	0	0
mt20	0	0	0	0.1953125
la01	0	0	0	0.175131349
la02	0	0	0	0
la03	0	0	0	0
la04	0	0	0	0
la05	0	0	0	0
la06	0	0	0	0
la07	0	0	0	0
la08	0	0	0	0
la09	0	0	0	0
la10	0	0	0	0
la11	0	0	0	0
la12	0	0	0	0
la13	0	0	0	0
la14	0	0	0	0
la15	0	0	0	0
la16	0	0	0	0
la17	0	0	0	0
la18	0	0	0	0.3008
la19	0	0	0	0.1618
la20	0	0	0	0.2639
la21	0.4975	0.7444	0.4975	0.7444
la22	0.5427	0.5427	0.6775	0.6775
la23	0.4920	0.4920	0.4920	0.4920
la24	0.5148	0.5148	0.5148	0.6427
la25	0.3979	0.3979	0.3979	0.3979
la26	0.0950	0.1898	0.0950	0.1898
la27	0.0922	0.0922	0.0922	0.0922
la28	0.0935	0.0935	0.0935	0.0935
la29	0.1006	0.1006	0.1006	0.1006
la30	0.0935	0.0935	0.0935	0.0935
la31	0	0	0	0.0657
la32	0.06031	0.06031	0.06031	0.0603
la33	0	0	0	0
la34	0	0	0	0
la35	0	0	0	0
la36	0	0	0	0.2105
la37	0	0	0	0
la38	0	0	0	0
la39	0	0	0	0
la40	0	0	0	0.1046

El algoritmo GLNSA muestra una distribución concentrada al igual que el HA, mientras que el algoritmo IJA su distribución es excesiva teniendo datos mas dispersos.

Por lo tanto en los cuatro algoritmos su tipo de distribución es asimetría positiva, ya que todos sus datos RPD se encuentran concentrados en la parte inferior de la distribución. Y al estar tan cercana a 0 indica que su estimación al hacer la comparación entre 2 medidas, BKN y el del makespan de cada algoritmo, esa diferencia porcentual relativa da la estimación de la precisión es muy cercana al valor real, por eso se acercan los valores a cero.

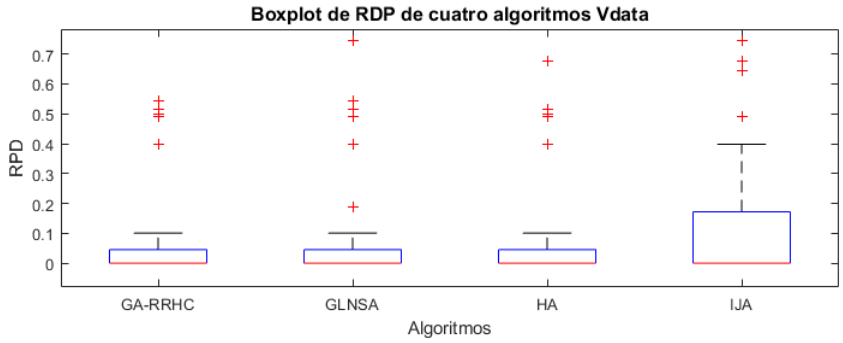

Figura V.13: Diagrama de caja de RPD para 4 algoritmos

Una vez que ya se obtuvieron los datos RPD Tabla V.9, se utilizan para obtener el RPD promedio en cada algoritmo como se muestra en la Tabla V.10, y a partir de este promedio se obtiene el ranking de los algoritmos; también se muestran los resultados de la prueba no-paramétrica de Friedman indicado como el valor p.

Tabla V.10: Ranking de algoritmos y prueba de Friedman para datos Vdata

Algoritmo	GA-RRHC	GLNSA	HA	IJA
Promedio RPD:	0.0693	0.0772	0.0724	0.1177
Rank:	1	3	2	4
p-value:	-	0.1573	0.3173	0.0005

blue

Los resultados muestran que de acuerdo al RPD promedio de las 43 instancias, el GA-RRHC obtuvo el primer lugar entre todos los algoritmos comparados, seguido del HA, en tercer lugar el GLNSA y por último IJA.

El análisis estadístico de la prueba no-paramétrica de Friedman, conociendo que el valor de significancia normalmente vale $p = 0.05$, valor que

delimita si $p < 0.05$, donde si el valor p calculado es menor al valor de significancia, la hipótesis nula H_o no se acepta , y si $p > 0.05$ la hipótesis H_o no se rechaza. Cuando no es aceptada es porque si existe una diferencia significativa entre las variables dependientes que son los algoritmos que se están comparando.

Los resultados del análisis estadístico de la prueba no-paramétrica de Friedman como se observa en la Tabla V.10, existe una diferencia significativa entre los algoritmos comparados.

Lo que se concluye con los datos obtenidos de la prueba de Friedman Tabla V.7,en este experimento si existe una diferencia entre los algoritmos, mostrando el algoritmo IJA es el único que muestra esa diferencia significativa, por lo que la Ho no se acepta, demostrando que si existen diferencias entre los 4 algoritmos de este dataset

En la Figura V.14 se muestra la diferencia del algoritmo RPD promedio con el algoritmo GA-RRHC comparable con los demás algoritmos. Donde un valor negativo del RPD promedio, significa un rendimiento inferior en comparación del GA-RRHC.

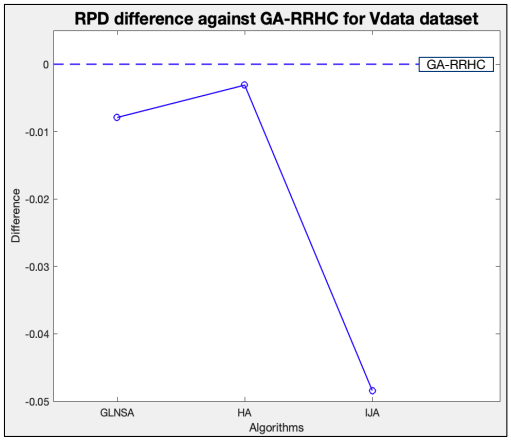

Figura V.14: Comparación de la diferencia RPD por algoritmo pare el conjunto Vdata [29]

Con los resultados obtenidos en este dataset, el algoritmo GA-RRHC enfocado a la resolución de problemas de instancias del FJSSP con una alta flexibilidad β, se confirma que tiene un desempeño comparable con algoritmos recientes, reconocidos por su robustez para optimizar este tipo de problemas data, obteniendo una complejidad computacional competitiva con aquellos

algoritmos actuales, en la cual el GA-RRHC resuelve con buenos resultados aquellas instancias con una baja y mediana flexibilidad , pero obtiene resultados excelentes en aquellas instancias de alta flexibilidad, obteniendo así el makespan mejor conocido para estas instancias.

A continuación se muestran algunos diagramas de Gantt de las programaciones de tareas obtenidas por el GA-RRHC. Se evidencian algunos de los diagramas de Gantt cuyos trabajos fueron en aumento en este dataset, la Figura V.15muestra la instancia la01 con 10 trabajos y 5 máquinas; la Figura V.16 muestra la instancia la11 con 20 trabajos y 5 máquinas; la Figura V.17 muestra la instancia la31 con 30 trabajos y 10 máquinas; y la Figura V.18 muestra la instancia la36 con 15 trabajos y 15 máquinas.

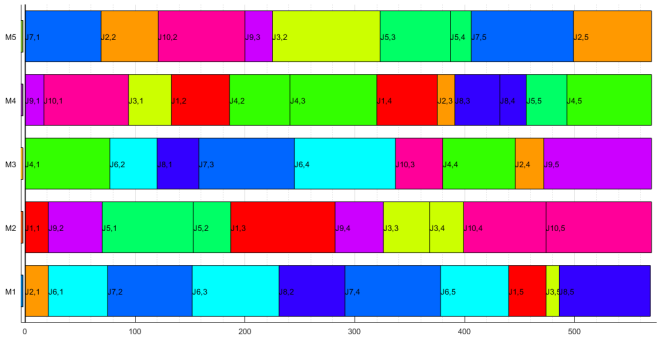

Figura V.15: Diagrama de Gantt de la solución obtenida aplicando el GA-RRHC en la instancia Vdata la-01 con un makespan de 570

V.0.5 Estudio de caso: Generación de un Dataset largo

En este trabajo se probaron 4 experimentos en los que variaba el tamaño de las instancias de todos los dataset, se trataron conjuntos de datos que van desde los 6 trabajos con 6 máquinas hasta los 30 trabajos con 10 máquinas, teniendo un máximo de 300 operaciones, que aplicado en la realidad, ese número de trabajos, máquinas y operaciones son insuficientes, ya que en el mundo real se maneja un número mucho mayor.

En este último experimento se abordan tres problemas de gran tamaño, en donde se demostrará también la eficiencia del algoritmo GA-RRHC. Se utilizaron los parámetros establecidos en [10] [83] . Para el desarrollo de este último experimento, se generaron tres instancias largas de manera aleatoria y

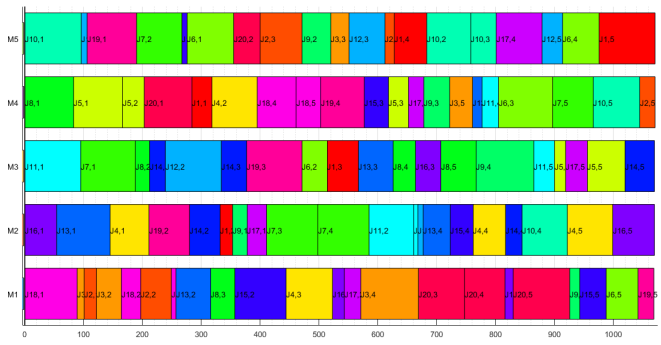

Figura V.16: Diagrama de Gantt de la solución obtenida aplicando el GA-RRHC en la instancia Vdata la-11 con un makespan de 1071

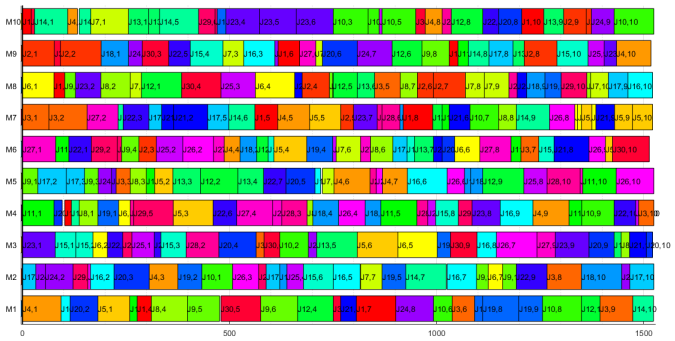

Figura V.17: Diagrama de Gantt de la solución obtenida aplicando el GA-RRHC en la instancia Vdata la-31 con un makespan de 1520

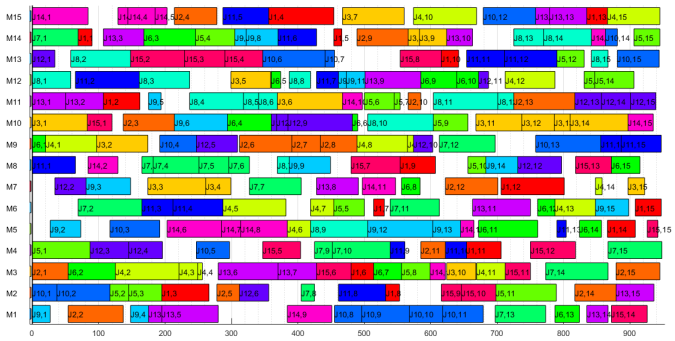

Figura V.18: Diagrama de Gantt de la solución obtenida aplicando el GA-RRHC en la instancia Vdata la-36 con un makespan de 948

se denominaron como VL01, VL02 y VL3; se consideraron en cada instancia valores que van de los 50 a 80 trabajos, de las 20 a 50 máquinas y de 704 a 2773 operaciones, para estas tres instancias se consideró una flexibilidad $beta = 0.75$, denominada como alta.

Las instancias propuestas no se han abordado en algún otro trabajo de investigación, por lo que para analizar la eficiencia del algoritmo en instancias grandes, solo se aplican dos algoritmos como método de comparación, aplicando el GA-RRHA y el GLNSA, siendo los algoritmos desarrollados en esta investigación.

La Tabla V.11 muestra los resultados que se obtuvieron aplicando las 3 instancias propuestas, mostrando que el GA-RRHC obtuvo mejores valores por arriba del GLNSA, obteniendo un mejor rendimiento en aquellas instancias largas con tienen una mayor dimensión, donde van en aumento sus trabajos, máquinas y operaciones.

Tabla V.11: Experimento con instancias largas

Instancia	$n \times m$	o	$beta$	**GLNSA**			**GA-RRHC**		
				Best	Promedio	RPD	Best	Promedio	RPD
VL01	50 × 20	704	0.75	592	617.3	6.9228675	551	570.9	0
VL02	60 × 30	1246	0.75	759	781.3	7.11462451	705	717.8	0
VL03	80 × 50	2773	0.75	1155	1183.1	9.87012987	1041	1058.4	0

La Figura V.19 muestra las tres instancias largas VL01, VL02 y la VL03, mostrando el proceso de optimización con el algoritmo que obtuvo mejor desempeño que fue el GA-RRHC. Se muestran 6 diagramas de Gantt de la solución dada del FJSSP, la primera fila son los primeros 3 diagramas de la solución de inicio de cada instancia, los siguientes 3 diagramas, son la solución final de cada una, y la última fila es la convergencia obtenida del desempeño del makespan de cada instancia.

Se puede concluir que en la solución de inicio donde son soluciones que se generaron de manera aleatoria, existen bastantes tiempos de inactividad, y comparada con la solución final, sus tiempos de inactividad disminuyen conforme continua la convergencia para calcular el makespan óptimo.

V.0.6 Conclusión del Capítulo

En los experimentos abordados de $Kacem, BRdata, RdatayVdata$ hicieron un total de 101 instancias diferentes, en la que se probó la eficiencia del

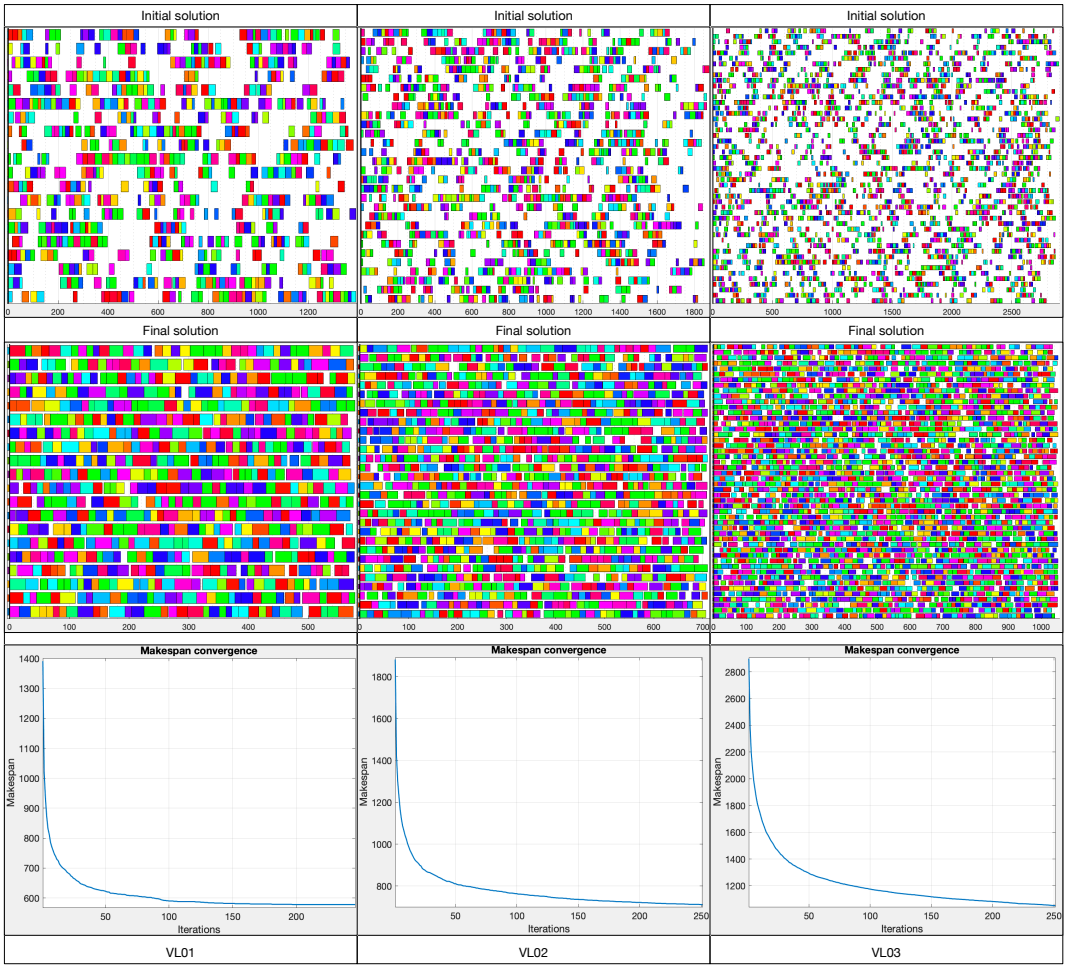

Figura V.19: Ejemplos vl [29]

algoritmo GA-RRHC.

Concluyendo que entre más alto sea su flexibilidad β mejores resultados se obtendrán. El algoritmo GA-RRHC obtuvo un buen desempeño en aquellas instancias donde $\beta < 0.5$ denominadas instancias de flexibilidad de media a baja, obteniendo valores cercanos al mejor valor conocido; pero obtuvo un excelente desempeño cuando $\beta > 0.5$ denominadas instancias de alta flexibilidad, alcanzando el mejor resultado conocido en aquella instancia.

Capítulo VI

Conclusiones

Este trabajo de investigación con base al estado del arte, se logro identificar el área de oportunidad donde se podría realizar investigación, donde primero se identifico ¿qué es lo que se quería realizar?, después el ¿porque? y finalmente el ¿para que? Analizando estas cuestiones se logro realizar todo este trabajo de tesis concluyendo lo siguiente:

Primero se decidió abordar un problema de programación de tareas, porque de acuerdo a la revisión de la literatura este tipo de problemas se han abordado desde décadas atrás por la necesidad de seguir proponiendo y buscar un buen algoritmo, ya que aún no existe alguno en especifico que siempre proporcione soluciones óptimas para resolver alguna función objetivo como cuándo se requiera la minimización de costos, tiempos, entre otros. Se eligió un problema Flexible por ser los que son más allegados a la vida real en un sistema de manufactura y por esa necesidad de adaptarse a la flexibilidad de máquinas, ya que hoy en día es necesario solventar problemas donde van en aumento el número de trabajos y operaciones, donde la mayoría de veces es necesario realizar varias operaciones, por eso la necesidad de la flexibilidad de tareas, en que la máquina realice varias operaciones, pero problema es encontrar una buena programación de esas tareas y poder ahorrar tiempos de procesamientos. Por eso se decidió abordar en este trabajo de tesis la programación de tareas para solucionar un problema Flexible, con el fin de desarrollar una nueva propuesta adoptando una estrategia diferente inspirada en la idea del autómata celular, y proponer un nuevo algoritmo que optimice instancias para resolver el problema del FJSSP. Se aplicó un algoritmo híbrido porque al utilizar dos técnicas metaheurísticas, fueron las encargados de realizar la exploración y la explotación de soluciones, empleando un algo-

ritmo para la búsqueda local y otro para la búsqueda global como método de optimización, y así se propuso un algoritmo en el que se obtuvieron buenos resultados encontrando la mejor $smart\_cells$ para la ruta de procesamiento y así obtener los mejores tiempos óptimos, y se verificó la funcionalidad del método propuesto comparándolo con otros algoritmos de la literatura.

Para llegar del modelo propuesto al modelo desarrollado se generó lo siguiente:

Se inicializó una población aleatoria inspirada en el CA para obtener un primer orden de secuenciación de operaciones y de maquinas. Se evaluó la población y se calculó el makespan de esa población inicial de $smart\_cells$. Si se cumplió el número de iteraciones se elige la mejor solución $smart\_cells$ con el makespan menor. Se inicia la primer etapa de optimización aplicando la búsqueda global, y con las soluciones $smart - cells$ anteriores, se vuelve a evaluar la población aplicando los algoritmos genéticos, mediante la generación de varios operadores, como el de mutación y cruce genético. Estos operadores optimizan principalmente la secuencia de operaciones para formar una nueva vecindad $smart\_cells$, en el que se selecciona la mejor modificación. Se inicia la segunda etapa de optimización con la búsqueda local, aplicando la escalada de colinas para refinar la búsqueda explotando la información existente de la $smart\_cells$ donde se asigna la mejor máquina a cada operación crítica para disminuir el valor del makespan.

La ventaja de haber utilizado una vecindad tipo CA, es que permitió la aplicación simultanea de operaciones genéticas, y dio al GA-RRHC una mejor capacidad de exploración del espacio de búsqueda. Mientras que con la escalada de colinas Hill-climbing fue fácil de implementar debido a que realizó la explotación de soluciones mediante la mejora iterativa, donde llegó con una configuración completa que se traía desde la parte genética, y a partir de esta, se aplicaron modificaciones a esta CA con el RRHC para mejorar su calidad para obtener el mejoramiento iterativo y reiniciar desde otro punto si fuese necesario obteniendo mejoras a la CA y así encontrar el mejor tiempo óptimo y evitar caer en un óptimo local.

Con la estrategia heurística implementada para la resolución del FJSSP se obtuvo un modelo capaz de encontrar un buen balance entre intensificación y diversificación, obteniendo un equilibrio en el proceso de búsqueda.

Se comprobó la eficiencia del método propuesto mediante el análisis estadístico de las muestras obtenidas, se aplicaron 4 conjuntos de datos, abordando un total de 101 instancias, para la experimentación numérica del GA-RRHC, en el que se obtuvo en cada una de ellas el makespan para los bench-

marks. El tiempo de ejecución se descarto por que cada algoritmo con los que se comparó el GA-RRHC tienen su propia arquitectura, sus propias características técnicas, su propia complejidad computacional donde ejecutaron sus algoritmos, por lo que solo se considero su complejidad en cuestión del orden de operaciones y asignación de máquinas en cada solución.

Se realizó un análisis estadístico por cada experimento para obtener la prueba de significancia de la hipótesis, con el fin de tomar decisiones y poder concluir con los datos del experimento, si la hipótesis nula (H0) y la hipótesis alternativa (H1) y se tomar la decisión si se acepta o rechaza la hipótesis nula, lo que significa que si se se aceptó significo que no hay diferencia significativa entre los algoritmos que se compararon y si se rechaza significó que si la hubo. Se aplicó la prueba no perimétrica de Friedman que fue la encargada de obtener el valor de significancia p que se utilizo para comparar el desempeño de cada algoritmo con el del GA-RRHC y en la toma de decisión para la hipótesis.

Concluyendo que en cada experimiento hubo diferencias significativas, y en base a los resultados, el GA-RRHC es estadísticamente competitivo con los otros algoritmos, con el que se alcanzó la optimización para el FJSSP aplicando el algoritmo propuesto. Con los resultados obtenidos se obtuvo un buen rendimiento en comparación con otros algoritmos que se tomaron como referencia, y se obtuvieron excelentes resultados en problemas de alta flexibilidad.

Con el GA-RRHC a presenta una nueva forma de resolver programación de tareas aplicando vecindarios inspirados en el autómata celular (CA) que se planteo en este trabajo de tesis. Este algoritmo propuesto se podría aplicar también en otro tipo de problemas de programación de tareas, donde se puede aplicar vecindarios tipos CA para aplicar aquella exploración concurrente y aquellas acciones de explotación de solución en el espacio de búsqueda.

VI.0.1 Trabajo a futuro

En la búsqueda local donde se tiene que realizar la explotación de soluciones, se puede implementar otro método de búsqueda o una variante al RRHC y proponer otro tipo de estrategias para implementar en problemas con baja flexibilidad y optimizar la secuencia de operaciones para optimiar el FJSSP y poder obtener el mejor valor conocido de las instancias que tengan menos flexibilidad .

Este algoritmo GA-RRHC que utilizó una programación inspirada en CA,

abordó problemas donde su función objetivo es encontrar el mejor makespan o también conocido como la función costo, esta idea se puede ampliar y extender la metodología para abordar problemas multiobjetivo.

Esperando sea implementado en un estudio de caso real para demostrar la eficiencia del modelo propuesto con datos reales en un SMF.

VI.0.2 Publicaciones

Las publicaciones realizadas a lo largo de esta tesis doctoral en relación al tema de investigación se muestran a continuación :

Escamilla-Serna NJ, Seck-Tuoh-Mora JC, Medina-Marin J, Hernandez-Romero N, Barragan-Vite I, Corona Armenta JR. A global-local neighborhood search algorithm and tabu search for flexible job shop scheduling problem. *PeerJ Computer Science 7:e574* , 2021. https://doi.org/10.7717/peerj-cs.574

Escamilla-Serna, NJ, Seck-Tuoh-Mora, JC, Medina-Marín, J, Barragan-Vite, I, Corona-Armenta, JR. Método híbrido para optimizar el Flexible Job Shop Scheduling Problem. *Pädi Boletín Científico De Ciencias Básicas E Ingenierías Del ICBI*,10(Especial2), 56-64, 2022. https://doi.org/10.29057/icbi.v10iEspecial2.8651

Escamilla-Serna NJ, Seck-Tuoh-Mora JC, Medina-Marin J, Barragan-Vite I, Corona-Armenta JR. A Hybrid Search Using Genetic Algorithms and Random-Restart Hill-Climbing for Flexible Job Shop Scheduling Instances with High Flexibility. *Applied Sciences*. 12(16):8050, 2022. https://doi.org/10.3390/app12168050

Bibliografía

[1] Ehsan Ahmadi, Mostafa Zandieh, Mojtaba Farrokh, and Seyed Mohammad Emami. A multi objective optimization approach for flexible job shop scheduling problem under random machine breakdown by evolutionary algorithms. *Computers & Operations Research*, 73:56–66, 2016.

[2] Malek Alzaqebah, Sana Jawarneh, Maram Alwohaibi, Mutasem K. Alsmadi, Ibrahim Almarashdeh, and Rami Mustafa A. Mohammad. Hybrid brain storm optimization algorithm and late acceptance hill climbing to solve the flexible job-shop scheduling problem. *Journal of King Saud University – Computer and Information Sciences*, page 12, 2020.

[3] M. Amiri, M. Zandieh, Yazdani M., and A. Bagheri. A variable neighbourhood search algorithm for the flexible job-shop scheduling problem. *International Journal of Production Research*, 48(19):5671—-5689, 2010.

[4] Muhammad Kamal Amjad, Shahid Ikramullah Butt, Rubeena Kousar, Riaz Ahmad, Mujtaba Hassan Agha, Zhang Faping, Naveed Anjum, and Umer Asgher. Recent research trends in genetic algorithm based flexible job shop scheduling problems. *Mathematical Problems in Engineering*, page 32, 2018.

[5] Mohammed A. Awad and Hend M Abd-Elaziz. A new perspective for solving manufacturing scheduling based problems respecting new data considerations. *processes*, 9(10):1–32, 2021.

[6] W. Barnes and J.B. Chambers. Flexible job shop scheduling by tabu search. *Graduate program in operations research and industrial engineering*, ORP:96–09, 1996.

[7] Adil Baykasoğlu, Fatma. S. Madenoğlu, and Alper Hamzadayı. Greedy randomized adaptive search for dynamic flexible job-shop scheduling. *Journal of Manufacturing Systems*, 56:425–451, 2020.

[8] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268—-308, 2003.

[9] R. S. Boyer and J. Strother Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[10] P. Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of operations research*, 41(3):157–183, 1993.

[11] P. Brucker and R. Schlie. Job-shop scheduling with multi-purpose machines. *Computing*, 45(4):369–375, 1990.

[12] Peter Brucker. *Scheduling Algorithms*, volume Fifth Edition. Springer, 2006.

[13] Peter Brucker and Sigrid Knust. *Complex Scheduling*. Springer-Verlag, 2006.

[14] A. Caldeira, R. H.and Gnanavelbabu. Solving the flexible job shop scheduling problem using an improved jaya algorithm. *Computers Industrial Engineering*, 137:1–16, 2019.

[15] Hao-Chin Chang, Yeh-Peng Chen, Tung-Kuan Liu, and Jyh-Horng Chou. Solving the flexible job shop scheduling problem with makespan optimization by using a hybrid taguchi-genetic algorithm. *IEEE Access*, pages 1740—-1754, 2015.

[16] Imran Ali Chaudhry and Abid Ali Khan. A research survey: review of flexible job shop scheduling techniques. *International Transactions in Operational Research*, 23:551–591, 2015.

[17] Ping Che, Zhenhao Tang, Hua Gong, and Xiaoli Zhao. An improved lagrangian relaxation algorithm for the robust generation self-scheduling problem. *Mathematical Problems in Engineering*, page 12, 2018.

[18] Haoxun Chen, Jiirgen Ihlow, and Carsten Lehmann. A genetic algorithm for flexible job-shop scheduling. *IEEE International Conference on Robotics & Automation*, 2:1120–1125, 1999.

[19] Ronghua Chen, Bo Yang, Shi Li, and Shilong Wang. A self-learning genetic algorithm based on reinforcement learning for flexible job-shop scheduling problem. *Computers Industrial Engineering*, 149(106778–), 2020.

[20] Min Dai, Dunbing Tang, Adriana Giret, and Miguel A. Salido. Multi-objective optimization for energy-efficient flexible job shop scheduling problem with transportation constraints. *Robotics and Computer Integrated Manufacturing*, 59:143–157, 2019.

[21] Vahid Majazi Dalfarda and Ghorbanali Mohammadi. Two meta-heuristic algorithms for solving multi-objective flexible job-shop scheduling with parallel machine and maintenance constraints. *Computers and Mathematics with Applications*, 64:2111–2117, 2012.

[22] S. Dauzère-Pérès and J. Paulli. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research*, 70(1997):281–306, 1997.

[23] Fantahun M. Defersha and Danial Rooyani. An efficient two-stage genetic algorithm for a flexible job-shop scheduling problem with sequence dependent attached/detached setup, machine release date and lag-time. *Computers & Industrial Engineering*, 147:106605, 2020.

[24] Qianwang Deng, Guiliang Gong, Xuran Gong, Like Zhang, Wei Liu, and Qinghua Ren. A bee evolutionary guiding nondominated sorting genetic algorithm ii for multiobjetive flexible job shop scheduling. *Computational Intelligence and Neuroscience*, pages 1–20, 2017.

[25] Joaquín Derrac, Salvador García, Daniel Molina, and Francisco Herrera. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1(1):3–18, 2011.

[26] Haojie Ding and Xingsheng Gu. Hybrid of human learning optimization algorithm and particle swarm optimization algorithm with scheduling

strategies for the flexible job-shop scheduling problem. *Neurocomputing*, 414:313–332, 2020.

[27] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, 1992.

[28] A. E. Eiben and J. E. Smith. Introduction evolutionary computing. *Natural Computing Serie*, 2:287, 2015.

[29] Nayeli Jazmin Escamilla-Serna, Juan Carlos Seck-Tuoh-Mora, Joselito Medina-Marin, Irving Barragan-Vite, and Jose Ramon Corona Armenta. A hybrid search using genetic algorithms and random-restart hill-climbing for flexible job shop scheduling instances with high flexibility. *PeerJ Computer Science*, 12:8050, 2022.

[30] Nayeli Jazmin Escamilla-Serna, Juan Carlos Seck-Tuoh-Mora, Joselito Medina-Marin, Norberto Hernandez-Romero, Irving Barragan-Vite, and Jose Ramon Corona Armenta. A global-local neighborhood search algorithm and tabu search for flexible job shop scheduling problem. *PeerJ Computer Science*, 7:e574, 2021.

[31] Jiaxin Fan, Weiming Shen, Liang Gao, Chunjiang Zhang, and Ze Zhang. A hybrid jaya algorithm for solving flexible job shop scheduling problem considering multiple critical paths. *Journal of Manufacturing Systems*, 60:298–311, 2021.

[32] P. Fattahi, M. Mehrabad Saidi, and F. Jolai. Mathematical modeling and heuristic approaches to flexible job shop scheduling problems. *Journal of Intelligent Manufacturing*, 18(3):331—-342, 2007.

[33] T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 2:1–7, 1995.

[34] H. Fisher and G. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. pages 225–251, 1963.

[35] Jie Gao, Linyan Sun, and Mitsuo Gen. A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. *Computers and Operations Research*, 35:2892–2907, 2008.

[36] Kai Zhou Gao, Ponnuthurai Nagaratnam Suganthan, Tay Jin Chua, Chin Soon Chong, Tian Xiang Cai, and Qan Ke Pan. A two-stage artificial bee colony algorithm scheduling flexible job-shop scheduling problem with new job insertion. *Expert Systems with Applications*, 42(21):7652–7663, 2015.

[37] Kaizhou Gao, Zhiguang Cao, Le Zhang, Zhenghua Chen, Han Yuyan, and Quanke Pan. A review on swarm intelligence and evolutionary algorithms for solving flexible job shop scheduling. *IEEE/CAA Journal of Automatica Sinica*, 6(4):1–13, 2019.

[38] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13(533-549), 1986.

[39] F. Glover. Tabu seach - part i. *ORSA Journal on Computing*, 1(3):190–206, Summer 1989.

[40] Andreas Goerler, Eduardo Lalla-Ruiz, and Stefan Voß. Late acceptance hill-climbing matheuristic for the general lot sizing and scheduling problem with rich constraints. *Algorithms*, 13(6):138, 2020.

[41] D. E. Goldberg. Genetic algorithms in search, optimization and machine learning. *Addison-Wesley, Reading, MA.*, 1989.

[42] Guiliang Gong, Raymond Chiong, Qianwang Deng, and Xuran Gong. A hybrid artificial bee colony algorithm for flexible job shop scheduling with worker flexibility. *International Journal of Production Research*, 58(14):4406–4420, 2019.

[43] Guiliang Gong, Qianwang Deng, Xuran Gong, Wei Liu, and Qinghua Ren. A new double flexible job-shop scheduling problem integrating processing time, green production, and human factor indicators. *Journal of Cleaner Production*, 174:560–576, 2018.

[44] D. G. Green and T. Leishman. *Computing and Complexity — Networks, Nature and Virtual Worlds*, volume 10: Philosophy of Complex Systems. Elsevier, 2011.

[45] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. *Congress on Numerical Methods in Combinatorial Optimization*, 1986.

[46] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press Ltd, 1975.

[47] Holger H. Hoos and Thomas Stutzle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[48] Manar Ibrahim Hosny. Investigating heuristic and meta-heuristic algorithms for solving pickup and delivery problems. *A thesis submitted in partial fulfilment of the requirement for the degree of Doctor of Philosophy*, pages 1–242, 2010.

[49] Johann Hurink, Bernd Jurisch, and Monika Thole. Tabu search for the job-shop scheduling problem with multi-purpose machines. *Operations-Research-Spektrum*, 15(4):205–215, 1994.

[50] S. M. Johnson. Optimal two and three stage production schedules with setup times included. *The Rand Corporation*, pages 1–10, 1953.

[51] I. Kacem and Borne P. Hammadi. Approach by localization and multiobjective evolutionary optimization for flexible job-shop scheduling problems. *IEEE Trans, Syst.*, 32(1):1–13, 2002.

[52] Imed Kacem, Slim Hammadi, and Pierre Borne. Pareto-optimality approach for flexible job-shop scheduling problems: hybridization of evolutionary algorithms and fuzzy logic. *Mathematics and computers in simulation*, 60(3-5):245–276, 2002.

[53] J. Kennedy and R. Eberhart. Particle swarm optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks*, 4:1942–1948., 1995.

[54] Eugene L. Lawler, Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, and David B. Shmoys. Chapter 9 sequencing and scheduling: Algorithms and complexity. 4:445–522, 1993.

[55] Stephen Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). *Graduate School of Industrial Administration*, 1984.

[56] Jun-qing Li, Quan-ke Pan, and Yun-Chia Liang. An effective hybrid tabu search algorithm for multi-objective flexible job-shop scheduling problems. *Computers & Industrial Engineering*, 59:647–662, 2010.

[57] Jun-Qing Li, Quan-Ke Pan, and M. Fatih Tasgetiren. A discrete artificial bee colony algorithm for the multi-objective flexible job-shop scheduling problem with maintenance activities. *Applied Mathematical Modelling*, 38:1111–1132, 2014.

[58] Xinyu Li and Liang Gao. An effective hybrid genetic algorithm and tabu search for flexible job shop scheduling problem. *Journal of Production Economics*, 174:93—-110, 2016.

[59] Xinyu Li and Liang Gao. An effective hybrid genetic algorithm and tabu search for flexible job shop scheduling problem. *International Journal of Production Economics*, 174:93–110, 2016.

[60] Xixing Li, Zhao Peng, Baigang Du, Jun Guo, Wenxiang Xu, and Kejia Zhuang. Hybrid artificial bee colony algorithm with a rescheduling strategy for solving flexible job shop scheduling problems. *Computers & Industrial Engineering*, 113:10—-26, 2017.

[61] Z.C. Li, B. Qian, R. Hu, L.L. Chang, and J.B. Yang. An elitist non-dominated sorting hybrid algorithm for multi-objective flexible job-shop scheduling problem with sequence-dependent setups. *Knowledge-Based Systems*, 173:83–112, 2019.

[62] Jian Lin, Lei Zhu, and Zhou-Jing Wang. A hybrid multi-verse optimization for the fuzzy flexible job-shop scheduling problem. *Computers & Industrial Engineering*, 127:1089–1100, 2019.

[63] González MA, Vela CR, and Varela R. Scatter search with path relinking for the flexible job shop scheduling problem. *European Journal of Operational Research*, 245(1):35–45, 2015.

[64] B. Mallia, M. Das, and C. Das. Fundamentals of transportation problem. *International Journal of Engineering and Advanced Technology (IJEAT)*, 10:90–103, 2021.

[65] Rafael Martí, Panos Pardalos, and ed. Resende, Mauricio. *Handbook of Heuristics*. 2018.

[66] L. M. Mastrolilli, M. Gambardella. Electronic references, 2000. A library of problem instances is given in FJSPLIB.

[67] Monaldo Mastrolilli and Luca Maria Gambardella. Effective neighborhood functions for the flexible job shop problem. *Journal of scheduling*, 3(1):3–20, 2000.

[68] Harold V McIntosh. *One dimensional cellular automata*. Luniver Press, 2009.

[69] Tao Meng, Quan-Ke Pan, and Hong-Yan Sang. A hybrid artificial bee colony algorithm for a flexible job shop scheduling problem with overlapping in operations. *International Journal of Production Research*, pages 1–15, 2018.

[70] B. Mihoubi, B. Bouzouia, and M. Gaham. Reactive scheduling approach for solving a realistic flexible job shop scheduling problem. *International Journal of Production Research*, 59(19):5790–5808, 2021.

[71] N. Mladenovic and P. Hansen. Variable neighbourhood search. *Computers and Operations Research*, pages 1097–1100, 1997.

[72] Sheldon. H. Morrison, David R.and Jacobson, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79—-102, 2016.

[73] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Number 2. Dover Publications, Inc., 1982, 1998.

[74] F. Pezzellaa, G. Morgantia, and G. Ciaschettib. A genetic algorithm for the flexible job-shop scheduling problem. *Computers & Operations Research*, 35:3202–3212, 2008.

[75] M. L. Pinedo. *Scheduling Theory, Algorithms, and Systems*. Number Fifth Edition. Springer, 2016.

[76] Ren Qing-dao-er ji and Yuping Wang. A new hybrid genetic algorithm for job shop scheduling problem. *Computers & Industrial Engineering*, pages 2291–2299, 2012.

[77] Edilson Reis Rodriguez Kato, Gabriel Diego de Aguiar Aranha, and Roberto Hideaki Tsunaki. A new approach to solve the flexible job shop

problem based on a hybrid particle swarm optimization and random-restart hill climbing. *Computers & Industrial Engineering*, 125:178–189, 2018.

[78] Valentina Salazar Alvarez. *Estado del arte del problema de secuenciaci´on de tareas implementando reglas de despacho*. Universidad Tecnológica de Pereira, 2019.

[79] Nasser Shahsavari-Pour and Behrooz Ghasemishabankareh. A novel hybrid meta-heuristic algorithm for solving multi objective flexible job shop scheduling. *Journal of Manufacturing Systems*, 32:771–780, 2013.

[80] Natalia V. Shakhlevich, Yuri N. Sotskov, and Frank Werner. Complexity of mixed shop scheduling problems: A survey. *European Journal of Operational Research*, 120(2):343—351, 2000.

[81] Liji Shen, Stéphane Dauzère-Pérès, and Janis S. Neufeld. Solving the flexible job shop scheduling problem with sequence-dependent setup times. *European Journal of Operational Research*, 265:503–516, 2018.

[82] M.B.S. Sreekara Reddy, Ch. Ratnam, G. Rajyalakshmi, and V.K. Manupati. An effective hybrid multi objective evolutionary algorithm for solving real time event in flexible job shop scheduling problem. *Measurement*, 114:78–90, 2018.

[83] Lu Sun, Lin Lin, and Haojie Lib. Large scale flexible scheduling optimization by a distributed evolutionary algorithm. *Computers & Industrial Engineering*, 128:894–904, 2018.

[84] Hongtao Tang, Rong Chen, Yibing Li, Zhao Peng, Shunsheng Guo, and Yuzhu Du. Flexible job-shop scheduling with tolerated time interval and limited starting time interval based on hybrid discrete pso-sa: An application from a casting workshop. *Applied Soft Computing Journal*, 78:176–194, 2019.

[85] Stephen Wolfram. *A new kind of science*, volume 5. Wolfram media Champaign, IL, 2002.

[86] J. Wu, G. Wu, and J. Wang. Flexible job shop scheduling problem based on hybrid aco algorithm. *International Journal of Simulation Modelling*, 16(3):497–505, 2017.

[87] Xiuli Wu, Xianli Shen, and Congbo Li. The flexible job-shop scheduling problem considering deterioration effect. *Computers & Industrial Engineering*, 2019:1004–1024, 2019.

[88] Weijun Xia and Zhiming Wu. An effective hybrid optimization approach for multi-objective flexible job-shop scheduling problems. *Computers & Industrial Engineering*, 48:409—-425, 2005.

[89] Naiming Xie and Nanlei Chen. Flexible job shop scheduling problem with interval grey processingtime. *Applied Soft Computing*, 70:513—524, 2018.

[90] Takeshi Yamada and RRyohei Nakano. Job-shop scheduling. *IEE control engineering*, 55:134–160, 1997.

[91] Yu Yang, Min Huang, Zhen Yu Wang, and Qi Bing Zhu. Robust scheduling based on extreme learning machine for bi-objective flexible job-shop problems with machine breakdowns. *Expert Systems with Applications*, 158:113545, 2020.

[92] Yingchen Yu. A research review on job shop scheduling problem. volume 253, page 5. 2021 International Conference on Environmental and Engineering Management (EEM 2021), E3S Web of Conferences, 2021.

[93] Yuan Yuan, Hua Xu, and Jiadong Yang. A hybrid harmony search algorithm for the flexible job shop scheduling problem. *Applied Soft Computing*, 13(7):3259–3272, 2013.

[94] K. Zahia and D. Messaoud. A meta-heuristics for the flexible manufacturing systemn problem. *International Review of Machanical Engineering*, 4(3):330–335, 2010.

[95] Gabriel Zambrano Rey, Abdelghani Bekrar, Damien Trentesaux, and Bing-Hai Zhou. Solving the flexible job-shop just-in-time scheduling problem with quadratic earliness and tardiness costs. *International Journal of Advanced Manufacturing Technology*, 81(9-12):1871–1891, 2015.

[96] R. Zarrouk and A. Bennour, I.E.and Jemai. A two-level particle swarm optimization algorithm for the flexible job shop scheduling problem. *Swarm Intell*, 13:145–168, 2019.

[97] Guohui Zhang, Xinyu Shao, Peigen Li, and Liang Gao. An effective hybrid particle swarm optimization algorithm for multi-objective flexible job-shop scheduling problem. *Computers & Industrial Engineering*, 56:1309—-1318, 2009.

[98] Xiao-long Zheng and Ling Wang. A knowledge-guided fruit fly optimization algorithm for dual resource constrained flexible job-shop scheduling problem. *International Journal of Production Research*, 54(18):5554–5566, 2016.

[99] G.I. Zobolas, C.D. Tarantilis, and G. Ioannou. Exact, heuristic and meta-heuristic algorithms for solving shop scheduling problems. *Studies in Computational Intelligence (SCI)*, 128:1–40, 2008.