Desarrollo de un algoritmo genético como solución al problema programación de producción

Juan Sebastián Herrera Cobo

Estudiante Maestría en Ingeniería

Octubre 2019

1. Introducción

La programación de producción es uno de los problemas más importantes para las grandes industrias de nuestros tiempos ya que, una solución ineficiente puede afectar directamente la capacidad de producción de las empresas y, por lo tanto, la rentabilidad de las mismas. Con esto, las metodologías de solución al problema han ido evolucionando con el tiempo.

Desde la optimización, se puede abordar el tema desde la búsqueda de soluciones exactas o aproximadas. Sin embargo, al ser la programación de producción un problema tipo NP-hard (Goncalves, Mendes y Resende, 2005), es recomendable proponer metodologías con soluciones aproximadas, como las heurísticas y metaheurísticas, dado que estás tienen aplicaciones en problemas de mediano y gran tamaño. Una de las metodologías más utilizadas para este tipo de problema es el algoritmo genético (GA).

Lo que busca el algoritmo genético es simular el crecimiento de una población de seres vivos a través de la genética con sus diferentes factores: una población inicial, un mecanismo para el apareamiento, mutaciones y la sobrevivencia de los individuos más fuertes para pasar de una generación a otra. Entonces, durante el proceso se genera una población inicial de soluciones que, con el tiempo, se combinan unas a otras para generar nuevas soluciones (apareamiento), se modifican según una probabilidad y una función de mutación, y se seleccionan los mejores individuos para que sobrevivan hasta la próxima generación.

Por esto, en el siguiente trabajo se presenta la implementación de un algoritmo genético a un problema de programación de producción tipo *Job Shop* donde se tienen M número de máquinas y J número de trabajos, los cuales están constituidos por n_i operaciones que se deben hacer en un orden predeterminado y donde cada operación O_{ij} puede realizarse solamente en un subconjunto de máquinas M_{ij}. Dentro de la implementación se presenta una estrategia para seleccionar los diferentes parámetros (tamaño de población inicial, número de generaciones y probabilidad de mutación) y luego se validad el rendimiento del algoritmo con diferentes problemas generados aleatoriamente.

2. Diseño del algoritmo

Para el diseño del algoritmo se deben de establecer la estructura del individuo, la función para crear un individuo aleatorio, la función *fitness* (la cual se encarga de medir la afinidad de un individuo frente a toda la población), la función *crossover* y la función de mutación.

2.1.Estructura del individuo

La mejor manera de ver una programación de producción es tener un cronograma por operación donde se consigne el tiempo de inicio de la operación S_{ij} y la máquina X_{ij} (Bierwirth y Mattfeld, 1999). Por lo tanto, teniendo *n* número de trabajos y *m* número de máquinas, y para hacer más fácil la implementación, el individuo se representa con una lista de $2*\sum n_i$ elementos (suma de todas las operaciones por cada trabajo) donde, por cada operación se tiene una variable entera S_{ij} y una variable entera X_{ij}, la cual se mueve entre 1 y *m*:

$$I = \left[S_{11}, X_{11}, S_{12}, X_{12}, \dots, S_{nn_i}, X_{ij}\right]$$

Con esto, teniendo 2 trabajos, cada uno con dos operaciones (es decir, 4 operaciones en total) y 2 máquinas, un individuo *[0,1,2,1,4,1,6,2]* significa que la operación 1 del trabajo 1 se comienza a realizar en el tiempo 0 en la máquina 1, la operación 2 del trabajo 1 se comienza en tiempo 2 en la máquina 1, la operación 1 del trabajo 2 se comienza en el tiempo 4 en la máquina 1 y la operación 2 del trabajo 2 se comienza en el tiempo 6 en la máquina 2.

2.2.Función para crear individuo aleatorio

Para tener una población inicial diversa, se generan soluciones factibles e infactibles a través de tres metodologías: 1. Factible desde adelante, 2. Factible desde atrás y 3. Aleatorio (infactible).

Factible desde adelante: se comienza a generar una solución desde la primera operación en el dataset (operación 1 trabajo 1) hasta la última, comenzando desde el tiempo cero. Para seleccionar la máquina de cada operación, se genera un número aleatorio entre las máquinas elegibles M_{ij}

Factible desde atrás: es igual que la anterior, sólo que no se comienza desde la primera operación sino desde la última.

Aleatorio: se generan tiempos de inicio aleatorios para la variable *S* entre 0 y el tiempo de procesamiento máximo de todas las operaciones, y se generan números aleatorios entre 1 y el número de máquinas para todas las operaciones del dataset.

En esta implementación se utilizó 40% de la población inicial con factible desde adelante, 30% con factible desde atrás y 30% con aleatorio.

2.3.Función *fitness*

Para el cálculo de la función *fitness* se recorre el individuo tres veces: la primera para calcular el C_{max} basado en todas las variables *S*, la segunda para castigar el *fitness* con el número de trabajos donde las operaciones no se hacen el orden predeterminado y la tercera para castigar el *fitness* con las operaciones que se cruzan en un tiempo en la misma máquina. A continuación, se muestra el pseudocódigo:

```
Inicio
I = individuo
P = matriz de tiempos de procesamiento
Fitness=0
s=0
while s < len(I):
  if I[s] + P[s]:
```

```
    Fitness = Fitness+I[s]+P[s]
  s+=2
Fouls=0
for job in jobs:
  for operation in jobs[job]:
        if I[operation] > I[operation+1]:
         fouls+=4
for machine in machines:
operations=operations_in_machine(machine)
  for op in operations:
    if I[op] == 0:
      count_zeros+=1
    start_reference=I[op]
    end_reference=I[op]+P[op]
    for op2 in operations:
    start=I[op2]
    end=I[op2]+P[op2]
      if start<=start_reference and
end>=end_reference:
        fouls+=2
      if start>=start_reference and
start<=end_reference and end<=end_reference:
        fouls+=2
      if start<=start_reference and
end>start_reference and end<=end_reference:
        fouls+=2
      if start>=start_reference and
start<end_reference and end>=end_reference:
        fouls+=2
Fitness=Fitness+(1000*fouls)
return Fitness
```

2.4.Función de mutación

Para la mutación se generan dos números aleatorios entre 0 y la longitud del individuo que representan los índices para intercambiar sus valores dentro el individuo. Si los dos números son pares (variables *S*) o los dos números son impares (variables *X*) se intercambian entre ellos. Si uno sale impar y otro par, se genera un número aleatorio para la variable *X* entre 1 y el número de máquinas, y para la variable *S* se encuentra otra variable *S* aleatoriamente en el individuo y se intercambia con ese.

2.5.Selección de parámetros

Luego de varios experimentos con un problema 4x2 (4 operaciones totales y dos máquinas), uno 6x2, uno 7x3, uno 9x5 y uno 11x6, se determina un tamaño inicial de 200 individuos y 500 generaciones, con el cual se encuentra una solución factible en todos los casos. Sin embargo, cuando no se encuentra una solución factible al final de las 500 generaciones, se aumenta en 100 las generaciones y se vuelve a correr el algoritmo. Este proceso tiene un tope de 10 aumentos de generaciones.

Por otra parte, para la probabilidad de mutación se tomó un valor del 30% teniendo en cuenta que la población inicial tiene una proporción grande de individuos infactibles y que la mutación ayudará que estos individuos mejoren el *fitness* de los demás.

3. Implementación del algoritmo

Para la implementación del algoritmo se utilizó Python 3.6.5 y la librería *pyeasyga* (https://pypi.org/project/pyeasyga/) la cual tiene la facilidad de darle como entrada la función para creación de individuo, la función de mutación, la función de *fitness* y demás parámetros de los algoritmos genéticos y realiza todo el proceso hasta finalizar las generaciones.

Las demás anotaciones sobre la implementación del algoritmo se pueden encontrar en el código.

4. Calidad de las soluciones

Al tener como criterio de parada de este algoritmo la factibilidad del mejor individuo de la última generación y el número de generaciones del parámetro de entrada, todas las soluciones encontradas son factibles y sirven como solución aplicable a los problemas de entrada.

Para la validación del algoritmo se utilizaron los mismos problemas con los que establecieron los parámetros y para cada uno de ellos se tomó el número de generaciones y el tiempo, en segundos, de ejecución. Cabe resaltar que, al tratarse de un proceso aleatorio, se pueden producir diferentes soluciones para un mismo problema, por lo que se corrió el algoritmo 3 veces por problema y se extrajo la mejor solución:

Problema	**Generaciones**	**Tiempo (s)**
4x2	500	13,66
6x2	500	16,33
7x3	500	15,87
9x5	500	21,23
11x6	500	27,76

Se puede observar que, para la mejor solución, no se necesitó aumentar el número de generaciones y los tiempos de ejecución están por debajo de los 60 segundos, lo que, teniendo en cuenta que el problema de programación de producción es táctico, estos tiempos son aplicables a una línea de producción real.

Comparándolo con algoritmos híbridos de gran rendimiento como el de Goncalvez, Mendes y Resend (2005), se tiene que el algoritmo desarrollado tiene un tiempo de ejecución de 27,76 segundos para un problema 11x6 y el algoritmo con el que se compara solucionó un problema 10x5 en 32 segundos, lo que, aunque no se puede comparar directamente dado que son problemas diferentes, sí da un buen panorama en cuanto a su eficiencia en relación con la dimensión del problema.

5. Conclusiones

En el presente trabajo se implementó un algoritmo genético a un problema de programación de producción, específicamente un *Flexible Job Shop*. Durante la implementación se utilizó una estructura del individuo basada en el tiempo de inicio de la operación y la máquina, una población inicial diversa donde se tiene individuos factible e infactibles, y una función *fitness* que minimiza el C_{max} *(makespan)* y castiga el no cumplimiento de restricciones aumentando el valor del mismo.

Luego de su implementación en Python, se validó el algoritmo con 5 problemas diferentes de diferente dimensión operaciones-máquinas. Esto dio como resultado un buen rendimiento con tiempos de solución menores a 60 segundos para el problema 11x6. Sin embargo, se debe explorar el rendimiento del algoritmo con problemas de mayor dimensión y, si es posible, compararlo con implementaciones conocidas en la literatura.

Por otra parte, con la realización de este trabajo, se puede concluir que las partes más críticas de un proceso de implementación de un algoritmo genético son el diseño de la estructura del individuo y la función *fitness* ya que, siempre se debe buscar que la estructura sea sencilla y entendible pero que no requiera de muchos cálculos para el *fitness* porque, de lo contrario, afectaría los tiempos de ejecución. Además, una población inicial diversa puede traer mejores resultados.

6. Referencias

Bierwirth, C., & Mattfeld, D. C. (1999). Production scheduling and rescheduling with genetic algorithms. Evolutionary computation, 7(1), 1-17.

Carrasco, F. (2016) Trabajo de grado: Modelado y Resolución de Problemas de Secuencia de Tareas. Aplicación a una Empresa de Fabricación de electrodos. Sevilla, España.

Gonçalves, J. F., de Magalhães Mendes, J. J., & Resende, M. G. (2005). A hybrid genetic algorithm for the job shop scheduling problem. European journal of operational research, 167(1), 77-95.

Márquez, José, (2012) Tesis doctoral: Optimización de la Programación (Scheduling) en Talleres de Mecanizado. Universidad Politécnica de Madrid. Madrid, España.

Murata, T., & Ishibuchi, H. (1994, June). Performance evaluation of genetic algorithms for flowshop scheduling problems. In Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence (pp. 812-817). IEEE.

Pezzella, F., Morganti, G., & Ciaschetti, G. (2008). A genetic algorithm for the flexible job-shop scheduling problem. Computers & Operations Research, 35(10), 3202-3212.

Pyeasyga: https://github.com/remiomosowon/pyeasyga/blob/develop/pyeasyga/pyeasyga.py