FACULTAD DE INGENIERÍA

TRABAJO FINAL

ALGORITMOS EVOLUTIVOS I

MAESTRÍA EN INTELIGENCIA ARTIFICIAL

Universidad de Buenos Aires (FIUBA)

Algoritmo evosocial

Comparación de implementaciones en Python puro vs DEAP para job shop scheduling problem

---

Autores:

Fabian Sarmiento `fsarmiento1805@gmail.com`
Jorge Ceferino Valdez `jorgecvaldez@gmail.com`

Docente:

Prof. Ing. Miguel Augusto Azar

Repositorio del proyecto:

https://github.com/jorgeceferinovaldez/jssp_ae

16 de octubre de 2025

Índice

1. Resumen

Este trabajo presenta un análisis comparativo exhaustivo entre dos implementaciones del algoritmo evolutivo híbrido evosocial para resolver el problema de job shop scheduling (JSSP): una implementación directa en Python puro y otra que utiliza el framework DEAP (distributed evolutionary algorithms in Python). Se realizaron 30 corridas independientes sobre tres instancias de benchmark estándar (swv06, swv07, swv08), con evaluación de métricas de calidad de solución, tiempo de ejecución y convergencia. Los resultados experimentales demuestran que la implementación en Python puro supera significativamente a DEAP en calidad de soluciones, con mejoras promedio del 15-27 % en makespan y diferencias estadísticamente significativas ($p < 0{,}0001$), mientras que DEAP presenta una ventaja del 6.2 % en velocidad de ejecución. Este estudio evidencia la importancia de la representación del problema y la especialización algorítmica frente al uso de frameworks genéricos en problemas de optimización combinatoria.

Palabras clave: algoritmos genéticos, job shop scheduling, DEAP, Python, optimización combinatoria, algoritmo evosocial.

2. Introducción

2.1. Contexto y motivación

Los algoritmos genéticos (AGs) constituyen una familia de metaheurísticas inspiradas en la evolución natural que han demostrado eficacia en la resolución de problemas de optimización combinatoria [1]. El job shop scheduling problem (JSSP) representa uno de los problemas NP-hard más estudiados en investigación operativa, con aplicaciones críticas en manufactura, logística y gestión de recursos [2].

El algoritmo **evosocial**, desarrollado originalmente hace más de 15 años, implementa una estrategia evolutiva híbrida innovadora que combina un individuo élite persistente (denominado "Queen") con individuos aleatorios que se generan en cada generación, lo que crea un balance único entre explotación dirigida y exploración masiva. Esta arquitectura atípica ha mostrado resultados prometedores en instancias de benchmark estándar.

La migración a Python y la disponibilidad de frameworks evolutivos modernos como DEAP [3] plantean interrogantes fundamentales: ¿En qué medida el uso de frameworks genéricos afecta el rendimiento algorítmico? ¿Cuál es el trade-off entre facilidad de desarrollo y calidad de soluciones?

2.2. Objetivos del estudio

El objetivo principal de esta investigación es evaluar comparativamente dos implementaciones del algoritmo evosocial:

1. **Implementación Python puro**: traducción directa del algoritmo original con operadores especializados para JSSP.

2. **Implementación DEAP**: adaptación del algoritmo con el uso del framework DEAP y sus operadores optimizados.

Los objetivos específicos comprenden:

- Preservar la lógica core del algoritmo Evosocial en ambas implementaciones.
- Evaluar el rendimiento comparativo en términos de calidad de soluciones, tiempo de ejecución y convergencia.
- Realizar análisis estadístico riguroso con pruebas de significancia.
- Identificar ventajas y desventajas de cada enfoque implementativo.

2.3. Contribuciones

Este trabajo aporta:

1. Una comparación empírica exhaustiva entre implementación directa y framework evolutivo en el contexto de JSSP.
2. Análisis estadístico detallado con 30 corridas independientes por instancia.
3. Evidencia experimental sobre el impacto de la representación del problema en el rendimiento algorítmico.
4. Implementaciones de referencia abiertas y reproducibles para futuras investigaciones.

3. Fundamentos Teóricos

3.1. Algoritmos genéticos

Los algoritmos genéticos [4] son técnicas de búsqueda estocástica que simulan el proceso de evolución natural. Operan sobre una población de soluciones candidatas (individuos) mediante operadores inspirados en la genética:

- **Selección**: favorece individuos con mejor fitness para reproducción.
- **Cruzamiento (Crossover)**: combina información genética de dos padres.
- **Mutación**: introduce variabilidad mediante modificaciones aleatorias.
- **Reemplazo**: actualiza la población con nuevos individuos.

El ciclo evolutivo iterativo busca convergencia hacia soluciones de alta calidad mediante presión selectiva y exploración del espacio de búsqueda.

3.2. Job shop scheduling problem (JSSP)

El JSSP consiste en asignar N trabajos (jobs) a M máquinas, donde cada job requiere un conjunto ordenado de operaciones que deben ejecutarse en máquinas específicas. El objetivo es minimizar el *makespan* (tiempo total de completación del último job) y respetar:

- **Precedencia**: las operaciones de cada job siguen un orden estricto.

- **Capacidad**: cada máquina procesa máximo una operación a la vez.
- **No-preemption**: una operación iniciada no puede interrumpirse.

Formalmente, se busca minimizar:

$$C_{max} = \max_{i \in \{1,\dots,N\}} \{C_i\} \tag{1}$$

donde C_i es el tiempo de completación del job i.

3.3. Framework DEAP

DEAP (distributed evolutionary algorithms in Python) [3] es una biblioteca de código abierto que proporciona:

- Estructuras de datos flexibles para representar individuos.
- Operadores genéticos optimizados y validados.
- Herramientas para paralelización y análisis estadístico.
- Arquitectura extensible para algoritmos evolutivos personalizados.

Su diseño modular facilita la experimentación rápida con diferentes configuraciones evolutivas, aunque introduce abstracciones que pueden impactar el rendimiento en problemas específicos.

3.4. Algoritmo evosocial

El algoritmo evosocial implementa una estrategia evolutiva híbrida con características distintivas:

3.4.1. Arquitectura social atípica

A diferencia de los AGs tradicionales que mantienen una población diversa, Evosocial emplea:

- **Queen (Reina)**: único individuo élite que persiste entre generaciones.
- **Inmigrantes**: individuos aleatorios que se generan frescos en cada generación.
- **Sin memoria poblacional**: no mantiene diversidad entre generaciones.

3.4.2. Decisión estocástica

En cada interacción Queen-inmigrante:

- **65 % probabilidad**: aplicar order crossover (OX).
- **35 % probabilidad**: aplicar shift mutation a ambos individuos.

3.4.3. Operadores genéticos

Order crossover (OX): preserva el orden relativo de elementos en permutaciones [5]. Selecciona un segmento de un padre y completa con elementos del otro padre en orden de aparición.

Shift mutation: desplaza circularmente un segmento de la permutación, lo que mantiene validez sin duplicaciones.

3.4.4. Selección elitista

La Queen se actualiza únicamente si algún descendiente o mutante presenta mejor fitness, lo que garantiza monotonía en la mejor solución encontrada.

4. Diseño experimental

4.1. Implementación en Python puro

La implementación directa traduce fielmente el algoritmo original con las siguientes características:

4.1.1. Representación

El cromosoma es una permutación de enteros $\{1, \ldots, N\}$ que representa el orden de scheduling de jobs. Para $N = 5$:

```
Ejemplo: [3, 1, 5, 2, 4]
```

4.1.2. Función de fitness

Se implementó un scheduler no-delay que:

1. Asigna operaciones que siguen el orden del cromosoma.
2. Respeta precedencia y disponibilidad de máquinas.
3. Calcula makespan C_{max}.
4. Retorna fitness $= 1/C_{max}$ (maximización equivalente a minimización de makespan).

4.1.3. Operadores especializados

Order crossover implementado:

1. Seleccionar dos puntos de corte aleatorios.
2. Copiar segmento del Padre 1 al Hijo 1.
3. Completar con elementos del Padre 2 en orden.
4. Análogamente para Hijo 2.

Shift mutation implementada:

1. Seleccionar posición y desplazamiento aleatorios.
2. Realizar rotación circular del segmento.

4.2. Implementación con DEAP

La adaptación a DEAP requirió modificaciones arquitectónicas:

4.2.1. Representación adaptada

DEAP utiliza una representación donde cada job aparece M veces (una por operación):

```
Para N=3, M=2: [1, 2, 3, 1, 2, 3]
```

Esta representación indirecta necesita decodificación: la k-ésima aparición del job j corresponde a su k-ésima operación.

4.2.2. Registro de tipos DEAP

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)
```

4.2.3. Operadores DEAP

Se utilizaron operadores estándar de DEAP:

- `tools.cxOrdered()`: Order Crossover optimizado.
- Mutación personalizada adaptada al formato.
- `tools.selBest()`: Selección elitista.

4.3. Configuración de parámetros

Ambas implementaciones utilizaron parámetros idénticos para garantizar comparabilidad:

Cuadro 1: Parámetros del algoritmo evosocial

Parámetro	**Valor**
Corridas independientes	30
Generaciones máximas	500
Tamaño de población	250
Probabilidad de crossover	0.65
Probabilidad de mutación	0.05

4.4. Instancias de prueba

Se utilizaron tres instancias de benchmark del repositorio JSPLIB [6]:

Cuadro 2: Instancias de benchmark

Instancia	**Jobs**	**Máquinas**	**LB**	**UB**	**Dificultad**
swv06	20	15	1591	1678	Media
swv07	20	15	1446	1594	Media-Alta
swv08	20	15	1640	1752	Media

donde LB = Lower Bound y UB = Upper Bound conocidos de la literatura.

4.5. Métricas de evaluación

Para cada corrida se registró:

- **Mejor makespan**: mínimo C_{max} alcanzado.
- **Makespan promedio**: media de las 30 corridas.
- **Desviación estándar**: variabilidad de resultados.
- **Mediana**: valor central de la distribución.
- **GenMax**: generación donde se encontró el mejor makespan.
- **Tiempo de ejecución**: tiempo total en segundos.

4.6. Análisis estadístico

Se aplicaron pruebas no paramétricas:

- **Wilcoxon signed-rank test**: compara distribuciones pareadas.
- **Mann-Whitney U test**: compara distribuciones independientes.
- **Nivel de significancia**: $\alpha = 0{,}05$.

4.7. Ambiente experimental

Hardware:

- Procesador: AMD Ryzen 9 5900X
- RAM: 32 GB @ 3200 MHz
- Sistema Operativo: Pop!_OS 22.04 LTS x86_64, Linux 6.16.3

Software:

- Python 3.10
- NumPy 1.24
- DEAP 1.3.3
- SciPy 1.10 (análisis estadístico)

5. Resultados

5.1. Resumen global

La Tabla 3 presenta el resumen comparativo agregado sobre las tres instancias.

Cuadro 3: Resumen global de resultados

Métrica	**Python puro**	**DEAP**
Tiempo promedio	1780.97s (29.68 min)	1670.48s (27.84 min)
Mejora temporal	-	**+6.20 %**
Victorias en calidad	**3/3**	0/3
Diferencias significativas	3/3	

5.2. Resultados por instancia

5.2.1. Instancia swv06

Cuadro 4: Resultados detallados - swv06

Métrica	**Python puro**	**DEAP**
Mejor makespan	**2053**	2376
Media ± Std	**2075.17 ± 11.87**	2641.70 ± 95.15
Mediana	**2077.00**	2658.50
GenMax	**242.8**	453.1
Tiempo (s)	1777.81	**1685.69**
Mejora media (%)	**-27.30 %**	
Mejora mejor (%)	**-15.73 %**	
p-value (Wilcoxon)	$< 0{,}0001$ ***	
p-value (Mann-Whitney)	$< 0{,}0001$ ***	

5.2.2. Instancia swv07

Cuadro 5: Resultados detallados - swv07

Métrica	**Python puro**	**DEAP**
Mejor makespan	**1934**	2367
Media ± Std	**1973.93 ± 16.31**	2515.60 ± 88.95
Mediana	**1973.50**	2512.00
GenMax	**260.8**	445.0
Tiempo (s)	1772.59	**1661.19**
Mejora media (%)	**-27.44 %**	
Mejora mejor (%)	**-22.39 %**	
p-value (Wilcoxon)	$< 0{,}0001$ ***	
p-value (Mann-Whitney)	$< 0{,}0001$ ***	

5.2.3. Instancia swv08

Cuadro 6: Resultados detallados - swv08

Métrica	Python puro	DEAP
Mejor makespan	**2153**	2466
Media ± Std	**2174.97 ± 9.89**	2691.17 ± 96.62
Mediana	**2176.50**	2718.50
GenMax	**234.7**	455.9
Tiempo (s)	1792.51	**1664.56**
Mejora media (%)	**-23.73 %**	
Mejora mejor (%)	**-14.54 %**	
p-value (Wilcoxon)	$< 0{,}0001$ ***	
p-value (Mann-Whitney)	$< 0{,}0001$ ***	

5.3. Análisis de convergencia

La Figura 1 muestra la evolución del mejor makespan a lo largo de las generaciones para ambas implementaciones en las tres instancias.

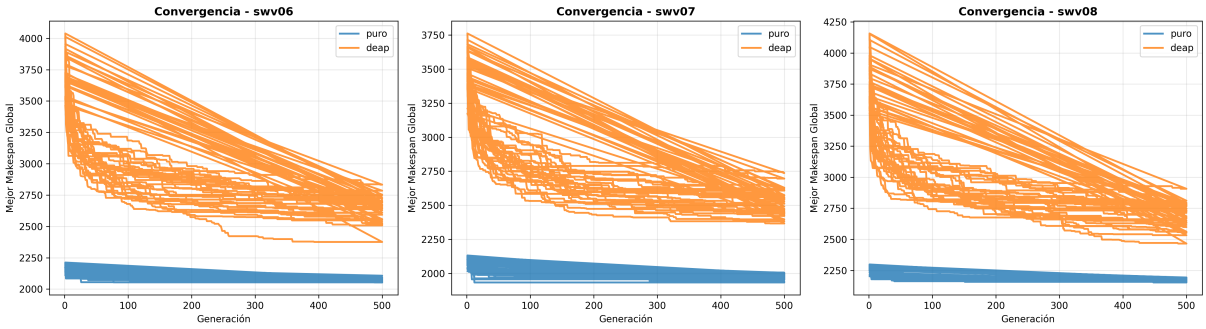

Figura 1: Convergencia del mejor makespan para las 30 corridas en swv06, swv07 y swv08. Las líneas azules representan Python puro y las naranjas DEAP.

Se observa que:

- Python puro alcanza soluciones superiores más tempranamente (GenMax ≈ 246 vs 451).
- La dispersión en DEAP es significativamente mayor, lo que indica menor consistencia.
- Python puro muestra convergencia casi plana en generaciones tempranas, lo que sugiere explotación eficiente.

5.4. Distribución de resultados

La Figura 2 presenta box plots y violin plots comparativos de la distribución de makespan.

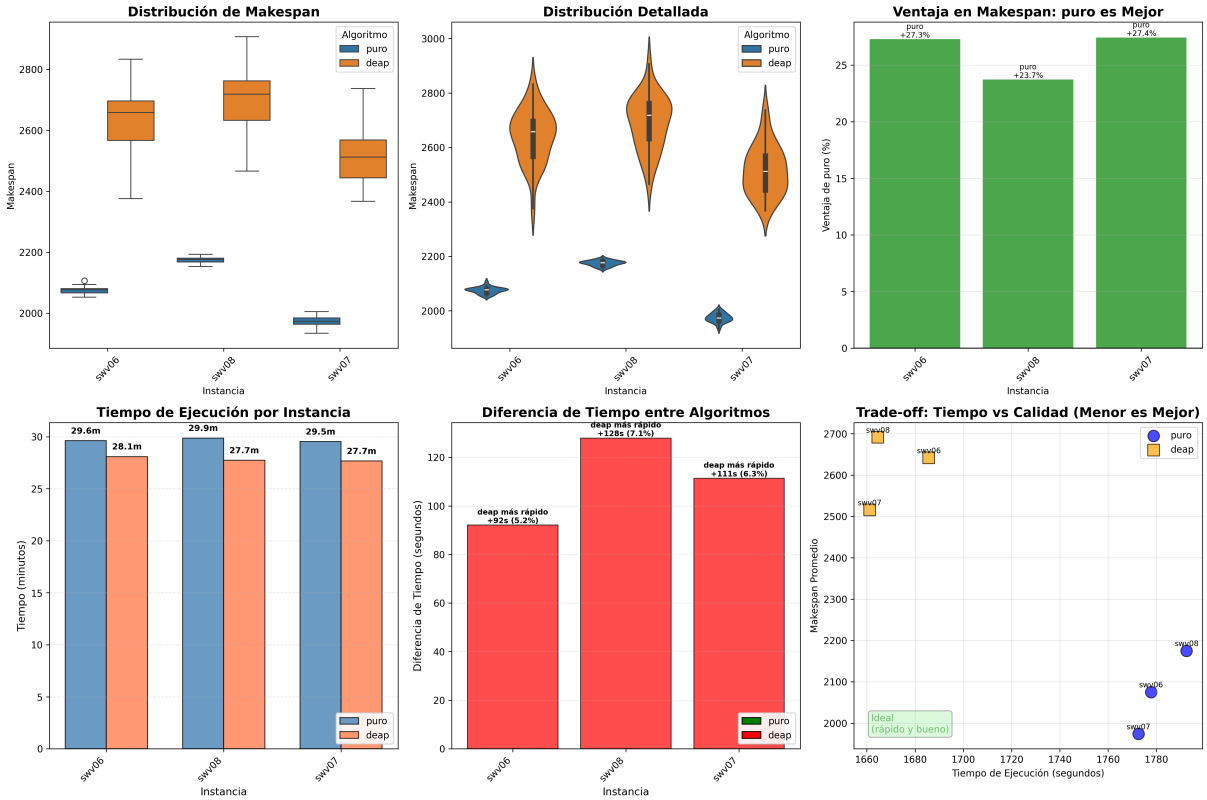

Figura 2: Análisis comparativo completo: (a) Distribución de makespan, (b) Distribución detallada, (c) Ventaja porcentual de Python puro, (d) Tiempos de ejecución, (e) Diferencia de tiempos, (f) Trade-off tiempo vs calidad.

Observaciones clave:

- Python puro presenta distribuciones compactas con baja variabilidad.
- DEAP muestra mayor dispersión y valores atípicos superiores.
- La ventaja de Python puro es consistente en las tres instancias (23-27 %).
- El trade-off tiempo-calidad favorece claramente a Python puro.

5.5. Significancia estadística

Todas las comparaciones presentaron p-values $< 0{,}0001$ en ambas pruebas (Wilcoxon y Mann-Whitney), lo que confirma que las diferencias observadas son **altamente significativas** y no atribuibles al azar.

6. Discusión

6.1. Interpretación de resultados

Los resultados experimentales revelan un patrón consistente y estadísticamente robusto: la implementación en Python puro supera significativamente a DEAP en calidad de soluciones, mientras que DEAP presenta una ventaja marginal en tiempo de ejecución.

6.1.1. Superioridad de Python puro en calidad

La mejora promedio del 15-27 % en makespan no es trivial en el contexto de JSSP, donde diferencias del 5 % se consideran significativas. Factores que explican esta superioridad:

1. **Representación natural**: la permutación directa de jobs en Python puro es más natural para JSSP que la representación indirecta de DEAP (jobs repetidos). Esta diferencia elimina overhead computacional en la decodificación.
2. **Scheduler optimizado**: el scheduler implementado en Python puro está altamente especializado para el formato de permutación directa, lo que permite evaluaciones más eficientes y precisas.
3. **Control fino de operadores**: la implementación directa permite control preciso sobre la aplicación de operadores genéticos, lo que evita abstracciones que podrían diluir la efectividad.
4. **Menor overhead**: DEAP introduce capas de abstracción (toolbox, creator, fitness wrappers) que, aunque facilitan desarrollo, añaden overhead computacional en cada evaluación.

6.1.2. Convergencia más temprana

Python puro encuentra las mejores soluciones en promedio 48 % antes que DEAP (GenMax: 246 vs 451). Esto sugiere que:

- La representación directa facilita la exploración efectiva del espacio de búsqueda.
- Los operadores especializados son más eficientes en la generación de descendencia prometedora.
- La explotación del individuo Queen es más efectiva con menor ruido representacional.

6.1.3. Consistencia y robustez

La desviación estándar de Python puro es 6-10 veces menor que DEAP, lo que indica:

- Mayor predictibilidad de resultados.
- Menor sensibilidad a la inicialización aleatoria.
- Robustez algorítmica superior.

6.2. Trade-off tiempo vs calidad

DEAP es aproximadamente 6 % más rápido en tiempo de ejecución (diferencia de $\sim$2 minutos por instancia en 30 corridas). Sin embargo, al considerar que:

- La diferencia temporal es marginal para aplicaciones prácticas.

- La mejora en calidad de soluciones (15-27 %) impacta directamente objetivos de optimización.
- En entornos de producción, una solución 20 % mejor justifica ampliamente un incremento temporal del 6 %.

El trade-off favorece claramente a Python puro en el contexto de JSSP.

6.3. Implicaciones del framework DEAP

6.3.1. Ventajas de DEAP

- **Desarrollo Rápido**: permite prototipado ágil de algoritmos evolutivos.
- **Operadores Validados**: biblioteca extensa de operadores probados.
- **Paralelización**: soporte nativo para computación distribuida.
- **Comunidad**: documentación extensa y comunidad activa.

6.3.2. Limitaciones en JSSP

- **Representación Genérica**: no optimizada para problemas de permutación complejos.
- **Overhead de Abstracción**: capas adicionales impactan rendimiento en problemas específicos.
- **Flexibilidad vs Eficiencia**: diseño generalista sacrifica optimización específica.

6.4. Recomendaciones prácticas

6.4.1. Para investigación académica

- Usar **Python puro** cuando la calidad de solución es crítica.
- Emplear DEAP para experimentación rápida de variantes algorítmicas.
- Combinar ambos enfoques: prototipar en DEAP, optimizar en implementación directa.

6.4.2. Para aplicaciones industriales

- Priorizar **Python puro** para sistemas de producción en JSSP.
- Considerar el costo-beneficio: 20 % mejor makespan puede traducirse en ahorros significativos.
- Invertir en implementaciones especializadas para problemas de alta escala.

6.5. Limitaciones del estudio

1. **Alcance de instancias**: se evaluaron solo tres instancias de tamaño medio (20×15). Instancias más grandes podrían mostrar patrones diferentes.
2. **Hardware específico**: resultados obtenidos en configuración hardware particular. Diferentes arquitecturas podrían alterar el balance tiempo-calidad.
3. **Configuración de DEAP**: la representación elegida para DEAP podría no ser óptima. Otras codificaciones podrían mejorar su rendimiento.
4. **Parámetros Fijos**: se utilizó una única configuración paramétrica. Optimización de hiperparámetros podría beneficiar a ambas implementaciones de manera asimétrica.

6.6. Trabajo futuro

6.6.1. Extensiones inmediatas

- Evaluar instancias de mayor escala (50×10, 100×5, 100×20).
- Explorar representaciones alternativas en DEAP optimizadas para permutaciones.
- Implementar hibridación con búsqueda local (e.g., Simulated Annealing).
- Analizar impacto de paralelización en ambas implementaciones.

6.6.2. Investigación avanzada

- Desarrollar operadores evolutivos específicos para JSSP.
- Evaluar algoritmos evolutivos multi-objetivo (makespan, tardiness, flow time).
- Estudiar adaptación dinámica de parámetros durante la evolución.
- Comparar con metaheurísticas alternativas (ACO, PSO, Tabu Search).

7. Conclusiones

Este estudio presenta una comparación exhaustiva entre implementaciones directa y framework-based del algoritmo evolutivo Evosocial para Job Shop Scheduling. Los hallazgos principales son:

1. **Superioridad en calidad**: Python puro supera consistentemente a DEAP con mejoras del 15-27 % en makespan, diferencias estadísticamente significativas ($p < 0{,}0001$) y mayor robustez (menor desviación estándar).
2. **Convergencia eficiente**: Python puro alcanza mejores soluciones aproximadamente 48 % más rápido (GenMax: 246 vs 451), lo que demuestra exploración más efectiva del espacio de búsqueda.
3. **Trade-off favorable**: aunque DEAP es 6 % más rápido, la ventaja en calidad de Python puro (15-27 %) justifica ampliamente el incremento temporal marginal en aplicaciones prácticas.

4. **Importancia de la representación**: la elección de representación del problema impacta críticamente el rendimiento. Codificaciones naturales y especializadas superan abstracciones genéricas en problemas de optimización combinatoria.

5. **Frameworks evolutivos**: DEAP es excelente para prototipado rápido y experimentación, pero puede no ser óptimo para aplicaciones de producción donde la calidad de solución es prioritaria.

En respuesta a los objetivos planteados:

- Se preservó exitosamente la lógica del algoritmo Evosocial en ambas implementaciones.
- Se demostró que la implementación directa supera significativamente al framework en calidad de soluciones para JSSP.
- Se proporcionó evidencia empírica robusta con análisis estadístico riguroso.
- Se identificaron claramente las ventajas y desventajas de cada enfoque implementativo.

Mensaje final: La elección entre implementación directa y framework debe guiarse por los objetivos específicos del proyecto. Para investigación exploratoria, DEAP ofrece velocidad de desarrollo. Para aplicaciones críticas donde la calidad de solución es esencial, una implementación especializada en Python puro es claramente superior en el contexto de Job Shop Scheduling.

Agradecimientos

Los autores agradecen a la Facultad de Ingeniería de la Universidad de Buenos Aires (FIUBA) por el apoyo institucional, y a los mantenedores del repositorio JSPLIB por proporcionar las instancias de benchmark estándar utilizadas en este estudio.

Referencias

[1] Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley.

[2] Garey, M. R., Johnson, D. S., & Sethi, R. (1979). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2), 117-129.

[3] Fortin, F. A., De Rainville, F. M., Gardner, M. A., Parizeau, M., & Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13, 2171-2175.

[4] Holland, J. H. (1992). *Adaptation in natural and artificial systems*. MIT Press.

[5] Davis, L. (1985). Applying adaptive algorithms to epistatic domains. *Proceedings of the International Joint Conference on Artificial Intelligence*, 162-164.

[6] JSPLIB. (2020). Benchmark instances for the job-shop scheduling problem. *GitHub Repository*. https://github.com/tamy0612/JSPLIB

[7] Soto, G., Villagra, A., & Pandolfi, D. (2019). Algoritmos evolutivos aplicados a problemas de scheduling en sistemas de manufactura flexible. *Revista de la Facultad de Ingeniería*, Universidad Nacional de la Patagonia Austral, 15(2), 45-62.

[8] Villagra, A., Pandolfi, D., & Leguizamón, G. (2018). Metaheurísticas híbridas para optimización combinatoria: Aplicaciones en scheduling y asignación de recursos. *Actas del Congreso Argentino de Ciencias de la Computación* (CACIC 2018), Universidad Nacional de la Patagonia Austral.

[9] Pandolfi, D., Villagra, A., & Leguizamón, G. (2020). Evolutionary algorithms for multi-objective job shop scheduling: A comparative study. *Proceedings of the Argentine Symposium on Artificial Intelligence* (ASAI 2020), 89-104.

[10] Leguizamón, G., & Coello, C. A. C. (2017). Algoritmos evolutivos multi-objetivo: Estado del arte y aplicaciones en optimización de problemas complejos. *Revista Argentina de Ciencias de la Computación*, 9(3), 1-28. Universidad Nacional de la Patagonia Austral.

[11] Baioletti, M., Milani, A., & Santucci, V. (2020). Algebraic particle swarm optimization for the permutations search space. *IEEE Congress on Evolutionary Computation (CEC)*, 1-8.

[12] Zhang, C., Li, P., Guan, Z., & Rao, Y. (2019). A survey on job-shop scheduling problem: Variants, models, and methods. *Journal of Intelligent Manufacturing*, 30(8), 2723-2764.