Algoritmos Genéticos

Alfonso Fierro

Semana 3, Octubre 2023

Hibridación utilizada

El algoritmo genético implementado utiliza, como mecanismo de mutación, una búsqueda local con inserción hacia adelante:

1. Verificación de condición para mutación.
2. Generación del vecindario.
3. Exploración del vecindario.
4. Medición del makespan.
5. Cambio del elemento en la población.

Para lograr disminuir la carga de iteraciones planteada en las búsquedas locales anteriores se implementan las siguientes restricciones

Restricción de vecindarios

- Se escogen aleatoriamente las actividades a_{Tm} y se fabrican sus vecindarios con el esquema respectivo
- Se itera solo hasta encontrar una primer mejora en el *makespan* del problema.

Para lograr que esta restricción conserve la capacidad del algoritmo de explorar vecindarios ”alejados”, es necesario que se construyan ciertas características sobre la solución Inicial.

Pseudocódigo de Mutación

Algorithm 4 Mutación

```
1: function MUTACION(num_maquinas, num_trabajos, programa, grupos)
2:     c_programa ← copia_profunda(programa)
3:     makespan_ini ← constructivo.makespan(num_maquinas, num_trabajos, programa, grupos)
4:     indice_mutado ← aleatorio.choice(longitud(programa))
5:     cromosoma_mutado ← busqueda_local(c_programa, grupos, indice_mutado, makespan_ini)
6:     return cromosoma_mutado
7: end function
```

Disposición de la solución inicial

- Se debe evitar que actividades relacionadas a un mismo trabajo estén en posiciones contigüas o muy cercanas.
- La factibilidad debe preservarse cuando se realizan perturbaciones pequeñas a la solución.

	Estación 1	Estación 2	
Trabajo 1	$a_{1,1}$	$a_{1,2}$	$\cdots$
Trabajo 2	$a_{2,1}$	$a_{2,2}$	$\cdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

Table: Matriz de solución inicial

La solución inicial $S = [a_{1,1}, a_{1,2}, ...]$ se construye recorriendo aleatoriamente la matriz 5 por columnas, desde arriba hacia abajo y de izquierda a derecha.
§) El orden de los trabajos solo sigue un criterio aleatorio.
§) Las Estaciones son las máquinas respectivas que debe atravesar cada trabajo.

Población

Se inicializa la población con una *piscina* de elementos factibles, que son las columnas de la matriz anterior. Cada vez que se escoge una actividad para la solución inicial se debe agregar a la piscina, si existe, la actividad consecuente de el respectivo trabajo. Este esquema permite que, potencialmente, se pueda explorara cualquier solución factible.

Algorithm 1 Generar Individuo

```
1:  function GenerarIndividuo(grupos)
2:      actividades ← []                          ▷ Lista para almacenar las actividades del individuo
3:      grupos_c ← copia_profunda(grupos)         ▷ Realizar una copia profunda de los grupos
4:      piscina ← []                              ▷ Lista para las actividades disponibles
5:      for grupo ∈ grupos_c do
6:          act ← grupo.pop(0)                    ▷ Extraer la primera actividad de cada grupo
7:          piscina.append(act)                   ▷ Agregar la actividad a la piscina
8:      end for
9:      while piscina do
10:         indice ← aleatorio.choice(len(piscina))   ▷ Seleccionar un índice aleatorio en la piscina
11:         act_escogida ← piscina.pop(indice)        ▷ Eliminar la actividad seleccionada de la piscina
12:         grupo ← act_escogida[0]
13:         if grupos_c[grupo] then
14:             act_siguiente ← grupos_c[grupo].pop(0)   ▷ Extraer la siguiente actividad del mismo grupo
15:             piscina.append(act_siguiente)            ▷ Agregar la actividad siguiente a la piscina
16:         end if
17:         actividades.append(act_escogida)          ▷ Agregar la actividad escogida a la lista de actividades
18:     end while
19:     return actividades
20: end function
```

Torneo Binario

Dado que los individuos (soluciones) se generan aleatoriamente, cualquier orden en que estos se pongan a competir es también un orden aleatorio. En particular, el algoritmo de torneo binario termina por comparar individuo s al inicio y al final del *stack* de soluciones.

Pseudocódigo Torneo Binario

Algorithm 2 Torneo Binario

```
1:  function TorneoBinario(num_maquinas, num_trabajos, programas, grupos)
2:      tamano_inicial ← longitud(programas)
3:      indices_elite ← []
4:      indice ← 0
5:      while longitud(programas) − indice > tamano_inicial / 2 do
6:          makespan_inic ← constructivo.makespan(num_maquinas, num_trabajos, programas[indice], grupos)
7:          makespan_fin ← constructivo.makespan(num_maquinas, num_trabajos, programas[-1-indice], grupos)
8:          if makespan_inic < makespan_fin then
9:              indices_elite.append(indice)
10:         else
11:             indices_elite.append(-1-indice)
12:         end if
13:         indice ← indice + 1
14:     end while
15:     poblacion_elite ← []
16:     for elemento ∈ indices_elite do
17:         poblacion_elite.append(programas[elemento])
18:     end for
19:     return poblacion_elite
20: end function
```

El torneo Binario se utiliza para depurar las generaciones una vez se hayan añadido las poblaciones hijas. Para la fabricación de individuos cruzados se utiliza un esquema de dominancia parental, esto es que, con base a una proporción fija p, los *hijos* heredarán $p\%$ de los alelos de un padre y otro $(1-p)\%$ de su otro progenitor.

Hibridación

La hibridación para esta implementación requiere solo de 2 progenitores. Usando la probabilidad p anteriormente descrita, se sigue el siguiente esquema:

- Generar un número aleatorio r.
- Si $r < p$ entonces se toma el alelo del primer padre.
- Se borra ese alelo de todas las programaciones para no escogerlo nuevamente.
- Se verifica la precedencia de las actividades.

Pseudocódigo de Hibridación

Algorithm 3 Cruce

```
1:  function CRUCE(programa_1, programa_2, dominancia, grupo)
2:      c_prog1 ← copia_profunda(programa_1)
3:      c_prog2 ← copia_profunda(programa_2)
4:      numero_alelos ← longitud(programa_1)
5:      cromosoma_cruzado ← []
6:      while longitud(cromosoma_cruzado) < numero_alelos do
7:          suerte ← aleatorio.random()
8:          alel_dominante ← []
9:          if suerte < dominancia then
10:             alel_dominante ← c_prog1[0]
11:         else
12:             alel_dominante ← c_prog2[0]
13:         end if
14:         cromosoma_cruzado.append(alel_dominante)
15:         c_prog1.remove(alel_dominante)
16:         c_prog2.remove(alel_dominante)
17:     end while
18:     if verificaciones.verificar_presedencia(cromosoma_cruzado, grupo) then
19:         return cromosoma_cruzado
20:     end if
21: end function
```

Algoritmo Genético

EL algoritmo genético implementado sigue el esquema especificado en las presentaciones de clase. El flujo de las generaciones es el siguiente:

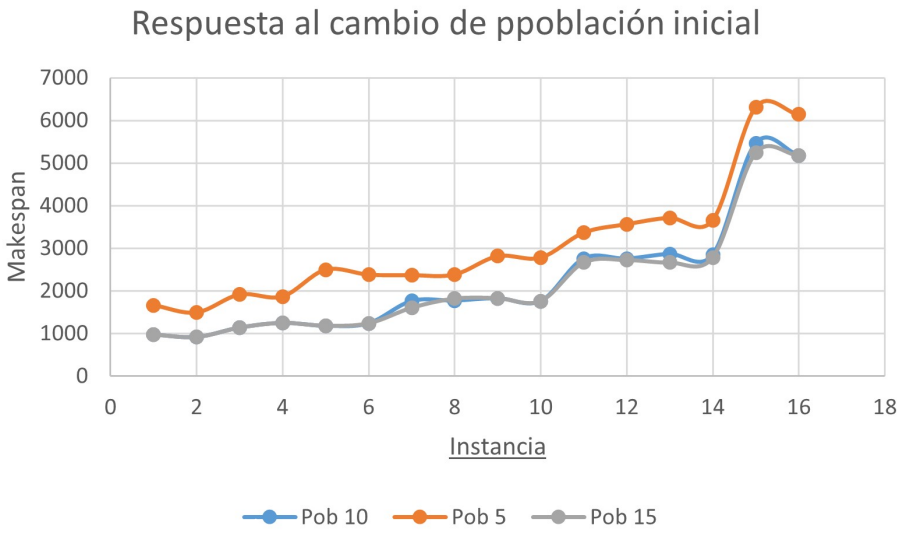
- Generar población.
- Generar población hija.
- Barrer sobre la población para mutar individuos.
- Realizar un torneo binario para determinar individuos sobrevivientes.
- Escoger mejor solución de cada generación.

Algorithm 5 Algoritmo Genético

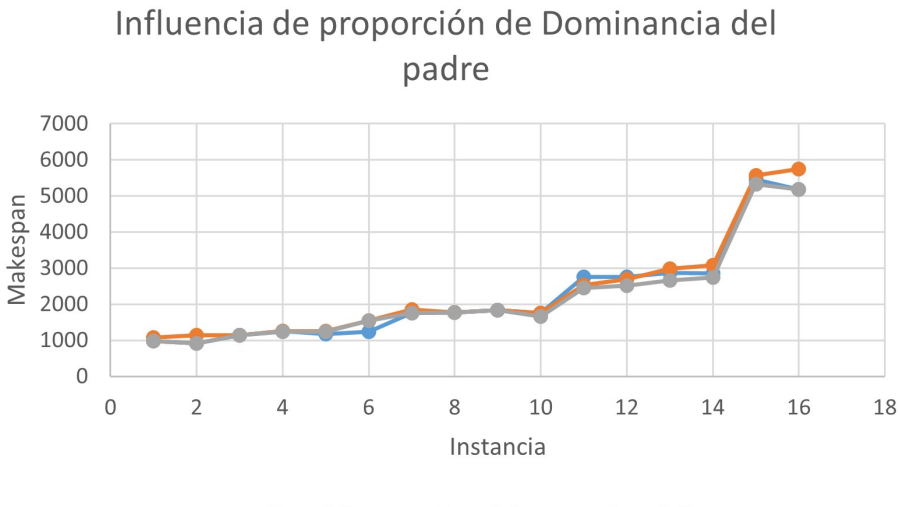
```
1:  function AlgoritmoGenetico(tamano_poblacion, generaciones, dominancia, probabilidad_mutacion)
2:      num_trabajos, num_maquinas, tiempos, secuencias ← lector.leer_txt(nombre_archivo)
3:      grups ← grupos.agrupamiento(num_trabajos, num_maquinas, secuencias, tiempos)
4:      poblacn ← poblacion.generar_poblacion(tamano_poblacion, grups)
5:      for generacion ∈ range(generaciones) do
6:          poblacion_hijos ← cruce.cruce_poblacional(poblacn, dominancia, grups)
7:          for hijo ∈ poblacion_hijos do
8:              dado ← aleatorio.random()
9:              if dado < probabilidad_mutacion then
10:                 hijo ← mutacion.mutacion(num_maquinas, num_trabajos, hijo, grups)
11:             end if
12:         end for
13:         poblacn ← torneo_binario.torneo_binario(num_maquinas, num_trabajos, poblacion_hijos, grups)
14:     end for
15:     mejor_solucion ← poblacn[0]
16:     makespan_mjr_sol ← constructivo.makespan(num_maquinas, num_trabajos, mejor_solucion, grups)
17:     for programa ∈ poblacn do
18:         makespan_programa ← constructivo.makespan(num_maquinas, num_trabajos, programa, grups)
19:         if makespan_programa ≤ makespan_mjr_sol then
20:             mejor_solucion ← copia_profunda(programa)
21:             makespan_mjr_sol ← makespan_programa
22:         end if
23:     end for
24:     return mejor_solucion
25: end function
```

Comparación de parámetros

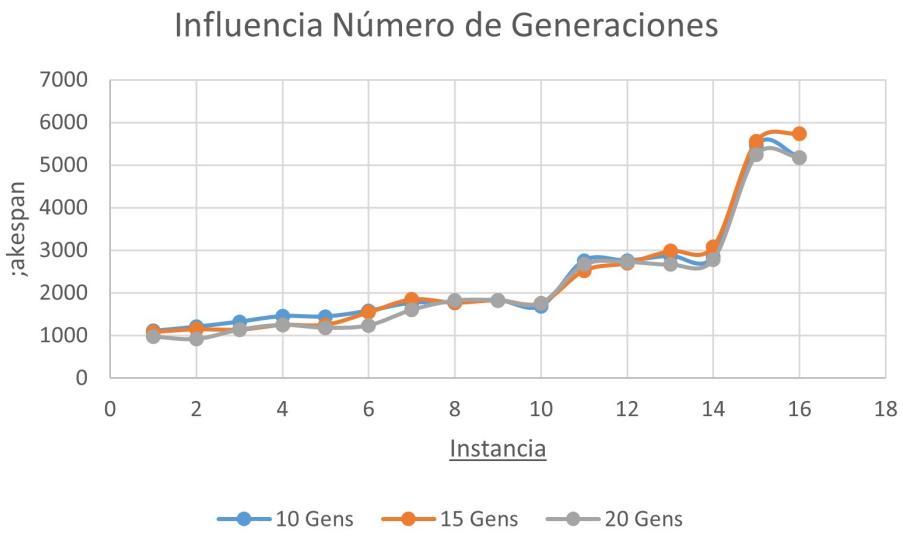
Tamaño de población 5 10 15



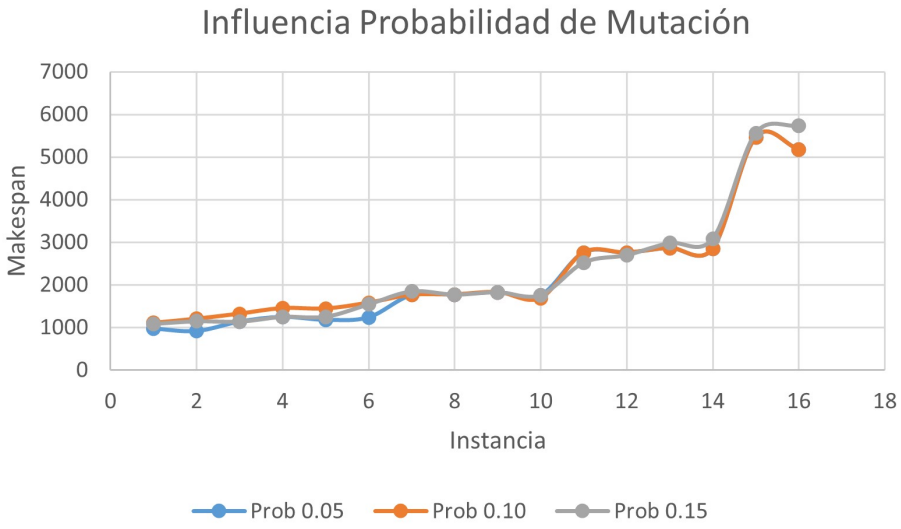
Dominancia del gen padre 0.3/0.5/0.7



Número de Generaciones 5/10/15



Probabilidad de Mutación 0.5/0.10/0.15



Mejores respuestas (según media)

15 individuos iniciales
Dominancia del 70% del padre
15 Generaciones de individuos
5% de probabilidad de mutar

Tiempo de Computo

Instancia	Tiempo Computo	Instancia	Tiempo Computo
1	2,056	9	74,98
2	2,055	10	74,945
3	2,025	11	135,28
4	2,024	12	138,28
5	74,87	13	298,52
6	73,85	14	300,025
7	75,025	15	1752,94
8	75,014	16	1795,15

Comparación con métodos anteriores

	Constructivo	GRASP1	GRASP2	LS6	VND	FRANK	GENÉTICO
JSSP1.txt	1879	1609	1553	1392	1392	1392	977
JSSP2.txt	1772	1597	1594	1357	1449	1449	919
JSSP3.txt	2109	1830	1936	1651	1775	1775	1139
JSSP4.txt	2349	1949	1916	1639	1745	1745	1251
JSSP5.txt	2380	2381	2245	1900	1851	1851	1182
JSSP6.txt	2771	2311	2223	1879	1878	1878	1240
JSSP7.txt	2630	2595	2508	2175	1972	1972	1604
JSSP8.txt	2985	2617	2445	2218	2165	2165	1820
JSSP9.txt	2893	2978	2797	2650	2650	2650	1830
JSSP10.txt	3056	2933	2837	2440	2488	2488	1761
JSSP11.txt	4229	3759	3665	3471	3471	3471	2665
JSSP12.txt	4179	3697	3611	3410	3410	3410	2725
JSSP13.txt	4570	4189	4185	3379	3379	3379	2668
JSSP14.txt	4558	4098	4084	3511	3511	3511	2784
JSSP15.txt	7473	7018	6971	6006	6006	6006	5240
JSSP16.txt	7070	6895	6915	5787	5787	5787	5181
Promedio	3556,4375	3278,5	3217,8125	2804,0625	2808,0625	2808,0625	2186,625

Figure: Comparación de resultados con métodos Anteriores

Conclusiones

Conclusiones

- Los algoritmos genéticos son eficaces para explorar el espacio de soluciones del JSSP, permitiendo la búsqueda de soluciones prometedoras en un conjunto amplio y complejo de posibles programaciones de trabajos en máquinas.
- La combinación de operadores genéticos, como la cruza y la mutación, permite un equilibrio adecuado entre la exploración del espacio de soluciones para encontrar nuevas áreas prometedoras y la explotación de soluciones conocidas para mejorar la calidad de las programaciones.

- La efectividad de un algoritmo genético en el JSSP puede depender de la elección adecuada de parámetros, la mejor influencia sobre el desempeño involucra un número más grande de elementos en la población inicial o un número más grande de generaciones.
- La convergencia del algoritmo a óptimos locales es dificil de romper si no se cuenta con una cantidad suficiente de individuos.
- Múltiples individuos pueden generar un mismo makespan, por lo que la exploración de vecindarios no suele ser muy efectiva para mejorar la solucion