**Solución del Job-Shop Scheduling Problem (JSSP) con algoritmos genéticos**

Integrantes:

- Quispe Quilhue, Edgar Antonio 15200143

- De la Cruz Vilchez, Harold Alexis 14200014

- Boada Cajo, Edinson Paolo 17200140

- Alderete Cruzatt, Miguel Humberto 17200131

**Introducción:**

El problema de programación del taller (JSP) es uno de los problemas extremadamente difíciles porque requiere un espacio de búsqueda combinatorio muy grande y la restricción de precedencia entre máquinas. El algoritmo tradicional utilizado para resolver el problema es el método de Branch and bound, que requiere un tiempo de cálculo considerable cuando el tamaño del problema es grande. Proponemos un nuevo método para resolver JSP mediante el algoritmo genético (GA). El diseño del cromosoma representa una solución factible y cumple con todas las restricciones. Se utilizó el mecanismo de selección por torneo, una reproducción basada en emparejamiento parcial con dos puntos de cruzamiento y una mutación que intercambia dos genes aleatorios del cromosoma.

**Definición del problema:**

El Problema de Programación de la Tienda de Trabajo, consiste básicamente en planificar un conjunto de N trabajos {J1,.., JN} sobre un conjunto de M recursos o máquinas físicas {R1,…, RM}. Cada trabajo Ji consta de un conjunto de operaciones o tareas {θi1,…,θiM} que deben ser ejecutadas de forma secuencial. Cada operación θij tiene asociado un tiempo de procesamiento sin interrupción de Tij unidades de tiempo durante el cual requiere del uso exclusivo de un único recurso. Cabe considerar que las operaciones de un mismo trabajo deben ejecutarse en un orden determinado, de forma que θi(j+1) no puede comenzar hasta que θij haya terminado completamente. Además, las operaciones que comparten una misma máquina son no interrumpibles y mutuamente exclusivas. Las restricciones del problema se pueden resumir de la siguiente manera:

La JSP tiene N trabajos para procesar en M máquinas y asumimos lo siguiente:

* Ninguna máquina puede procesar más de un trabajo a la vez.
* Ningún trabajo puede ser procesado por más de una máquina a la vez.
* La secuencia de máquinas en las que una visita de trabajo está completamente especificada y tiene una estructura de precedencia lineal.
* Se conocen los tiempos de procesamiento;
* Todos los trabajos deben procesarse en cada máquina una sola vez.

Subrayamos a continuación los elementos esenciales para comprender el problema de job shop scheduling.

* Máquina (machine). Es cada uno de los entes que realizan alguna tarea. Disponemos de un grupo de ellas. Suponemos que una máquina cualquiera no puede realizar más de una única operación a la vez.
* Operación (operation). Es cada uno de los pasos necesarios para lograr un producto terminado. Una operación consiste en aplicar una máquina determinada durante un período determinado de tiempo ininterrumpido.
* Trabajo (job). Establece el objetivo de lograr un producto en particular. Y por comprensión, indica el conjunto de operaciones para lograr dicha meta.
* Programa (schedule). Es una asignación que fija a cada operación un intervalo de tiempo para ser efectuada.
* Intervalo operativo (makespan). Dado un programa, es el período o lapso que abarca la totalidad de la producción, es decir, la realización de todas las operaciones.
* Relación de precedencia entre operaciones de un trabajo. Indica el orden en que deben ser efectuadas las operaciones correspondientes a un mismo trabajo. Este orden queda determinado por el problema.
* Relación de precedencia entre máquinas. Indica el orden en que deben ser efectuadas las operaciones correspondientes a una misma máquina. Este orden puede ser elegido entre algunos admisibles.

**Objetivo del trabajo:**

Utilizar la metaheurística de algoritmos genéticos para resolver un difícil problema combinatorio de optimización.

**Importancia del tema:**

El Job Shop Scheduling es uno de los problemas más complejos de resolver debido al alto número de posibilidades que pueden ser consideradas en la secuenciación de pedidos. La secuenciación implica determinar el orden de entrada de los pedidos a las máquinas con el objetivo de conseguir los mejores desempeños en cierto tipo de variables, entre las cuales, el tiempo de procesamiento (makespan) ha sido uno de los más abordados. El problema de programación en ambientes job shop se considera del tipo NP-hard debido a que la cantidad de posibles soluciones son del orden de , donde n corresponde al número de pedidos y m al número de máquinas que conforman el sistema. La resolución del problema es importante para el sector de empresas del ramo industrial como también para muchos de otros ámbitos industriales, porque siempre la competencia exige una mejora continua en sus sistemas de producción y planificación.

**Representación del problema:**

La representación indirecta necesita un programa de interpretación para traducirlo a un horario válido. La ventaja de este esquema es la simplicidad de la estructura individual y los operadores, y el inconveniente es que el algoritmo genético está limitado a buscar en el espacio de toda posible permutación de genes individuales. La representación puede verse como una especie de lista de trabajos ampliada, que contiene N x M genes, donde N es el número de trabajos y M es el número de máquinas. Debido a que se supone que cada trabajo se procesa en cada máquina solo una vez, cada trabajo aparece en la lista de trabajos exactamente M veces. Como programa válido, el orden de procesamiento de los trabajos se da para cada máquina y el individuo indirecto se puede traducir en un programa válido de acuerdo con el orden de procesamiento predeterminado de las máquinas.

Dada M máquinas y J tareas.

Dado un vector de elementos que contiene los números 1,2,3,4,…,J exactamente M veces cada uno de ellos.

[v1, v2, v3, … , vM\*J]

Y cada elemento del vector se puede traducir en una operación de una tarea en una máquina, es decir, esta define una secuencia de operaciones a ejecutar.

**Definición de la función fitness:**

La función fitness calcula el makespan de la programación que define el cromosoma.

Sea Cj el tiempo de terminación de la tarea j.

Fitness(individual) = MAX(C1,…, CJ)

**Definición de la función de cruzamiento:**

Proponemos un horario parcial de intercambio cruzado. Primero, los horarios parciales se seleccionan al azar en los padres, respectivamente. El horario parcial se identifica con el mismo trabajo en la primera y última posición del horario parcial. Y luego intercambiamos los dos horarios parciales. Por lo general, los programas parciales tienen una longitud diferente (contiene el número diferente de genes). Las crías generadas después del intercambio pueden ser más grandes o cortas que la longitud determinada, es necesario eliminar genes excesivos o agregar genes necesarios para que se conviertan en las crías válidas.

Ejemplo:

Sea J = 3 y M = 3

P1 = [3, 1, 2, **2, 1**, 2, 3, 3, 1]

P2 = [1, 2, **1, 3, 3, 2, 2**, 3, 1]

O1 = [3, 1, 2, **1, 3, 3, 2, 2,** 2, 3, 3, 1]

O2 = [1, 2, **2, 1,** 3, 1]

El hijo O1 tiene la tarea 3,2 en exceso. Se eliminan las tareas 3,2 que estén fuera de la programación parcial **1, 3, 3, 2, 2** hasta que haya exactamente 3 instancias de cada una.

El hijo O2 tiene la tarea 3,2 sin el número de ocurrencias necesarias. Se agregarán las tareas 3,2 hasta que haya tantas instancias de estas como máquinas haya disponibles. La inserción debe hacerse fuera de la programación parcial **2, 1.**

En ambos caso, la inserción y eliminación, deben hacerse de manera aleatoria.

Pueden obtenerse los siguientes hijos:

O1 = [3, 1, 2, **1, 3, 3, 2, 2**, 1] O1 = [1, **1, 3, 3, 2, 2,** 2, 3, 1] O1 = [1, 2, **1, 3, 3, 2, 2**, 3, 1]

O2 = [1, 3, 3, 2, **2, 1,** 3, 1, 2] O2 = [3, 1, 2, 3, **2, 1,** 3, 1, 2] O2 = [1, 2, **2, 1,** 3, 1, 3, 3, 2]

Definición del operador de mutación:

Definimos un operador de mutación de la siguiente manera: generar aleatoriamente dos posiciones en la lista e intercambiar sus genes, y si los dos genes son el mismo, volver a intentar seleccionar nuevas posiciones.

Ejemplo

Sea J = 3 y M = 3

P1 = [3, **1**, 2, 2, 1, 2, **3**, 3, 1]

O1 = [3, **3**, 2, 2, 1, 2, **1**, 3, 1]

Definición del operador de selección:

Se usará el método de selección por torneo, cuyo tamaño puede ser variable. Una selección por torneo de tamaño n elige aleatoriamente n individuos de la población y luego elige el mejor de la muestra.

Generación de la población inicial:

Se generarán individuos de manera aleatoria.

Desarrollo en Python:

import random

import numpy as np

import copy

**class** **Chromosome**(object):

**def** \_\_init\_\_(self, genes):

self.genes = genes

self.fitness = 0

**def** \_\_repr\_\_(self):

**return** repr((self.fitness, self.genes))

**class** **JSSP**(object):

**def** \_\_init\_\_(self,J,M,times,ordering\_machines,

population\_size,generations,

crossover\_probability,mutation\_probability,

tournament\_size,elitism):

self.J = J

self.M = M

self.N = J\*M

self.times = times

self.positions = list(np.arange(0,self.N))

self.ordering\_machines = ordering\_machines

self.seed\_data = list(np.arange(1,self.J+1))

self.population\_size = population\_size

self.generations = generations

self.crossover\_probability = crossover\_probability

self.mutation\_probability = mutation\_probability

self.tournament\_size = tournament\_size

self.elitism = elitism

self.current\_generation = []

self.minimize\_fitness = **False**

**def** assignment\_machines(list\_jobs):

times = np.zeros(self.J,dtype=int)

list\_machines = np.zeros(self.N,dtype=int)

**for** j,job **in** enumerate(list\_jobs):

list\_machines[j] = self.ordering\_machines[job-1][times[job-1]]

times[job-1] += 1

**return** list\_machines

**def** fitness(individual):

list\_machines = assignment\_machines(individual)

free = np.zeros(self.M)

final = np.zeros(self.N)-1

present = 0

**while** -1 **in** final:

**for** j **in** self.positions:

job = individual[j]

**if** final[j] != -1:

**continue**

machine = list\_machines[j]

**if** present < free[machine-1]:

**continue**

ready = **True**

elapsed = 0

**for** i **in** range(0,j):

cond1 = individual[i]==job **and** (final[i]>present **or** final[i]==-1)

cond2 = list\_machines[i]==machine **and** final[i]==-1

**if** cond1 **or** cond2:

ready = **False**

**break**

**if** ready:

**for** i **in** range(j-1, -1, -1):

**if** individual[i] == job:

elapsed = final[i]

**break**

final[j] = max(free[machine-1], elapsed) + times[job-1][machine-1]

free[machine-1] = final[j]

**else**:

**continue**

i = 0

past = present

**while** free[i] <= present:

i += 1

present = free[i]

i += 1

**while** i < self.M:

**if** free[i]>past **and** free[i]<present:

present = free[i]

i += 1

**return** max(free)

**def** ordering\_jobs(self,list\_jobs, list\_machines):

times = np.zeros(self.M, dtype=int)

M\_J = np.zeros((self.M,self.J), dtype=int)

**for** m **in** self.positions:

temp = list\_machines[m] -1

M\_J[temp][times[temp]] = list\_jobs[m]

times[temp] +=1

**return** M\_J

**def** mutate(individual):

index\_1,index\_2 = random.sample(self.positions,2)

individual[index\_1], individual[index\_2] = individual[index\_2], individual[index\_1]

**def** delete\_job(list\_jobs,job,repeats,p1,p2):

posi = []

n = len(list\_jobs)

j = 0

**for** i **in** range(0,n):

**if** job == list\_jobs[i] **and** (i<p1 **or** i>p2 ) :

posi.append(i)

posi = random.sample(posi,repeats)

save\_p1 = p1

**for** i **in** posi:

**if** i < save\_p1:

p1 -= 1

p2 -= 1

list\_jobs[i] = -1

**for** \_ **in** range(0,repeats):

list\_jobs.remove(-1)

**return** p1,p2

**def** delete(list\_jobs,p1,p2):

**for** i **in** self.seed\_data:

repeats = 0

**for** job **in** list\_jobs:

**if** job == i:

repeats += 1

repeats = repeats - self.M

**if** repeats > 0:

p1,p2 = delete\_job(list\_jobs,i,repeats,p1,p2)

**return** p1,p2

**def** complete\_job(list\_jobs,job,repeats,p1,p2):

n = len(list\_jobs)

**for** \_ **in** range(0,repeats):

randi = random.randint(0,n)

**while** randi>p1 **and** randi<=p2:

randi = random.randint(0,n)

**if** randi <= p1:

p1 += 1

p2 += 1

list\_jobs.insert(randi,job)

n += 1

**return** p1,p2

**def** complete(list\_jobs,p1,p2):

**for** i **in** self.seed\_data:

repeats = 0

**for** job **in** list\_jobs:

**if** job == i:

repeats += 1

repeats = self.M - repeats

**if** repeats > 0:

p1,p2 = complete\_job(list\_jobs,i,repeats,p1,p2)

**return** p1,p2

**def** crossover(parent\_1,parent\_2):

index\_1 = random.sample(self.positions,2)

index\_2 = random.sample(self.positions,2)

index\_1.sort()

index\_2.sort()

cross\_11,cross\_12 = index\_1

cross\_21,cross\_22 = index\_2

child\_1a = parent\_1[:cross\_11]

child\_1b = parent\_2[cross\_21:cross\_22+1]

child\_1c = parent\_1[cross\_12+1:]

child\_2a = parent\_2[:cross\_21]

child\_2b = parent\_1[cross\_11:cross\_12+1]

child\_2c = parent\_2[cross\_22+1:]

child\_1 = child\_1a + child\_1b + child\_1c

child\_2 = child\_2a + child\_2b + child\_2c

jump\_1 = cross\_22 - cross\_21

jump\_2 = cross\_12 - cross\_11

cross\_11 = len(child\_1a)

cross\_12 = cross\_11 + jump\_1

cross\_21 = len(child\_2a)

cross\_22 = cross\_21 + jump\_2

cross\_11,cross\_12 = delete(child\_1,cross\_11,cross\_12)

cross\_11,cross\_12 = complete(child\_1,cross\_11,cross\_12)

cross\_21,cross\_22 = delete(child\_2,cross\_21,cross\_22)

cross\_21,cross\_22 = complete(child\_2,cross\_21,cross\_22)

**return** child\_1, child\_2

**def** selection(population):

members = random.sample(population,self.tournament\_size)

members.sort(key=**lambda** x: x.fitness,reverse=self.minimize\_fitness)

**return** members[0]

**def** create\_individual(data):

individual = self.positions[:]

pos = self.positions[:]

**for** job **in** data:

in\_machine = random.sample(pos,self.M)

**for** machine **in** in\_machine:

individual[machine] = job

pos.remove(machine)

**return** individual

self.fitness = fitness

self.selection = selection

self.create\_individual = create\_individual

self.mutate = mutate

self.crossover = crossover

**def** create\_initial\_population(self):

initial\_population = []

**for** \_ **in** range(self.population\_size):

genes = self.create\_individual(self.seed\_data)

individual = Chromosome(genes)

initial\_population.append(individual)

self.current\_generation = initial\_population

**def** calculate\_population\_fitness(self):

**for** individual **in** self.current\_generation:

individual.fitness = self.fitness(individual.genes)

**def** rank\_population(self):

self.current\_generation.sort(key=**lambda** x: x.fitness,

reverse=self.minimize\_fitness)

**def** create\_new\_population(self):

new\_population = []

elite = copy.deepcopy(self.current\_generation[0])

**while** len(new\_population) < self.population\_size:

parent\_1 = copy.deepcopy(self.selection(self.current\_generation))

parent\_2 = copy.deepcopy(self.selection(self.current\_generation))

child\_1, child\_2 = parent\_1, parent\_2

child\_1.fitness, child\_2.fitness = 0, 0

can\_crossover = random.random() < self.crossover\_probability

can\_mutate = random.random() < self.mutation\_probability

**if** can\_crossover:

child\_1.genes, child\_2.genes = self.crossover(parent\_1.genes,

parent\_2.genes)

**if** can\_mutate:

self.mutate(child\_1.genes)

self.mutate(child\_2.genes)

new\_population.append(child\_1)

**if** len(new\_population) < self.population\_size:

new\_population.append(child\_2)

**if** self.elitism:

new\_population[0] = elite

self.current\_generation = new\_population

**def** create\_first\_generation(self):

self.create\_initial\_population()

self.calculate\_population\_fitness()

self.rank\_population()

**def** best\_individual(self):

best = self.current\_generation[0]

**return** (best.fitness, best.genes)

**def** create\_next\_generation(self):

self.create\_new\_population()

self.calculate\_population\_fitness()

self.rank\_population()

**def** run(self):

self.create\_first\_generation()

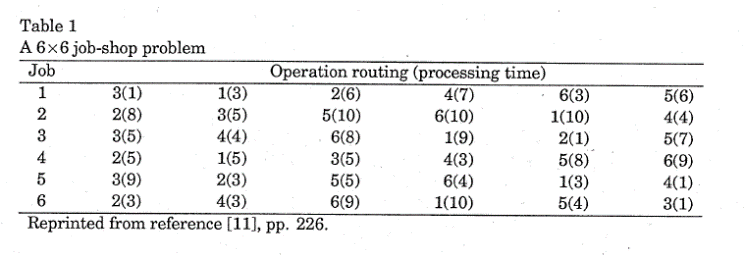
**for** \_ **in** range(1, generations):

self.create\_next\_generation()

**def** last\_generation(self):

**return** ((member.fitness, member.genes) **for** member **in** self.current\_generation)

Ejecuciones:



Parámetros:

* population\_size = 50
* generations = 50
* crossover\_probability = 0.8
* mutation\_probability = 0.2
* tournament\_size = 4
* elitism = True

Resultados:

(58.0, [6, 1, 3, 2, 3, 2, 6, 1, 3, 4, 4, 1, 5, 2, 5, 1, 6, 3, 4, 5, 1, 2, 4, 4, 6, 6, 3, 5, 3, 2, 6, 1, 5, 2, 5, 4])

(58.0, [3, 6, 6, 2, 1, 6, 2, 5, 4, 3, 5, 1, 4, 3, 4, 2, 4, 2, 6, 5, 1, 3, 1, 4, 6, 1, 3, 6, 2, 5, 4, 5, 2, 3, 5, 1])

(58.0, [2, 1, 6, 4, 1, 6, 5, 2, 4, 6, 6, 3, 3, 5, 4, 3, 3, 1, 1, 2, 2, 5, 5, 4, 6, 3, 1, 2, 6, 2, 5, 5, 4, 4, 3, 1])

(59.0, [3, 6, 4, 1, 2, 5, 5, 2, 1, 6, 4, 3, 5, 2, 4, 6, 3, 5, 6, 4, 1, 2, 4, 6, 3, 1, 6, 1, 5, 2, 2, 3, 4, 5, 3, 1])

(55.0, [3, 1, 2, 4, 3, 2, 3, 5, 2, 1, 4, 6, 5, 6, 3, 6, 4, 1, 6, 4, 3, 2, 5, 2, 5, 1, 4, 6, 2, 3, 5, 5, 1, 6, 4, 1])

(58.0, [2, 1, 3, 4, 6, 2, 3, 1, 5, 4, 3, 2, 1, 4, 5, 3, 2, 4, 3, 6, 1, 5, 6, 2, 3, 6, 1, 4, 1, 6, 6, 5, 4, 2, 5, 5])

(58.0, [3, 6, 2, 1, 2, 4, 6, 1, 5, 3, 2, 6, 1, 4, 4, 5, 1, 3, 5, 4, 6, 2, 5, 3, 1, 2, 3, 5, 2, 4, 3, 4, 6, 5, 1, 6])

(58.0, [6, 1, 3, 6, 1, 3, 2, 4, 2, 5, 2, 4, 3, 1, 6, 1, 5, 3, 4, 5, 3, 2, 6, 3, 4, 4, 1, 2, 5, 6, 4, 5, 1, 5, 2, 6])

(57.0, [3, 2, 3, 6, 4, 1, 1, 6, 2, 1, 2, 1, 3, 6, 5, 4, 4, 3, 5, 5, 4, 6, 1, 4, 2, 2, 3, 5, 5, 2, 6, 5, 3, 4, 1, 6])

(59.0, [1, 5, 6, 2, 5, 6, 4, 6, 1, 6, 3, 3, 4, 1, 5, 2, 2, 1, 5, 4, 4, 3, 1, 3, 5, 6, 5, 6, 4, 2, 2, 3, 1, 4, 3, 2])

* population\_size = 100
* generations = 100
* crossover\_probability = 0.8
* mutation\_probability = 0.2
* tournament\_size = 4
* elitism = True

(58.0, [4, 3, 2, 6, 1, 3, 6, 2, 3, 4, 1, 3, 2, 4, 6, 5, 1, 4, 6, 4, 5, 5, 3, 2, 2, 1, 5, 6, 3, 2, 1, 4, 1, 5, 6, 5])

(57.0, [3, 3, 1, 2, 6, 2, 3, 5, 4, 1, 4, 2, 5, 1, 6, 3, 4, 4, 6, 5, 2, 6, 4, 3, 6, 1, 5, 6, 1, 5, 1, 2, 3, 5, 4, 2])

(55.0, [1, 3, 2, 2, 3, 6, 6, 4, 1, 2, 5, 1, 4, 5, 4, 3, 6, 4, 2, 5, 4, 3, 3, 6, 1, 2, 1, 5, 5, 4, 2, 3, 5, 6, 1, 6])

(55.0, [2, 3, 4, 3, 6, 1, 2, 5, 3, 4, 6, 5, 2, 6, 5, 3, 1, 3, 4, 2, 1, 4, 6, 3, 2, 4, 6, 5, 1, 2, 5, 1, 1, 4, 5, 6])

(55.0, [2, 1, 3, 3, 3, 6, 6, 2, 5, 1, 4, 4, 6, 5, 3, 4, 6, 1, 2, 5, 3, 4, 1, 2, 5, 4, 3, 1, 2, 6, 5, 2, 4, 1, 5, 6])

(55.0, [1, 2, 3, 3, 1, 2, 4, 4, 5, 6, 3, 1, 4, 6, 4, 3, 2, 5, 5, 6, 1, 4, 2, 5, 6, 1, 2, 4, 3, 2, 6, 3, 5, 5, 6, 1])

(58.0, [5, 1, 2, 6, 6, 2, 6, 3, 4, 3, 2, 1, 4, 3, 1, 6, 5, 3, 1, 2, 2, 5, 4, 6, 1, 4, 4, 5, 5, 3, 1, 2, 4, 3, 5, 6])

(57.0, [1, 2, 3, 6, 3, 4, 1, 2, 1, 5, 4, 6, 2, 4, 5, 3, 3, 6, 5, 3, 2, 1, 6, 4, 5, 4, 6, 2, 3, 5, 2, 1, 6, 4, 1, 5])

(57.0, [2, 1, 4, 3, 6, 1, 2, 3, 3, 1, 4, 6, 1, 6, 5, 3, 2, 4, 2, 6, 5, 1, 4, 5, 4, 2, 6, 1, 5, 5, 3, 3, 4, 5, 6, 2])

(55.0, [3, 2, 4, 3, 1, 6, 2, 1, 5, 4, 2, 3, 6, 1, 4, 3, 1, 6, 5, 3, 4, 6, 5, 3, 2, 2, 4, 6, 1, 5, 2, 5, 4, 6, 1, 5])

* population\_size = 100
* generations = 300
* crossover\_probability = 0.8
* mutation\_probability = 0.2
* tournament\_size = 4
* elitism = True

(55.0, [2, 1, 4, 1, 3, 2, 5, 6, 3, 4, 6, 3, 4, 2, 6, 2, 5, 3, 5, 6, 5, 3, 1, 3, 4, 2, 5, 1, 4, 1, 6, 4, 2, 6, 5, 1])

(55.0, [2, 4, 3, 1, 2, 3, 4, 5, 3, 6, 4, 3, 2, 6, 6, 5, 1, 4, 2, 6, 5, 1, 2, 3, 4, 5, 1, 3, 5, 6, 6, 1, 4, 2, 5, 1])

(58.0, [6, 6, 3, 6, 1, 1, 3, 2, 2, 4, 5, 5, 2, 5, 4, 6, 3, 3, 2, 5, 5, 2, 4, 4, 1, 6, 4, 6, 1, 1, 3, 4, 1, 5, 3, 2])

(55.0, [2, 3, 4, 6, 4, 3, 1, 2, 3, 6, 6, 1, 5, 2, 3, 2, 5, 5, 4, 6, 1, 4, 3, 1, 2, 4, 3, 6, 6, 1, 5, 5, 1, 2, 4, 5])

(55.0, [2, 1, 3, 6, 4, 2, 3, 1, 2, 5, 6, 1, 5, 4, 3, 6, 2, 4, 4, 1, 5, 4, 5, 3, 3, 6, 1, 2, 4, 2, 5, 3, 6, 6, 1, 5])

(55.0, [1, 2, 3, 4, 1, 3, 6, 2, 2, 5, 4, 3, 4, 5, 1, 6, 6, 3, 2, 4, 5, 5, 1, 4, 6, 2, 3, 6, 3, 5, 1, 1, 2, 4, 5, 6])

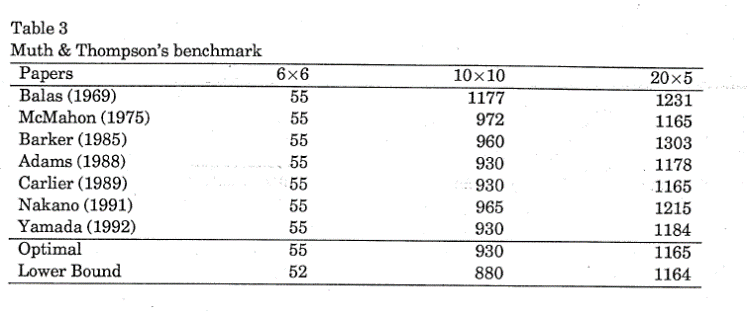
(55.0, [1, 1, 3, 2, 3, 4, 2, 5, 6, 4, 2, 4, 5, 5, 6, 4, 4, 3, 1, 6, 2, 3, 1, 6, 6, 5, 3, 2, 6, 3, 1, 5, 1, 2, 4, 5])

(55.0, [1, 2, 3, 2, 6, 1, 2, 4, 3, 3, 1, 5, 6, 4, 4, 3, 6, 4, 1, 5, 6, 3, 2, 1, 5, 4, 3, 2, 5, 5, 2, 6, 1, 6, 4, 5])

(55.0, [1, 3, 3, 1, 2, 2, 5, 4, 4, 6, 6, 3, 2, 5, 6, 3, 4, 3, 5, 2, 1, 4, 1, 5, 4, 6, 6, 3, 2, 1, 1, 6, 5, 2, 5, 4])

(55.0, [2, 1, 3, 3, 1, 2, 4, 6, 3, 5, 4, 5, 6, 6, 3, 6, 1, 2, 4, 5, 4, 4, 2, 1, 1, 6, 5, 4, 2, 3, 6, 3, 5, 1, 2, 5])

Resultados:



Conclusiones:

* La generación de individuos a partir del operador de cruzamiento y mutación no son complejos en codificación porque la representación indirecta solo debe cumplir una restricción para ser un individuo válido. La mayor dificultad está en la función de evaluación pues está tiene que hallar el makespan a partir de una jerarquía de operaciones, que es muy compleja en tiempo de ejecución y que retrasa mucho el algoritmo.
* El algoritmo genético puede resolver instancias del JSSP que plantean tareas que no se ejecutan en todas las máquinas calibrando correctamente los parámetros.
* Aumentar el tamaño de la población y el número de las generaciones hace más frecuente la obtención de las soluciones muy cercanas al óptimo.

Referencias:

M. Gen, Y. Tsujimura and E. Kubota, "Solving job-shop scheduling problems by genetic algorithm," Proceedings of IEEE International Conference on Systems, Man and Cybernetics, San Antonio, TX, USA, 1994, pp. 1577-1582 vol.2, doi: 10.1109/ICSMC.1994.400072.

Yamada, T. and R. Nakano: A Genetic Algorithm Applicable to Large-scale Job-Shop Problems, *Parallel Problem Solving from Nature* 2, Manner and Manderick(ed), pp. 281- 290, 1992.

Y. Li, D. Tang and Y. Chen, "Research on Agile Job-shop Scheduling Problem Based on Genetic Algorithm," 2009 Second International Symposium on Electronic Commerce and Security, Nanchang, 2009, pp. 590-593, doi: 10.1109/ISECS.2009.105.