



Instituto Politécnico Nacional

Escuela Superior de Cómputo

Ing. en Inteligencia Artificial



7BM1

**“Práctica 2. Algoritmo genético poliploide para planificación de tareas.”**

Docente:

Miriam Pescador Rojas

Equipo:

Diaz Ruiz Israel

Ignacio Cortés Atzin Maxela

30 de octubre 2025

# Marco Teórico

## Algoritmo Poliploide

Los algoritmos poliploides son una extensión de los algoritmos genéticos clásicos inspirada en la biología evolutiva, específicamente en el fenómeno de la poliploidía, que se refiere a la existencia de múltiples conjuntos de cromosomas dentro de un mismo organismo. En la naturaleza, la poliploidía confiere a las especies una mayor variabilidad genética, resistencia y capacidad de adaptación a cambios ambientales. De forma análoga, en los algoritmos evolutivos, la poliploidía permite representar cada individuo mediante varios cromosomas (en lugar de uno solo), incrementando la diversidad y la estabilidad del proceso evolutivo.

En un algoritmo genético tradicional, cada individuo posee una única cadena de genes que representa una posible solución al problema, y su desempeño se evalúa directamente sobre esa estructura. En contraste, un algoritmo poliploide asocia a cada individuo con múltiples cromosomas, de los cuales uno o varios pueden contribuir a la expresión fenotípica, es decir, a la solución que se evalúa en cada iteración. Este mecanismo de dominancia genética permite conservar información genética “oculta” en cromosomas no expresados, la cual puede manifestarse posteriormente si las condiciones del entorno o la presión de selección cambian.

Esta característica otorga a los algoritmos poliploides varias ventajas importantes respecto a los algoritmos genéticos clásicos. En primer lugar, aumentan la diversidad genética de la población, evitando la convergencia prematura hacia óptimos locales. En segundo lugar, mejoran la capacidad de adaptación ante entornos dinámicos o ruidosos, ya que los genes no expresados pueden ofrecer alternativas evolutivas en generaciones futuras. En tercer lugar, incrementan la robustez del proceso evolutivo, al permitir la coexistencia de soluciones potencialmente útiles sin necesidad de reiniciar la búsqueda.

Gracias a estas propiedades, los algoritmos poliploides se utilizan ampliamente en problemas de optimización compleja, entornos dinámicos, modelos multiobjetivo y sistemas de aprendizaje adaptativo, donde la estabilidad y la diversidad de las soluciones son factores determinantes para alcanzar resultados óptimos y generalizables.

# Desarrollo de la actividad

En esta actividad se implemento un algoritmo genético poliploide basado en el enfoque NSGA-II (Non-Dominated Sorting Genetic Algorithm II) para resolver un problema de planificación de tareas. El objetivo principal es optimizar simultáneamente el tiempo total de ejecución (makespan) y el consumo energético, considerando múltiples políticas de atención de trabajos.

El diseño se inspira en la naturaleza poliploide de los organismos biológicos, donde cada individuo posee varios conjuntos de cromosomas. En este contexto, cada individuo de la población está compuesto por seis cromosomas, y cada uno representa una política de planificación distinta:

- **FIFO (First In, First Out):** Atiende las tareas en el orden en que llegan al sistema. Es una estrategia sencilla en la que no se asignan prioridades dinámicas; el primer trabajo que entra es el primero en ser ejecutado. Su principal ventaja es la simplicidad de implementación y su comportamiento justo en cuanto al orden de llegada. Sin embargo, puede generar largos tiempos de espera promedio si las primeras tareas en la cola requieren mucho tiempo de procesamiento (efecto convoy).

```
•     fifo_order = []
•         for job_id in sorted(self.jobs.keys()):
•             for op_idx in range(len(self.jobs[job_id])):
•                 fifo_order.append((job_id, op_idx))
•
•     orders['FIFO'] = fifo_order
```

- **LTP (Long Time Processing):** La política LTP (Long Time Processing) da prioridad a las operaciones con mayor tiempo de procesamiento promedio. En este enfoque, las tareas que tardan más en completarse se ejecutan primero. Esta política busca minimizar los tiempos de espera de los trabajos más largos, reduciendo el riesgo de que queden pendientes por demasiado tiempo. No obstante, puede afectar negativamente a las tareas más cortas, las cuales podrían quedar retrasadas.

```
•     operations_with_time = []
•         for job_id, operations in self.jobs.items():
•             for op_idx, operation in enumerate(operations):
•                 avg_time = np.mean(self.processing_times[operation])
•                 operations_with_time.append((job_id, op_idx,
•
•                 avg_time))
•
•         # Ordenar por tiempo promedio descendente (mayor tiempo
•         primero)
•         operations_with_time.sort(key=lambda x: x[2], reverse=True)
```

- # Crear orden manteniendo restricciones de precedencia
- ltp\_order =
 

```
self._apply_precedence_constraints(operations_with_time)
```
- orders['LTP'] = ltp\_order
- **STP (Short Time Processing):** La política STP (Short Time Processing) es la contraparte de LTP. En este caso, las operaciones con menor tiempo de procesamiento promedio son atendidas primero. El objetivo principal es reducir el tiempo promedio de espera y mejorar la eficiencia global del sistema, ejecutando rápidamente las tareas más cortas. Sin embargo, puede provocar que las tareas más largas sufran starvation (falta de atención prolongada), especialmente si llegan continuamente tareas cortas.

```
• operations_with_time.sort(key=lambda x: x[2])
• stp_order =
  self._apply_precedence_constraints(operations_with_time)
• orders['STP'] = stp_order
```

- **RRFIFO (Round Robin + FIFO):** La política RRFIFO (Round Robin + FIFO) combina el principio Round Robin (asignación cíclica) con la prioridad FIFO. En esta estrategia, las operaciones de los distintos trabajos se atienden de manera alternada, siguiendo un ciclo entre los trabajos disponibles, pero respetando el orden de llegada interno de las operaciones de cada uno. Este enfoque permite distribuir equitativamente los recursos entre los trabajos, evitando bloqueos y manteniendo un flujo de ejecución más equilibrado entre tareas de distinta duración.

```
• rrfifo_order = []
• job_pointers = {j: 0 for j in self.jobs.keys()} # Puntero de operación por trabajo
•
• # Mientras haya operaciones pendientes
• while any(job_pointers[j] < len(self.jobs[j]) for j in self.jobs.keys()):
•     for job_id in sorted(self.jobs.keys()):
•         # Si el trabajo aún tiene operaciones pendientes
•         if job_pointers[job_id] < len(self.jobs[job_id]):
•             rrfifo_order.append((job_id,
•                  job_pointers[job_id]))
•             job_pointers[job_id] += 1
• orders['RRFIFO'] = rrfifo_order
```

- **RRLTP (Round Robin + Long Time Processing):** La política RRLTP (Round Robin + Long Time Processing) aplica el esquema Round Robin, pero priorizando los trabajos que poseen operaciones con mayor tiempo promedio de procesamiento. Primero se establece un orden de los trabajos según su carga de trabajo (de mayor a menor tiempo promedio), y posteriormente se ejecutan de forma alternada siguiendo ese orden. Esta política equilibra la equidad del Round Robin con la necesidad de dar prioridad a las tareas más costosas, evitando que se acumulen al final del ciclo de atención.

```

• job_avg_times = []
•     for job_id, operations in self.jobs.items():
•         total_time = sum(np.mean(self.processing_times[op]) for op
in operations)
•         job_avg_times.append((job_id, total_time /
len(operations)))
•
•     # Ordenar trabajos por tiempo promedio descendente
•     job_avg_times.sort(key=lambda x: x[1], reverse=True)
•     job_order = [j[0] for j in job_avg_times]
•
•     # Aplicar Round Robin con este orden
•     rrltp_order = []
•     job_pointers = {j: 0 for j in self.jobs.keys()}
•
•     while any(job_pointers[j] < len(self.jobs[j]) for j in
self.jobs.keys()):
•         for job_id in job_order:
•             if job_pointers[job_id] < len(self.jobs[job_id]):
•                 rrltp_order.append((job_id, job_pointers[job_id]))
•                 job_pointers[job_id] += 1
•
orders['RRLTP'] = rrltp_order

```

- **RRECA (Round Robin + Energy Consumption Average):** La política RRECA (Round Robin + Energy Consumption Average) introduce un criterio energético dentro del esquema Round Robin. En este caso, los trabajos se ordenan de acuerdo con su consumo energético promedio, dando prioridad a aquellos que requieren menor energía. El objetivo de esta política es optimizar la eficiencia energética global del sistema, reduciendo el consumo total sin sacrificar significativamente el rendimiento temporal. Esta estrategia es particularmente útil en entornos donde la sostenibilidad y el ahorro energético son factores relevantes de optimización.

```

job_avg_energy = []
for job_id, operations in self.jobs.items():
    total_energy = sum(np.mean(self.energy_consumption[op]) for op in operations)
    job_avg_energy.append((job_id, total_energy / len(operations)))

# Ordenar trabajos por consumo energético promedio ascendente
job_avg_energy.sort(key=lambda x: x[1])
job_order = [j[0] for j in job_avg_energy]

# Aplicar Round Robin con este orden
rreca_order = []
job_pointers = {j: 0 for j in self.jobs.keys()}

while any(job_pointers[j] < len(self.jobs[j]) for j in self.jobs.keys()):
    for job_id in job_order:
        if job_pointers[job_id] < len(self.jobs[job_id]):
            rreca_order.append((job_id, job_pointers[job_id]))
            job_pointers[job_id] += 1
orders['RRECA'] = rreca_order

return orders

```

Cada cromosoma contiene una cadena de genes que codifican la asignación de operaciones a máquinas, y su longitud corresponde al número total de operaciones del sistema. De este modo, el individuo poliploide representa de forma simultánea distintas estrategias de ejecución posibles sobre el mismo conjunto de tareas.

La clase *JobShopData* gestiona toda la información del entorno de planificación:

- Número de máquinas disponibles (4).
- Conjunto de trabajos y las operaciones que los componen.

J1: {O2, O4, O5}

J2: {O1, O3, O5}

J3: {O1, O2, O3, O4, O5}

J4: {O4, O5}

J5: {O2, O4}

J6: {O1, O2, O4, O5}

- Se utilizaron las tablas de tiempos de procesamiento y consumos energéticos, dadas en el ejercicio
- Cálculo automático de los órdenes de atención para cada política mediante el método `_calculate_policy_orders()`.

```

class JobShopData:
    def __init__(self):
        """
        Inicializa los datos del problema según las tablas del documento.
        """
        # Número de máquinas disponibles
        self.num_machines = 4

        # Número de operaciones totales
        self.num_operations = 5

        # Definición de trabajos y sus operaciones
        # Cada trabajo es una lista de operaciones (índice base 0)
        self.jobs = {
            1: [1, 3, 4],           # J1: {02, 04, 05}
            2: [0, 2, 4],           # J2: {01, 03, 05}
            3: [0, 1, 2, 3, 4],     # J3: {01, 02, 03, 04, 05}
            4: [3, 4],              # J4: {04, 05}
            5: [1, 3],              # J5: {02, 04}
            6: [0, 1, 3, 4]         # J6: {01, 02, 04, 05}
        }

        # Número total de trabajos
        self.num_jobs = len(self.jobs)

        # Tabla de tiempos: tiempo[operacion][maquina]
        # Tiempos de procesamiento por operación en cada máquina
        self.processing_times = np.array([
            [3.5, 6.7, 2.5, 8.2],   # 01
            [5.5, 4.2, 7.6, 9.0],   # 02
            [6.1, 7.3, 5.5, 6.7],   # 03
            [4.8, 5.3, 3.8, 4.7],   # 04
            [3.8, 3.4, 4.2, 3.6]    # 05
        ])

```

```

# Tabla de consumo energético: energy[operacion][maquina]
# Consumo energético por operación en cada máquina
self.energy_consumption = np.array([
    [1.2, 4.7, 3.5, 4.2],   # 01
    [7.5, 1.5, 6.6, 3.5],   # 02
    [1.1, 5.3, 8.5, 1.7],   # 03
    [7.8, 3.3, 8.8, 9.7],   # 04
    [1.9, 5.9, 7.5, 3.6]    # 05
])

# Calcular el número total de operaciones a planificar
self.total_operations = sum(len(ops) for ops in self.jobs.values())

# Nombres de las políticas
self.policy_names = ['FIFO', 'LTP', 'STP', 'RRFIFO', 'RRLTP', 'RRECA']

# Calcular los órdenes de atención según cada política
self.policy_orders = self._calculate_policy_orders()

def _calculate_policy_orders(self) -> Dict[str, List[Tuple[int, int]]]:

```

La clase *Individual* modela a cada solución candidata.

Cada individuo contiene:

- Un diccionario chromosomes con los seis cromosomas, uno por política.
- Un conjunto de valores objetivos (objectives), que guardan el makespan y el consumo energético total calculados para cada política.
- Métricas evolutivas como rango de no-dominancia (rank) y distancia de crowding.

El método \_evaluate() calcula automáticamente los dos objetivos aplicando la secuencia de operaciones definida por cada política y considerando las restricciones de precedencia.

El tiempo total de finalización se obtiene como el máximo de los tiempos de las máquinas (makespan), y el consumo energético total se calcula sumando los consumos de todas las máquinas.

```
class Individual:  
    def __init__(self, data: JobShopData, chromosomes: Dict[str, np.ndarray] = None):  
        self.data = data  
        # Si no se proporcionan cromosomas, generar aleatoriamente  
        if chromosomes is None:  
            self.chromosomes = self._generate_random_chromosomes()  
        else:  
            self.chromosomes = chromosomes  
  
        # Inicializar métricas  
        self.objectives = {} # {policy: (makespan, energy)}  
        self.rank = float('inf') # Nivel de no-dominancia  
        self.crowding_distance = {} # {policy: distance}  
  
        # Calcular objetivos para cada política  
        self._evaluate()
```

La clase *GeneticOperators* define tres operadores fundamentales:

- Cruza uniforme poliploide (uniform\_crossover\_polyplloid):  
Combina aleatoriamente los genes de los cromosomas de ambos padres usando un booleano diferente para cada política, generando así dos hijos con combinaciones híbridas de asignaciones.

```
class GeneticOperators:  
    @staticmethod  
    def uniform_crossover_polyplloid(parent1: Individual, parent2: Individual,  
                                       data: JobShopData) -> Tuple[Individual, Individual]:  
        child1_chromosomes = {}  
        child2_chromosomes = {}  
  
        for policy in data.policy_names:  
            # Crear máscara aleatoria (True = tomar de parent1, False = tomar de parent2)  
            mask = np.random.rand(data.total_operations) < 0.5  
  
            # Crear cromosomas de los hijos  
            child1_chrom = np.where(mask, parent1.chromosomes[policy],  
                                   parent2.chromosomes[policy])  
            child2_chrom = np.where(mask, parent2.chromosomes[policy],  
                                   parent1.chromosomes[policy])  
  
            child1_chromosomes[policy] = child1_chrom  
            child2_chromosomes[policy] = child2_chrom  
  
        # Crear nuevos individuos  
        child1 = Individual(data, child1_chromosomes)  
        child2 = Individual(data, child2_chromosomes)  
  
        return child1, child2
```

- Mutación inter-cromosoma (inter\_chromosome\_mutation):  
Intercambia cromosomas completos entre diferentes políticas dentro del mismo individuo, simulando la transferencia de información genética entre conjuntos de cromosomas homólogos.

```
@staticmethod
def inter_chromosome_mutation(individual: Individual, data: JobShopData):
    # Seleccionar aleatoriamente 2 o 3 políticas
    num_swaps = random.choice([2, 3])
    policies_to_swap = random.sample(data.policy_names, num_swaps)

    # Intercambiar cromosomas de forma circular
    if num_swaps == 2:
        # Intercambio simple entre dos políticas
        p1, p2 = policies_to_swap
        individual.chromosomes[p1], individual.chromosomes[p2] = \
            individual.chromosomes[p2].copy(), individual.chromosomes[p1].copy()
    else: # num_swaps == 3
        # p1 -> p2, p2 -> p3, p3 -> p1
        p1, p2, p3 = policies_to_swap
        temp = individual.chromosomes[p1].copy()
        individual.chromosomes[p1] = individual.chromosomes[p2].copy()
        individual.chromosomes[p2] = individual.chromosomes[p3].copy()
        individual.chromosomes[p3] = temp

    # Re-evaluar objetivos
    individual._evaluate()
```

- Mutación por intercambio recíproco y desplazamiento:  
Introducen variaciones locales intercambiando pares de genes o moviendo segmentos de operaciones dentro de cada cromosoma, favoreciendo la exploración del espacio de soluciones.

```
@staticmethod
def reciprocal_exchange_mutation(individual: Individual, data: JobShopData,
                                 num_swaps: int = 2):
    # Para cada cromosoma
    for policy in data.policy_names:
        chromosome = individual.chromosomes[policy]

        # Realizar num_swaps intercambios
        for _ in range(num_swaps):
            # Seleccionar dos posiciones aleatorias
            pos1, pos2 = random.sample(range(data.total_operations), 2)

            # Intercambiar valores
            chromosome[pos1], chromosome[pos2] = chromosome[pos2], chromosome[pos1]

    # Re-evaluar objetivos
    individual._evaluate()
```

```

@staticmethod
def displacement_mutation(individual: Individual, data: JobShopData,
                           segment_length: int = 3):
    # Para cada cromosoma
    for policy in data.policy_names:
        chromosome = individual.chromosomes[policy]

        # Seleccionar posición inicial del segmento
        start_pos = random.randint(0, data.total_operations - segment_length)

        # Extraer segmento
        segment = chromosome[start_pos:start_pos + segment_length].copy()

        # Seleccionar nueva posición (diferente de la actual)
        possible_positions = list(range(0, start_pos)) + \
            list(range(start_pos + segment_length,
                       data.total_operations - segment_length + 1))

        if possible_positions:
            new_pos = random.choice(possible_positions)

            # Eliminar segmento de posición original
            remaining = np.delete(chromosome, range(start_pos, start_pos + segment_length))

            # Insertar en nueva posición
            chromosome = np.insert(remaining, new_pos, segment)

            individual.chromosomes[policy] = chromosome

    # Re-evaluar objetivos
    individual._evaluate()

```

Cada mutación reevalúa al individuo inmediatamente, manteniendo la coherencia de los objetivos.

La clase *PolyploidNSGAI* implementa el flujo principal del algoritmo evolutivo. Las etapas son:

Se generan individuos aleatorios, garantizando diversidad inicial. Donde cada individuo calcula su desempeño en las seis políticas, mediante el algoritmo rápido de NSGA-II se construyen los frentes de Pareto, agrupando las soluciones no dominadas y con crowding distance se puede medir la densidad de soluciones cercanas para mantener la diversidad del frente, se aplica selección, cruza y mutación a los operadores genéticos con las probabilidades establecidas, generando nuevas soluciones.

Usamos la selección por torneo binario para elegir los individuos más aptos de la generación actual para participar en la cruxa y mutación, asegurando así que las características genéticas más favorables se propaguen hacia las generaciones siguientes.

Para la actualización del frente de Pareto, se almacenan los resultados por política y generación, junto con el hipervolumen de dominancia alcanzado.

```

class PolyploidNSGAI:

    def __init__(self, data: JobShopData, population_size: int = 20,
                 generations: int = 100, crossover_rate: float = 0.8,
                 mutation_rates: Dict[str, float] = None, seed: int = None):
        self.data = data
        self.population_size = population_size
        self.generations = generations
        self.crossover_rate = crossover_rate

        # Tasas de mutación por defecto
        if mutation_rates is None:
            self.mutation_rates = {
                'inter_chromosome': 0.3,
                'reciprocal_exchange': 0.2,
                'displacement': 0.1
            }
        else:
            self.mutation_rates = mutation_rates

        # Establecer semilla si se proporciona
        if seed is not None:
            np.random.seed(seed)
            random.seed(seed)

        # Inicializar población
        self.population = []

        # Estadísticas de mutaciones
        self.mutation_stats = {
            'inter_chromosome': 0,
            'reciprocal_exchange': 0,
            'displacement': 0
        }

```

```

def tournament_selection_with_chromosome_exchange(self,
                                                population: List[Individual]) -> Individual:
    """
    Selección por torneo binario con intercambio cromosómico.
    Dos individuos compiten y se recombinan basándose en dominancia y crowding.
    """
    # Seleccionar dos individuos aleatoriamente
    ind1, ind2 = random.sample(population, 2)

    # Crear nuevo individuo seleccionando mejor cromosoma por política
    new_chromosomes = {}

    for policy in self.data.policy_names:
        # Comparar basándose en nivel de dominancia y crowding distance
        # (en este contexto usamos los objetivos directamente)

        # Determinar mejor cromosoma
        obj1 = ind1.objectives[policy]
        obj2 = ind2.objectives[policy]

        # Calcular frentes para determinar nivel de dominancia
        temp_pop = [ind1, ind2]
        fronts = self.fast_non_dominated_sort(temp_pop, policy)

        # Asignar ranks
        for rank, front in enumerate(fronts):
            for ind in front:
                ind.rank = rank

```

A continuación se muestran los resultados de las 6 políticas con la frente de Pareto cercana a la rodilla y los diagramas de Gantt enfocados en ENERGÍA.

# Resultados

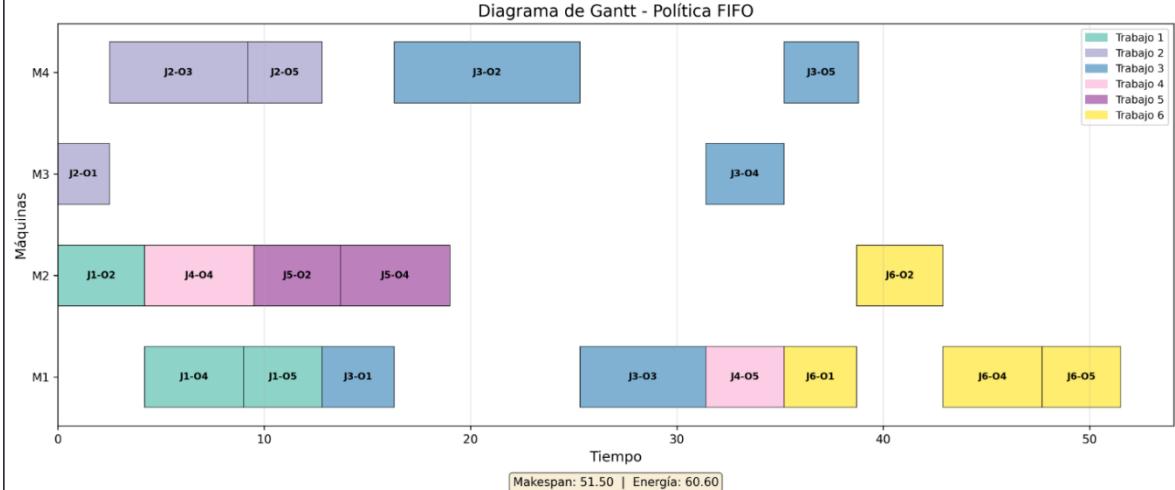
Para Fifo

Política utilizada: FIFO  
Makespan: 51.50  
Consumo Energético Total: 60.60

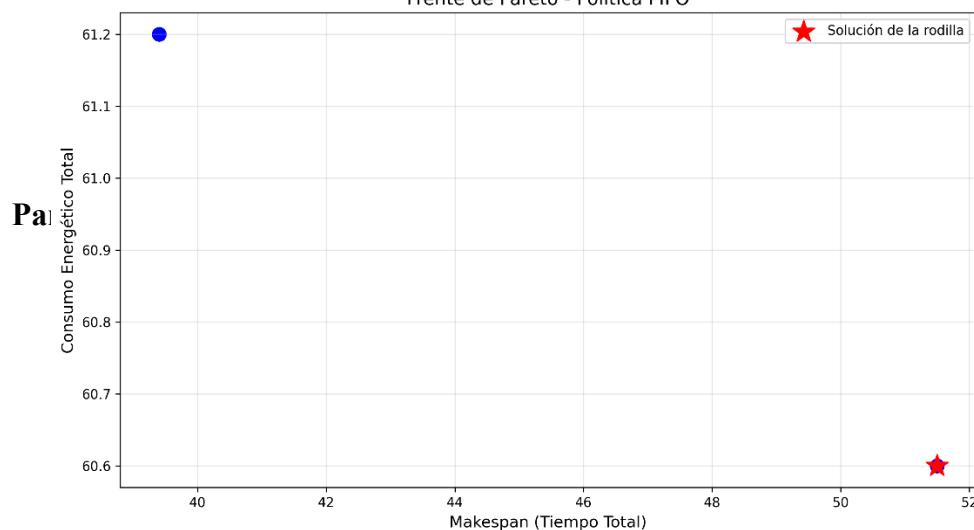
## ASIGNACIÓN DE OPERACIONES

Posición	Trabajo	Operación	Máquina	Tiempo	Energía
1	J1	02	M2	4.20	1.50
2	J1	04	M1	4.80	7.80
3	J1	05	M1	3.80	1.90
4	J2	01	M3	2.50	3.50
5	J2	03	M4	6.70	1.70
6	J2	05	M4	3.60	3.60
7	J3	01	M1	3.50	1.20
8	J3	02	M4	9.00	3.50
9	J3	03	M1	6.10	1.10
10	J3	04	M3	3.80	8.80
11	J3	05	M4	3.60	3.60
12	J4	04	M2	5.30	3.30
13	J4	05	M1	3.80	1.90
14	J5	02	M2	4.20	1.50
15	J5	04	M2	5.30	3.30
16	J6	01	M1	3.50	1.20
17	J6	02	M2	4.20	1.50
18	J6	04	M1	4.80	7.80
19	J6	05	M1	3.80	1.90

Diagrama de Gantt - Política FIFO



Frente de Pareto - Política FIFO



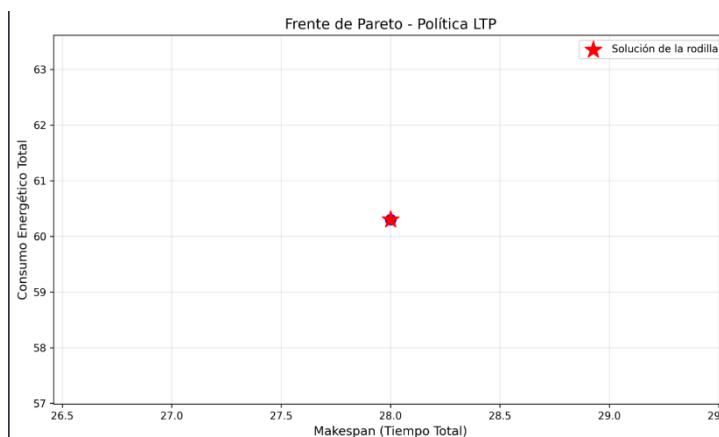
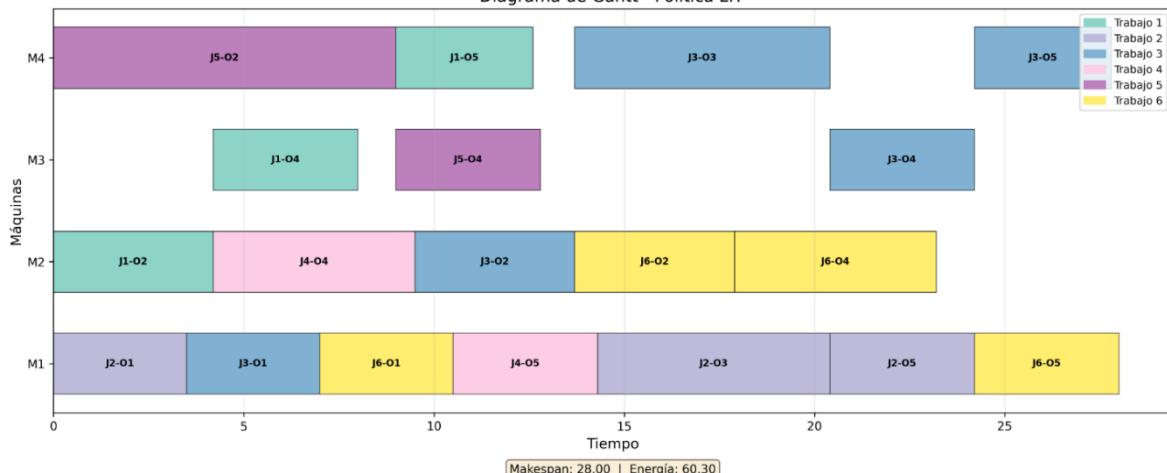
**Política utilizada: LTP**  
**Makespan: 28.00**  
**Consumo Energético Total: 60.30**

## Para LTP

### ASIGNACIÓN DE OPERACIONES

Posición	Trabajo	Operación	Máquina	Tiempo	Energía
1	J1	02	M2	4.20	1.50
2	J5	02	M4	9.00	3.50
3	J2	01	M1	3.50	1.20
4	J3	01	M1	3.50	1.20
5	J6	01	M1	3.50	1.20
6	J1	04	M3	3.80	8.80
7	J4	04	M2	5.30	3.30
8	J5	04	M3	3.80	8.80
9	J1	05	M4	3.60	3.60
10	J4	05	M1	3.80	1.90
11	J3	02	M2	4.20	1.50
12	J6	02	M2	4.20	1.50
13	J2	03	M1	6.10	1.10
14	J3	03	M4	6.70	1.70
15	J3	04	M3	3.80	8.80
16	J6	04	M2	5.30	3.30
17	J2	05	M1	3.80	1.90
18	J3	05	M4	3.60	3.60
19	J6	05	M1	3.80	1.90

Diagrama de Gantt - Política LTP



Política utilizada: RRECA

Makespan: 30.80

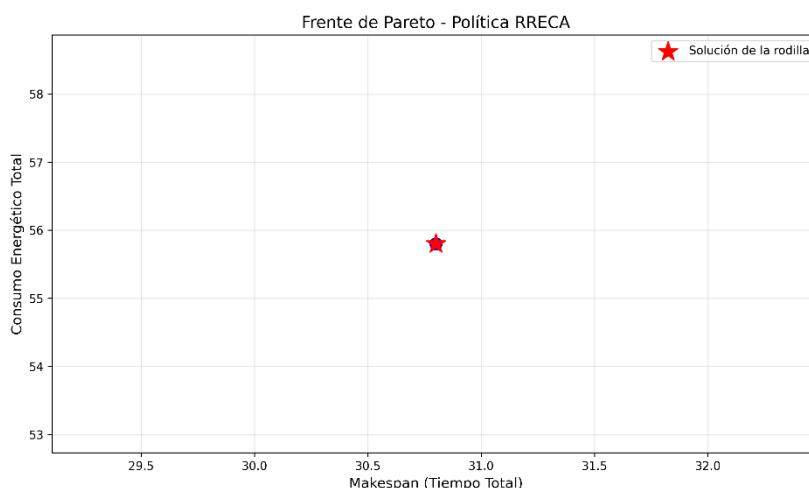
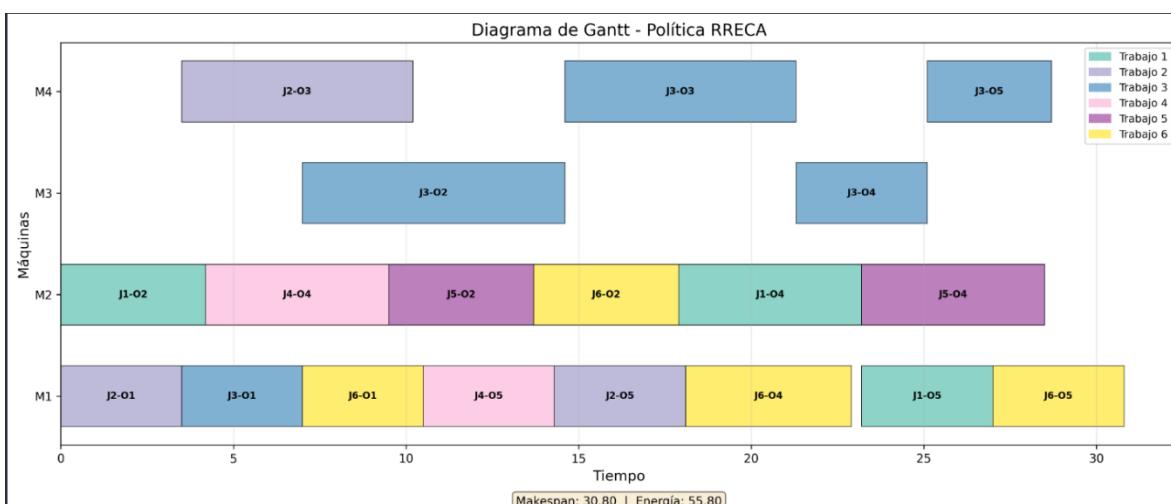
Consumo Energético Total: 55.80

## Para RRECA

### ASIGNACIÓN DE OPERACIONES

Posición	Trabajo	Operación	Máquina	Tiempo	Energía
1	J2	01	M1	3.50	1.20
2	J3	01	M1	3.50	1.20
3	J6	01	M1	3.50	1.20
4	J1	02	M2	4.20	1.50
5	J4	04	M2	5.30	3.30
6	J5	02	M2	4.20	1.50
7	J2	03	M4	6.70	1.70
8	J3	02	M3	7.60	6.60
9	J6	02	M2	4.20	1.50
10	J1	04	M2	5.30	3.30
11	J4	05	M1	3.80	1.90
12	J5	04	M2	5.30	3.30
13	J2	05	M1	3.80	1.90
14	J3	03	M4	6.70	1.70
15	J6	04	M1	4.80	7.80
16	J1	05	M1	3.80	1.90
17	J3	04	M3	3.80	8.80
18	J6	05	M1	3.80	1.90
19	J3	05	M4	3.60	3.60

=====



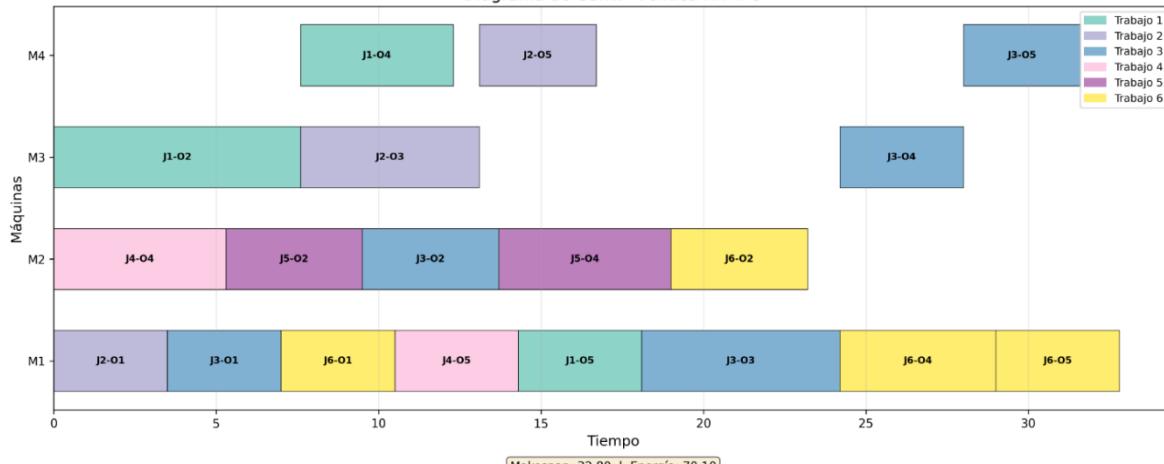
Política utilizada: RRFIFO  
 Makespan: 32.80  
 Consumo Energético Total: 70.10

## Para RRFIFO

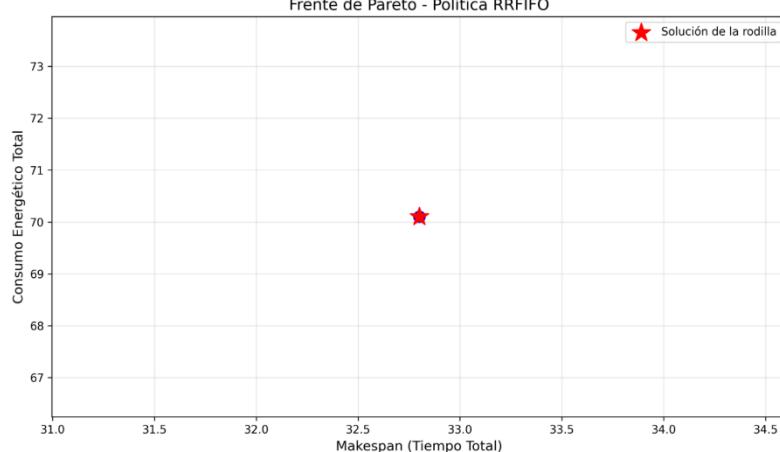
### ASIGNACIÓN DE OPERACIONES

Posición	Trabajo	Operación	Máquina	Tiempo	Energía
1	J1	02	M3	7.60	6.60
2	J2	01	M1	3.50	1.20
3	J3	01	M1	3.50	1.20
4	J4	04	M2	5.30	3.30
5	J5	02	M2	4.20	1.50
6	J6	01	M1	3.50	1.20
7	J1	04	M4	4.70	9.70
8	J2	03	M3	5.50	8.50
9	J3	02	M2	4.20	1.50
10	J4	05	M1	3.80	1.90
11	J5	04	M2	5.30	3.30
12	J6	02	M2	4.20	1.50
13	J1	05	M1	3.80	1.90
14	J2	05	M4	3.60	3.60
15	J3	03	M1	6.10	1.10
16	J6	04	M1	4.80	7.80
17	J3	04	M3	3.80	8.80
18	J6	05	M1	3.80	1.90
19	J3	05	M4	3.60	3.60

Diagrama de Gantt - Política RRFIFO



Frente de Pareto - Política RRFIFO



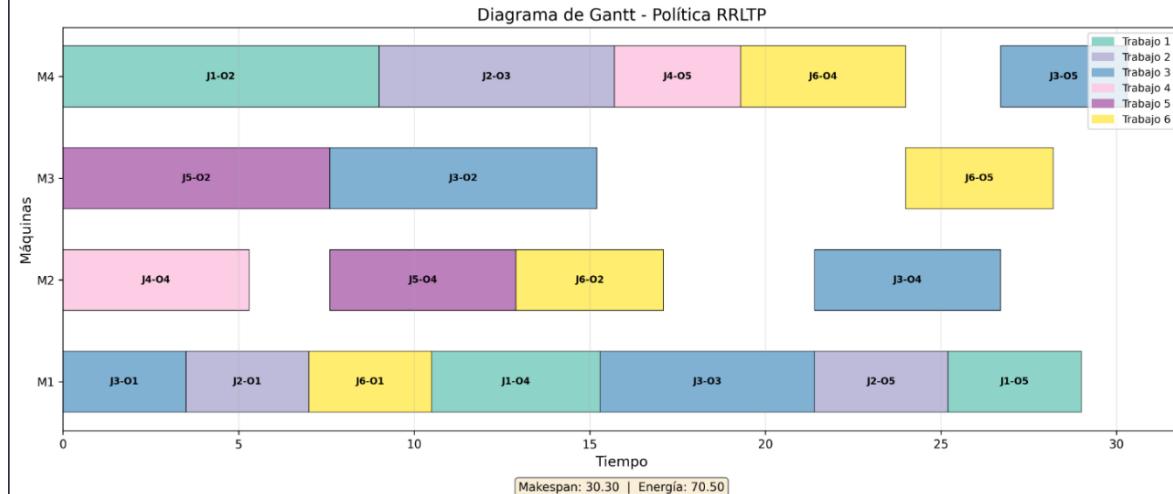
Política utilizada: RRLTP  
 Makespan: 30.30  
 Consumo Energético Total: 70.50

## Para RRLTP

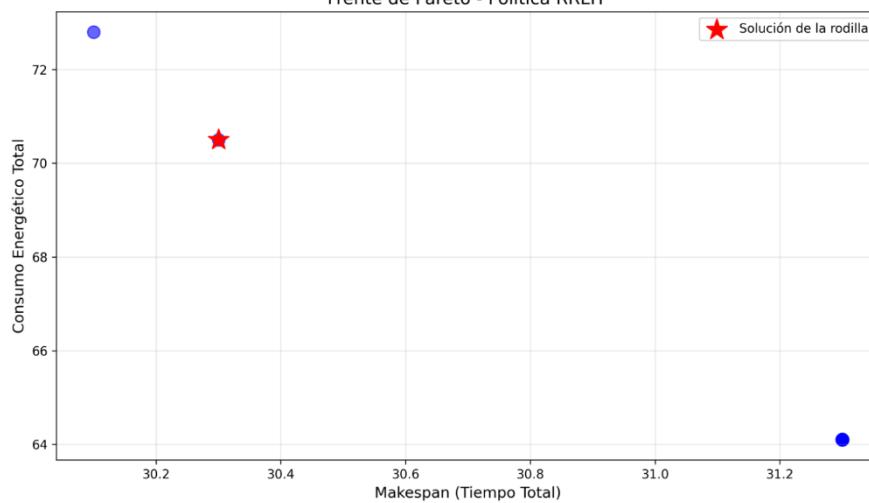
### ASIGNACIÓN DE OPERACIONES

Posición	Trabajo	Operación	Máquina	Tiempo	Energía
1	J5	02	M3	7.60	6.60
2	J3	01	M1	3.50	1.20
3	J2	01	M1	3.50	1.20
4	J6	01	M1	3.50	1.20
5	J1	02	M4	9.00	3.50
6	J4	04	M2	5.30	3.30
7	J5	04	M2	5.30	3.30
8	J3	02	M3	7.60	6.60
9	J2	03	M4	6.70	1.70
10	J6	02	M2	4.20	1.50
11	J1	04	M1	4.80	7.80
12	J4	05	M4	3.60	3.60
13	J3	03	M1	6.10	1.10
14	J2	05	M1	3.80	1.90
15	J6	04	M4	4.70	9.70
16	J1	05	M1	3.80	1.90
17	J3	04	M2	5.30	3.30
18	J6	05	M3	4.20	7.50
19	J3	05	M4	3.60	3.60

Diagrama de Gantt - Política RRLTP



Frente de Pareto - Política RRLTP



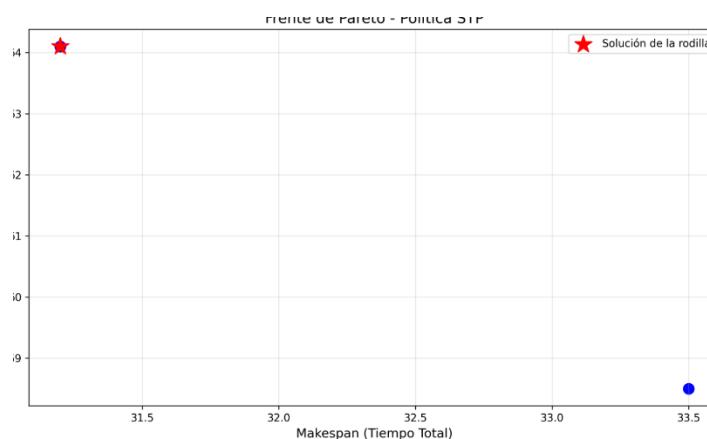
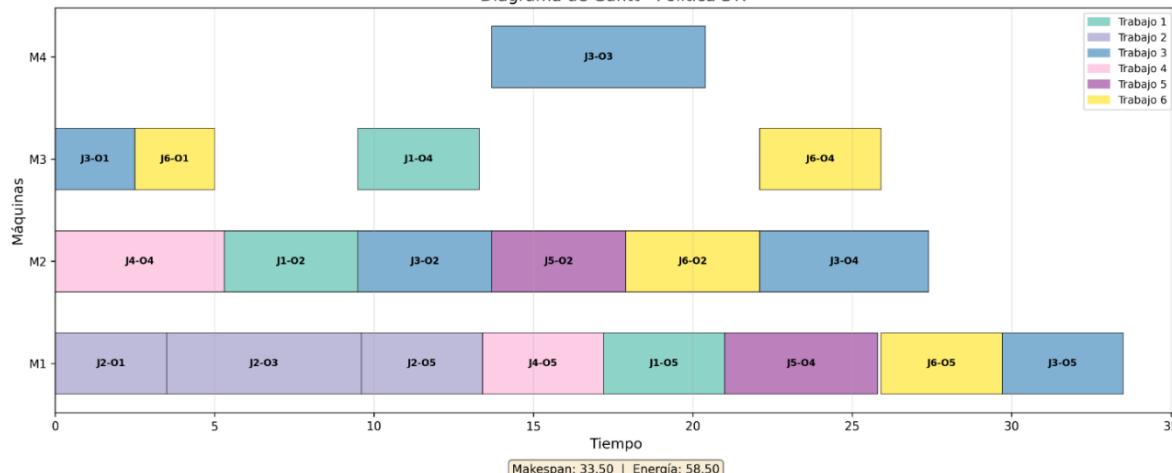
**Política utilizada:** STP  
**Makespan:** 31.20  
**Consumo Energético Total:** 64.10

## Para STP

### ASIGNACIÓN DE OPERACIONES

Posición	Trabajo	Operación	Máquina	Tiempo	Energía
1	J4	04	M2	5.30	3.30
2	J2	01	M1	3.50	1.20
3	J3	01	M3	2.50	3.50
4	J6	01	M3	2.50	3.50
5	J2	03	M1	6.10	1.10
6	J1	02	M2	4.20	1.50
7	J3	02	M2	4.20	1.50
8	J5	02	M2	4.20	1.50
9	J6	02	M2	4.20	1.50
10	J2	05	M1	3.80	1.90
11	J4	05	M1	3.80	1.90
12	J1	04	M3	3.80	8.80
13	J1	05	M1	3.80	1.90
14	J5	04	M1	4.80	7.80
15	J6	04	M3	3.80	8.80
16	J6	05	M3	4.20	7.50
17	J3	03	M4	6.70	1.70
18	J3	04	M2	5.30	3.30
19	J3	05	M1	3.80	1.90

Diagrama de Gantt - Política STP



## Estadísticas para Hipervolumen

política	FIFO			LTP			STP		
Generación	min	max	Prom (desv)	min	max	Prom (desv)	min	max	Prom (desv)
20	290.19	755.86	434.29 (144.51)	245.00	501.77	345.73 (88.61)	193.20	885.65	369.82 (190.65)
40	215.20	401.20	297.80 (57.41)	193.82	368.20	265.60 (48.25)	171.02	535.57	312.85 (107.48)
60	215.20	381.55	306.48 (57.45)	168.84	303.52	240.11 (42.46)	202.83	409.07	265.06 (58.86)
80	215.20	862.70	362.20 (183.79)	168.84	338.52	238.70 (56.68)	218.02	1113.65	376.67 (263.39)
100	215.20	417.79	291.20 (62.40)	168.84	575.98	291.49 (147.20)	171.02	362.99	221.64 (54.37)

política	RR-FIFO			RR-LTP			RR-ACE		
	min	max	Prom (desv)	min	max	Prom (desv)	min	max	Prom (desv)
Generación									
20	218.67	600.64	317.79 (103.77)	186.48	440.35	280.83 (73.27)	214.62	365.91	285.78 (48.32)
40	202.54	849.80	343.21 (185.15)	192.06	329.72	241.39 (40.99)	178.95	294.20	235.11 (40.89)
60	192.97	543.63	271.46 (108.22)	192.06	496.58	256.65 (83.71)	186.69	880.10	295.35 (200.12)
80	195.86	500.49	248.96 (85.58)	192.06	355.50	238.42 (47.27)	171.86	299.37	221.94 (36.14)
100	194.89	332.47	232.11 (38.91)	176.46	261.41	216.54 (27.68)	164.86	208.86	190.44 (14.05)

# Respuestas de las preguntas

- a. ¿Hay evidencia de que una política sea más efectiva que otra?, explique por qué.

Basándonos en el hipervolumen promedio en la generación 100.

La política LTP muestra mejor desempeño en términos de hipervolumen, lo que indica que encuentra un frente de Pareto con mejor convergencia y diversidad. Sin embargo, la efectividad de cada política puede depender del contexto: si se prioriza el makespan o la energía, algunas políticas pueden ser más adecuadas que otras.

- b. ¿Cuál fue el operador de mutación que tuvo mejor desempeño en las pruebas y por qué consider que es mejor?

La mutación inter-cromosoma se aplicó más frecuentemente y es particularmente efectiva en este contexto poliploide porque permite intercambiar estrategias completas entre políticas, lo que genera diversidad significativa en el espacio de búsqueda. La mutación por intercambio recíproco proporciona refinamiento local, mientras que el desplazamiento mantiene la estructura, pero explora diferentes secuencias.

- c. Explique si empleo otros valores de parámetros diferentes a los propuestos, diga qué valores y explique por qué se tuvieron que cambiar, si no cambio los valores explique qué parámetros tienen mayor efecto en la convergencia y distribución de las soluciones encontradas en el frente de Pareto.

- Tamaño de población: 20 individuos, esto es de acuerdo al tamaño del cromosoma y para que la convergencia sea más rápida.

- Generaciones: 100, para el espacio de búsqueda

- Tasa de crusa: 0.8, para que no se quede estancado

- Probabilidad mutación inter-cromosoma: 0.3

- Probabilidad mutación intercambio recíproco: 0.2

- Probabilidad mutación desplazamiento: 0.1

Para la exploración global.

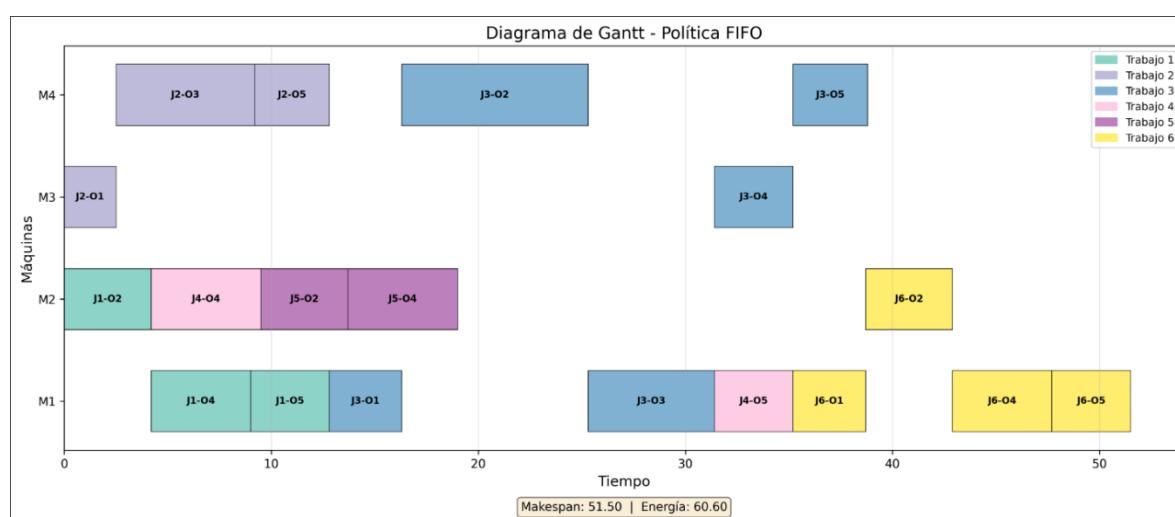
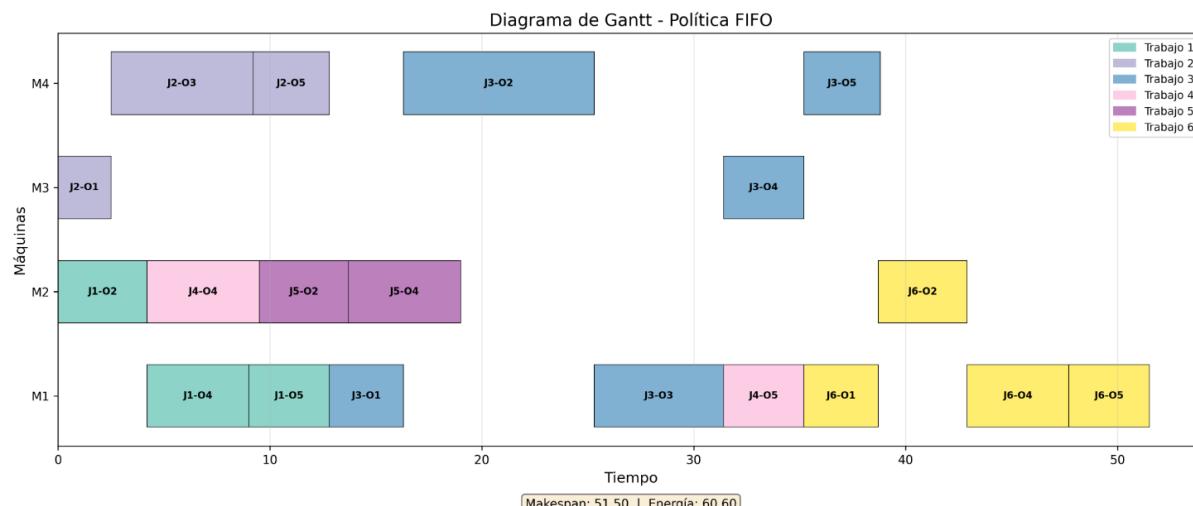
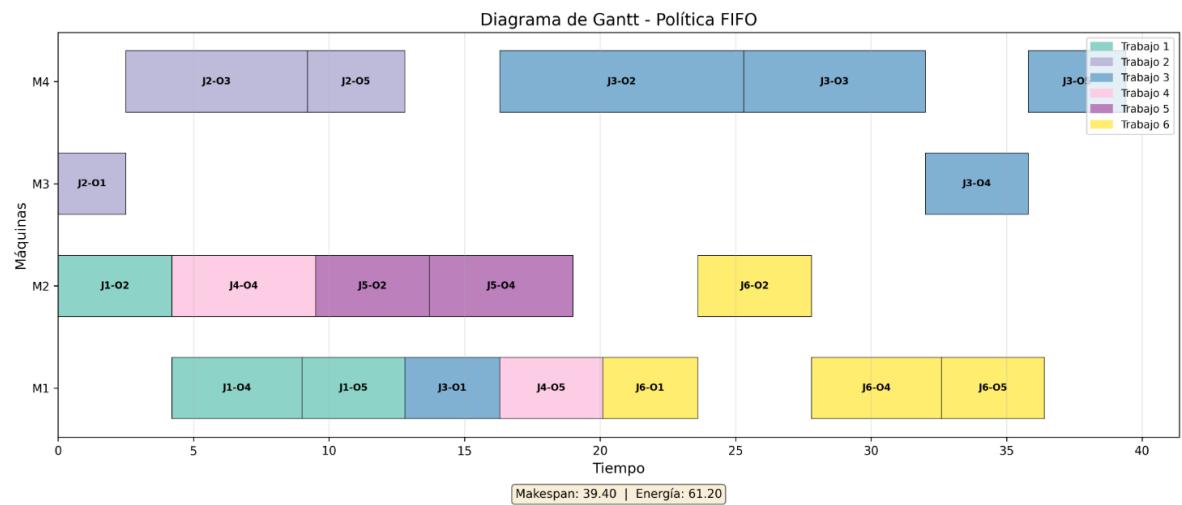
- d. Dibuje el frente de Pareto que se obtiene de la población final (generación 100), en la ejecución con desempeño medio y diga si alguna de las políticas tiende a estar en la rodilla o en los extremos del frente. Diga por qué ocurre esto.

- Las políticas Round Robin tienden a producir mejores soluciones en makespan debido a su balanceo de carga entre trabajos.

- Las políticas basadas en tiempo (LTP/STP) pueden encontrar soluciones en los extremos del frente (optimizando un objetivo a costa del otro).

e. Muestre 3 diagramas de Gantt asociados con los extremos del frente y la solución cercana a la rodilla.

Para FIFO



- El diagrama de min\_makespan muestra mejor utilización de máquinas en paralelo
- El diagrama de min\_energy puede tener mayor tiempo total, pero usa máquinas más eficientes energéticamente
- La solución de rodilla balancea ambos aspectos

## Conclusiones Individuales

### Diaz Ruiz Israel

Las 6 políticas son diferentes, pero ayudaron mucho para resolver el problema desde diferentes ángulos, cada uno con su propia estrategia. NSGA-II es el principal que evalúa todas las propuestas y selecciona las mejores. La codificación poliploide permite que estas políticas compartan información entre sí, creando soluciones híbridas aún mejores.

### Ignacio Cortés Atzin Maxela

Una de las principales dificultades al implementar este problema fue asegurar el estricto cumplimiento de la restricción de precedencia  $O_{ij}$  antes que  $O_{(i+1)j}$  al calcular el orden de las operaciones para las políticas LTP y STP, ya que sus criterios de ordenamiento por tiempo promedio podían fallar la regla fundamental de la planificación. Para resolver esto, se utilizó la clase JobShopData para encapsular y precalcular el orden válido de las 6 política, asegurando consistencia y evitando errores de precedencia durante la ejecución. Además, este modularidad facilitó la extensión y mantenimiento del código, permitiendo agregar nuevas políticas o métricas de rendimiento sin comprometer la integridad de la planificación existente.