CONCIENCIA TECNOLÓGICA No. 56, Julio-Diciembre 2018: 54-58

Implementación de un Algoritmo Genético para la Programación de Operaciones en Configuración FJSSP (Flexible Job Shop Scheduling Problem)

Implementation of a Genetic Algorithm for the FJSSP (Flexible Job Shop Scheduling Problem)

Reporte de Proyecto

Ing. Mario E. Marín-Limón[1], Ing. Guillermo C. Rangel-Sánchez1, Ing. Javier Barrios-Aviña1, Ing. Demetrio R. Briones-Briones1, Ing. Saúl Domínguez-Casasola1, Dr. Ricardo Pérez-Rodríguez[1,2.]
[1]Maestría en Modelación y Optimización de Procesos.
Centro de Investigación en Matemáticas CIMAT, A.C.
Bartolomé de las Casas 314, Barrio la Estación, Aguascalientes, Ags., México.
email: ricardo.perez@cimat.mx
[2]CONACYT - CIMAT, A.C.

Resumen

El FJSSP, por sus siglas en inglés (Flexible Job Shop Scheduling Problem), consiste en asignar de manera inteligente órdenes de producción a un conjunto de máquinas, de tal modo que se minimice el tiempo de operación. Tal tiempo se refleja mayormente por la suma de los tiempos de operación en todas las máquinas, o bien por el makespan, es decir, el tiempo en que se termina la última operación en ser procesada en un conjunto de órdenes de producción. Dada la complejidad del problema, y la recurrencia con la que se presenta, un método de optimización exacto generalmente no es viable, por ello se han propuesto diversas estrategias, entre ellas el uso de heurísticas. En este reporte de proyecto, se propone un algoritmo genético, resolviendo instancias reportadas en la literatura, discutiendo los resultados analizados en términos de eficiencia y estabilidad. Con el método propuesto se puede mejorar el desempeño en piso de producción y se ofrecen ideas para aplicaciones futuras de este tipo de algoritmos al problema en cuestión.

Palabras clave: FJSSP, algoritmo genético, programación de la producción.

Abstract

The Flexible Job Shop Scheduling Problem (FJSSP), consists of intelligently assigning customer orders to a set of machines, in such a way as to minimize the whole time of operation. It is been mostly due to the sum of the times of the operations in all the machines, or the time in which the last operation to be processed in a set of production orders is finished, i.e., the makespan. Given the complexity of the problem, an exact optimization method is generally not viable. Various strategies have been proposed, including the use of diverse heuristics. In this report, a genetic algorithm is proposed, resolving instances reported in the literature, and the results are analyzed in terms of efficiency and stability. With the proposed method we can improve the performance on the production floor and we offer ideas for future research of this type of algorithms to resolve the aforementioned problem.

Keywords: FJSSP, genetic algorithm, production scheduling.

Introducción

El FJSSP por sus siglas en inglés (Flexible Job Shop Scheduling Problem) es un problema de la planeación de la producción, que es conocido por la dificultad de resolverlo, producto del aspecto combinatorio que lleva consigo [1]. En términos generales consiste en determinar qué operación debe ser procesada en qué máquina, de un conjunto dado. El problema consiste en asignar cada operación a la máquina más idónea, tal que el tiempo máximo en que se completen las operaciones, es decir, el makespan, sea minimizado [2]. La aplicación inmediata, claramente se encuentra en la planeación de la producción de diversas industrias, principalmente la metalmecánica y sistemas reales de manufactura; sin embargo, también puede ser utilizado en la optimización de procesos de asignación, por ejemplo, para la simulación y optimización de sistemas de transporte. Dada la complejidad y dificultad para obtener la solución exacta, se han utilizado diversas heurísticas para obtener idóneas soluciones, entre los cuales se pueden resaltar la aplicación del Recocido Simulado (SA - Simulated Annealing), Búsqueda Tabú (TS - Tabu Search), Algoritmos Genéticos (GA - Genetic Algorithm), Colonia de Hormigas (ACO - Ant Colony Optimization), Redes Neuronales (NN - Neural Network), Algoritmos Evolutivos (EA - Evolutionary - Algorithm), entre otras más [1].

El problema se formula del siguiente modo [1]:

1. Se tiene un conjunto de n jobs que se deben procesar en m máquinas.

2. El conjunto de máquinas puede denotarse como $M=\{M_1, M_2...., M_m\}$

3. Cada job i consiste en una secuencia de n_i operaciones $0_{i1}, o_{i2},..., 0_{i,n}$

4. La ejecución de cada operación j de un job i (denotado como $O_{i,j}$ requiere ser realizada en una máquina del conjunto de máquinas posibles, denotada como $M_{i,j} \subseteq M$. Se denota como p,i,j,k al tiempo de procesamiento de $0_{i,j}$ en la máquina M_k.

El problema entonces consiste tanto en determinar la secuencia de las operaciones, como la asignación de cada operación a la máquina que mejor convenga, considerando los siguientes supuestos.

1. C_{max} = *Makespan,* el tiempo máximo en que se completa la última operación en la última máquina trabajando.

2. Cada job o trabajo, es independiente de cada uno de los otros trabajos.

3. Las máquinas son independientes entre sí.

4. Los tiempos de configuración de las máquinas son despreciables.

5. Los tiempos de movimiento entre las operaciones son despreciables.

6. No se puede procesar más de una operación del mismo trabajo al mismo tiempo, y se debe seguir una secuencia dada, es decir, precedencia entre operaciones del mismo trabajo.

7. No se tiene la restricción de precedencia entre las operaciones de diferentes trabajos o jobs.

8. En un tiempo dado, una máquina puede ejecutar cuando más una operación. La máquina se considera disponible, únicamente si la operación en proceso ha sido completada.

Para resolver el FJSSP, en este caso se utiliza un algoritmo genético. Los algoritmos evolutivos (EA) son la meta heurística de optimización con más influencia actualmente, de las cuales, los algoritmos genéticos son los más populares [3]. Los algoritmos genéticos son una estrategia de búsqueda para resolver problemas de optimización, que se caracterizan por ser métodos adaptativos. Parten del proceso genético de los organismos vivos, y como la interacción entre ellos y sus mutaciones, van cambiando sus características de generación a generación [4]. Es decir, se observa que, con el paso del tiempo, las generaciones de las especies presentes en la naturaleza evolucionan, ello acorde a los principios de la selección natural y la supervivencia de los más fuertes, postulados por Darwin en 1859. En un algoritmo genético, se busca imitar este proceso, de tal modo que se tiene la intención de ser capaz de crear soluciones para problemas del mundo real, mejorando las soluciones de generación a generación. En este punto, es importante resaltar que la aproximación de las soluciones de generación a generación hacia la solución óptima, depende de la codificación que se haga del algoritmo, y de las características de la propia solución [4]. Holland en 1975, estableció los principios básicos de los algoritmos genéticos, y a partir de tal año, se han estudiado y desarrollado muchas estrategias, que toman de base la evolución de las especies observada en la naturaleza. En el medio ambiente natural, las especies compiten entre sí en la búsqueda de su supervivencia y lo que ello implique como la obtención de recursos tales como espacio y alimento. Más aún, esta competencia se da también dentro de la comunidad de individuos de la misma especie. Algunos miembros de la población, tienen características particulares que le facilitan sobrevivir en tales condiciones de competencia, esas condiciones reflejan la capacidad de adaptación que heredaron de sus padres, que a su vez heredaron de sus ascendientes [4].

Los algoritmos genéticos modelan el proceso de evolución como una sucesión de frecuentes cambios en las soluciones candidatas. El espacio de soluciones posibles es explorado aplicando transformaciones a éstas soluciones candidatas tal y como se observa en los organismos vivientes [4].

Materiales y métodos

Instancias y su contenido.

Se utilizaron instancias provistas en [2] para su análisis y resolución. La Tabla 1 describe un ejemplo de una de ellas.

En la primera línea se tienen dos números. El primero es el número de trabajos (jobs) y el segundo es el número de máquinas disponibles.

3	**4**													
3	**1**	**1**	**2**	**2**	**2**	**4**	**3**	**5**	**1**	**4**	**4**			
4	**1**	**1**	**1**	**1**	**4**	**2**	**2**	**2**	**3**	**3**	**5**	**1**	**4**	**1**
3	**1**	**1**	**3**	**2**	**2**	**2**	**3**	**4**	**1**	**4**	**3**			

Tabla 1. Ejemplo de instancia en el FJSSP.

En algunas instancias se tiene un tercer número en este mismo renglón que se refiere al promedio de máquinas por operación, pero de existir dicho tercer número puede ser ignorado. A partir del segundo renglón y hasta el final de la instancia, cada uno de estos renglones representa un trabajo (job). El primer número es el número de operaciones de dicho trabajo, el segundo número, al que se le puede llamar *k* y siempre es mayor o igual a 1, representa el número de máquinas que pueden procesar dicha primera operación. De acuerdo con este último número siguen a continuación *k* pares de números que representan el número de máquina y el tiempo de procesamiento en dicha máquina para dicha operación [2].

Generación de población inicial.

Primeramente, se construye una matriz de equivalencia que identifica cada operación de cualquier otra en el universo de tareas a procesar de acuerdo a cada instancia. Dicha matriz tiene un renglón por cada trabajo, y cada renglón tiene tantos elementos con números consecutivos de las operaciones asociadas a cada trabajo. A continuación, se muestra en la Tabla 2 la matriz de equivalencia de la instancia de muestra previamente detallada.

	Índice de operaciones			
Job 1	0,	1,	2	
Job 2	3,	4,	5,	6
Job 3	7,	8,	9	

Tabla 2. Ejemplo de matriz de equivalencia.

Esta matriz es de suma importancia y utilidad ya que nos permite tanto generar secuencias factibles. En este caso, factibilidad se define como el hecho de que en una secuencia de operaciones nunca una primera operación que tenga precedencia sobre una segunda operación esté después de esta última [2].

Basado en dicho principio de factibilidad se programa una función que siempre genera secuencias de operaciones factibles de la siguiente forma:

1. Se elige aleatoriamente un renglón de la matriz de equivalencia.

2. Se toma el primer elemento de dicho renglón y se añade a la secuencia.

3. Se borra de dicho renglón el elemento tomado.

4. Se regresa al paso 1 hasta que todos los renglones de la matriz de equivalencia estén vacíos.

Con este proceso se generan una población inicial de *n* vectores de secuencias de operaciones factibles de tamaño *m* . Donde *n* es el tamaño de la población y *m* es la cantidad de operaciones en la secuencia. Un ejemplo de una secuencia completa de operaciones se puede apreciar en la Figura 1.

3	**0**	**1**	**4**	**7**	**8**	**9**	**2**	**5**	**6**

Figura 1. Ejemplo de vector de secuencia de operaciones factible.

Adicional, se construyen vectores de asignación de máquinas a las correspondientes operaciones de acuerdo a los datos de cada instancia. La Figura 2 presenta un vector de asignación de máquina basado en el vector ejemplo de la Figura 1 y sobre la instancia presentada en la Tabla 1. Igualmente, se genera una población de vectores de tamaño *m* .

4	**1**	**1**	**3**	**2**	**2**	**4**	**2**	**1**	**1**

Figura 2. Ejemplo de vector de asignación de máquinas factible.

Selección.

La selección de los individuos se hace mediante torneo binario, es decir, elegimos dos individuos de manera aleatoria, se elige el que mejor aptitud tenga, definiendo el mejor como el menor makespan entre ellos. Este individuo participa ahora para las siguientes etapas o procesos en el algoritmo.

Cruza.

El mecanismo se describe a continuación, y se detalla en la Figura 3, donde se muestran dos padres seleccionados previamente e identificados como (P1) y (P2), estos padres contienen secuencias de operaciones factibles. Se elige un punto de cruza al azar, en este caso la posición 3 indicada con flecha dentro de la Figura, del padre P1, se toman los elementos desde el inicio del vector, hasta el punto de cruza y se copian al descendiente H1, de igual forma hacemos esto para el padre P2 y su respectivo descendiente H2. El descendiente H1, se completa con los elementos del padre P2, para hacer esto, a partir del punto de cruza, se busca cada elemento en la parte copiada del descendiente H1, es decir, la operación 8 no está en la secuencia 4-1-5, por lo tanto, se agrega, la 6 tampoco se encuentra en la secuencia 4-1-5, se adiciona, el 7, no se encuentra en la secuencia 4-1-5, se incluye, y así sucesivamente hasta completar los elementos del descendiente H1, las operaciones que ya se encuentren en la secuencia 4-1-5, no se agregan. El mismo procedimiento se realiza para el descendiente H2, ahora llenándolo a partir del padre P1, es decir, la

operación 6 no se encuentra en la secuencia 4-5-1, se agrega, la 8 tampoco se encuentra en la secuencia 4-5-1, se incluye, y así para todos los elementos restantes.

Cabe mencionar, que cada padre tiene una secuencia de máquinas donde es posible realizar el trabajo, aplica exactamente el mismo procedimiento de cruza que se acaba de ilustrar. Se verifica que los vectores obtenidos sean factibles, de no serlo, se reparan recolocando operaciones en el orden que respeten precedencia.

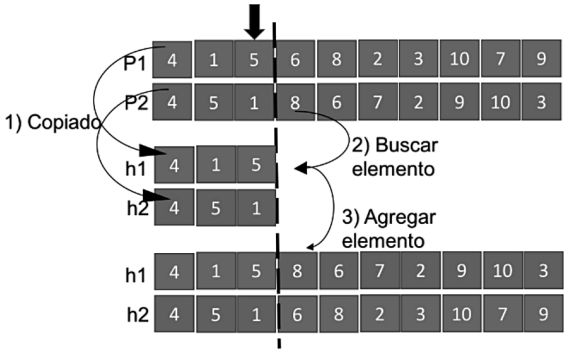

Figura 3. Ejemplo de operador de cruza.

Mutación.

El procedimiento es el siguiente, se eligen dos operaciones al azar, para este ejemplo ilustrativo las operaciones 4 y 5, que se detallan en la Figura 4, y se intercambian sus posiciones si y solo si no pertenecen al mismo trabajo o job y si no violan precedencia entre las operaciones de los trabajos involucrados. El resultado se visualiza en la Figura 5.

Figura 4. Ejemplo de operador de mutación.

Figura 5. Nuevo individuo.

Reemplazo y elitismo.

Para el proceso de reemplazo se tienen 2 métodos: aleatoriamente y eligiendo a los mejores de cada generación. En el método aleatorio, simplemente se eligen 2 secuencias, una de las secuencias padres y otra de los descendientes, se evalúa su aptitud y se opta por la mejor, o la secuencia padre en caso de empate. Por el segundo método, se ordenan las listas de padres e hijos por aptitud y se hace el reemplazo con el mejor 50% de la lista de padres y el mejor 50% de la lista de hijos.

Parámetros.

Los parámetros utilizados en la ejecución del algoritmo genético son siguiendo las recomendaciones de [5], los cuales consisten en:
Individuos = 200
Generaciones = 100
pc = 0,90 Probabilidad de cruza
pm = 0,10 Probabilidad de mutación
Ejecuciones o corridas = 30
Este último parámetro no es sugerido por [5], se establece para tener una muestra estadísticamente válida.

Resultados y discusión

Para observar la convergencia del algoritmo genético se ilustra una de las instancias, es decir, la instancia 'mt10x' en la Figura 6. En el eje horizontal se presentan las generaciones, y en el eje vertical corresponde a la aptitud, es decir, el makespan. El promedio de la aptitud por generación marcada en línea continua azul, y la solución incumbente ilustrada en línea roja.

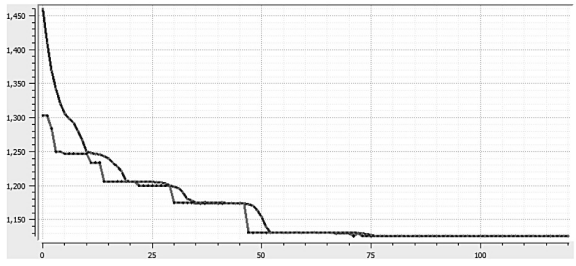

Figura 6. Gráfica de resultados de instancia mt10x

Con excepción de las relativamente simples primeras dos instancias que convergen alrededor de la generación 25, la gran mayoría de las instancias convergen alrededor de la generación 75.

El tiempo de ejecución del algoritmo oscila entre 400 segundos y 2500 segundos dependiendo de la instancia a resolver. Es importante mencionar que el tiempo mencionado es por las 30 corridas que se ejecuta el algoritmo, por lo que en el caso de la instancia que más tiempo demora, es decir, la instancia 'seti5x', se requiere poco más de 83 segundos por cada ejecución. En consecuencia, la velocidad obtenida es resultado de que el algoritmo fue programado en C++, lenguaje reconocido por ser robusto, y de rápida ejecución.

Para el caso de la estabilidad del algoritmo, la mejor solución encontrada es la más frecuente de presentarse en cada generación, lo que no es sorpresa tratando de algoritmos genéticos en donde se puede comenzar a explorar una nueva parte del espacio de solución de manera aleatoria. Esto habla de la robustez del algoritmo

que tiende a dar los mismos resultados la mayoría de las veces, aunque estos resultados sean óptimos locales dados las restricciones de tiempo computacional.

Conclusiones

Debido a la elevada dificultad de las instancias, con un gran número de máquinas y trabajos, es necesario una mayor exploración de diferentes parámetros como las tasas de mutación y cruza, así como diferentes combinaciones de métodos de reemplazo, selección, cruza y mutación. Aunque se puede decir que el método converge con relativa rapidez en cuanto al número de generaciones, dicha implementación converge hacia óptimos locales y difícilmente logra avanzar hacia otro lugar del espacio de búsqueda. Así mismo, la experimentación con las instancias del conjunto de [2] indica que una mayor población y mayor número de generaciones, combinado con valores diferentes en la tasa de mutación permiten encontrar soluciones con una mejor aptitud.

Estudios previos consideran dentro de la evaluación de las soluciones, tiempos de ajuste conocidos como tiempos set up. Estos tiempos se presentan a menudo cuando los equipos utilizados requieren ajustes específicos para así realizar las tareas u operaciones que siguen en la secuencia. Los resultados presentados en este documento no consideran tiempos de ajuste por lo que, como parte del trabajo futuro, se podría integrar más condiciones relevantes de operación en la propuesta de solución, así como mejorar el algoritmo con una siguiente fase de mejora local, por ejemplo, implementando una búsqueda tabú, para realizar búsquedas locales sobre las soluciones obtenidas, con la intención de mejorar aún más los resultados, evitando así el estancamiento en óptimos locales.

Agradecimientos

Los autores agradecen las sugerencias de todas las personas que revisaron en numerosas ocasiones este reporte de proyecto.

Referencias

[1] Xing, L.N., Chen, Y.W., & Yang, K.W. (2008). Double layer ACO algorithm for the multi-objective FJSSP. *New Generation Computing*, 26(4), 313-327.

[2] Mastrolilli, M. (2002). Flexible Job Shop Problem. *IDSIA Dalle Molle Institute for Artificial Intelligence Research*. Recuperado de<http://people.idsia.ch/˜monaldo/fjsp.html#ProblemInstances>

[3] Du, K.L., & Swamy, M.N.S. (2016). *Search and optimization by metaheuristics* (p. 434). New York City: Springer.

[4] Grefenstette, J.J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Trans*. Syst. Man Cybern. 16(1), 122-128.

[5] Gonçalves, J.F., & Resende, M.G. (2015). A biased random-key genetic algorithm for the unequal area facility layout problem. *European Journal of Operational Research*, 246(1), 86-107.

Recibido: 28 de marzo de 2018

Aceptado: 24 de agosto de 2018