Algoritmo Híbrido Aplicado al Problema de Asignación de Trabajos en un Taller Flexible

Miguel Federico Hernández Molina

Trabajo de Grado para Optar al Título de Ingeniero Industrial

Director

Henry Lamos Diaz

Ph.D en Matemática Aplicada

Universidad Industrial de Santander

Facultad de Ingenierías Físico Mecánicas

Escuela de Estudios Industriales y Empresariales

Bucaramanga

2018

Tabla de Contenido

Lista de Tablas

Pág

Lista de Figuras

Pág

Lista de Apéndices

Pág

Resumen

Título**:** Algoritmo híbrido aplicado al problema de asignación de trabajos en un taller flexible*

Autor: Miguel Federico Hernández Molina**

Palabras clave: Optimización, problema del taller flexible, secuenciamiento y programación de operaciones, modelo del grafo disyunto, algoritmos híbridos.

Descripción:

Los requerimientos actuales de los mercados conducen permanentemente a las empresas a buscar la manera más eficaz y eficiente de programar sus recursos disponibles. Lo anterior conlleva cierto grado de complejidad incluso en procesos de producción a pequeña escala. El problema de la *programación* y *secuenciación* de operaciones *(scheduling and sequencing)* estudia la asignación de recursos de manera óptima a las diferentes actividades de la producción. Su contribución a la mejora de los procesos productivos resalta la importancia de estudiar esta clase de problemas. El objeto de investigación del presente proyecto de grado es la implementación de un algoritmo híbrido para resolver el problema del taller flexible, *FJSP*. El problema del *FJSP* consiste en asignar *n* trabajos que deben ser procesados en *m* máquinas con un nivel mayor de flexibilidad que su antecesor, el *JSP*. Su propósito radica en minimizar uno o varios objetivos previamente establecidos. Entre el objetivo(s) a optimizar, se encuentra generalmente, la minimización del tiempo de la última operación o *makespan*.

El proyecto de grado implementa un *algoritmo genético* híbrido con *búsqueda local* para resolver el *FJSP* con tres criterios de minimización: makespan, carga máxima por máquina y total de carga entre las máquinas. Su desempeño se medirá mediante la validación de instancias de benchmarking. El código se escribió en el programa de computación Matlab.

*Trabajo de grado

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería Industrial. Director: Henry Lamos Díaz, Matemático.

Abstract

Title: Hybrid algorithm applied to the flexible job *

Author: Miguel Federico Hernández Molina.**

Keywords: Optimization, flexible job shop problem, scheduling and sequencing, disjunctive graph model, hybrid algorithms.

Description:

The current requirements of the markets permanently lead companies to find the most effective and efficient way to program their available resources. The above entails a degree of complexity even in small-scale production processes.

Scheduling and sequencing address the problem of how to allocate resources optimally to different production activities. Their contribution to productive processes improvements highlights the importance of studying these kind of problems originated from operations research. The main purpose of this undergraduate research is to solve the problem of the flexible workshop, FJSP, in regard to the implementation of a hybrid algorithm. The problem of the FJSP consist of assign n jobs that must be processed in m machines with a higher level of flexibility than its predecessor, the JSP. Its purpose is to minimize one or more objectives previously established. Among the objective (s) to be optimized, it might be found usually the minimization of the last time operation or makespan.

The project implements a hybrid genetic algorithm with *local search* component to solve the FJSP with three minimization criteria: makespan, maximum load per machine and total load between machines. Its performance will be measured by the validation of benchmarking instances. The code was written in Matlab.

*Bachelor Thesis

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería Industrial. Director: Henry Lamos Díaz, Matemático.

Introducción

Los requerimientos actuales de los mercados conducen permanentemente a las empresas a buscar la manera más eficaz y eficiente de programar sus recursos disponibles. Lo anterior conlleva cierto grado de complejidad incluso en procesos de producción a pequeña escala. La investigación de operaciones ha desarrollado técnicas de optimización que permiten resolver problemas de naturaleza práctica en amplios sectores de la industria y la economía; contribuyendo a mejorar la programación de operaciones de alta calidad y bajo costo.

El problema de la *programación* y *secuenciación* de operaciones *(scheduling and sequencing)* consiste en cómo asignar los recursos de manera óptima a las diferentes actividades de la producción. Su contribución a la mejora de los procesos productivos resalta la importancia de estudiar esta clase de problemas. El objeto de estudio del presente proyecto de grado es la implementación de un algoritmo híbrido para resolver el problema del taller flexible, *FJSP*.

El problema del Taller Flexible (*Flexible Job Shop Problem, FJSP*) es una generalización del problema del Taller (*Job Shop Problem, JSP*). Los problemas tipo Job-shop modelan diversos ámbitos de la planificación y control de la producción y son especialmente adecuados para mejorar los procesos de manufactura. El problema del *FJSP* consiste en asignar *n* trabajos que deben ser procesados en *m* máquinas con un nivel mayor de flexibilidad que su antecesor, el *JSP*. Su propósito radica en minimizar uno o varios objetivos previamente establecidos. Entre el objetivo(s) a optimizar, se encuentra generalmente, la minimización del tiempo de la última operación o *makespan*.

El empleo de métodos exactos para encontrar soluciones óptimas al *FJSP* sólo es posible en las instancias más pequeñas. Al igual que el *JSP*, se trata de un problema combinatorio, clase *NP-Hard*. La imposibilidad práctica de resolver problemas combinatorios del tipo *JSP* o *FJSP* en un tiempo computacionalmente razonable, ha orientado parte de la investigación a desarrollar técnicas heurísticas y metaheurísticas que permiten encontrar soluciones subóptimas de gran calidad. Entre el grupo de metaheurísticas más significativas que se han implementado para resolver problemas tipo Job Shop, se encuentran: recocido simulado *(simulated annealing)*, búsqueda tabú *(tabu search)*, algoritmos evolutivos *(evolutionary algorithms)*, optimización por colonia de hormigas *(ant colony optimization)* y optimización por enjambre de partículas *(particle swarm optimization)*.

La hibridación entre algunas de las metaheurísticas antes mencionadas, ha supuesto un avance adicional en el desempeño de estos algoritmos. El concepto de hibridación consiste en reforzar las fortalezas de dos o más técnicas con el objeto de que el algoritmo resultante posea un desempeño superior a los originales.

El presente proyecto de grado implementa un *algoritmo genético* híbrido con *búsqueda local* para resolver el *FJSP* con tres criterios de minimización: makespan, carga máxima por máquina y total de carga entre las máquinas. Su desempeño se medirá mediante la validación de instancias de benchmarking. El código se escribirá en el programa de computación Matlab.

1 Objetivos

1.1 Objetivo General

Desarrollar un algoritmo de optimización híbrido para minimizar el makespan del problema de asignación de trabajos en un taller flexible (Flexible Job Shop Problem FJSP).

1.2 Objetivos Específicos

- Revisar la literatura existente concerniente a los métodos metaheurísticos con enfoques evolutivos que plantean diferentes metodologías para resolver el problema del taller flexible *FJSP*.
- Desarrollar el marco de trabajo basado en un algoritmo híbrido para el problema de minimizar el *makespan* en el taller flexible *FJSP*.
- Implementar en MATLAB el algoritmo híbrido para el problema de minimizar el *makespan* en el taller flexible *FJSP*.
- Validar el algoritmo construido a partir del uso de instancias encontradas en la revisión de la literatura.
- Elaboración de artículo académico que presente el desarrollo de la investigación y los resultados obtenidos.

2 Optimización matemática

Los modelos, en general, buscan capturar y abstraer la esencia de un evento u objeto del cual se indaga algo. Los modelos en Investigación de Operaciones sirven para mostrar relaciones que

permiten describir problemas de la vida real, y así facilitar su análisis. Debido a que el conocimiento adquirido está basado en una representación, es muy posible que las conclusiones que se derivan del análisis del modelo corresponden al modelo utilizado y no tanto al problema directamente. En este sentido, se espera que los modelos objetos de estudios tengan validez, y por esta razón se prefieren métodos robustos basados en la matemática para la modelación de problemas (Hillier & Lieberman. 2001).

La motivación principal de la Optimización Matemática es desarrollar un cuerpo teórico que sirva de fundamento al diseño de algoritmos eficientes que permitan resolver problemas de cierta complejidad y puedan ser implementados en un sistema computacional.

2.1 Modelos de optimización matemática

Existe una gran cantidad de problemas de optimización de naturaleza práctica que no pueden ser resueltos usando métodos exactos, es decir, no es posible encontrar su solución óptima con esfuerzos computacionales aceptables; aún en el caso de poder contar con computadores de alta velocidad operando en paralelo. La complejidad matemática y el tamaño son elementos comunes a esta clase de problemas.

Los problemas de optimización se pueden encontrar en diferentes campos: problemas de carácter puramente teórico en análisis combinatorio, investigación de fenómenos físicos en las ciencias naturales, estudio de métodos para resolver sistemas de ingeniería o diferentes aplicaciones en el área de la administración y los negocios.

Un problema de optimización puede definirse como una pareja (S, f), donde $\boldsymbol{S} \subseteq \mathbb{R}^{\boldsymbol{n}}$ representa el conjunto de soluciones factibles y $f: \boldsymbol{S} \rightarrow \mathbb{R}^{\boldsymbol{n}}$la función objetivo a optimizar. Si $\boldsymbol{S} \in$

$\mathbb{R}$, la función objetivo asigna a cada solución $s \in \boldsymbol{S}$ del espacio solución un número real que indica su valor. La función objetivo f permite definir una relación total de orden entre cualquier par de soluciones en el espacio solución.

A una solución $s^* \in \boldsymbol{S}$ se le denomina óptimo global, si $\forall s \in \boldsymbol{S}, f(s^*) \leq f(s)$. El principal objetivo al resolver un problema de optimización es encontrar el óptimo global s^*. Un problema puede poseer más de un óptimo global o no poseer alguno en absoluto. En caso de ser requerido, el problema puede reformularse con el propósito de que encuentre todas las soluciones óptimas globales (en caso de existir).

Diferentes familias de modelos de optimización son empleados en situaciones prácticas para formular y resolver problemas del tipo de *toma de decisiones.* Uno de los modelos más comunes en programación matemática es el de programación lineal (PL), el cual puede ser formulado de la siguiente manera:

$$Minimizar\ c \cdot \boldsymbol{x}$$

$$sujeto\ a:$$

$$\boldsymbol{A} \cdot \boldsymbol{x} \leq \boldsymbol{b}$$

$$\boldsymbol{x} \geq 0$$

Donde $\boldsymbol{c} = (c_1, c_2, \dots, c_n)$ es un vector fila, $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ es un vector columna, $\boldsymbol{A} = (a_{ij})$ una matriz de $m \times n$, $\boldsymbol{b} = (b_1, b_2, \dots, b_m)$ es un vector columna y 0 es un vector columna n dimensional nulo. La función objetivo $c_1x_1, c_2x_2, \dots, c_n$ es la función a minimizar y generalmente es denotada con la letra *z*. Los coeficientes $c_1, c_2, \dots, c_n$ son los coeficientes de costo (conocidos),

y $x_1, x_2, \dots, x_n$ son las variables de decisión que deben determinarse (Bazara, Jarvis, & Sherali, 1998).

En un problema de programación lineal, tanto la función objetivo a ser optimizada, $c \cdot \boldsymbol{x}$, como las restricciones de la forma $\boldsymbol{A} \cdot \boldsymbol{x} \leq \boldsymbol{b}$, son funciones lineales. El modelo anterior es uno de los más eficientes para resolver un tipo de problemas en optimización.

La programación *no lineal* resuelve problemas de programación matemática en los cuales la función objetivo y/o las restricciones no son lineales. Un problema de optimización no lineal continuo, consiste en minimizar una función $f: \boldsymbol{S} \subset \mathbb{R}^{\boldsymbol{n}} \to \mathbb{R}$ en un dominio continuo. Los modelos no lineales continuos son, sin embargo, mucho más difíciles de resolver que los problemas de programación lineal. No obstante, varias técnicas de modelado pueden llegar a emplearse para linealizar un problema de este tipo: linealizar un producto de variables, condiciones lógicas, conjunto ordenado de variables, etc. Las técnicas de linealización suelen introducir en general variables y condiciones extras al modelo y en algunos casos, algún grado de aproximación (Talbi, 2009).

Heurísticas específicas inspiradas en el método simplex como el caso del algoritmo de Nelder-Mead es usado con frecuencia para resolver problemas de programación no lineal. En el caso de problema cuadrático y convexo continuo, algunos algoritmos exactos pueden resolver ciertos problemas de baja complejidad. Desafortunadamente, algunas propiedades de los problemas no lineales como alta dimensionalidad, multimodalidad, epístasis (interacción de los parámetros) y no diferenciabilidad, pueden dejar fuera de alcance a los enfoques tradicionales.

En términos de algoritmos de optimización, la teoría de optimización continua se encuentra más desarrollada que la discreta. Sin embargo, existen abundantes casos de situaciones prácticas que

exigen modelamiento en variables discretas. Los problemas de optimización discreta surgen con relativa naturalidad al considerar recursos indivisibles (máquinas, personal, etc.).

Los problemas de programación entera mixta generalizan el modelo de programación lineal y permite variables de decisiones discretas y continuas. Las técnicas de relajación, el enfoque de descomposición y los algoritmos de planos cortantes han mejorado la capacidad de resolución de la programación entera mixta. Los algoritmos enumerativos como ramificación y poda (*branch and bound*) pueden ser usados en instancias pequeñas para resolver problemas de programación entera y entera mixta.

2.2 Problemas de optimización combinatoria

Una clase más general de problemas de programación entera son los *problemas de optimización combinatoria*. Se caracterizan por permitir que la función objetivo y las restricciones puedan ser de tipo diferente (no lineal, no analítica, caja negra, etc.) Permite únicamente variables de decisión discreta y un espacio de búsqueda finito. El ag*ente viajero* (*traveling salesman problem, TSP*) es seguramente el problema de optimización combinatoria más conocido y ampliamente estudiado. El problema consiste en un conjunto de *n* de ciudades y una matriz de distancias $d_{n,n}$, donde cada elemento $d_{i,j}$ representa la distancia entre las ciudades *i* y *j*. El problema exige encontrar un tour (comenzando y terminado en el mismo punto) de costo mínimo, el cual puede ser expresado en términos de tiempo o distancia, es decir, recorrer el mínimo de kilómetros o hacerlo en el menor tiempo posible. Un tour visita cada ciudad exactamente una vez (*ciclo hamiltoniano*). El tamaño del espacio de búsqueda es n !. La Tabla 2 muestra la explosión combinatoria del número de soluciones dependiendo del número de ciudades.

Tabla 1
Efecto del número de ciudades en el tamaño del espacio solución.

Número de ciudades n.	**Tamaño del espacio solución.**
5	120
10	3´628.800
75	$2{,}5 \times 10^{109}$

Nota: Adaptado de Talbi, E. (2009). Metaheuristics: From design to implementation. New Jersey: John Wiley & Sons.

Se han planteado algoritmos para resolver problemas combinatorios en función del tamaño de la instancia del problema, que se juzgan son difíciles de resolver. La instancia de un problema permite medir y contrastar los resultados obtenidos en la aplicación de los diferentes métodos de solución y están compuestas por un conjunto de datos característicos del problema.

En avances recientes se han estudiado casos especiales de solución polinomial en problemas de optimización combinatoria denominados “difíciles”. Tales casos se consideran especiales por su estructura de costos, geometría del problema, la estructura del grafo subyacente o su tratamiento por análisis de algoritmos especiales (Burkard, 1997).

En este tipo de problemas, las variables se agrupan en varios conjuntos que representan objetos que incluyen una estructura de datos compleja, como permutaciones, grafos, árboles, etc. Cada una de las variables del problema ocupa determinada posición, generando una configuración particular de la estructura. El enfoque combinatorio examina las posibles configuraciones. Los problemas combinatorios buscan establecer cuál es la mejor configuración. Con ese propósito se construyen

una o más funciones de valor sobre el espacio de las configuraciones; si la cantidad de funciones de valor es unitario, el problema se denomina uniobjetivo, si por el contrario, posee dos o más funciones a optimizar, será un problema multiobjetivo.

2.3 Complejidad computacional

En general, los problemas de optimización combinatoria (CO por sus siglas en inglés) se encuentran clasificados de acuerdo a su complejidad computacional. La complejidad computacional de un algoritmo está directamente relacionada con la dificultad del problema a resolver y su tiempo de ejecución, el cual determina la eficiencia del algoritmo; sin embargo, se está empezando a considerar el tiempo en paralelo y la arquitectura del hardware como medidas de complejidad.

2.4 Clases de complejidad

La teoría de la complejidad Computacional proporciona herramientas para medir la dificultad de problemas abstractos, a la vez, en términos absolutos (complejidad intrínseca de un problema) y en términos comparativos con otros problemas (clases de complejidad). Su objetivo fundamental es la clasificación de problemas en función de la resolubilidad algorítmica práctica de los mismos. Para ello, se define un concepto de eficiencia práctica que trata de capturar la idea intuitiva de su solución por medio de recursos computacionales convencionales. Un algoritmo se considera eficiente si la cantidad de recursos necesarios para su ejecución, en el peor caso, está acotada por un polinomio del tamaño del dato de entrada. Ello permite fijar una frontera entre la resolubilidad algorítmica práctica (tratabilidad) y la no resolubilidad algorítmica práctica (intratabilidad)

2.5 Clase P

Un algoritmo es polinómicamente acotado si su complejidad en el peor de los casos está acotada por una función polinómica del tamaño de las entradas (es decir, si existe un polinomio $\boldsymbol{p}$ tal que, para toda entrada de tamaño n, el algoritmo termine después de cuando más $\boldsymbol{p}(\boldsymbol{n})$ pasos). $\boldsymbol{P}$ se define como la clase de problemas de decisión que están polinómicamente acotados (Baase & Van Gelder, 2002).

A pesar que no necesariamente todos los problemas en *P* tienen un algoritmo de eficiencia aceptable, sí es posible asegurar que, si un problema no pertenece a *P*, será extremadamente costoso y probablemente imposible de resolver en la práctica. Si bien es factible que la definición de *P* sea demasiado amplia para servir como criterio en el caso de problemas con necesidades de tiempo verdaderamente razonables, el hecho de estar o no en *P* sí es un criterio útil en el caso de problemas renuentes.

Una segunda razón para usar una cota polinómica para definir *P* está dada por las propiedades de *"cierre"* que exhiben los polinomios. Es posible obtener un algoritmo para un problema complejo combinando varios algoritmos para problemas más sencillos. Algunos de los algoritmos más simples podrían trabajar con las salidas o los resultados intermedios de otros. La complejidad del algoritmo compuesto podría estar acotada por la suma, multiplicación y composición de las complejidades de sus algoritmos constitutivos. El hecho que los polinomios sean cerrados bajo estas operaciones, cualquier algoritmo que se construya de diversas formas naturales a partir de varios algoritmos polinómicamente acotados también será polinómicamente acotado. Ninguna

clase más pequeña de funciones que sean cotas de complejidad útiles tiene estas propiedades de cierre (Baase & Van Gelder, 2002).

Un razón adicional para emplear una cota polinómica, reside en hacer a P independiente del modelo de cómputo formal específico que se use. Se pueden emplear varios modelos formales (definiciones formales de algoritmos) para demostrar teoremas rigurosos acerca de la complejidad de los algoritmos y los problemas. Los modelos difieren en cuanto al tipo de operaciones permitidas, los recursos de memoria disponibles y los costos asignados a diferentes operaciones. Un problema que requiere $\Theta(f(n))$ pasos con un modelo podría requerir más de $\Theta(f(n))$ pasos con otro, pero en el caso de prácticamente todos los modelos, si un problema está acotado polinómicamente con uno, estará acotado polinómicamente con los demás.

2.6 Clase NP

La clase NP está formada por todos aquellos problemas que se pueden resolver por algoritmos no deterministas cuyo tiempo de ejecución está acotado por un polinomio; problemas cuyas posibles soluciones pueden ser chequeadas en tiempo polinomial a fin de decidir si realmente son o no soluciones correctas.

Los problemas de la clase NP serían resolubles en tiempo polinomial mediante maquinas que tuvieran la capacidad de realizar en paralelo y de manera independiente, un número no acotado de computaciones. Por tanto, esta clase jugaría el papel de clase de problemas tratables en modo no determinista (Pérez y Sancho, 2003).

2.7 Clase NP completo

Dentro de la clase NP se destaca una subclase de problemas que tienen especial interés: los problemas que son los más difíciles de la clase, en el sentido de que cualquier otro problema de la clase NP puede ser resuelto a través de él con un coste en tiempo adicional de tipo polinomial (mediante una reducción en tiempo polinomial). Es la clase de los problemas denominados $\boldsymbol{NP}$–completos**.**

Figura 1 Relación entre las diferentes clases de complejidad computacional

El interés en la clase de problemas NP–completos (que notaremos NPC) radica principalmente en el hecho de que son candidatos idóneos para atacar la cuestión $P = NP$?. Si existe un problema NP-completo que fuera tratable (es decir, que pertenece a la clase P), entonces la respuesta a la cuestión es afirmativa (es decir, todo problema de la clase NP pertenece a la clase P); y si existe un problema NP-completo que no es tratable, entonces la respuesta es negativa (y, además, resulta que ningún problema NP–completo pertenece a la clase P). Esto significaría que las clases de complejidad P y NPC o bien son iguales o son disyuntas. En 1971, S.A. Cook proporciona el primer ejemplo de un problema NP-completo: el problema SAT de la satisfactibilidad de la lógica

proposicional. Un año después, teniendo presente que la deducibilidad en tiempo polinomial permite generar nuevos problemas NP-completos a partir de otros conocidos, R.M. Karp proporciona 24 ejemplos nuevos de problemas NP-completos. Entre ellos, destacan el problema del recubrimiento de vértices, el problema del recubrimiento exacto, el problema del número cromático, el problema del circuito hamiltoniano y el problema del *agente viajero*. Karp introdujo las notaciones P y NP que ahora son consideradas estándares, y redefinió el concepto de NP–completitud en los términos antes descritos. (Baase & Van Gelder, 2002).

2.8 NP hard

Los problemas NP – duros no son un subconjunto de los problemas NP. Es decir, para los problemas NP –duros no existe un algoritmo polinómico que permita verificar una solución. Son al menos tan difíciles de resolver como los problemas NP – completos, aunque es posible que sean incluso más difíciles (Cote & Mendoza, 2012).

3 El problema del taller flexible *FJSP*

El *FJSP (Flexible Job Shop Problem)* es un tipo de problema mucho más exigente que el *JSP (Job Shop Problem)*, es por ello que se encuentra clasificado entre los *especialmente NP-difíciles* (*strongly NP-hard*), para los cuales aún no se disponen de algoritmos de resolución eficientes que los ejecuten en tiempos computacionales razonables, sobre todo para instancias grandes y/o complejas.

3.1 Estado del arte

Los métodos heurísticos y metaheurísticos desarrollados en el estudio del *FJSP*, se pueden clasificar en dos categorías: *enfoque integrado y jerárquico.* Con el objeto de disminuir la complejidad del *FJSP*, el enfoque *jerárquico* descompone el problema original en dos subproblemas. Resuelve en primer lugar el subproblema de *ruteo*, asignando a cada operación una máquina a partir de un conjunto de máquinas permitidas; en segundo lugar resuelve el subproblema de *programación de actividades,* generando la secuencia de operaciones asignadas en las respectivas máquinas. El enfoque integrado por su parte, no descompone el problema en los subproblemas de *ruteo y programación de operaciones*. De acuerdo con Yazdani, Amiri y Zandieh (2009), en las pruebas de rendimiento, generalmente el enfoque integrado obtiene mejores resultados; no obstante, la mejora en rendimiento conlleva un incremento notable en la dificultad de su implementación

Brucker y Schlie (1990) fueron pioneros en el estudio del *FJSP*; desarrollaron un algoritmo polinomial para resolver el problema con dos trabajos. Brandimarte (1993) aplicó el *enfoque jerárquico* al *FJSP*. Primero resolvió el subproblema de ruteo usando reglas de despacho para luego concentrarse en el subproblema de secuenciamiento, resuelto por un algoritmo genético.

Empleando enfoques opuestos, Hurink, Jurisch y Thole (1994) con base en el enfoque integrado y Barnes & Chambers (1996) en el enfoque jerárquico, propusieron algoritmos tabú para resolver el *FJSP*. Dauzére & Paulli (1997) definieron una nueva estructura de vecindad para el problema, en la cual no se hace distinción entre reasignar y resecuenciar una operación; propusieron un algoritmo tabú eligiendo el enfoque integrado. Mastrolilli & Gambardella (2000) perfeccionaron las técnicas de búsqueda tabú de Dauzére-Perés y presentaron dos funciones de vecindad. Chen, Ihlow y Lehmann (1999) desarrollaron un algoritmo genético para el *FJSP*; ellos dividieron la representación cromosómica en dos partes, la primera definía una política de ruteo y la segunda

delineaba la secuencia de operaciones en cada máquina. Kacem, Hammadi & Borne (2002) propusieron un algoritmo genético controlado por el modelo de asignación, el cual es generado por el enfoque de localización; lo emplearon para resolver el FJSP de un objetivo y el FJSP multi objetivo. Zhang y Gen proponen un algoritmo genético, basado en operaciones multi etapa, tratando el problema desde el punto de vista de la programación dinámica.

Xia y Wu trabajaron el enfoque jerárquico en el estudio del FJSP multi-objetivo. Emplearon optimización por enjambre de partículas (*Particle Swarm Optimization,* PSO) en la asignación de las operaciones a máquinas y un algoritmo tipo recocido simulado, para secuenciar los trabajos en cada máquina. Fattahi, Saidi Mehrabad y Jolai propusieron un modelo matemático y dos algoritmos metaheurísticos (recocido simulado y búsqueda tabú) como solución al problema. Más aún, considerando las dos metaheurísticas previamente desarrolladas y analizando los dos enfoques (integrado y jerárquico), presentaron seis algoritmos diferentes. Pezzella, Morganti y Ciaschetti diseñaron un algoritmo genético para el *FJSP* usando el enfoque integrado, en el cual se presenta una combinación de diferentes estrategias para generar la población inicial, la selección de los individuos para la reproducción y la futura descendencia de nuevos individuos.

Gao, Sun & Gen (2008) estudiaron el FJSP con tres objetivos a *minimizar: el* makespan, la máxima carga de trabajo de cada máquina y la carga total de trabajo sobre todas las máquinas. Ellos desarrollaron un algoritmo genético híbrido con base en el enfoque integrado para este problema. La Búsqueda de Vecindad Variable *(Variable neighborhood search VNS)* es una metaheurística moderna y se fundamenta en cambios sistemáticos de la estructura de la vecindad dentro de una búsqueda de optimización combinatoria.

Las metodologías descritas anteriormente, constituyen los desarrollos más significativos en tiempos recientes, en el estudio y solución por medio de métodos no exactos, del problema del *FJSP*.

4 Metodologías de solución para problemas de Scheduling

Un algoritmo es un conjunto de instrucciones o pasos utilizados para realizar una tarea o resolver un problema. Formalmente, un algoritmo es una secuencia finita de operaciones que se realizan sin ambigüedades, cuya aplicación ofrece una solución a un problema. Los algoritmos se utilizan para el cálculo, procesamiento de datos y el razonamiento automatizado.

A partir de un estado inicial y una entrada inicial (quizás vacía), las instrucciones describen un cómputo que, cuando se ejecuta, se procede a través de un número finito de estados sucesivos bien definidos que generalmente producen una salida y terminan con un estado final. La transición de un estado a otro no es necesariamente determinista; algunos algoritmos, conocidos como algoritmos aleatorios, incorporan una entrada aleatoria.

4.1 Clasificación de algoritmos

Los algoritmos pueden clasificares bajo ciertos criterios. En primer lugar, se puede diferenciar entre algoritmos *deterministas* y algoritmos *no deterministas*. Un algoritmo es determinista si en un conjunto de problemas, todas las ejecuciones del algoritmo producen el mismo resultado final (y además, todos los resultados intermedios también son iguales). Un algoritmo no es determinista si

se introduce una componente de aleatoriedad en el proceso de encontrar la solución y por ello tanto los resultados finales e intermedios no tienen necesariamente que coincidir (Vidal, 2013).

En problemas de optimización y de acuerdo al criterio de la precisión, se clasificaría según:

- Algoritmos exactos son algoritmos que siempre devuelven una solución óptima.
- Algoritmos aproximados son algoritmos que producen soluciones que están dentro de un cierto porcentaje del óptimo. Un algoritmo λ-aproximado devuelve una solución x tal que:

$$OPT \leq c(x) \leq \lambda OPT \; si \; \lambda > 1 \qquad (minimización)$$

$$\lambda OPT \leq c(x) \leq OPT \; si \; \lambda < 1 \qquad (maximización)$$

- Los algoritmos heurísticos son aquellos que producen soluciones sin ninguna garantía de optimalidad y generalmente tienen un tiempo de ejecución mucho menor. Dentro del grupo de los algoritmos heurísticos se encuentran los denominados métodos metaheurísticos, que imitan fenómenos simples observados en la naturaleza y que están asociados con la inteligencia artificial. Estos algoritmos tratan de adaptar el comportamiento de diferentes especies a soluciones de problemas altamente complejos mediante optimización. Cabe destacar los siguientes: enfoques metaheurísticos: algoritmos evolutivos (genéticos): basado en modelos biológicos que emulan el proceso natural de evolución. Algoritmos basados en el comportamiento de las comunidades de hormigas, abejas, etc., simulated annealing, búsqueda heurística (tabú, aleatorios...) y sistemas multiagente.

4.2 Algoritmos exactos

Estos algoritmos encuentran soluciones óptimas en un tiempo determinado, sin embargo para la mayoría de los problemas de aplicación real de Scheduling requieren un tiempo computacional extenso; existen diferentes algoritmos exactos, los más comunes en solución de problemas de Scheduling son el algoritmo Branch and Bound y métodos de descomposición.

El algoritmo *Branch and Bound* es el método exacto más eficiente para la solución de problemas tipo *job shop*. Su principio fundamental es la representación en forma de árbol de todas las posibles soluciones factibles de tal manera que las características y propiedades no compatibles con la solución óptima son detectadas rápidamente y removidas del árbol. Aunque este algoritmo genera una solución óptima, tiende a ser muy lento en problemas de optimización combinatoria debido a sus altos requisitos computacionales. Su principal limitación es la falta de fuertes límites para cortar las ramas del árbol de enumeración lo más pronto posible y reducir el tiempo requerido.

4.3 Algoritmos heurísticos

Para la mayoría de problemas de optimización combinatorial, no existe un algoritmo exacto con complejidad polinómica que encuentre la solución óptima. Más aún, el tamaño del espacio de búsqueda de estos problemas suele ser enorme, lo cual hace inviable el uso de algoritmos exactos ya que la cantidad de tiempo que necesitaría para encontrar una solución no sería computacionalmente aceptable. Debido a estos dos motivos, se necesita utilizar algoritmos *aproximados* o *heurísticos* que permiten obtener una solución de calidad en un tiempo razonable.

El término heurística proviene del vocablo griego *heuriskein*, que puede traducirse como encontrar, descubrir o hallar. Desde un punto de vista científico, el término heurística se debe al

matemático George Polya quien lo empleó por primera vez en su libro How to solve it (Vidal, 2013). Con este término, Polya englobaba las reglas con las que los humanos abordan el conocimiento:

- Buscar un problema parecido que ya haya sido resuelto.
- Determinar la técnica empleada para su resolución así como la solución obtenida.
- En el caso en el que sea posible, utilizar la técnica y solución descrita en el punto anterior para resolver el problema planteado.

Existen dos interpretaciones posibles para el término heurística. La primera de ellas concibe las heurísticas como un procedimiento para resolver problemas. La segunda interpretación, entiende que son una función que permite evaluar la bondad de un movimiento, estado, elemento o solución.

4.4 Algoritmos metaheurísticos

El término metaheurística fue introducido por F. Glover en 1986. Se pretendía definir un procedimiento que guía y modifica otras heurísticas para explorar soluciones más allá de la simple optimalidad local.

Las metaheurísticas son un tipo de métodos aproximados que están diseñados para resolver problemas de optimización combinatoria, especialmente en casos en los cuales las técnicas heurísticas clásicas pierden efectividad. Las metaheurísticas proporcionan un marco general para diseñar nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los procedimientos estadísticos

Existe una limitación de carácter teórica en lo que respecta al desempeño de las metaheurísticas:

> Según el teorema NFL (No Free Lunch), los métodos generales de búsqueda, entre los que se encuentran las metaheurísticas, se comportan exactamente igual cuando se promedian sobre todas las funciones objetivo posibles, de tal forma que si un algoritmo A es más eficiente que un algoritmo B en un conjunto de problemas, debe existir otro conjunto de problemas de igual tamaño, para los cuales el algoritmo B sea más eficiente que el A. Esta aseveración establece que, en promedio, ninguna metaheurística (algoritmos genéticos, búsqueda dispersa, búsqueda tabú, etc.) es mejor que la búsqueda completamente aleatoria (Cruz, 2013, p. 41).

Los enfoques metaheurísticos no ofrecen ningún tipo de prueba sobre su convergencia hacia el óptimo global, por lo tanto, no es posible tener certeza sobre si el problema converge a algún valor, ni la calidad de la solución obtenida.

A pesar de las limitaciones anteriores, el desempeño experimental de la mayoría de las metaheurísticas es notable, siendo una alternativa confiable a muchos problemas difíciles donde se requiere encontrar una solución de calidad en un tiempo computacionalmente razonable.

4.5 Metaheurísticas de relajación

Una cuestión relevante al abordar un problema real es la obtención de un modelo que permita emplear una técnica de resolución apropiada. Si con este modelo el problema resulta difícil de resolver se acude a modelos *modificados* en los que es más sencillo encontrar buenas soluciones o en los que los procedimientos son más eficientes. Una relajación de un problema es un modelo simplificado obtenido al excluir, atenuar o modificar restricciones (u objetivos) del problema real. En cualquier formulación siempre existe algún grado de simplificacíon, lo que puede afectar en

mayor o menor medida el ajuste a la realidad de los procedimientos de resolución y de las soluciones del problema propuestas. Los modelos que se ajustan mucho a la realidad suelen ser muy difíciles de resolver, y sus soluciones difíciles de implementar en condiciones reales, por lo que se acude a modelos relajados (Santana, et al., 2004)

Muchas heurísticas de relajación modifican elementos del problema para proponerlas como solución heurística del problema original. Las buenas relajaciones son las que simplifican el problema y hacen más eficientes los procedimientos de solución, pero cuya resolución proporciona muy buenas soluciones del problema original. Por ejemplo, para un problema de programación lineal entera, su relajación lineal consiste en ignorar la restricción de que las variables sean enteras. Se utiliza frecuentemente para aplicar procedimientos eficientes de programación lineal, como el método del Simplex, a dicha relajación y proponer una solución entera muy próxima a la solución del problema relajado.

Entre las metaheurísticas de relajación se encuentran los métodos de relajación lagrangiana o de restricciones subordinadas. Otras metaheurísticas de relajación alteran las restricciones o los objetivos del problema para usar su solución en la conducción de la búsqueda de la solución del problema original. Esta modificación puede estar encaminada a relajar las restricciones a las que debe estar sometida la solución, permitiendo que el recorrido bordee la región factible para acercarse al óptimo global incluso desde la región no factible. Otras estrategias modifican la función objetivo para obtener, de forma más rápida, valoraciones aproximadas (por exceso o por defecto) de la calidad de la solución que orientan la búsqueda, al menos en los estados iniciales. Es frecuente encontrar problemas en los que evaluar la función objetivo equivale a resolver otro problema de gran dificultad, realizar un proceso de simulación o realizar algún tipo de inversión o consumo de recursos. Para estos problemas es muy útil encontrar funciones sencillas de calcular

que den una idea aproximada de la calidad de las soluciones sin necesidad de una evaluación ajustada de la función objetivo.

4.6 Metaheurísticas constructivas

Santana et al (2004) define a las heurísticas constructivas como aquellas que aportan soluciones del problema por medio de un procedimiento que incorpora iterativamente elementos a una estructura, inicialmente vacía, que representa a la solución. Las metaheurísticas constructivas establecen estrategias para seleccionar las componentes con las que se construye una buena solución del problema. Entre las metaheurísticas originarias en este contexto se encuentra la popular estrategia voraz o greedy, que implica la elección que da mejores resultados inmediatos, sin tener en cuenta una perspectiva más amplia. Dentro de este tipo de metaheurísticas, destaca la aportación de la metaheurísticas GRASP que, en la primera de sus dos fases, incorpora a la estrategia greedy pasos aleatorios con criterios adaptativos para la selección de los elementos a incluir en la solución.

4.7 Algoritmos genéticos.

Los Algoritmos Genéticos (AG's) son métodos de optimización basados en una simulación parcial de los mecanismos de la evolución natural. Están basados en la teoría de la evolución que surge con las investigaciones de Charles Darwin (Darwin, 1859). Los algoritmos genéticos fueron creados en la década de los 60's por John Holland (Holland, 1975), como un modelo para el estudio del fenómeno de adaptación natural y para el desarrollo de mecanismos que permitieran incorporar este fenómeno a los sistemas de cómputo. Los algoritmos genéticos alcanzaron popularidad a raíz de la publicación del libro de Goldberg que sentaba las bases para su aplicación en problemas

prácticos. Los algoritmos evolutivos constituyen una parte importante de la Computación Evolutiva, un área de la Inteligencia Artificial en constante crecimiento. Actualmente son innumerables las aplicaciones exitosas en las más diversas áreas industriales, comerciales y de la ingeniería

Un algoritmo genético, desde el punto de vista de la optimización, es un método poblacional de búsqueda dirigida basada en probabilidad. Bajo una condición muy débil (que el algoritmo mantenga elitismo, es decir, guarde siempre al mejor elemento de la población sin hacerle ningún cambio) se puede demostrar que el algoritmo converge en probabilidad al óptimo (Melián, 2009). En otras palabras, al aumentar el número de iteraciones, la probabilidad de tener el óptimo en la población tiende a 1 (uno). Un algoritmo genético emula el comportamiento de una población de individuos que representan soluciones y que evoluciona en base a los principios de la evolución natural: reproducción mediante operadores genéticos y selección de los mejores individuos, correspondiendo éstos a las mejores soluciones del problema a optimizar.

Para aprovechar las ventajas del modelo del proceso evolutivo en la resolución de un problema de optimización, se deben establecer los siguientes procesos (Melián, 2009)

1. Una apropiada codificación de las posibles soluciones del problema representará a éstas de la misma forma que el cromosoma representa a los individuos de la especie. Dada esta unívoca relación, se usarán indistintamente los términos solución, codificación, cromosoma o individuo.

2. La adaptabilidad de cada solución será una medida del comportamiento de ésta en el problema particular considerado. Normalmente, es el valor objetivo de la solución. Así, una solución está más adecuada a un problema cuanto mejor sea su valor objetivo.

3. Se definen operadores genéticos que, al actuar sobre una o varias soluciones, suministren una o más soluciones al alterar genéticamente los cromosomas. Juegan el papel del cruce y la mutación en el proceso evolutivo natural.

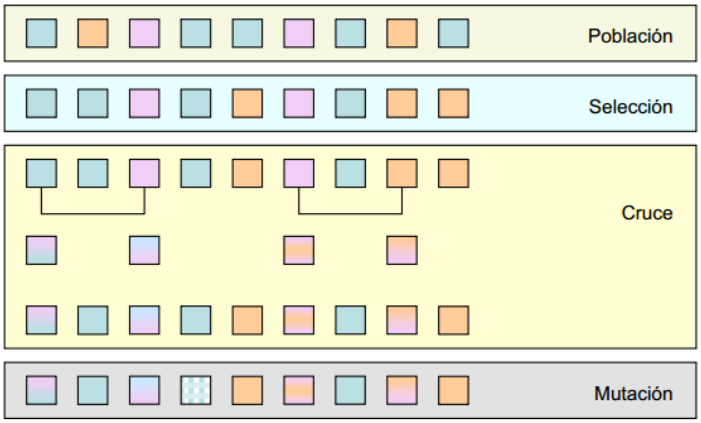

Figura 2 Esquema de funcionamiento de un algoritmo genético
Nota: Adaptado de Melían, B, (2009, 4 de agosto). Algoritmos Genéticos: una visión práctica. Didáctica de las matemáticas, (71), p. 29-47.

Las acciones básicas del algoritmo genético:

1. Generar una población inicial de soluciones.

2. Seleccionar, de la población actual, las soluciones mejor adaptadas.

3. Cruzar algunas soluciones para obtener su descendencia.

4. Mutar algunas soluciones para obtener las soluciones mutadas.

5. Elegir las soluciones que sobreviven y formarán la nueva generación.

6. Si no se alcanza el criterio de parada volver al paso 2. Al finalizar los pasos anteriores, la mejor solución de la población es la que se propone como solución del problema.

4.8 Metaheurísticas de búsqueda

Las metaheurísticas de búsqueda constituyen quizás el grupo más importante; se caracterizan por establecer estrategias para recorrer el espacio de soluciones del problema transformando de forma iterativa soluciones de partida. Las búsquedas evolutivas se distinguen de éstas, en que es un conjunto de soluciones, generalmente llamado población de búsqueda, el que evoluciona sobre el espacio de búsqueda

En sus inicios, la concepción de heurística era la de alguna regla inteligente para mejorar la solución de un problema que se aplicaba iterativamente mientras fuera posible obtener nuevas mejoras. El enfoque se conoció como búsquedas monótonas (descendentes o ascendentes), algoritmos escaladores (hill-climbing) o búsquedas locales.

4.9 Búsqueda local

El término búsqueda local, obedece a que la mejora se obtiene a partir del análisis de soluciones similares a la que realiza la búsqueda; denominadas soluciones vecinas. Las búsquedas locales operan a partir del estudio de soluciones del vecindario o entorno de la solución que realiza el recorrido. Las metaheurísticas de búsqueda local son las estrategias o pautas generales para diseñar métodos de búsqueda. Un ejemplo es la estrategia voraz o greedy, la cual establece como pauta, una vez son consideradas cuáles son las soluciones que intervienen en el análisis local, elegir iterativamente la mejor de tales soluciones mientras exista alguna mejora posible.

La principal desventaja que presenta la búsqueda local es el riesgo de "*quedar atrapada*" en un óptimo local, una solución que no puede ser mejorada por un análisis local. Por ello, el propósito

fundamental de las primeras metaheurísticas era extender una búsqueda local para continuarla más allá de los óptimos locales, denominándose *búsqueda Global*.

Las metaheurísticas de búsqueda global incorporan procedimientos básicos con el objeto de evitar "*quedar atrapada*" en óptimos locales de baja calidad. Un enfoque alternativo ha consistido en iniciar la búsqueda desde otra solución de arranque, modificar la estructura de entornos que se está aplicando y permitir movimientos o transformaciones de la solución de búsqueda que no sean de mejora (Santana, et al., 2004). Los anteriores procedimientos dan lugar respectivamente a las metaheurísticas de arranque múltiple, las metaheurísticas de entorno variable y las metaheurísticas de búsqueda no monótona.

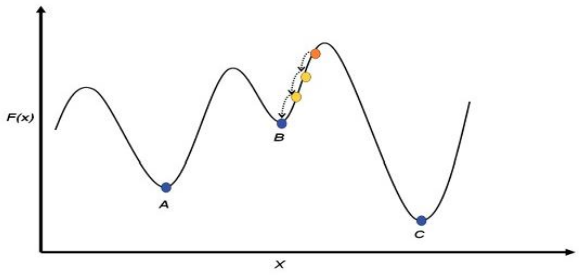

Figura 3 Transición desde una solución inicial a un óptimo local.

Las metaheurísticas de arranque múltiple establecen pautas para reiniciar de forma inteligente las búsquedas descendentes. Las metaheurísticas de entorno variable modifican de forma sistemática el tipo de movimiento con el objeto de evitar que la búsqueda "*quede atrapada"* por una estructura de entornos rígida. Las búsquedas que también aplican movimientos de no mejora durante el recorrido de búsqueda se denominan búsquedas *no monótonas*.

4.10 Búsqueda Tabú

Las metaheurísticas para búsquedas no monótonas controlan los posibles movimientos de empeoramiento de la solución mediante criterios de aceptación estocásticos o utilizando la memoria del proceso de búsqueda. Las metaheurísticas de búsqueda estocástica establecen criterios para regular la probabilidad de aceptar transformaciones que no mejoren la solución.

El Recocido Simulado es el exponente más importante de este tipo de metaheurísticas donde la probabilidad de aceptación es una función exponencial del empeoramiento producido. Las metaheurísticas de búsqueda con memoria utilizan información sobre el recorrido realizado para evitar que la búsqueda se concentre en una misma zona del espacio solución. Esencialmente se trata de la Búsqueda Tabú cuya propuesta original prohíbe temporalmente soluciones muy parecidas a las últimas soluciones del recorrido.

4.11 Entornos de búsqueda

Los procedimientos de búsqueda por entornos recorren el espacio de soluciones U mediante un conjunto de transformaciones o movimientos. Las soluciones que se obtienen a partir de otra mediante uno de los movimientos posibles se denominan *vecinas* de ésta y constituyen su entorno. El conjunto de movimientos posibles da lugar a una relación de vecindad y una estructura de entornos en el espacio de soluciones cuya elección es un aspecto significativo en el éxito de los procesos de búsqueda. Además de una implementación y evaluación eficiente de los movimientos, las propiedades de la estructura de entorno resultante intervienen en esta elección. El esquema general de un procedimiento de búsqueda por entornos consiste en generar una solución inicial y, hasta que se cumpla el criterio de parada, seleccionar iterativamente un movimiento para modificar la solución. Las soluciones son evaluadas mientras se recorren y se propone la mejor solución del problema encontrada.

El entorno de una solución está constituido por las soluciones a las que se puede acceder desde ella por uno de los movimientos posibles. Formalmente, una estructura de entornos sobre un espacio o universo de búsqueda U es una función $E: U \rightarrow 2^U$ que asocia a cada solución $x \in U$ un entorno $E(x) \subseteq U$ de soluciones vecinas a x. Gran cantidad de métodos heurísticos propuestos en la literatura pertenece a la clase de procedimientos de búsqueda por entornos.

4.12 Estructura de entornos

La elección de la estructura de entornos es fundamental en el éxito de los procesos de búsqueda ya que determina la calidad del conjunto de movimientos aplicados. Aparte de la factibilidad y el grado de mejora de los movimientos aplicados es importante la versatilidad de los mismos. Los movimientos combinados aparecen al ejecutar sucesivamente varios movimientos sobre una solución. Una adecuada combinación de movimientos enriquece los entornos, con lo que se pueden realizar pasos más amplios en el acercamiento al óptimo, pero se corre el riesgo de perjudicar la eficiencia del algoritmo al tener que contemplar un número mayor de movimientos posibles en el proceso de selección.

Una característica adicional e importante de los movimientos es la factibilidad de las soluciones aportadas. Los movimientos factibles son aquellos que siempre proporcionan una solución factible. Esto puede estar ligado o no al hecho de que se aplique sólo a soluciones factibles. En muchos casos, aplicar movimientos más simples, pero no necesariamente factibles, y descartar las soluciones producidas que no sean factibles, es menos eficiente que adaptar el diseño de los movimientos para que sean factibles, sobre todo cuando dicha comprobación es costosa o cuando la probabilidad de que resulte factible es baja. Formalmente, los procedimientos que sólo consideran movimientos factibles están asociados al concepto, algo más restrictivo, de estructura

de entornos como una función $E: U \rightarrow 2^U$ que asocia a cada solución factible $x \in S$ un entorno $E(x) \subset S$ de soluciones factibles vecinas a x. Gran cantidad de métodos heurísticos propuestos en la literatura pertenece a la clase de procedimientos de búsqueda por entornos.

El libro de Talbi (2009) subraya la importancia en la elección de la estructura de entornos en el éxito de los procesos de búsqueda porque determina la calidad del conjunto de movimientos aplicados. Aparte de la factibilidad y el grado de mejora de los movimientos aplicados es importante la versatilidad de los mismos. Los movimientos combinados aparecen al ejecutar sucesivamente varios movimientos sobre una solución. Una adecuada combinación de movimientos enriquece los entornos, con lo que se pueden realizar pasos más amplios en el acercamiento al óptimo, pero se corre el riesgo de perjudicar la eficiencia del algoritmo al tener que contemplar un número mayor de movimientos posibles en el proceso de selección.

5 El problema original del taller JSP

El primer modelo matemático que se propuso para abordar la optimización de este tipo de sistemas productivos fuel el JSP (*job shop problem*). El problema considera un conjunto de m máquinas $M = \{M_1, M_2, \dots, M_m\}$ y n trabajos $J = \{J_1, J_2, \dots, J_n\}$ Todo trabajo $j_i \in J$, se compone de una secuencia *ordenada* de m operaciones, $O_{i,1}$, $O_{i,2}$, … , $O_{i,k}$, … , $O_{i,m}$, necesarias para completarlo. La operación *k-ésima* del trabajo i se designa como $O_{i,k}$. Cada operación $O_{i,k}$ tiene asignada biunívocamente una máquina $M_{i,k}$ del conjunto $M = \{M_1, M_2, \dots, M_m\}$ en la cual se ejecutará. Su tiempo de procesamiento se denota con: $p_{i,k}$.

En su forma más general, el problema del *JSP* está sujeto a un conjunto de restricciones que condiciona el procesamiento de las operaciones. El libro de *Simon French, Sequencing and*

Scheduling. An introduction to the mathematics of the Job Shop, presenta las restricciones del *JSP* clásico (French, 1982).

- Cada trabajo es una entidad. A pesar que cada trabajo está compuesto de *m* operaciones distintas, no existen dos operaciones del mismo trabajo que puedan ser procesadas simultáneamente. Quedan excluidas ciertas situaciones reales en las cuales ciertas piezas del producto son fabricadas simultáneamente antes de ser ensamblado como producto terminado.

- Cada trabajo tiene *m* operaciones distintas, una en cada máquina. No se permite la posibilidad que un trabajo pueda requerir ser procesado dos veces en una misma máquina. Tampoco podrá omitir una o más máquinas.

- Una vez una operación ha iniciado su procesamiento, hasta que ésa operación no sea concluida, no podrá otra operación iniciar en la misma máquina.

- No se permite la cancelación de un trabajo. Todo trabajo debe ser procesado hasta su finalización.

- Los tiempos de procesamiento son independientes de la programación. En particular se están asumiendo un par de situaciones. En primer lugar, que el tiempo de ajuste para preparar una máquina es independiente del trabajo a ser procesado. En segundo lugar, los tiempos para desplazar los trabajos entre máquinas son despreciables.

- El inventario de producto en proceso es permitido. Esta no es una suposición trivial. En algunos problemas, el procesamiento de trabajos debe ser continuo conforme va siendo transformado en cada una de las operaciones que lo compone.

- De cada tipo de máquina sólo existe una. No se permite que haya un conjunto de máquinas disponibles para ejecutar una operación en particular. Esta condición elimina la posibilidad de duplicar un recurso con el objeto de eliminar un cuello de botella.

- Las máquinas podrán encontrarse en inactividad entre la finalización de una operación y el comienzo de otra

- Ninguna máquina puede procesar más de una operación al mismo tiempo.
- Las máquinas nunca se descomponen y se encontrarán siempre disponibles durante el tiempo que tome el procesamiento de todos los trabajos.
- Los parámetros que definen al problema se conocen completamente. El número de máquinas, los trabajos y los tiempos de procesamiento de las operaciones son conocidos y tampoco son susceptibles a ser modificados.

En algunos casos especiales, los investigadores muestran interés en examinar el problema, relajando una o varias de estas restricciones. En general, hacerlo conlleva un aumento notable en el nivel de dificultad para tratarlo. En lo referente al *FJSP*, salvo dos importantes excepciones, que serán presentadas a continuación en su formulación matemática básica, sus restricciones son iguales.

5.1 Definición matemática del *FJSP*

El problema del *Flexible Job Shop (FJSP)* se define en términos matemáticos muy similares al *JSP*. Toma el mismo conjunto de *m* máquinas, $M = \{M_1, M_2, \dots, M_m\}$ y *n* trabajos $J = \{J_1, J_2, \dots, J_n\}$. En el *JSP todos* los trabajos se componen de *m* operaciones; por el contrario, en el *FJSP*, cada trabajo está compuesto por una secuencia *ordenada* de $\boldsymbol{n_i}$ operaciones: $O_{i,1}, O_{i,2}, \dots, O_{i,k}, \dots, O_{i,n_i}$. El número de operaciones que requiere cada trabajo del conjunto J, puede diferir. Esta es la primera diferencia con respecto al *JSP*.

La segunda diferencia reside en la capacidad del modelo de permitir seleccionar entre un conjunto $A_{i,k}$ de operaciones , la máquina que procesará la operación k-ésima, $O_{i,k}$. El tiempo de procesamiento por parte de una máquina $M_j \in A_{i,k}$, para la operación $O_{i,k}$, se indica como: $p_{i,k,j}$.

Cuando el problema exhibe flexibilidad total (*T-FJSP*), en caso que todas las operaciones son procesadas por todas las máquinas, $A_{i,k} = M$, Por el contrario, cuando al menos una operación $O_{i,k}$ no puede ser procesada en todas las máquinas, es decir $A_{i,k} \subset M$, la flexibilidad del problema es parcial (*P-FJSP*).Esta flexibilidad le permite al FJSP aproximarse mejor a las condiciones de operación que se encuentran frecuentemente en ambientes reales.

6 Diseño del algoritmo híbrido para el problema del *FJSP*

El algoritmo propuesto en el artículo: *A hybrid genetic and variable neighoborhood descent algorithm for flexible job scheduling problems*, cuyos autores: *Gao Jie, Sun Linyan, Gen Mitsuo*, presenta una metodología híbrida de algoritmo genético equipado con una componente de búsqueda local para resolver el problema del *FJSP* (Gao, Sun, & Gen, 2008).

El diseño del algoritmo posee tres criterios de minimización: el makespan, la carga de trabajo por máquina y la carga de trabajo total. Su estructura genética emplea un par-cromosoma para almacenar el contenido genético de cada solución particular; el primer cromosoma del par codifica la secuencia de operaciones, el segundo, la asignación de máquinas a operaciones.

Los operadores genéticos de cruce y mutación, aprovechan la estructura de par-cromosoma y contribuyen a mejorar la calidad de las soluciones, garantizando un grado de diversidad necesario que evite que la búsqueda converja prematuramente a un óptimo local.

La población inicial es generada aleatoriamente. Subsecuentemente, en cada generación, los operadores de cruzamiento y mutación forman individuos adicionales que enriquecen la diversidad

genética de la población. Previamente a la selección de los individuos más aptos, el makespan de cada individuo es mejorado por un proceso de búsqueda local. La etapa de selección imita al fenómeno evolutivo de la *selección natural*, su propósito es determinar cuáles serán los individuos que pasarán sus genes a la siguiente generación. El algoritmo itera un número predeterminado de generaciones. Al concluir la última iteración, se escoge el mejor makespan de la población. Se espera que solución escogida se aproxime al óptimo global.

Con el objetivo de robustecer la explotación del espacio solución, el algoritmo incorpora un mecanismo de búsqueda local. En cada generación, todos los individuos de la población son sometidos uno por uno a una transformación que resulta en un individuo óptimo local que reemplaza al individuo original. El tipo anterior de búsqueda local se conoce como: *vecindad variable de descenso*.

La metodología anterior planteada en el artículo, consiste en mover de posición sucesivamente cada una de las operaciones que componen la ***ruta crítica*** de un individuo, es decir, eliminar la operación de su actual posición y buscar su reasignación en otra máquina, si es posible. El proceso inicia con la primera operación de su ruta crítica (Gao, et al., 2008). En caso de no llegar a ser posible mover alguna de sus operaciones, se selecciona la siguiente operación. Al mover una operación de la ruta crítica, se genera una programación de operaciones nueva (nuevo individuo) con su correspondiente nueva ruta crítica. El proceso anterior se repite hasta llegar a una programación en la cual no sea posible *mover* ninguna operación de su ruta crítica. Cuando la condición anterior se cumple, el individuo es óptimo local y garantiza que el makespan de su nueva programación sea igual o menor al makespan del individuo original.

La rutina para mejorar cada individuo a un óptimo local se le denomina mover una operación. El mecanismo de mover una operación, es reforzado mediante el movimiento análogo de mover dos operaciones simultáneamente.

6.1 Representación de una solución particular

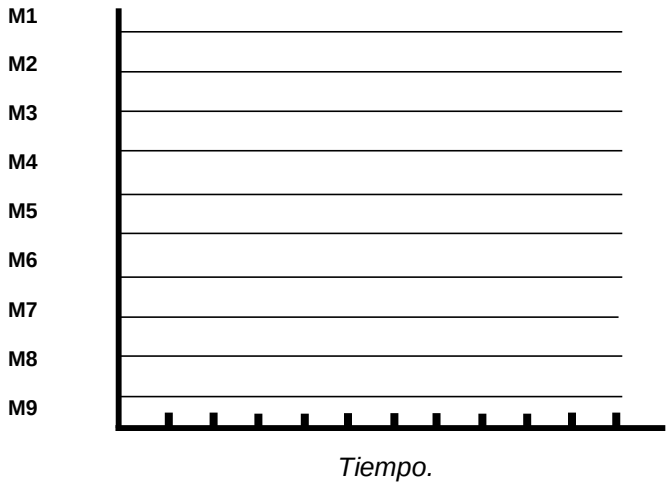

Figura 4. Diagrama de Gantt de problema de 15x10, aún sin operaciones asignadas a las máquinas

En el diagrama de la figura 4 no existen operaciones asignadas en ninguna de las diez máquinas; es una manera visual e intuitiva para comprender el proceso de *"crear"* un individuo o solución *particular* al problema del *FJSP*.

Al iniciar el proceso de creación de una solución particular, su diagrama de Gantt se encuentra vacío. Cada operación $O_{i,k}$, debe ser asignada a una máquina que se encuentre en el conjunto $A_{i,k}$. Sin embargo el intervalo de tiempo que debe ocupar en la máquina asignada, se encuentra condicionada a la siguiente restricción

La restricción obedece a la existencia de su operación *predecesora de trabajo*. Un trabajo está compuesto por una secuencia *ordenada* de operaciones, un orden que no es posible modificar. El concepto es sencillo si se considera los pasos necesarios para manufacturar cualquier producto. En la fabricación de un carro, por ejemplo, se requiere una secuencia de pasos ordenados. Simplemente no es posible llevar a cabo la operación de pintura antes de completar la soldadura de la carrocería. Al anterior tipo de limitación se le denomina *restricción tecnológica*.

El inicio del procesamiento de la operación $O_{i,k}$ en la máquina $M(O_{i,k}) \in A_{i,k}$ está condicionado por la finalización de su operación predecesora de trabajo , $O_{i,k-1}$, en la máquina $M(O_{i,k-1}) \in A_{i,k-1}$. En la figura 5 por ejemplo, se observa que la operación $O_{2,2}$ inicia su procesamiento en la máquina M_3, sólo después que su operación predecesora de trabajo , $O_{2,1}$, ha concluido en la máquina M_4.

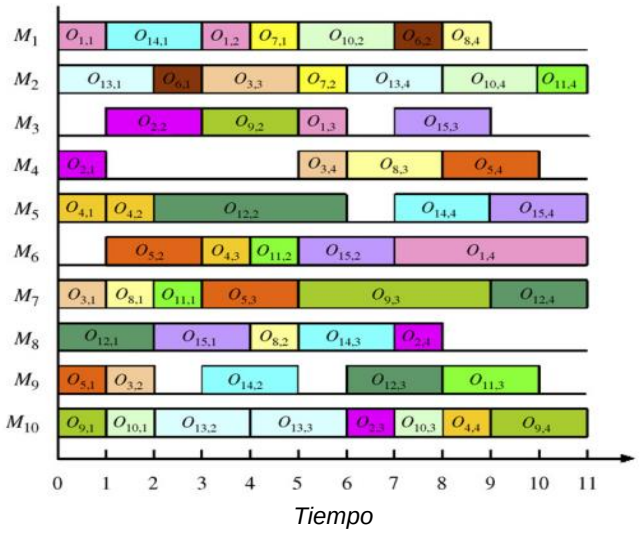

Figura 5. Diagrama de Gantt obtenido de una solución para un problema FJSP de 15x10. Esta representación es equivalente a la programación de operaciones o solución particular del problema.

El ejemplo anterior, ilustra que el inicio de toda operación $O_{i,k}$ se encuentra condicionado por el tiempo de finalización de su operación predecesora: de trabajo.

En la figura 5, al finalizar la asignación de todas las operaciones, cada máquina $M_j \in M$, tiene asociada una secuencia de operaciones que indica el orden en que son procesadas las operaciones. La secuencia de operaciones por ejemplo en la máquina M_8 es: $O_{12,1}$, $O_{15,1}$, $O_{8,2}$, $O_{14,3}$ y $O_{2,4}$. Una ***solución particular*** es precisamente el conjunto de secuencias de operaciones que fueron programadas en todas las máquinas. El diagrama de Gantt de la figura 5, representa gráficamente una solución particular *o* programación de operaciones (***schedule***).

Si todas las operaciones de un problema han sido asignadas en sus respectivas máquinas, respetando la restricción de su operación predecesora de trabajo, la programación de operaciones es una ***solución factible***. *La figura 5* representa gráficamente una ***solución factible*** a un problema *FJSP 15x10*. Una vez las operaciones han sido asignadas, al período de tiempo que ha tomado concluirlas se le denomina ***makespan***.

6.2 Representación de una solución particular por un par de cromosomas

En los organismos biológicos, la estructura genética básica de todo ser vivo es el cromosoma. Su función consiste en ***codificar*** y ***transmitir*** la información de progenitores a descendientes. De manera análoga, los algoritmos genéticos desarrollan mecanismos para que una *solución particular* o *programación*, pueda ser *codificada* en una configuración de cromosoma (Gao, et al., 2008).

En el caso particular del algoritmo híbrido estudiado en el presente proyecto de grado, la estructura tradicional de un solo cromosoma por individuo, ha sido extendida a una que emplea un par de cromosomas: el par Cr_1 y Cr_2. (Gao, et al., 2008).

Cr_1 codifica la secuencia de procesamiento de las operaciones en las diferentes máquinas. Su codificación, emplea una convención ingeniosa para identificar las distintas operaciones de un

mismo trabajo. En la siguiente secuencia de operaciones para un problema 4×4, se presenta el orden de asignación de todas las operaciones a las distintas máquinas: $O_{2,1} \rightarrow O_{4,1} \rightarrow O_{3,1} \rightarrow O_{1,1} \rightarrow O_{4,2} \rightarrow O_{1,2} \rightarrow O_{4,3} \rightarrow O_{3,2} \rightarrow O_{2,2} \rightarrow O_{1,3} \rightarrow O_{3,3} \rightarrow O_{1,4} \rightarrow O_{2,3} \rightarrow O_{4,4} \rightarrow O_{3,4} \rightarrow O_{2,4}$.

La anterior secuencia es codificada en Cr_1 de acuerdo a la figura 6:

2	4	3	1	4	1	4	3	2	1	3	1	2	4	3	2
$O_{2,1}$	$O_{4,1}$	$O_{3,1}$	$O_{1,1}$	$O_{4,2}$	$O_{1,2}$	$O_{4,3}$	$O_{3,2}$	$O_{2,2}$	$O_{1,3}$	$O_{3,3}$	$O_{1,4}$	$O_{2,3}$	$O_{4,4}$	$O_{3,4}$	$O_{2,4}$

Figura 6 Cromosoma tipo 1.

El contenido de Cr_1 no hace referencia *explícita* a las operaciones. No obstante, es sencillo observar, que las posiciones que contienen el número dos, hacen referencia *implícita* a las *operaciones del segundo trabajo*. Por ejemplo, en la primera posición de Cr_1, su valor de dos corresponde a la primera operación del segundo trabajo: $O_{2,1}$. En la novena posición, corresponde a la segunda operación del segundo trabajo: $O_{2,2}$. Cada posición de Cr_1 toma algún valor de i, tal que, $1 \leq i \leq n$; el cual representa alguno de los n trabajos a realizar. El número de veces que se repetirá un determinado número i se encuentra predeterminado por n_i, término que hace referencia al número de operaciones es que se descompone un trabajo i.

La existencia de Cr_2 obedece al elemento de *flexibilidad* que incorpora el *FJSP* frente al *JSP* original. En un problema clásico *JSP* $n \times m$, a cada $O_{i,k}$.le corresponde una y sólo una máquina $M_j \in M$ y las operaciones del mismo trabajo se encuentran en biyección con el conjunto M de máquinas. El *FJSP* por el contrario, permite establecer para cada operación $O_{i,k}$, un conjunto de

máquinas $A_{i,k}$, que se encuentran en capacidad de procesar la operación y los distintos trabajos i se pueden descomponer en un número arbitrario, $n_{i,}$ de operaciones. Cr_2 codifica precisamente la asignación de cada operación O_{ik}.a una máquina $M_j \in A_{ik}$.

Cada posición de Cr_2 indica el número de máquina asignada a cada $O_{i,k}$. La asignación sigue la numeración ascendente de los distintos trabajos, como se muestra en la siguiente ilustración. Por ejemplo, la primera posición indica que la operación $O_{1,1}$ será procesada en la máquina número cuatro de acuerdo a la figura 7.

4	3	3	1	2	4	1	4	3	1	2	1	2	4	4	3
$O_{1,1}$	$O_{1,2}$	$O_{1,3}$	$O_{1,4}$	$O_{2,1}$	$O_{2,2}$	$O_{2,3}$	$O_{2,4}$	$O_{3,1}$	$O_{3,2}$	$O_{3,3}$	$O_{3,4}$	$O_{4,1}$	$O_{4,2}$	$O_{4,3}$	$O_{4,4}$

Figura 7 Cromosoma tipo 2

La ventaja más notable del mecanismo de codificación del par-cromosoma, consiste en que siempre produce secuencias de operaciones ***factibles*** y su implementación representa un bajo costo desde el punto de vista computacional.

6.3 Tipos de programación de operaciones (*Schedules*)

En el subcapítulo 6.1, se establecieron las condiciones para programar cualquier operación $O_{i,k}$, .en su máquina respectiva. No obstante, entre el numeroso conjunto de programaciones factibles para un problema determinado, existen unos subconjuntos de programaciones (*schedules*) que facilitan enormemente la búsqueda del óptimo global. La figura 8 presenta una programación factible para un problema 2x2. En ella se pueden observar los tiempos de inicio de las cuatro operaciones.

Las operaciones $O_{2,1}$ y $O_{1,1}$ pueden iniciar su procesamiento en $t = 0$ dado que ninguna de ellas cuenta con predecesor de trabajo y son las primeras operaciones en ser programadas en M_1 y M_2 de acuerdo al cromosoma Cr_asig_maq. Si bien el predecesor de máquina de la operación $O_{2,2}$ finaliza en $t = 3$, su predecesor de trabajo lo hace en $t = 4$; por ello $O_{2,2}$ no puede iniciar antes de $t = 4$. La operación $O_{1,2}$ por el contrario, podría hacerlo antes de $t = 5$ sin violar ninguna restricción, mejorando en una unidad de tiempo el makespan original. Por tanto, para disminuir el makespan de siete unidades de tiempo a seis, la operación $O_{1,2}$ debe iniciar en $t = 4$.

Si en una programación *es posible* iniciar más temprano una operación $O_{i,k}$, sin alterar la secuencia de operaciones de la máquina $M(O_{i,k})$ donde se procesa, la programación presenta *tiempo ocioso* entre sus operaciones (Gao, et al., 2008). Ajustar el tiempo de inicio en cualquier operación $O_{i,k}$ donde haya tiempo ocioso, equivale a desplazar lo más posible hacia la izquierda de la máquina la operación $O_{i,k}$. A este tipo de ajuste se le conoce como: *cambio local a izquierda* (*local left-shift*). En la figura 8, la operación $O_{1,2}$ es susceptible de un cambio local a izquierda.

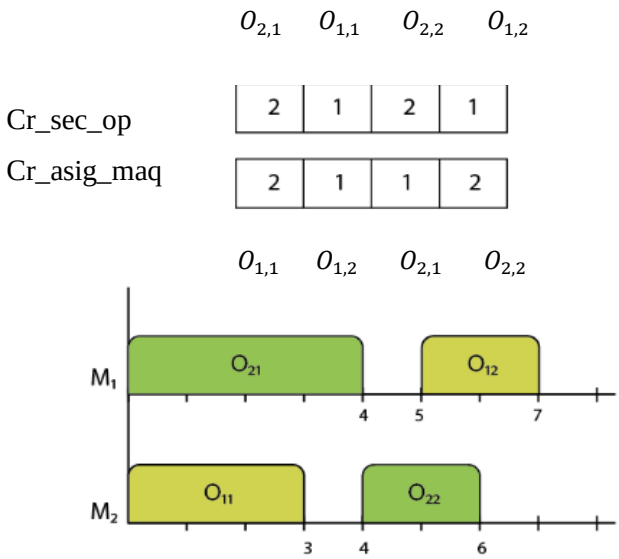

Figura 8 Ejemplo de programación de un problema 2x2, sin cambio local a izquierda en O12.

6.4 Programaciones semiactivas y activas

Al conjunto de todas las soluciones factibles se le suele llamar *espacio solución*. La búsqueda de la solución óptima se limita al anterior conjunto. No obstante, debido a la naturaleza combinatoria del problema, resulta impráctico desde el punto de vista computacional, examinar individualmente todas las soluciones del espacio solución.

Debido a lo anterior, se han propuesto métodos algorítmicos que permiten una exploración sistemática del *espacio solución*, facilitando la tarea de contar con soluciones que se aproximen lo más posible, a la *solución óptima*.

Todas las programaciones en las cuales no es posible un *cambio local a izquierda*, conforman el conjunto de programaciones semiactivas (*semiactive schedules*) El tamaño del conjunto anterior, acota el número de programaciones posibles que se deben examinar en la búsqueda de la solución óptima. A pesar que su número pueda llegar a ser elevado, es al menos finito. Un buen estimativo de su número es calcular el número de programaciones brutas.

Para un problema *FJSP* $n \times m$, resulta conveniente considerar el problema desde la perspectiva del *JSP clásico*. En cada máquina $M_j \in M$, es posible realizar un número de $n!$ permutaciones. En principio existe un número de $n!$ secuencias posibles por máquina. Si se supone que todas las secuencias generan soluciones factibles y existen m máquinas, el número total de permutaciones posibles para programar todas las operaciones es: $(n!)^m$. En un problema 10x10, el número de programaciones *brutas* asciende a: $(10!)^{10} = 3{,}95\text{x}10^{65}$.

Sin embargo, el orden de precedencia de las operaciones del mismo trabajo, reduce considerablemente el número de programaciones que son *factibles.* Aunque el valor exacto puede

resultar técnicamente muy difícil de calcular, el nuevo valor es estrictamente menor al valor original de programaciones brutas.

El enfoque para reducir sistemáticamente el número de posibles soluciones, consiste en definir subconjuntos con características que permiten encontrar soluciones de calidad, a pesar que no exista certeza de encontrar la solución óptima. El subconjunto debe ser construido de manera inmediata y su tamaño lo más pequeño posible.

En una programación *factible*, el tiempo de inicio de una operación $O_{i,k}$ particular se encuentra restringido por el tiempo de inicio de su operación predecesora de *trabajo*. Adicionalmente, en una programación semiactiva no existe tiempo ocioso entre operaciones adyacentes, esto es, no es posible realizar *cambios locales a izquierda* sobre alguna operación $O_{i,k}$. Resulta notable que aún sea posible mejorar el *makespan* sobre un subconjunto de las programaciones semiactivas.

En la figura 9(a), se observa una programación semiactiva en un problema 4×3. A pesar de no ser posible llevar a cabo *cambios locales a izquierda*, es posible, por ejemplo, iniciar la operación $O_{1,1}$ en la máquina M1, más temprano sin *retrasar* el inicio de alguna operación. El inicio más temprano para la operación $O_{1,1}$ en M_1 sería en el tiempo $t = 0$; las demás operaciones del primer trabajo, también podrían iniciar más temprano sin causar retraso en las demás operaciones. La modificación de la programación de operaciones en la figura 9(b), equivale a iniciar la operación $O_{1,1}$ lo *más temprano* posible en la máquina M_1. El tipo de ajuste anterior, sin causar retardo en las demás operaciones de la programación, se llama *cambio global a izquierda* (global left shift). El conjunto de las programaciones en las cuales no es posible un *cambio global a la izquierda* se denomina *programaciones activas* y constituye un subconjunto de las programaciones *semiactivas* (Gao, et al., 2008).

Así como las programaciones *semiactivas* dominan sobre el conjunto de todas las programaciones, el conjunto de las programaciones *activas* domina sobre el conjunto de las programaciones *semiactivas*. La importancia de las *programaciones activas* reside en que la solución *óptima* se encuentra precisamente en él.

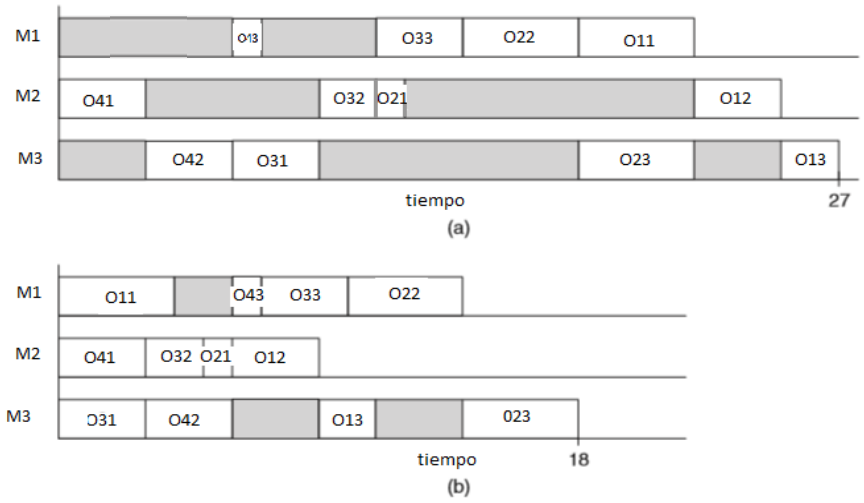

Figura 9 A partir de una programación semiactiva (a) se obtiene una programación activa (b)

6.5 Concepto de población en el algoritmo genético

La metodología del algoritmo propuesto sigue la estructura de las metaheurísticas evolutivas-genéticas. En ellas se busca imitar el mecanismo biológico por medio del cual las especies vivas evolucionan.

A través de múltiples generaciones y por medio del proceso reproductivo, los organismos vivos tienen la capacidad de heredar los rasgos que más favorecen su adaptación al medio ambiente. En biología a este principio se le conoce como: *selección natural.* Cada *solución particular* en el

algoritmo representa un organismo vivo (*individuo*) cuyo material genético se encuentra codificado por medio de la estructura del par de cromosomas Cr_1 y Cr_2.

La diversidad es otro aspecto relevante en la concepción de los algoritmos genéticos. La diversidad hace referencia a la existencia, en una misma población, de individuos que posean factores de herencia altamente variables entre sí. La coexistencia de los individuos considerados "buenos" con aquellos considerados "malos", es una condición necesaria para el proceso evolutivo de la especie. En principio una población diversa favorece el surgimiento de individuos muy bien adaptados a las nuevas condiciones del entorno.

El tamaño sugerido de la población inicial por el artículo, establece un intervalo entre 300 y 3000 individuos. El algoritmo genera la población inicial creando un número igual de par-cromosomas Cr_1 y Cr_2 aleatoriamente.

Determinar que tan bien "*adaptado*" se encuentra un *individuo* (*solución particular*) requiere una evaluación del valor de su función objetivo. En los procesos de optimización matemática, la eliminación completa de los individuos considerados "malos" en cada generación, conlleva a la homogenización de la población y a una convergencia prematura; la cual generalmente conduce al algoritmo a una solución óptima local. Después de su homogenización, es difícil que el método siga evolucionando. Para superar esta dificultad en los algoritmos genéticos, el proceso de selección debe ser un mecanismo que permita a los individuos "buenos" tener una alta probabilidad de supervivencia y a los considerados "malos" una baja probabilidad pero sin permitir su eliminación.

El algoritmo híbrido a implementar, posee tres criterios a minimizar:

- Minimizar el makespan (CM).

- Minimizar la carga de trabajo, esto es, el tiempo máximo de trabajo en cualquier máquina. Se busca prevenir que una solución asigne demasiado trabajo a una máquina particular y asegurar una distribución equilibrada entre todas las máquinas.

- Minimizar la carga total de trabajo, representada en el tiempo total asignado sobre todas las máquinas. Este objetivo es de interés, si existe una diferencia de eficiencia entre la maquinaria.

6.6 Decodificación con base en prioridad y reordenamiento.

En el subcapítulo 6.4 se presentó el mecanismo que *codifica* una solución particular. No obstante, tan útil como resulta la tarea de codificar, es su tarea inversa: *decodificar* el par-cromosoma a una *solución particular* o *programación*.

En principio, decodificar un par-cromosoma Cr_1 y Cr_2, podría generar un número infinito de programaciones. Lo anterior obedece a la manera en que se asignan las operaciones a sus respectivas máquinas. El inicio de cualquier operación $O_{i,k}$ debe llevarse a cabo posterior a la finalización de su operación predecesora de trabajo $O_{i,k-1}$.

Existe una infinita cantidad de tiempos que cumplen la anterior restricción. Una manera equivalente de expresarlo es que el tiempo entre el fin de $O_{i,k-1}$ y el inicio de $O_{i,k}$, es arbitrario (Gao, et al., 2008). Evidentemente retrasar *deliberadamente* el inicio de las operaciones, no contribuye a minimizar el *makespan* de la programación. Aunque es preciso que el proceso de asignación no permita la introducción de tiempo ocioso entre operaciones de un mismo trabajo, aún sin tiempo ocioso entre operaciones, es posible aún mejorar las programaciones de operaciones

Asignar una operación $O_{i,k}$, consiste en mover lo *más posible* a la *izquierda* la operación en su máquina asignada, $M(O_{i,k})$, sin retrasar alguna operación en la máquina. Lo anterior equivale a un *cambio global a izquierda* (global left shift) y garantiza la existencia sólo de *programaciones activas* en la población.

La restricción de precedencia entre operaciones de un mismo trabajo, causa la existencia de tiempo libre entre las operaciones que se programan en una máquina. Se debe precisar que el concepto de tiempo libre no es equivalente al de tiempo ocioso. La existencia de intervalos de tiempo libre entre operaciones es el fundamento del mecanismo que decodifica el par-cromosoma Cr_1 y Cr_2.

Se definen los siguientes términos: $s_{i,k}$ y $c_{i,k}$ el tiempo de inicio y finalización de la operación $O_{i,k}$, una vez ha sido programada. Un intervalo de tiempo libre se denota como: $\left[t^S_{M(O_{i,k})}\ ,\ t^E_{M(O_{i,k})}\right]$, con un inicio y final en los instantes t^S y t^E, en la máquina $M(O_{i,k})$ asignada.

La elección del intervalo de tiempo $\left[t^S_{M(O_{i,k})}\ ,\ t^E_{M(O_{i,k})}\right]$, requiere que se vayan examinando todos los intervalos libres de izquierda a derecha que existan en $M(O_{i,k})$ hasta que sea posible encontrar uno en que $O_{i,k}$ pueda ser asignada.

El inicio de $O_{i,k}$, en el intervalo de tiempo libre se define como: $s_{i,k}$

$$s_{i,k} = \begin{cases} \max\{t_j^S, c_{i,k-1}\} & si\ k \geq 2 \\ t_j^S & si\ k = 1 \end{cases}$$

La operación $O_{i,k}$ se programa en el intervalo $\left[t^S_{M(O_{i,k})}\ ,\ t^E_{M(O_{i,k})}\right]$, si puede contener la operación desde su inicio hasta su final. El final de $O_{i,k}$, se denota como: $c_{i,k}$.

$$c_{i,k} = \begin{cases} \max\{t_j^S, c_{i,k-1}\} + p_{ikj} \leq t_j^E & si\ k \geq 2 \\ t_j^S + p_{ikj} \leq t_j^E & si\ k = 1 \end{cases}$$

Si considerados todos los intervalos de tiempo libres, en ninguno de ellos es posible que sea programada, $O_{i,k}$ será programada después de la última operación en la máquina. Todo el proceso anterior es ilustrado a través de un ejemplo en la figura 10.

Es posible observar que el mecanismo anterior permite que en principio sea posible que una operación O_B pueda ser procesada antes que una operación, O_A, aun cuando O_A la preceda en la secuencia de operaciones del cromosoma Cr_1. En la figura 8, la operación $O_{1,2}$ precede a la operación $O_{2,1}$ en Cr_1, ocupando la posición segunda y tercera respectivamente. Las dos primeras operaciones se asignan siguiendo el orden de Cr_1 : $O_{1,1}$ y $O_{1,2}$. La primera se procesa en la máquina M_1 y la segunda en M_2. La siguiente operación en la secuencia es $O_{2,1}$. Debido a que no cuenta con operación predecesora de trabajo y existe un intervalo de tiempo libre desde $0 \leq t \leq 5$, en M_2, $O_{2,1}$ es programada en M_2 iniciando $t = 0$ y finalizando en $t = 3$, siendo procesada antes que la operación $O_{1,2}$.

En los casos donde se presente la anterior situación, se debe recodificar la secuencia de tal manera que refleje el nuevo orden introducido en el proceso de decodificación. La recodificación es llevada a cabo antes de que el individuo sea sometido a los operadores genéticos de cruce y mutación. Así, los potenciales individuos que desciendan parcialmente del par-cromosoma original, pueden aprovechar las mejoras introducidas por el mecanismo de decodificado.

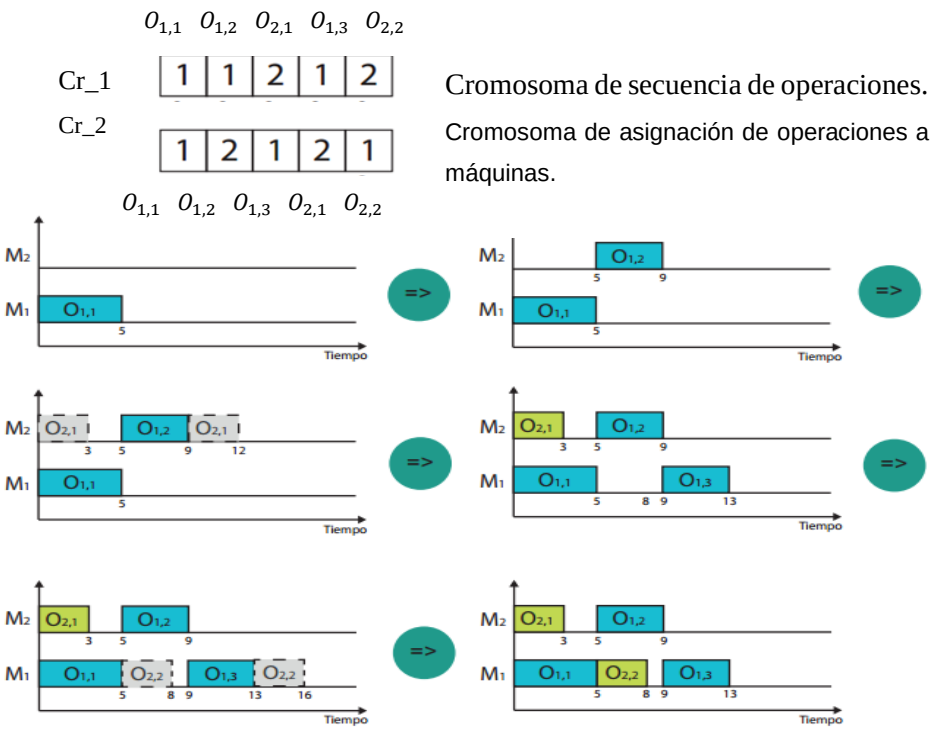

Figura 10 Proceso de decodificar par-cromosoma a una solución particular en un problema 2 x2.

6.7 Operador genético de cruzamiento

En los algoritmos genéticos el concepto de derivar una nueva población a partir de una actual, se logra a través de los operadores genéticos de cruzamiento y mutación (Gao, et al., 2008).

Cada nuevo individuo es generado por medio del cruzamiento de dos individuos pertenecientes a la población. El cruzamiento se lleva a cabo sobre la información del par-cromosoma de cada uno de los individuos progenitores. Cada nuevo descendiente, heredará cierta parte de la estructura genética de sus progenitores.

La principal ventaja del tipo de representación del cromosoma Cr_1, encargado de codificar la secuencia de operaciones, reside en que siempre garantiza una programación factible. Ello le exime llevar a cabo una costosa validación desde el punto de vista computacional. No obstante, su falta de referencia explícita respecto a la operación que se encuentra en cada alelo del cromosoma, causa

que se deba emplear otro tipo de representación adicional que refleje la estructura heredada de los progenitores.

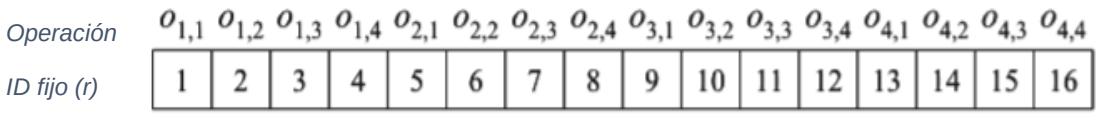

Figura 11 Asignación de número de identificación a cada operación

Antes de ser sometidos los individuos de la población al proceso de cruzamiento se debe modificar del tipo de representación original a la nueva representación, denominada representación por permutación. En ella, a cada operación le es asignado un número de identificación ID fijo. La figura 11 presenta la asignación a cada operación de su respectivo número de identificación fijo ID en un problema 4x4.

La información contenida en el cromosoma Cr_1 contendrá estos números ID fijos para referenciar las operaciones a procesar. El orden de las operaciones sigue indicando la prioridad de la programación de la operación. La figura 12 muestra la transformación de la representación de Cr_1 a partir de la representación original a la de tipo permutación. El cromosoma Cr_2 por el contrario, no sufre modificación alguna.

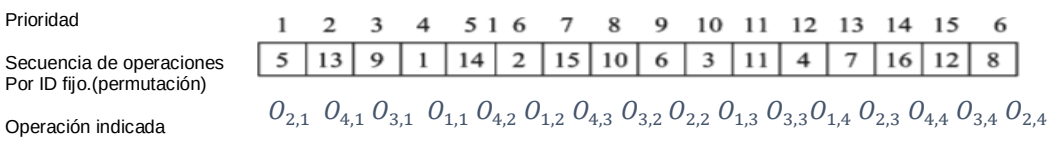

Figura 12 Transformación del cromosoma Cr_1 de la representación original al tipo permutación

La figura 13 exhibe el cruzamiento entre un par de cromosomas Cr_1 de dos individuos de acuerdo a los siguientes pasos:

1. Seleccionar una subsección de la secuencia de operaciones de un progenitor.
2. copiar la subsección del progenitor manteniendo las respectivas posiciones a la de un individuo descendiente.
3. Borre las operaciones del segundo progenitor que ya se encuentren copiadas en el descendiente. La secuencia resultante contendrá las operaciones que necesita el descendiente.
4. Ubique las operaciones resultantes del segundo progenitor en las posiciones aún sin ocupar del descendiente.

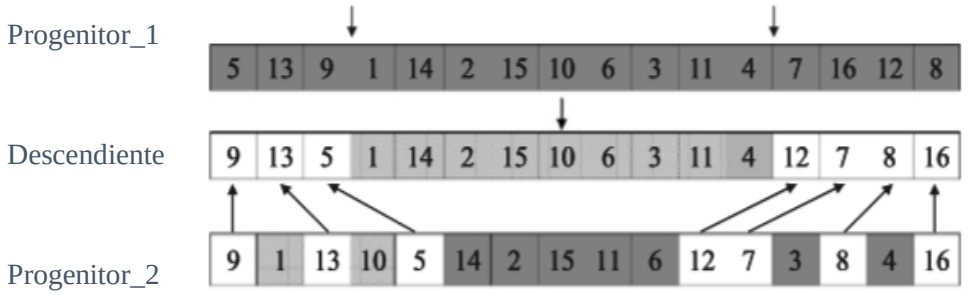

Figura 13 Cruzamiento de cromosomas.

Como resultado del cruzamiento se crea el nuevo cromosoma Cr_1 del individuo descendiente que hereda la estructura genética de sus progenitores. El artículo recomienda un índice de cruzamiento de 0,4.

El cruzamiento de un par de individuos comprende también el cruzamiento del cromosoma tipo Cr_2, el cual se controla por medio de un parámetro de cruzamiento uniforme. Se genera un número aleatorio entre 1 y 0; si el valor es menor al parámetro de cruzamiento uniforme, se copia el alelo

de Cr_2 del progenitor 1; de lo contrario, el alelo de Cr_2 del otro progenitor. El valor recomendado del parámetro de cruzamiento uniforme es 0,4.

6.8 Operador genético de mutación

El algoritmo híbrido implementa dos tipos de mutación: por alelo e inmigración. La mutación por alelo determina si el alelo de un cromosoma debe ser mutado de acuerdo a un parámetro de mutación. El valor recomendado por el artículo es 0,4. La mutación por inmigración examina a todos los individuos de la población en la primera generación. A cada uno de aquellos se le asigna un valor aleatorio entre1 y 0. Si el valor asociado a un individuo es menor al parámetro y el individuo no se encuentra en la población de la generación actual, es agregado; de lo contrario se omite. El valor recomendado del parámetro de mutación por inmigración es 0,4 (Gao, et al., 2008).

6.9 Operador genético de selección

El operador de selección exime al 30% de los individuos con los menores makespan, de ser sometidos al proceso de selección. Los individuos que hacen parte del anterior grupo, pasan a formar parte de la siguiente generación de manera inmediata (Gao, et al., 2008).

La probabilidad de un individuo de ser seleccionado depende en primera instancia de su puntaje de makespan. El intervalo de tamaño de población en el artículo se encuentra entre 300 y 3000 individuos. Aún con poblaciones entre 20 y 50 individuos, las diferencias entre los makespan de los individuos tendrán un margen muy estrecho. El algoritmo emplea un método de ranking para asignar las probabilidades a cada individuo (Yu & Gen, 2010). El individuo con el mejor makespan tendrá un ranking de cero y el peor uno igual al $(tamaño\ de\ población - 1)$. La probabilidad de cada individuo i a ser seleccionado se define como:

$$p_i = \frac{\propto + \frac{ranking_{\ i}}{tamaño\ de\ población\ -\ 1}\ (\beta - \alpha)}{tamaño\ de\ población}$$

Alfa y beta son parámetros que controlan la presión de selección y fueron establecidos a 1,3 y 0,7 respectivamente. El método asocia a cada individuo una probabilidad $p_{\ i}$ y una probabilidad acumulada $\sum_1^{tam\_pblc} p_i$. La Tabla 3 presenta el ranking, la probabilidad asignada y la probabilidad acumulada correspondiente. El método de ruleta genera un número aleatorio entre el intervalo abierto (0,1). Se compara el número con la probabilidad acumulada de cada individuo. El primer individuo con una probabilidad acumulada mayor es seleccionado. Si el número aleatorio fuera 0,85 por ejemplo, el individuo 6 saldría seleccionado.

Tabla 2.
Generación de probabilidades para cada individuo con el método de ranking

individuo	makespan	ranking	ρ_i	$\sum_1^{tam\_pblc} \rho_i$
indv_1	24	0	0,16	0,16
indv_2	24	1	0,15	0,31
indv_3	27	2	0,14	0,46
indv_4	28	3	0,12	0,71
indv_5	28	4	0,11	0,81
indv_6	29	5	0,1	0,91
indv_7	30	6	0,09	1

6.10 Datos de entrada del algoritmo híbrido

Los datos de entrada requeridos para la ejecución del algoritmo propuesto son:

- El número n de trabajos a procesar.
- El número m de máquinas disponibles.
- El número de operaciones n_i en que se descompone cada trabajo.
- Para cada una de las operaciones O_{ik}, se debe especificar el conjunto de máquinas M_{ik} en capacidad de procesarla, con sus tiempos de procesamiento respectivos.

Concluida la etapa de introducción de los datos de entrada, se establecen algunos parámetros como el número de generaciones, el tamaño de la población y las tasas de cruzamiento y mutación de individuos.

El algoritmo inicia con la generación de un conjunto de soluciones aleatorias a manera de población inicial. Cada pareja de cromosomas es decodificada a una *programación de operaciones* (representada como un diagrama de gantt) y su *makespan* calculado.

Por medio de un mecanismo de selección y cruce entre los miembros de la población y mutación, se generan nuevos individuos. Los individuos con el mejor *makespan* son seleccionados para conformar la siguiente generación. El proceso de búsqueda local contribuye a mejorar la calidad de los individuos en cada nueva generación. Este proceso se repite un número determinado de iteraciones. El algoritmo finaliza al alcanzar la última generación. Llegado a ese punto, se selecciona la solución que cuenta con el menor *makespan*. La solución encontrada debe ser cercana a la *solución óptima*.

7 Modelo de grafo dirigido

Una solución factible del FJSP puede ser también modelada por un grafo dirigido, $G = (N, A, E)$; La estructura del grafo G se compone de un conjunto N de nodos, un conjunto A de arcos ordinarios y un conjunto E de arcos disyuntos. Los nodos representan todas las operaciones que componen los n trabajos del problema. Dos nodos ficticios, S y T, a modo de fuente y sumidero, son introducidos en la estructura del grafo. El número asociado a cada nodo representa el tiempo de procesamiento de la operación.

El conjunto de arcos ordinarios (representados en línea continua) indican la secuencia ordenada de operaciones en cada uno de los n trabajos. Los arcos disyuntos (representados en línea entrecortada) indican las operaciones, que serán procesadas por cada maquina $M_j \in M,\ 1 \leq j \leq m$. La figura 12 representa una solución particular como grafo dirigido.

7.1 Tiempo de evento temprano y tardío

Los conceptos de tiempo temprano y tardío son centrales en el modelo de grafo dirigido. El tiempo de inicio temprano, representado por $S^E(O_r)$, es el instante de tiempo más temprano en que puede

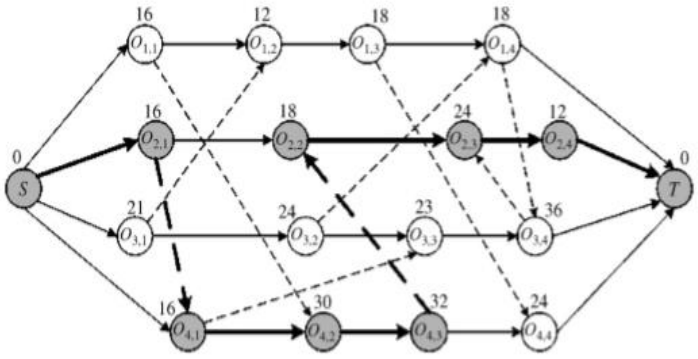

Figura 14 Esquema de representación de una solución particular a un problema FJSP 4x5, por medio de un grafo dirigido. Los arcos en negrilla representan la ruta crítica del grafo

iniciar el procesamiento de O_r. El tiempo de inicio tardío, representado por $S^L(O_r)$, representa lo más tarde que puede iniciar el procesamiento de O_r sin retrasar el makespan de la programación.

A partir de las definiciones anteriores, se establece el tiempo de finalización temprano y el de finalización tardío.

$$C^E(O_r) = S^E(O_r) + t(O_r), \quad C^L(O_r) = S^L(O_r) + t(O_r).$$

$t(O_r)$ denota el tiempo de procesamiento de la operación O_r. El tiempo de finalización temprano del primer nodo, el nodo S, se establece como cero. El tiempo de inicio temprano de un nodo O_r. se encuentra determinado por los tiempos de finalización temprano de sus operaciones predecesora de trabajo y predecesora de máquina.

$$S^E(O_r) = \max \{ \ c^E[PJ(O_r)], \qquad c^E[PM(O_r)] \ \}.$$

En caso de un nodo no poseer operación predecesora de trabajo u operación predecesora de máquina, el tiempo correspondiente a aquella operación será cero.

El cálculo del tiempo de finalización tardío se establece a partir del último nodo T. El tiempo de inicio tardío del nodo T se define como el makespan de la programación. El tiempo de finalización tardío de un nodo O_r se encuentra determinado por los tiempos de inicio tardío de sus operaciones sucesoras de trabajo y sucesora de máquina.

$$c^L(O_r) = \min \{ s^L[SJ(O_r)], \qquad s^L[SM(O_r)] \}.$$

En caso de un nodo no poseer operación sucesora de trabajo u operación sucesora de máquina, el tiempo correspondiente a aquella operación será infinito.

Cualquier nodo O_r cuyo inicio sea posterior a $s^L(O_r)$ incrementa el makespan. La diferencia entre $s^L(O_r)$ y $S^E(O_r)$ se denomina tiempo flotante. Las operaciones cuyo tiempo flotante son cero, son operaciones críticas. Un camino entre los nodos S y T, comprendido sólo por operaciones críticas, se llama ruta crítica. La longitud de la ruta crítica de una programación es precisamente su makespan.

7.2 Movimiento de una operación

La capacidad de búsqueda local bajo un marco conceptual de algoritmo evolutivo, caracteriza el diseño del algoritmo híbrido analizado en el presente proyecto de grado. Debido a ello, el algoritmo permite mejorar la explotación de soluciones particulares, a través del *movimiento* de *una* y *dos* operaciones.

Mover una operación O_r es eliminarla de su ruta crítica y reasignarla en otra posición factible del grafo solución. La etapa de eliminación comprende las siguientes acciones:

- Borrar los arcos disyuntos hacia y desde O_r.
- Trazar un arco disyunto desde el $PM(O_r)$ al $SM(O_r)$.
- Establecer un tiempo para el nodo O_r igual a cero.

El proceso completo de *mover una operación* O_r genera tres grafos diferentes. El grafo original G, antes de ser eliminada O_r. El grafo G^-, una vez ha sido eliminada O_r. Por último, el grafo resultante G', una vez O_r ha sido reasignado a una nueva máquina.

Reasignar la operación eliminada O_r exige encontrar una posición factible en el grafo G^-. Al tomar a $C_M(G)$ como el makespan "requerido" de G^- en el cálculo de los tiempos tardíos, se garantiza un $C_M(G^-)$ no mayor a $C_M(G)$.

El concepto de tiempo temprano y tardío permite establecer si una operación O_r puede ser reasignada en otra máquina. Lo más tarde que puede iniciar una operación O_v en la máquina $M(O_v)$ en G^-, sin causar retraso en el makespan es $s^{L-}(O_v)$; por el contrario, lo más temprano que puede terminar una operación $PM(O_v)$ es $c^{E-}[PM(O_v)]$. Ambos extremos de tiempo forman el máximo intervalo de tiempo *inactivo* previo a la operación O_v en la máquina $M(O_v)$. Si la operación O_v no tiene una operación predecesora de máquina, $c^{E-}[PM(O_v)]$ es cero.

Un intervalo de tiempo que inicia en t_j^S y finaliza en t_j^E en la máquina j se representa como $[t_j^S, t_j^E]$. La operación eliminada O_r sólo puede iniciar después de finalizar su operación predecesora de trabajo y finalizar antes que inicie su operación sucesora de trabajo. El intervalo es factible para asignar a O_r si:

$$\max\{t_j^S, c^{E-}[PJ(O_r)]\} + p_{r,j} \leq \min\{t_j^E, s^{L-}[SJ(O_r)]\}$$

La variable $p_{r,j}$ representa el tiempo de procesamiento de la operación O_r en la máquina j. Un intervalo de tiempo $[t_j^S, t_j^E]$ en la máquina j se denomina *asignable* a una operación O_r si:

$$\max\{t_j^S, c^{E-}[PJ(O_r)]\} + p_{r,j} < \min\{t_j^E, s^{L-}[SJ(O_r)]\}$$

De acuerdo con la definición anterior de intervalo asignable, el intervalo de tiempo máximo previo a O_v en G^-, es asignable para O_r, si:

$$\max\{c^{E-}[PM(O_v)], c^{E-}[PJ(O_r)]\} + p_{r,j} < \min\{s^{L-}(O_v), s^{L-}(O_r)\}$$

En caso de serlo, O_r es reasignada en la secuencia de la máquina $M(O_v)$ en la posición anterior a O_v. La reasignación de O_r comprende por último los siguientes pasos:

- Establecer el tiempo de procesamiento del nodo O_r en la máquina $M(O_r)$.
- Borrar el arco discontinuo desde $PM(O_v)$ hacia O_v.
- Conectar $PM(O_v)$ a O_r .
- Conectar O_r con O_v.

El makespan de una solución está determinado por la longitud de su ruta crítica. El propósito de la búsqueda local es identificar y romper las rutas críticas existentes una por una hasta obtener una programación de makespan $C_M(G')$ igual o menor a la del grafo original $C_M(G)$.

El mecanismo de búsqueda local inicia su rutina al seleccionar la primera operación crítica del grafo original G para ser movida. Si no es posible, selecciona la segunda operación; en general cuando no sea posible mover alguna operación, se considera la siguiente operación crítica, hasta encontrar una que sea susceptible a ser movida. Cada movimiento de una operación, representa una iteración del mecanismo de búsqueda y genera un grafo nuevo.

El grafo recién creado, garantiza un makespan igual o menor al anterior. Mover una operación es romper una ruta crítica. El mecanismo de búsqueda continuará rompiendo rutas críticas hasta que eventualmente encuentre un grafo en el cual ninguna operación crítica puede ser movida. Llegado al caso, la solución es un óptimo local. Lo anterior implica que no hay posibilidad de encontrar un intervalo de tiempo asignable para ninguna operación de $O_r \in P^*$. P^* denota la ruta crítica del grafo óptimo local.

7.3 Movimiento de dos operaciones

La concepción de *mover dos operaciones* $O_r \; y \; O_s, \; tal \; que \; O_r \neq O_s,$ es una extensión natural de *mover una operación.* Eliminar una segunda operación O_s genera mayor tiempo inactivo y la posibilidad de intervalos de tiempo asignables al par de operaciones $O_r \; y \; O_s$.

El mecanismo de *mover dos operaciones* inicia con una solución óptimo local obtenido a partir del mecanismo de mover una operación. La solución es susceptible a ser mejorada si se eliminan dos operaciones simultáneamente, una de las cuales debe pertenecer a la ruta crítica. El par de operaciones $O_r \; y \; O_s$ ($O_r$es la operación crítica) son eliminadas del grafo G, y se obtiene el grafo G^{--}. Se inicia la búsqueda de un intervalo asignable para O_r en G^{--}. Si el intervalo es encontrado, O_r es reasignada y se inicia la búsqueda de un intervalo asignable para O_s en G^{-}. En caso de no ser posible reasignar a O_r u O_s, se elige otra operación O_s, para formar una nueva pareja ($O_r, \; O_s$) de operaciones a eliminar. Si O_s es reasignada, el grafo G' garantiza un makespan igual o menor que la solución de óptimo local original.

La solución resultante del mecanismo de mover dos operaciones no llega a un nuevo óptimo local. Sin embargo, lo anterior es precisamente lo que permite que el individuo se encuentre en capacidad de seguir evolucionando en cada generación (Gao, et al., 2008).

El mayor efecto positivo de hibridar un algoritmo de tipo genético con mecanismos de búsqueda local es mejorar la velocidad de convergencia a óptimos locales. Sin embargo, la mejora en la calidad de las soluciones es muy exigente en términos de poder computacional. El nivel de complejidad computacional del movimiento de dos operaciones es mucho mayor que el de una operación. El movimiento de una operación lleva a cabo iterativamente movimientos sobre una

solución hasta ser transformada a un óptimo local. Por el contrario, el movimiento de *dos operaciones* termina al reasignar el par de operaciones O_r y O_s.

A continuación se presenta el pseudocódigo para el movimiento de una operación, el movimiento de dos operaciones y el algoritmo híbrido genético con búsqueda local estudiado (Gao, et al., 2008).

Pseudocódigo del algoritmo híbrido propuesto.

$P(t)$: $progenitores\ \ de\ una\ determinada\ generación.$

$C(t)$: $descendientes\ de\ una\ determinada\ generación.$

Procedimiento: algoritmo híbrido genético.
Entrada: datos de entrada del problema, parámetros de la componente genética.
Salida: una programación cercana al óptimo.

Inicio

$t \leftarrow 0$;

Inicializar $P(t)$ con la representación de pareja de cromosomas.

Evaluar aptitud (P) decodificando basado en prioridad.

Reordenar la secuencia de operaciones de acuerdo al tiempo de inicio.

Mientras ***(***no sea la última generación.) *hacer*

Cruzar $P(t)$ para producir $C(t)$ por medio de cruce mejorado y cruce ordenado;

Mutar $P(t)$ para producir $C(t)$ por mutación de alelos y mutación por inmigración;

Mejorar $P(t)\ y\ C(t)$ a su óptimo local para producir $P'(t)\ y\ C'(t)$ por búsqueda local de movimiento de una operación;

Mejorar $P'(t)\ y\ C'(t)$ para producir $P''(t)\ y\ C''(t)$ por búsqueda local de movimiento de dos operaciones;

Evaluar aptitud ($P''(t)\ y\ C''(t)$) por decodificado basado en prioridad;

Seleccionar $P(t+1)$ de $P''(t)\ y\ C''(t)$ por muestreo combinado;

Reordenar la secuencia de operaciones de acuerdo al tiempo de inicio;

$t \leftarrow t+1$;

Fin

Salida: una programación de operaciones cercano al óptimo

Fin.

Pseudocódigo de búsqueda local moviendo una operación.

1. Identificar una ruta crítica P dada una solución particular S.
2. Sea O_r la primera operación de P.
3. Repetir
    a) Eliminar la operación O_r del grafo solución G de S para obtener G^-.
    b) Buscar un intervalo de tiempo asignable para O_r en G^-.
    c) En caso de no encontrar un intervalo asignable, definir O_r como la siguiente operación de P.
4. Hasta encontrar un intervalo de tiempo asignable o que la operación O_r sea la última operación de la ruta crítica.
5. Si se encuentra un intervalo de tiempo asignable, asigne la operación O_r en el intervalo. De lo contrario, S en un óptimo local de mover una operación.

Pseudocódigo del procedimiento de búsqueda local moviendo dos operaciones.

1. Identificar una ruta crítica P dada una solución particular S.
2. Sea O_r la primer operación de P.
3. Repetir

a. Sea O_s $(O_s \neq O_r)$ una operación de la solución S.
b. Repetir
    i. Eliminar las operaciones $O_r \; y \; O_s$ del grafo solución G de S para obtener G^{--}.
    ii. Buscar un intervalo de tiempo asignable para la operación O_r en G^{--}.
    iii. Si el intervalo de tiempo asignable para O_r es encontrado, programar a O_r en el intervalo para obtener G^{-}; de lo contrario diríjase al paso vi).
    iv. Buscar un intervalo de tiempo asignable para la operación O_s en G^{-}.
    v. Si el intervalo de tiempo asignable para O_s es encontrado, insertar a la operación O_s en el intervalo para producir G'.
c. Si no es encontrado un intervalo de tiempo para la operación $O_r \; u \; O_s$, escoger una nueva operación O_s $(O_s \neq O_r)$ en G.
d. Hasta que sean encontrados intervalos para ambas operaciones $O_r \; y \; O_s$; u O_s sea la última operación en G
e. Si el intervalo de tiempo no es encontrado para $O_r \; u \; O_s$, establecer a O_r como la próxima operación crítica de P.

4. **Hasta** que ambos intervalos para $O_r \; y \; O_s$ sean encontrados u O_r sea la última operación de la ruta crítica.

8 Presentación de resultados

8.1 Pruebas de rendimiento

El algoritmo propuesto fue sometido a un conjunto de pruebas de benchmarking con el propósito de medir su rendimiento. Se ejecutaron dos de las tres instancias de los problemas de Xia y Wu: Xwdata_8x8 y Xwdata_10x10. Igualmente, se ejecutaron en su totalidad las diez instancias de Brandimarte.

Para la primera prueba fue seleccionada la instancia Brdata_Mk02_10x6 con índice de flexibilidad 4,1. El índice anterior indica el número de máquinas que en promedio pueden procesar

una operación. La referencia 10x6 en el nombre de la instancia, denota el tamaño del problema: 10 trabajos y 6 máquinas. En el anexo A puede ser consultada la matriz de procesamiento de Brdata_Mk02_10x6. El tamaño de población se mantuvo igual a 6 en toda la primera prueba. El algoritmo híbrido fue ejecutado para 9 valores distintos de número de generaciones.

La Tabla 4 presenta un resumen de las 9 rondas de la primera prueba. Los resultados se presentan por columnas, de acuerdo al número de generaciones establecido en cada ronda. El segundo renglón es el promedio del makespan del 30% de los individuos más aptos, esto es, los individuos con los menores makespan. El menor makespan por ronda se encuentra en el tercer renglón. En el anexo C se encuentran los resultados discriminados por ronda.

Los datos de la Tabla 4 indican una relación entre el makespan y el número de generaciones. La figura 15 presenta la gráfica de makespan Vs. número de generaciones de la Tabla 4, para el menor makespan y para el 30% de la población con menor makespan. La curva de mejor ajuste para ambos conjuntos de puntos es una función logarítmica.

Tabla 3

Primera prueba de instancia Brdata_Mk02_10x6 con población fija de seis y número de generaciones variable

Número de generaciones por ronda.	4	6	8	10	12	14	16	20	24
Promedio de makespan del 30% más apto.	32,67	32,67	31,67	31,67	31,00	31,67	31,33	31,33	30,67
Mejor makespan individual.	31	31	30	31	30	31	30	29	29

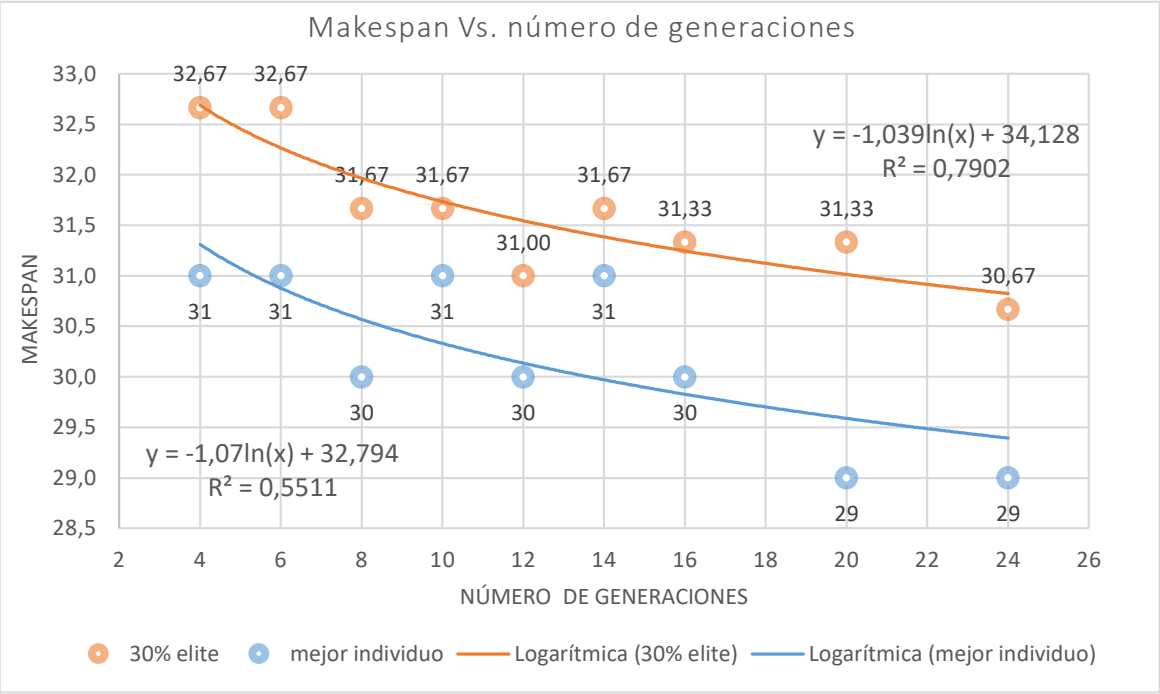

Figura 15 Gráfica de primera prueba sobre la instancia Brdata_Mk02_10x6 con tamaño de población fija y número de generaciones variable.

En la segunda prueba fue empleada nuevamente la instancia Brdata_Mk02_10x6. Se estableció un número constante de generaciones igual a dos y tamaño de población variable. En total se ejecutaron 6 rondas, con los siguientes tamaños de población: 8, 16, 28, 36, 56 y 76. En el segundo y tercer renglón de la Tabla 5, se encuentra el registro del promedio de los 8 menores makespan y el menor makespan por ronda respectivamente.

Tabla 4
Segunda prueba con número de generación fijo y población variable

Tamaño de la población	8	16	28	36	56	76
Promedio de los mejores 8 individuos	35,87	33,87	33,87	33	32,75	32,5
Mejor makespan	33	33	33	32	31	31
Tiempo de procesamiento [minutos]	4	8,6	14,8	19,6	30,5	38

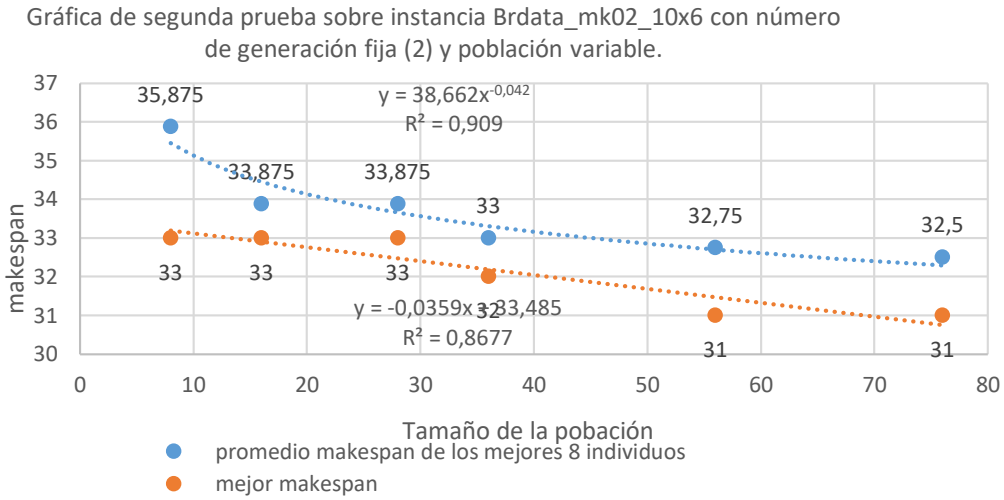

Figura 16 Gráfica de segunda prueba sobre instancia Brdata_Mk02_10x6

La figura 16 presenta la gráfica del makespan Vs. el tamaño de la población de la Tabla 4 para el promedio de los 8 menores makespan y el menor makespan por ronda respectivamente. Adicionalmente la figura 17 muestra la relación entre el tamaño de la población por ronda Vs. el tiempo de procesamiento del algoritmo híbrido en minutos. El ajuste del conjunto de puntos establece una relación lineal con un coeficiente de determinación de 0,91. En general, las dos primeras pruebas muestran que el comportamiento del valor del makespan depende tanto del tamaño de la población como del número de generaciones.

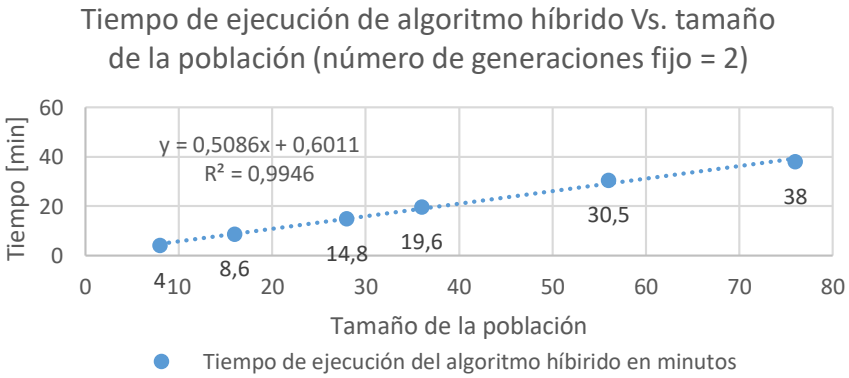

Figura 17 Tiempo de ejecución de algoritmo híbrido Vs. tamaño de la población

En la tercera prueba se emplea una nueva instancia, Brdata_Mk05_15x4, con una población constante de 4 individuos y un número de generaciones igual a 13. La Tabla 6 representa la historia evolutiva de los individuos con los 4 mejores makespan de la prueba. Cada columna en la Tabla 6 describe la generación en que cada individuo entró a la población y las generaciones durante las cuales mejoró su makespan.

Tabla 5

Historial evolutivo de los 4 individuos con menor makespan

	Generaciones												
	1	2	3	4	5	6	7	8	9	10	11	12	13
indv_1		186	185	181									
indv_2											200	196	195
indv_3													197
indv_4													212

8.2 Pruebas de benchmarking

La Tabla que se presenta en el apéndice C es una comparativa de rendimiento entre el algoritmo de los autores del artículo y el implementado en el presente proyecto de grado. Cada renglón de la tabla hace referencia a una instancia de Brandimarte. En la segunda y tercera columna se especifica el tamaño $n\ x\ m$ de la instancia y su índice de flexibilidad correspondiente. La flexibilidad se define como la cantidad promedio de máquinas capacitadas para procesar las operaciones $O_{i,k}$ del problema

En cada renglón, la instancia presenta dos parámetros de entrada ajustados a juicio del programador: el tamaño de la población (T.P) y el número de generaciones (N.G). Un mayor tamaño de población o un mayor número de generaciones incrementa las posibilidades de individuos con menor makespan, menor maximal de carga y menor carga total pero también conlleva un costo computacional considerable.

Es posible comparar para cada instancia de la familia de problemas Brandimarte, las tres medidas de desempeño del problema para ambos algoritmos: makespan, maximal de carga y carga total. En la última columna de la tabla del apéndice C se presenta la diferencia porcentual del makespan entre los dos algoritmos.

9 Conclusiones

La totalidad de los objetivos propuestos en el proyecto de grado han sido cumplidos exitosamente. La ejecución del algoritmo sobre las diez instancias de Brandimarte, evidenció un rendimiento homologable en las instancias pequeñas y medianas respecto al algoritmo original de los autores del artículo. En instancias superiores, el rendimiento del algoritmo implementado en el proyecto, a pesar de obtener resultados satisfactorios, no pudo igualar o superar los resultados del algoritmo del artículo original.

Sin embargo, es notable la diferencia en los tamaños de población y el número de generaciones empleados en ambos algoritmos. El algoritmo del artículo usó tamaños de poblaciones que variaron entre 1000 y 3000 individuos de acuerdo a la instancia ejecutada. Adicionalmente, los autores

establecieron un número fijo de 200 generaciones para todas las instancias Brandimarte. El algoritmo del proyecto, por el contrario, no superó en ninguna instancia, un número de 30 generaciones y ninguna población fue mayor a 30 individuos. Es razonable esperar que con tamaños de población y número de generaciones semejantes a las del artículo, el desempeño del algoritmo implementado en el proyecto, igualaría o incluso podría exceder el del algoritmo original.

El componente de búsqueda local representa la mayor carga computacional del algoritmo implementado en el presente proyecto de grado. La rutina de mover una o dos operaciones, exige tres tareas de carga considerable:

- Una vez eliminada la operación O_r u O_s, los tiempos tempranos y tardíos en todos los nodos del grafo G deben ser recalculados.
- verificar que la posición candidata a ser ocupada por O_r u O_s, no genera bucles en las operaciones predecesoras o sucesores de la operación reasignada.
- Una vez reasignada la operación O_r u O_s, recalcular los tiempos tempranos y tardíos en todos los nodos del grafo G.

La segunda tarea, siendo la más intensiva en términos computacionales, es susceptible a ser rediseñada. Lo anterior implica encontrar un enfoque alternativo para recalcular los tiempos tempranos y tardíos una vez las operaciones O_r u O_v, son reasignadas en sus nuevas máquinas.

Teniendo en cuenta los resultados obtenidos en las pruebas de rendimiento se comprobó que el algoritmo presentado en el artículo funciona dentro del marco de los algoritmos evolutivos; igualmente, su componente de búsqueda local cumple el objetivo de llevar a cabo explotación sobre las soluciones producto de los operadores genéticos. De acuerdo a las pruebas, aumentar el tamaño

de la población y el número de generaciones incrementa la calidad de las soluciones encontradas y contribuye a encontrar el menor makespan de la población.

La elaboración del presente proyecto ha exigido el desarrollo de habilidades implícitas en la metodología de la investigación científica. La lectura y el estudio de artículos especializados han sido fuente de ideas y conceptos nuevos requeridos para la comprensión precisa del problema del *FJSP*. El requerimiento del proyecto de implementar el algoritmo híbrido en un programa de computadora, ha conducido a resolver un sin número de cuestiones de naturaleza teórica que previamente no habían sido advertidas. Se ha resuelto también un problema de carácter práctico relativo al modelamiento del grafo como objeto matemático. La etapa final de la escritura del programa ha consistido en la depuración del código. Se escribieron subprogramas en el algoritmo principal con el propósito de probar que el algoritmo no presentaba algún tipo de falla. Se llevaron las respectivas pruebas de rendimiento y se presentaron los resultados.

El anterior proceso de comprender un problema a nivel teórico y tener la capacidad de transformarlo a unas condiciones concretas, es quizá el logro más importante del presente proyecto de grado. En trabajos futuros, se sugiere adaptar esta heurística a otros problemas de Scheduling para darle un tratamiento más cercano a los casos reales de la industria.

10 Recomendaciones

Examinar y plantear un nuevo mecanismo de eliminar/insertar operaciones en la metodología propuesta en el presente proyecto de grado. Se sugiere a manera de ejercicio, la siguiente propuesta: examinar por etapas el conjunto de operaciones sucesoras de trabajo y de máquina a partir de la

operación eliminada O_r. En la primera etapa se establecen los tiempos de inicio y final tempranos de los sucesores de O_r. En la segunda etapa, se examina el conjunto de operaciones sucesoras del conjunto de operaciones anterior. Este proceso se repite hasta que no existan más operaciones que cuenten con operaciones sucesoras de trabajo o de máquina. Se ha logrado de esta manera actualizar los tiempos de inicio y final temprano de todas las operaciones afectadas por la reasignación de O_r. Los tiempos de inicio y finalización tardíos se calculan recorriendo en sentido inverso las operaciones comprendidas en cada una de las etapa anteriores hasta llegar nuevamente a O_r. La segunda parte del mecanismo es la actualización de los tiempos de inicio y finales tardíos del conjunto de operaciones predecesoras a partir de O_r. La descripción conceptual del nuevo mecanismo podría representar una economía en la carga computacional y una mejora significativa frente algoritmo propuesto.

El desarrollo de la habilidad de programar se debe reforzar por medio de ejercicios planteados por profesores(as), modificando o refinando funciones de algoritmos ya implementados en proyectos de grado anteriores.

Brindar la oportunidad de dar a conocer los trabajos de investigación a empresas del ámbito local y nacional. Muchos métodos heurísticos son susceptibles a ser aplicados en problemas que surgen de la programación de operaciones en la mayoría de organizaciones.

Referencias Bibliográficas

BAASE, Sara y VAN GELDER, Allen. Algoritmos computacionales : Introducción al análisis y diseño. 3 ed. México: Pearson Educación, 1002. p. 549

BARNES, J Y CHAMBERS, J. Tabu search for the job shop scheduling. En: Technical report ORP 9609, University of Texas [en línea]. Vol. 9 (1996) [consultado mar 2014] disponible en <http://www.cs.utexas.edu/users/jbc/>

BAZARA, Mokhtar S; JARVIS John y SHERALI, Hanif D. Programación lineal y flujo de redes. México: Limusa-Noriega editores, 1998. p.2.

BRANDIMARTE, Paolo. Routing and Scheduling in a flexible job shop taboo search. . En: Annals of Operation Research. Vol. 41, No. 3 (1993); p. 157-183.

BRUCKER, P y SCHLIE, R. Job-shop scheduling with multipurpose machines. En: Springer-Verlag [bases de datos en línea]. Vol. 45, No.4 (ene 1990); p.369-375. Disponible en <http://rd.springer.com/article/10.1007%2FBF02238804>

BURKARD, Rainer, Efficiently solvable special cases of hard combinatorial optimization problems. En: Mathematical Programming, 1997, Vol. 79, pp 55-69.

CHEN, H; IHLOW, J y LEHMAN, C. A genetic algorithm for flexible Job-shop scheduling. En: IEEE international conference on robotics and automation. Vol. 2 (1999); p. 1120-1125. Disponible en <http://ieeexplore.ieee.org/Xplore/home.jsp >

COTE MARTÍNEZ, Gina Marcela y MENDOZA MORENO, Diana Carolina. Metaheurística de honey bees.matting optimization (HBMO) aplicada al problema de distribución de planta FLP de un solo nivel y departamentos de plantas de áreas iguales o desiguales. Bucaramanga, 2012, 155 p. Proyecto de grado de Ingeniería Industrial. Universidad Industrial de Santander. Facultad de Físico-mecánicas.

CRUZ, Cesar. Optimización de enjambre de partículas (PSO) aplicada al problema de la P-mediana. Bucaramanga, 2013, Trabajo de grado en Ingeniería Industrial. Universidad Industrial de Santander. Facultad de Físico-Mecánicas.

DAUZÉRE-PERES, S y PAULLI, J. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search.En: Annals of Operations Research, Vol. 70, No. 0 (1997); p.281–306.

FATTAHI, P; SAIDI MEHRABAD, M y JOLAI F. Mathematical modelling and heuristic approaches to flexible job shop scheduling problems. En: Journal of intelligent Manufacturing. Vol.18, No. 3 (2007); p. 331-342.

FRENCH, Simon. Sequencing and Scheduling: An introduction to the mathematics of the Job Shop. Jon Wiley & Sons, 1982. P. 8-9.

GAO Jie, SUN Linyan y GEN ,Mitsuo. A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. En: Computers and operations research. Vol 35(2007), p. 2892-2907.

GAO, J; SUN, L y GEN, M. A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. En: Computers & Operations Research. Vol. 35, No. 9 (2008); p. 2892-2907.

HILLIER, Frederick S. y LIEBERMAN, Gerald J. Introduction to Operations Research.ed. 7.Mc-Graw Hill. New York, 2001. p. 10-11.

HURINK Johann; JURISCH Bernd y THOLE Monik. Tabu search for the job shop scheduling. En: Operations Research Spektrum. Vol. 41, No. 4 (1994); p. 205-215.

MASTROLILI, M y GAMBARDELLA, L. Effective neighborhood functions for the flexible job shop problem. En: Journal of Scheduling. Vol. 3, No. 1 (2000); p. 3-20.

MELIÁN BATISTA, Belén; MORENO PÉREZ José y MORENA VEGA, J. Marcos. Algoritmos Genéticos: una visión práctica. En: Números: Revista Didáctica de las matemáticas. Vol 71, No. 09 (Agosto 2009); p. 29-47

MUÑOZ NEGRÓN, David. Administración de las operaciones : enfoque de administración de procesos de negocios. 2 ed. México : CENGAGE Learning, 2009. p. 17-18.

OSORIO, Juan Carlos y MOTOA, Tulio Gerardo. Planificación jerárquica de la producción de un job shop flexible. En: Revista de facultad de ingenierías, Universidad de Antioquia. No. 44 (2008); p. 158-171.

PÉREZ JIMÉMNEZ, Mario y SANCHO CAPARRINI, Fernando. Máquinas moleculares basadas en ADN. España. Universidad de Sevilla, secretariado de publicaciones, 2003. P. 19-29 (Colección divulgación científica; no.2)

PEZZELA, F; MORGANTI, G y GIASCHETTI, F. A genetic algorithm for the flexible job-shop problem. En: Computers & Operations Research. Vol.35, No. 10 (2008); p. 3202-3212.

TALBI, El-Ghazali. Metaheuristic : From design to implementation. New Jersey : John Wiley & Sons, 2009. p. 121-122.

VIDAL ESMORÍS, Aitana. Algoritmos heurísticos en optimización Santiago de Compostella, 2013, 94 p. Proyecto de grado de Master en Técnicas estadísticas. Universidad de Santiago de Compostela. Facultad de Matemáticas.

XIA, W y WU, Z. An effective hybrid optimization approach for multiobjective flexible job-shop problem. En: Computers & Industrial Engineering, Vol. 48, No. 2 (2005); p. 409-425.

YAZDANI, M; AMIRI, M y ZANDIEH, M. Flexible job-shop scheduling with parallel variable neighborhood search algorithm. En: Expert Systems with Applications. Vol. 37, No.1 (ene 2010); p. 678-687.

YU ,Xinjie y GEN, Mitsuo. Introduction to evolutionary algorithms. En: Springer-Verlag, 2010, pp 72-73.

ZHANG, H y GEN, M. Multistage-based genetic algorithm for flexible Job-Shop scheduling problem. En: Journal of Complexity International. Vol. 48 (2005); p. 409-425.

Apéndices

Apéndice A

Matriz de procesamiento de la instancia Brandimarte Mk_02_10X6. Cada renglón de la matriz representa las unidades de tiempo que le toma a la operación $k-ésima\ del\ trabajo\ i,\ O_{ik}$ ser procesada en una determinada máquina ($M_1, M_2, \ldots M_6$). Un tiempo de procesamiento igual a cero representa que la operación no puede ser procesada en una máquina determinada.

	M1	M2	M3	M4	M5	M6
O1,1	3	2	3	5	3	6
O1,2	0	0	4	0	0	5
O1,3	1	6	3	3	6	5
O1,4	0	6	0	0	0	0
O1,5	0	0	0	0	6	3
O1,6	0	1	2	0	4	0
O2,1	3	4	0	2	6	1
O2,2	0	0	0	0	6	3
O2,3	0	0	0	0	2	0
O2,4	0	4	3	0	0	0
O2,5	0	1	2	0	4	0
O2,6	3	2	3	5	3	6
O3,1	1	6	3	3	6	5
O3,2	2	4	6	6	3	6
O3,3	0	1	2	0	4	0
O3,4	4	3	5	0	2	3
O3,5	5	4	3	1	5	3
O3,6	4	0	6	6	3	3
O4,1	4	3	5	0	2	3
O4,2	3	4	0	2	6	1
O4,3	0	6	0	0	0	0
O4,4	1	6	3	3	6	5
O4,5	4	3	0	5	4	3
O4,6	5	4	3	1	5	3
O5,1	2	4	6	6	3	6
O5,2	4	3	0	5	4	3

O5,3	0	0	0	3	0	0
O5,4	4	0	6	6	3	3
O5,5	3	4	0	2	6	1
O5,6	0	4	3	0	0	0
O6,1	4	0	6	6	3	3
O6,2	0	0	4	0	0	5
O6,3	4	3	5	0	2	3
O6,4	2	4	6	6	3	6
O6,5	0	6	0	0	0	0
O6,6	3	4	0	2	6	1
O7,1	5	4	3	1	5	3
O7,2	0	0	0	0	2	0
O7,3	2	4	6	6	3	6
O7,4	3	2	3	5	3	6
O7,5	4	0	6	6	3	6
O8,1	0	4	3	0	0	0
O8,2	4	3	5	0	2	3
O8,3	2	4	6	6	3	6
O8,4	4	0	6	6	3	3
O8,5	4	3	0	5	4	3
O8,6	3	4	0	2	6	1
O9,1	0	6	0	0	0	0
O9,2	0	0	4	0	0	5
O9,3	3	4	0	2	6	1
O9,4	4	3	0	5	4	3
O9,5	0	4	3	0	0	0
O10,1	0	0	0	3	0	0
O10,2	2	4	6	6	3	6
O10,3	4	0	6	6	3	3
O10,4	5	4	3	1	5	3
O10,5	0	0	0	0	6	3
O10,6	3	4	0	2	6	1

Apéndice B

Matriz de procesamiento Brandimarte, instancia Mk_05_15X4. Cada renglón de la matriz representa las unidades de tiempo que le toma a la operación $k-ésima\ del\ trabajo\ i,\ O_{ik}$ ser procesada en una determinada máquina ($M_1, M_2, \ldots M_4$). Un tiempo de procesamiento igual a cero representa que la operación no puede ser procesada en una máquina determinada.

	M1	M2	M3	M4
O1,1	0	7	5	0
O1,2	8	0	0	8
O1,3	6	5	0	0
O1,4	0	0	7	0
O1,5	0	6	0	5
O1,6	5	0	0	5
O2,1	0	0	7	0
O2,2	6	5	0	0
O2,3	0	0	0	6
O2,4	0	6	0	5
O2,5	8	6	0	0
O3,1	0	0	9	7
O3,2	0	7	5	0
O3,3	5	0	0	5
O3,4	8	0	0	8
O3,5	6	5	0	0
O3,6	0	0	0	6
O3,7	8	6	0	0
O3,8	0	0	6	9
O4,1	5	0	0	5
O4,2	0	0	9	7
O4,3	8	0	0	8
O4,4	0	0	0	8
O4,5	8	6	0	0
O4,6	0	6	0	5
O4,7	0	0	0	6
O5,1	5	0	7	0
O5,2	0	7	0	6
O5,3	0	0	9	7
O5,4	0	0	8	0
O5,5	0	7	5	0

O5,6	8	6	0	0
O6,1	0	0	0	6
O6,2	0	6	0	5
O6,3	0	0	8	0
O6,4	5	0	7	0
O6,5	0	7	0	6
O6,6	0	0	0	8
O6,7	8	6	0	0
O6,8	8	0	0	8
O6,9	5	0	0	5
O7,1	0	0	8	0
O7,2	0	0	9	7
O7,3	6	5	0	0
O7,4	0	7	0	6
O7,5	0	0	7	0
O8,1	5	0	7	0
O8,2	0	0	8	0
O8,3	0	0	9	7
O8,4	5	0	0	5
O8,5	0	0	7	0
O8,6	0	0	0	8
O8,7	0	0	6	9
O8,8	6	5	0	0
O9,1	0	7	5	0
O9,2	0	0	0	8
O9,3	0	6	0	5
O9,4	6	5	0	0
O9,5	0	0	0	6
O9,6	8	0	0	9
O9,7	8	0	0	8
O9,8	8	6	0	0
O9,9	0	0	7	0
O10,1	8	6	0	0
O10,2	8	0	0	8
O10,3	8	0	0	9
O10,4	0	0	6	9
O10,5	6	5	0	0
O10,6	0	0	8	0
O10,7	0	0	7	0
O10,8	0	0	0	6
O10,9	0	6	0	5

O11,1	8	6	0	0
O11,2	8	0	0	8
O11,3	6	5	0	0
O11,4	0	0	7	0
O11,5	0	0	0	6
O11,6	0	0	8	0
O11,7	0	0	6	9
O12,1	0	0	0	8
O12,2	0	0	7	0
O12,3	0	0	9	7
O12,4	6	5	0	0
O12,5	0	0	8	0
O12,6	8	0	0	8
O13,1	0	0	0	8
O13,2	0	0	6	9
O13,3	8	0	0	8
O13,4	0	7	0	6
O13,5	0	7	0	6
O13,6	8	6	0	0
O13,7	5	0	7	0
O14,1	6	5	0	0
O14,2	5	0	7	0
O14,3	8	0	0	8
O14,4	8	6	0	0
O14,5	5	0	0	5
O14,6	0	7	0	6
O14,7	0	0	0	6
O15,1	0	0	8	0
O15,2	8	0	0	9
O15,3	0	0	6	9
O15,4	0	0	7	0
O15,5	0	6	0	5
O15,6	8	6	0	0
O15,7	6	5	0	0

Apéndice C

Comparativa de rendimiento entre el algoritmo del artículo y el implementado en el presente proyecto de grado

			Algoritmo del artículo					Algoritmo del proyecto					Diferencia porcentual entre ambos algoritmos
Serie de problemas de familia Brandimarte			T.P	N.G	MKS	M.C	C.T	T.P	N.G	MKS	M.C	C.T	
INSTANCIA	TAMAÑO *n x m*	FLEXIBI-LIDAD											
Mk01	10 X 6	2,09	3000	200	40	36	167	20	20	41	38	193	2,5%
Mk02	10 X 6	4,1	3000	200	26	26	151	30	31	29	29	163	11,5%
Mk03	15 X 8	3,01	2000	200	204	204	850	12	8	204	204	1116	0,0%
Mk04	15 X 4	1,91	3000	200	60	60	375	8	8	68	67	400	13,3%

Mk05	10 X 15	1,71	1000	200	172	172	687	18	12	180	178	702	4,7%
Mk06	10 X 15	3,27	2000	200	58	58	427	12	4	73	71	481	25,9%
Mk07	20 X 5	2,83	1000	200	139	139	693	14	14	149	149	715	7,2%
Mk08	20 X 10	1,43	1000	200	523	523	2524	2	2	556	548	2674	6,3%
Mk09	20 X 10	2,53	1000	200	307	307	2312	4	2	338	358	2593	10,1%
Mk10	20 X 15	2,98	2000	200	197	197	2029	4	3	237	238	2215	20,3%