Informe Técnico: Optimización de Scheduling en TecnoPrecision Global LLC

Integrantes:

Gabriel Alonso Coro — Grupo C412
Mauro Campver Barrios — Grupo C411
Josue Rolando Naranjo Sieiro — Grupo C411

Resumen

Este informe presenta una solución integral para el problema de optimización de la programación de la producción en la planta de *TecnoPrecision Global LLC*. El desafío se formaliza técnicamente como un problema de máquinas paralelas idénticas con restricciones de recursos acumulativos ($P_m \mid res \mid C_{max}$), cuya complejidad computacional se clasifica como NP-hard. Para abordar este reto, se diseñaron e implementaron tres aproximaciones algorítmicas: una búsqueda exhaustiva por fuerza bruta (*BruteForceSolver*) para la obtención de óptimos globales en instancias reducidas, una heurística constructiva de tipo *list-scheduling* (*EarliestStartSolver*) y una metaheurística poblacional basada en Algoritmos Genéticos (*GeneticSolver*).

El análisis experimental, realizado sobre un conjunto de 30 instancias de prueba, revela que el algoritmo de fuerza bruta alcanza su límite de computabilidad práctico al superar los 11 trabajos debido a la explosión exponencial del espacio de búsqueda. En contraste, el Algoritmo Genético demostró una eficacia excepcional, logrando un error relativo medio del 0.0 % respecto al óptimo en tiempos de ejecución escalables (≈ 10 s). Finalmente, se concluye con una recomendación estratégica que prioriza la metaheurística para la planificación diaria y el uso de la heurística *greedy* para la gestión de contingencias en tiempo real.

1 Fase 1: Formalización del Problema

1.1 El Fabricante Chino: Enunciado del Problema

En *TecnoPrecision Global LLC*, nos enorgullece ser líderes en la fabricación de componentes electrónicos de alta precisión, operando bajo un modelo de producción "bajo demanda" para nuestros clientes más exigentes. Nuestra reputación se basa en la calidad y, fundamentalmente, en la rapidez y fiabilidad de nuestras entregas.

Actualmente, nos enfrentamos a un desafío crítico que impacta directamente nuestra capacidad para cumplir con los plazos de entrega prometidos. Cuando recibimos un pedido de un cliente, este suele consistir en una lista de varios componentes electrónicos distintos que debemos fabricar. Contamos con varias **líneas de producción idénticas** en nuestra planta, lo que nos da flexibilidad para fabricar cualquier componente en cualquiera de ellas.

Sin embargo, la fabricación de cada uno de estos componentes requiere el uso de **herramientas especializadas** de alta tecnología. Estas herramientas son muy costosas y, por lo tanto, tenemos un número limitado de cada tipo. Por ejemplo, solo disponemos de un cierto número de cabezales de soldadura láser de precisión o de equipos de calibración específicos. Un componente puede necesitar varias de estas herramientas, y algunas herramientas pueden ser compartidas por diferentes tipos de componentes.

El problema que necesitamos resolver es el siguiente: **¿Cómo podemos organizar la fabricación de todos los componentes de un pedido de cliente de la manera más eficiente posible para que el pedido completo esté listo en el menor tiempo total?**

Esto implica tomar decisiones clave:

- ¿En qué orden debemos empezar a fabricar cada componente?
- ¿Qué línea de producción debe utilizarse para cada componente?
- **¿Cómo gestionamos la disponibilidad de nuestras herramientas especializadas?** Si una herramienta es necesaria para fabricar un componente en una línea, no puede usarse simultáneamente para otro componente en otra línea. La escasez de una herramienta específica puede crear un cuello de botella y detener la producción de varios componentes a la vez.

Necesitamos una solución que nos permita planificar la secuencia de fabricación de cada componente, asignándolos a nuestras líneas de producción y asegurando que las herramientas necesarias estén disponibles en el momento justo, para que el último componente del pedido esté terminado lo antes posible.

1.2 Clasificación del Problema

El problema de programación de la producción en la planta se define formalmente como un **Problema de Programación de Tareas en Máquinas Paralelas Idénticas con Restricciones de Recursos**. Bajo la clasificación técnica de tres campos propuesta por **Graham et al. (1979)**, este modelo se denota como:

$$P_m \mid res \mid C_{max}$$

Esta notación identifica que disponemos de m máquinas idénticas (P_m), restricciones de recursos adicionales (res) y el objetivo de minimizar el tiempo de finalización de la última tarea o *makespan* (C_{max}). El problema es inherentemente no preventivo (*non-preemptive*), lo que añade una restricción de integridad temporal a cada tarea: una vez iniciada la fabricación de un componente, no puede ser interrumpida.

1.3 Definición de Estructuras de Datos

Para formalizar el modelo, se requieren las siguientes entradas:

- **Conjunto de Trabajos (J):** $\{j_1, j_2, \ldots, j_n\}$, donde cada j es un componente del pedido.
- **Conjunto de Máquinas (M):** $\{i_1, i_2, \ldots, i_m\}$, representando las líneas de producción idénticas.
- **Conjunto de Recursos (R):** $\{k_1, k_2, \ldots, k_r\}$, que agrupa los tipos de herramientas especializadas.
- **Vector de Tiempos (P):** donde p_j es la duración estimada de la tarea j.
- **Matriz de Recursos (D):** donde r_{jk} indica la cantidad del recurso k que requiere el trabajo j.
- **Vector de Capacidades (Q):** donde Q_k es la cantidad máxima disponible del recurso k.

1.4 Modelo Matemático

1.4.1 Función Objetivo

Minimizar el tiempo de término del último trabajo del pedido (*Makespan*):

$$\min C_{max}$$

1.4.2 Restricciones Fundamentales

1. **Garantía de Procesamiento**
   Cada trabajo j debe ser asignado a una única máquina i:
   $$\sum_{i=1}^{m} x_{ij} = 1, \quad \forall j \in J$$
2. **No Solapamiento en Líneas (Disyunción)**
   Si dos trabajos j y l comparten la misma línea i, sus intervalos de ejecución no deben cruzarse:
   $$x_{ij} \cdot x_{il} = 1 \implies (S_j + p_j \leq S_l) \cup (S_l + p_l \leq S_j)$$
3. **Disponibilidad de Herramientas (Capacidad Acumulativa)**
   Para cada recurso k y en todo instante t, la demanda agregada de los trabajos activos debe ser menor o igual a la oferta total:
   $$\sum_{j:S_j \leq t < S_j + p_j} r_{jk} \leq Q_k, \quad \forall k \in R$$
4. **Cota Superior del Tiempo**
   $$C_{max} \geq S_j + p_j, \quad \forall j \in J$$

1.5 Propiedades de la Salida (Solución Óptima)

Una solución válida debe entregar:

1. **Plan de Inicio (S):** un valor de tiempo de comienzo preciso para cada componente.
2. **Asignación (X):** el mapeo de cada componente a su línea de producción correspondiente.
3. **Eficiencia de Red:** el valor de C_{max} debe ser el mínimo global, garantizando que no existan retrasos por subutilización de máquinas mientras los recursos estén disponibles.

2 Fase 2: Análisis de Complejidad Computacional

2.1 Preliminares: Optimización vs. Decisión

Para establecer la complejidad computacional del problema de optimización planteado $(P_m \mid res \mid C_{max})$, primero debemos formular su versión de **Problema de Decisión** asociada. La teoría de la NP-completitud se aplica directamente a problemas de decisión. Si demostramos que la versión de decisión es **NP-completa**, entonces el problema de optimización correspondiente es **NP-duro**.

Definición del Problema de Decisión Π_{sched}:

- **Instancia:** Un conjunto de trabajos J, máquinas M, recursos R, tiempos p_j, capacidades Q_k, demandas r_{jk} y un valor entero K (el límite de tiempo o *deadline*).
- **Pregunta:** ¿Existe un cronograma factible $\mathcal{S}$ tal que el tiempo de finalización de todos los trabajos sea menor o igual a K ($C_{max} \leq K$), respetando todas las restricciones de recursos y máquinas?

2.2 Teorema de Complejidad

Teorema 1. El problema de decisión Π_{sched} es **NP-completo**. En consecuencia, el problema de optimización asociado es **NP-duro**.

Estrategia de Demostración: Para demostrar que un problema es NP-completo, deben cumplirse dos condiciones:

1. **Condición 1 (Pertenencia a NP):** Demostrar que, dada una solución propuesta (certificado), es posible verificar su validez en tiempo polinomial.
2. **Condición 2 (NP-dureza):** Demostrar que un problema conocido como NP-completo (en este caso **PARTITION**) se reduce polinomialmente a una instancia restringida de nuestro problema.

2.3 Demostración de Pertenencia a NP

Sea $\mathcal{S}$ un certificado (solución candidata) que describe un cronograma: para cada trabajo j, especifica una máquina m_i y un tiempo de inicio t_j. Un algoritmo verificador $\mathcal{V}$ debe comprobar las siguientes restricciones:

1. **Unicidad y No-Solapamiento:** Verificar que cada trabajo se asigna exactamente a una máquina y que, en dicha máquina, el intervalo $[t_j, t_j + p_j)$ no se solapa con otros trabajos. Esto requiere ordenar los trabajos por máquina y tiempo, con complejidad $O(n \log n)$.
2. **Restricciones de Recursos:** Identificar los eventos discretos (tiempos de inicio y fin). Para cada intervalo entre eventos, sumar el consumo de cada recurso k y verificar que no exceda Q_k. Dado que hay a lo sumo $2n$ eventos y $|R|$ recursos, esto toma $O(n \cdot |R|)$.
3. **Límite de Tiempo:** Verificar para todo j que $t_j + p_j \leq K$. Complejidad $O(n)$.

Dado que todas las verificaciones se ejecutan en tiempo polinomial respecto al tamaño de la entrada, concluimos que $\Pi_{sched} \in \mathbf{NP}$.

2.4 Definición del Problema Base: PARTITION

Para la segunda condición (NP-dureza), utilizaremos el problema de la Partición, uno de los 21 problemas NP-completos originales de Karp (1972).

Problema: PARTITION

- **Entrada:** Un conjunto finito $A = \{a_1, a_2, \ldots, a_n\}$ de enteros positivos. Sea $W = \sum_{i=1}^{n} a_i$ la suma total.
- **Pregunta:** ¿Existe un subconjunto $A' \subseteq A$ tal que

$$\sum_{a \in A'} a = \sum_{a \in A \setminus A'} a = \frac{W}{2}?$$

2.5 Demostración de la Reducción Polinomial (PARTITION $\propto P_2 \parallel C_{max}$)

Sea una instancia arbitraria de PARTITION definida por el conjunto A. Construiremos una instancia de Scheduling con 2 máquinas ($P_2 \parallel C_{max}$) en tiempo polinomial.

Construcción de la Instancia de Scheduling:

1. Definimos el conjunto de trabajos $J = \{1, 2, \ldots, n\}$, donde cada trabajo j corresponde a $a_j \in A$.
2. El tiempo de procesamiento es $p_j = a_j$.
3. Definimos $m = 2$ máquinas.

4. No existen restricciones de recursos ($Q_k = \infty$).

5. Definimos el límite del *makespan* como $K = W/2$.

Esta transformación es lineal, $O(n)$.

Prueba de Equivalencia:

(⇒) Directa: Si existe A' tal que suma $W/2$, asignamos los trabajos de A' a la Máquina 1 y el resto a la Máquina 2. Ambas terminan en $W/2 = K$. El cronograma es válido.

(⇐) Inversa: Si existe un cronograma válido con $C_{max} \leq W/2$, y la suma total de trabajo es W, entonces ambas máquinas deben terminar exactamente en $W/2$ (no pueden terminar antes, pues sobraría trabajo, ni después, pues violaría K). Los trabajos en la Máquina 1 forman el conjunto A'.

2.6 Generalización y Conclusión

Hemos demostrado que $P_2 \parallel C_{max}$ es NP-completo. Ahora conectamos esto con el problema general de *TecnoPrecision Global LLC* ($P_m \mid res \mid C_{max}$).

Utilizamos el argumento de **inclusión de instancias**. Sea Ψ_{Gen} el espacio de instancias del problema general y Ψ_{P2} el espacio de instancias sin recursos con 2 máquinas. Se cumple que $\Psi_{P2} \subset \Psi_{Gen}$.

Conclusión: Dado que el problema general contiene como caso particular a un problema NP-completo, el problema general es al menos tan difícil como este. Por tanto, el problema de optimización $P_m \mid res \mid C_{max}$ es **NP-duro**.

3 Fase 3: Diseño de Soluciones Algorítmicas

Algoritmos implementados

En esta sección se detallan las estrategias algorítmicas diseñadas para resolver el problema de *scheduling* con restricciones de recursos, abarcando desde el enfoque exacto hasta heurísticas avanzadas y metaheurísticas.

1. BruteForceSolver (`bruteforce.py`)

Este solver implementa un enfoque de búsqueda exhaustiva total que garantiza el óptimo global al explorar tanto el espacio de asignación de máquinas como el de secuenciación de trabajos, optimizando el tiempo de cómputo mediante la eliminación de simetrías.

- **Idea algorítmica:** El algoritmo explora el producto cartesiano de dos espacios: las permutaciones de los trabajos ($n!$) y las asignaciones únicas de máquinas. Para evitar la redundancia debida a que las máquinas son idénticas, se utilizan *Particiones de un Conjunto* (basadas en números de Stirling de segunda especie), lo que reduce el factor

de búsqueda de máquinas de m^n a una fracción significativamente menor. Para cada combinación de orden y asignación, se simula un horario mediante una política de eventos discretos.

- **Complejidad Temporal:** $O(n! \cdot S(n,m) \cdot (n \cdot m \cdot D))$, donde $S(n,m)$ es el número de Stirling de segunda especie y D es la duración promedio de los trabajos (debido al chequeo de recursos).
- **Complejidad Espacial:** $O(n + m + (n \cdot D))$, para almacenar el *timeline* de recursos y las estructuras de las colas.

Pseudocódigo Principal

```
Entrada: problem (n trabajos, m máquinas)
mejor_sol ← None; mejor_makespan ←

for cada permutación orden  Permutaciones(problem.jobs):
    # Generar asignaciones únicas para máquinas idénticas
    for cada asignación única assign  Particiones(orden, m):
        machine_queues ← construir colas siguiendo 'orden' y 'assign'
        sol ← BuildScheduleForAssignment(machine_queues)

        if sol != None y sol.makespan < mejor_makespan:
            mejor_makespan ← sol.makespan
            mejor_sol ← sol
return mejor_sol
```

Análisis de Correctitud La correctitud del algoritmo para hallar el óptimo global se basa en dos pilares:

1. **Completitud del espacio de búsqueda:** En problemas de agendamiento con recursos, el óptimo siempre reside en un *horario activo*. Un horario activo puede ser unívocamente definido por una lista de prioridades y una asignación de recursos. Al explorar todas las permutaciones $n!$, el algoritmo garantiza evaluar todas las jerarquías de prioridad posibles. Al evaluar todas las particiones de conjuntos, cubre todas las asignaciones de máquinas posibles, eliminando solo aquellas que son redundantes por simetría (máquinas idénticas).
2. **Simulación ASAP (*As Soon As Possible*):** La subrutina `BuildSchedule` coloca cada trabajo en el primer instante de tiempo disponible que satisface tanto la disponibilidad de la máquina como la capacidad de los recursos. Esto garantiza que para una secuencia y asignación dadas, el *makespan* resultante es el mínimo posible.

Análisis de Complejidad Temporal Detallado El coste total se define por la multiplicación de tres factores:

1. **Factor de Ordenación ($n!$):** El número de formas de priorizar los trabajos.
2. **Factor de Asignación ($S(n,m)$):** El número de formas de repartir n elementos en m subconjuntos no vacíos. Aunque $S(n,m) < m^n$, el crecimiento sigue siendo exponencial.

3. **Factor de Simulación (C_{sched}):** Por cada iteración, se procesan n trabajos. Para cada uno, se buscan candidatos en el conjunto de tiempos de finalización (hasta n eventos) y se verifica la disponibilidad de recursos durante la duración del trabajo (D). Esto resulta en un coste aproximado de $O(n^2 \cdot D)$.

En conjunto, el algoritmo es de clase exponencial, garantizando la solución óptima a cambio de un tiempo de ejecución que escala según $O(n! \cdot m^n \cdot n^2)$.

Subrutina: BuildScheduleForAssignment Simula la ejecución de una asignación y orden fijos.

```
function BuildScheduleForAssignment(machine_queues):
    machine_free_time[m] ← 0; timeline ← {}; completion_times ← {0}
    remaining ← copia de machine_queues; solution_jobs ← []

    while existe trabajo en remaining:
        candidates ← []
        for cada máquina i con cola no vacía:
            job ← remaining[i].head()
            # Buscar el primer instante factible en los eventos conocidos
            for t  sorted(completion_times) donde t  machine_free_time[i]:
                if CheckResources(t, job.duration, job.resources, timeline):
                    candidates.append((t, i, job))
                    break

        if candidates vacío: return None # Inviable
        elegir (start, m, job) con menor start (y menor machine_id como desempate)

        # Registrar en la solución y marcar recursos
        job.start ← start; solution_jobs.add(job)
        machine_free_time[m] ← start + job.duration
        completion_times.add(start + job.duration)
        MarcarRecursosEnTimeline(start, job.duration, job.resources, timeline)
        remaining[m].pop()

    return Solution(solution_jobs, max(finish_times))
```

2. EarliestStartSolver (`earliest_start_solver.py`)

Es una heurística voraz (*greedy*) de tipo *list-scheduling* diseñada para obtener soluciones rápidas y factibles.

- **Idea algorítmica:** En cada iteración, el algoritmo evalúa todos los trabajos no asignados y calcula para cada uno el par (*earliest_start_time*, máquina) más temprano posible. Se consideran "tiempos relevantes" basados en los eventos de finalización de tareas ya programadas. Se selecciona el trabajo que pueda iniciar antes en el tiempo global y se fija en el cronograma.

- **Complejidad:** $O(n \cdot (n \cdot m \cdot C_{check}))$. En cada una de las n asignaciones, se exploran m máquinas para los trabajos restantes.

```
while unassigned no vacío:
    earliest_list ← []
    current_candidates ← sorted(completion_times)

    for cada job en unassigned:
        best_start ← ; best_machine ← None
        for m in 1..m:
            m_free ← machine_free_time[m]
            valid_t ← {t  current_candidates | t  m_free}
            found ← primer t tal que CheckResources(t, job, timeline)
            if found < best_start:
                best_start ← found; best_machine ← m
        if best_machine != None:
            earliest_list.append((best_start, best_machine, job))

    (start, m, job) ← elegir menor start en earliest_list
    asignar job; actualizar estados; eliminar de unassigned
```

Análisis de cotas para el peor caso:

Dado que el `EarliestStartSolver` opera como una heurística de tipo *List Scheduling*, su desviación respecto al óptimo en el peor escenario está acotada teóricamente por los resultados de Garey y Graham (1975). El modelo considerado en dicho estudio se alinea con el problema de este proyecto bajo las siguientes definiciones:

- **Procesadores (n):** Un conjunto de n máquinas idénticas disponibles en paralelo.
- **Tareas:** Un conjunto finito de trabajos con tiempos de ejecución τ_i. En este trabajo, el orden parcial es vacío ($\leq \emptyset$), lo que implica que las tareas son independientes.
- **Recursos (s):** Existen s tipos de recursos $R = \{R_1, \ldots, R_s\}$ con capacidades normalizadas.
- **Restricción:** La suma de las demandas de un recurso R_i por las tareas activas en cualquier instante t no puede exceder su capacidad total.

Bajo estas condiciones, la razón entre el *makespan* obtenido por la heurística (ω) y el óptimo (ω^*) cumple con:

- **Teorema (Máquinas abundantes):** Si el número de procesadores no restringe el paralelismo frente a los recursos, la cota depende únicamente de la diversidad de herramientas:
$$\frac{\omega}{\omega^*} \leq s + 1$$
Donde s es el número de tipos de recursos. En el peor caso, una gestión ineficiente de la contención puede degradar la solución hasta $(s + 1)$ veces.

- **Teorema (Cota general):** Para un número finito de procesadores ($n \geq 2$), la cota integra tanto la limitación de máquinas como la congestión de recursos:

$$\frac{\omega}{\omega^*} \leq \text{mín}\left\{2n+1,\ s+2-\frac{n}{2s+1}\right\}$$

Estas cotas actúan como una garantía matemática de que el `EarliestStartSolver` nunca excederá estos límites de ineficiencia, independientemente de la complejidad de la instancia.

3. GeneticSolver (`metaheuristic.py`)

Metaheurística poblacional que explora el espacio de permutaciones de trabajos para minimizar el *makespan*. Este enfoque permite evadir óptimos locales mediante operadores estocásticos de evolución.

- **Representación:** Cada individuo del algoritmo se codifica como una permutación de los n trabajos. El *fitness* (aptitud) se determina mediante el decodificador `SolutionBuilder.build_from_sequence`, que transforma la secuencia en un cronograma factible respetando las restricciones de recursos.
- **Operadores Evolutivos:**
  - **Selección:** Torneo probabilístico ($k = 3$), que garantiza una presión selectiva equilibrada.
  - **Cruce (Crossover):** Operador de Cruce de Orden (OX), diseñado específicamente para preservar la precedencia relativa de las tareas.
  - **Mutación:** Intercambio (*swap*) de posiciones aleatorias para reintroducir diversidad genética.
- **Estrategias de Convergencia:** Implementa **elitismo** para preservar las mejores soluciones encontradas y un **mecanismo de reinicio** (*restart*) que reinyecta individuos aleatorios si la población converge prematuramente.

Parametrización del Modelo Para garantizar la convergencia y robustez en las instancias evaluadas, se han definido los siguientes parámetros de control:

Parámetro	Definición técnica	Valor
`pop_size`	Tamaño de la población; número de soluciones candidatas mantenidas simultáneamente.	100
`generations`	Límite máximo de iteraciones del ciclo evolutivo.	300
`mutation_rate`	Probabilidad de que un individuo sufra una alteración aleatoria en su estructura.	0.25
`crossover_rate`	Probabilidad de que dos padres intercambien material genético para generar descendencia.	0.9
`restart_threshold`	Umbral de generaciones sin mejora antes de forzar un reinicio de la población.	40

La configuración de un `mutation_rate` relativamente alto (0,25) junto con el `restart_threshold` asegura una exploración agresiva del espacio de soluciones, mitigando el riesgo de estancamiento en mesetas de *makespan* subóptimas.

```
Entrada: pop_size, generations, mutation_rate, restart_threshold
población ← [shuffle(base_jobs) for _ in 1..pop_size]

for gen in 1..generations:
    evaluar fitness de cada individuo; actualizar mejor_sol

    if gens_sin_mejora  restart_threshold:
        población ← [élite + nuevos_aleatorios]
        continue

    nueva_población ← [elitismo de 2]
    while len(nueva_población) < pop_size:
        p1, p2 ← seleccionar_padres_torneo(k=3)
        hijo ← OX(p1, p2) if random < crossover_rate else p1
        nueva_población.append(mutar_swap(hijo))
    población ← nueva_población

return mejor_sol
```

Complejidad aproximada: $O(\text{generaciones} \cdot \text{población} \cdot C_{sched})$.

4 Fase 4: Implementación y análisis experimental

En esta fase se evalúa el desempeño computacional y la eficacia de los algoritmos implementados. El análisis se sustenta en los datos recolectados en los archivos `bruteforce_scaling.csv` (escalabilidad del método exacto) y `results.csv` (comparativa de rendimiento en 30 instancias de prueba).

4.1 Límite de computabilidad del algoritmo exacto (Fuerza Bruta)

El estudio de escalabilidad evidencia la naturaleza exponencial del problema de *scheduling* con restricciones de recursos, confirmando su clasificación como NP-hard.

- **Tamaño Máximo Resuelto:** Según los registros de ejecución, el algoritmo de fuerza bruta logró resolver de manera óptima instancias de hasta **12 trabajos** dentro de un tiempo razonable.
- **Comportamiento de la curva de tiempo:** Para instancias pequeñas ($n < 8$), los tiempos de respuesta son triviales (inferiores a 0.1s). Sin embargo, al superar los 10 trabajos, el tiempo de ejecución crece de forma exponencial debido a la explosión del espacio de búsqueda.

4.2 Instancias de prueba

Para validar el comportamiento del sistema y comparar el desempeño de las heurísticas frente al óptimo global, se diseñaron cinco instancias de prueba específicas y se generaron

25 instancias aleatorias, todas con una dimensión máxima de 11 trabajos. Esta reducción permite que el algoritmo de fuerza bruta obtenga una solución exacta en un tiempo razonable, sirviendo como métrica de error para el resto de los *solvers*. A continuación una explicación de cada una de las instancias específicas elaboradas.

1. **Resource Bottleneck (Mini):** Consta de 11 trabajos que compiten por un único recurso crítico con capacidad limitada (2). El objetivo es verificar si el algoritmo gestiona correctamente la ociosidad forzada en las máquinas cuando el factor limitante no es la capacidad de procesamiento, sino la disponibilidad de herramientas.
2. **High Contention (Mini):** Diseñada con una estructura de dependencia circular de recursos. Cada trabajo requiere un par de herramientas específicas, creando una red de incompatibilidades. Evalúa la capacidad de los algoritmos para encontrar combinaciones de trabajos que puedan ejecutarse en paralelo sin violar las restricciones acumulativas.
3. **The Rock & Sand (Mini):** Presenta un escenario de heterogeneidad extrema con un trabajo de muy larga duración ("la roca") que requiere un recurso exclusivo, frente a 10 trabajos cortos ("la arena") que no requieren recursos. La prueba mide si el *solver* es capaz de balancear la carga rellenando los huecos temporales de las máquinas libres mientras se procesa la tarea larga.
4. **Irrelevant Resources (Mini):** Actúa como escenario de control o *sanity check*. En esta instancia, la disponibilidad de recursos iguala al número de máquinas. El objetivo es confirmar que el algoritmo converge a una solución de *List Scheduling* clásica, donde el recurso deja de ser un cuello de botella y la optimización depende únicamente de la duración de los trabajos.
5. **Greedy Killer (Mini):** Un escenario contraintuitivo donde existen varios trabajos cortos que requieren herramientas y un trabajo de duración media libre de recursos. Está diseñada para penalizar heurísticas puramente voraces que, al intentar maximizar la ocupación inmediata de máquinas, podrían posponer innecesariamente el trabajo largo, resultando en un *makespan* subóptimo.

4.3 Comparativa de calidad de soluciones (Makespan)

Para evaluar la eficacia de los algoritmos propuestos, se realizó una comparativa utilizando como referencia los resultados del `BruteForceSolver`. Dado que este es un algoritmo exacto, su solución representa el óptimo global (C^*_{max}).

De las 30 instancias de prueba, el algoritmo de fuerza bruta logró resolver exitosamente **23 instancias** antes de exceder el tiempo límite de ejecución(10 minutos). El análisis de calidad se restringe a este subconjunto para garantizar la precisión del cálculo del error relativo, definido como $E_{rel} = (C_{algo} - C^*)/C^*$.

- **GeneticSolver:** Demostró una precisión excepcional, alcanzando un **error relativo medio del 0.0 %** respecto al óptimo en todas las instancias evaluadas. Esto indica que la configuración de la población y las tasas de mutación son suficientes para converger al óptimo global en problemas de dimensiones moderadas (hasta 11-12 trabajos).

- **EarliestStartSolver:** Al ser una heurística constructiva *greedy*, presenta una desviación mayor pero controlada, con un **error relativo medio del 3.52 %** y una desviación estándar del 7.09 %. Es capaz de encontrar el óptimo en escenarios de baja contención (error mediano de 0.0 %), pero su rendimiento se degrada ligeramente en instancias con alta competencia por recursos donde una decisión local inmediata perjudica el horizonte lejano del cronograma.

4.4 Análisis de tiempos de ejecución

El tiempo de cómputo es un factor crítico para la aplicabilidad del sistema en un entorno de producción real. Los resultados reflejan el compromiso entre la calidad de la solución y el esfuerzo computacional requerido.

Solver	Tiempo Medio (s)	Mediana (s)	Desv. Est. (s)
`EarliestStart`	0.0013	0.0008	0.0014
`GeneticSolver`	10.5228	8.8879	8.2910
`BruteForce`*	7.9481	0.0076	32.5693

Cuadro 1: Estadísticas de tiempo de ejecución por algoritmo (*Solo instancias exitosas).

1. **Eficiencia de la Heurística:** El `EarliestStartSolver` destaca por su rapidez extrema, resolviendo cualquier instancia en milisegundos. Esta característica lo posiciona como la herramienta ideal para escenarios de reprogramación dinámica ante imprevistos en la planta.
2. **Escalabilidad de la Metaheurística:** Aunque el `GeneticSolver` es más lento en promedio ($\approx 10{,}5s$), su tiempo de ejecución se mantiene estable incluso en las instancias donde la fuerza bruta falla. El coste computacional está dominado por el número de generaciones y el tamaño de la población, lo que permite predecir el tiempo de espera del usuario.
3. **Inviabilidad de la Fuerza Bruta:** El `BruteForceSolver` presenta una volatilidad extrema (desviación estándar de 32.5s). Mientras que resuelve instancias triviales instantáneamente, su tiempo escala de forma exponencial, alcanzando el estado de *timeout* (300s) en el 23.3 % de los casos probados, confirmando su limitación teórica.

4.5 Conclusión experimental

El análisis empírico realizado permite extraer conclusiones definitivas sobre la capacidad de respuesta y precisión del sistema frente a las necesidades de **TecnoPrecision Global LLC**:

- **Inviabilidad del Enfoque Exacto:** Los experimentos de escalabilidad confirman que el `BruteForceSolver` presenta una barrera infranqueable cerca de los $n = 11$ trabajos. El crecimiento de las combinaciones (m^n) y la complejidad de la validación de recursos hacen que el cálculo del óptimo global sea impracticable para las instancias de producción real (20-25 trabajos), donde el algoritmo superó consistentemente el tiempo de espera de 300 segundos.

- **Eficacia de la Metaheurística:** El `GeneticSolver` se posiciona como el algoritmo más equilibrado. Su capacidad para obtener un **error relativo del 0.0 %** en todas las instancias comparables, manteniendo tiempos de ejecución estables ($\approx$ 10 s), demuestra que la exploración del espacio de permutaciones mediante operadores genéticos es capaz de sortear las restricciones de recursos de manera mucho más eficiente que la búsqueda exhaustiva.
- **Robustez de las Heurísticas:** Aunque el `EarliestStartSolver` presenta un error marginal del 3.52 %, su velocidad de respuesta es inigualable. La proximidad de sus resultados al óptimo en la mayoría de las instancias sugiere que, para el conjunto de herramientas y máquinas actual, las decisiones *greedy* son una aproximación aceptable.

Recomendación Final: Se recomienda la integración del **GeneticSolver** como motor principal de planificación diaria, asegurando entregas fiables con el menor *makespan* posible. El **EarliestStartSolver** debe reservarse como un mecanismo de contingencia para re-planificación instantánea en caso de averías en las líneas de producción o cambios de última hora en los pedidos del cliente.

Referencias Bibliográficas

- Graham, R. L., Lawler, E. L., Lenstra, J. K., & Rinnooy Kan, A. H. G. (1979). *Optimization and approximation in deterministic sequencing and scheduling: A survey. Annals of Discrete Mathematics*, 5, 287–326. `https://doi.org/10.1016/S0167-5060(08)70356-X`
- Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.
- Garey, M. R., & Graham, R. L. (1975). *Bounds for Multiprocessor Scheduling with Resource Constraints. SIAM Journal on Computing*, 4(2).