# Optimización del Llenado Manual de Contenedores con Paquetes Heterogéneos

## Máster Universitario en Estadística Computacional y Ciencia de Datos para la Toma de Decisiones

Jose Gustavo Quilca Vilcapoma
Tutor: Javier Alcaraz Soria

Instituto Centro de Investigación Operativa
Universidad Miguel Hernández

17 Junio 2024



# Introducción

- El problema del llenado de contenedores (CLP) es un problema clásico en la optimización combinatoria.
- Consiste en encontrar la mejor manera de llenar un contenedor con paquetes de diferentes tamaños y pesos, optimizando la utilización del espacio.
- Este problema tiene aplicaciones en logística, transporte, almacenamiento, entre otros.

# Restricciones

- Restricciones Básicas
  - No superposición de paquetes.
  - No superar el peso máximo del contenedor.
  - Colocación dentro de los límites del contenedor.
- Restricciones Prácticas
  - Estabilidad de los paquetes.
  - Centro de gravedad del contenedor.
  - Prioridad de carga.
  - Restricción de contigüidad de tipos.

# Métodos de Solución

- **Métodos exactos**:
  - Garantizan la solución óptima, pero son computacionalmente costosos.
- **Métodos heurísticos**:
  - Proporcionan soluciones de buena calidad en un tiempo razonable.
- **Métodos metaheurísticos**:
  - Estrategias flexibles de alto nivel para desarrollar algoritmos que resuelven problemas específicos.

# Contexto del Problema

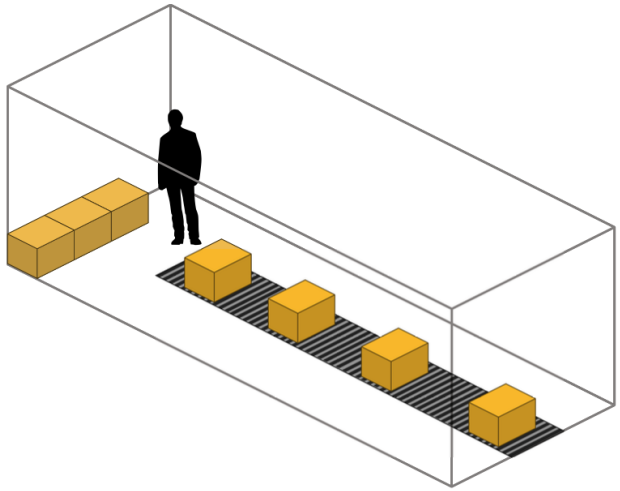
- Empresa en el sector de envío de paquetes en contenedores marítimos.
- No tiene control sobre las medidas de los paquetes.
- Cuenta con una cantidad máxima de paquetes de cada tipo.
- Envía un solo contenedor a la vez.
- Usan procedimientos establecidos de llenado manual.

# Definición del Problema

- Maximizar el beneficio de la carga en un contenedor, con los paquetes disponibles.
- Los paquetes apilados deben mantener su estabilidad durante la carga.
- Todos los paquetes del mismo tipo son cargados de forma contigua.
- Los paquetes del mismo tipo son rotados en una misma dirección.

# Definición Formal

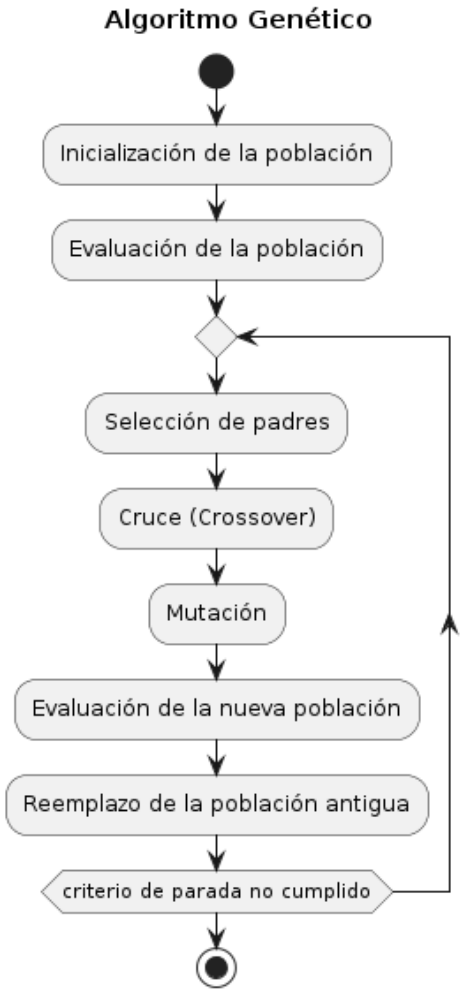
Optimizar la carga de un contenedor, determinando el **orden**, la **cantidad** y **rotación** de los tipos de paquetes, cumpliendo con las restricciones prácticas del **llenado manual**. El objetivo principal es maximizar el beneficio total de la carga.



# Algoritmo Genético

## Es una técnica de optimización inspirada en la evolución natural:

1. Generación y evaluación de una población inicial.
2. Selección de los mejores individuos.
3. Cruce de los individuos seleccionados.
4. Mutación de los individuos resultantes.
5. Evaluación y reemplazo de la nueva población.
6. Repetir hasta cumplir un criterio de parada.



# Algoritmo Genético

## Diseño de la Codificación de las Soluciones:

- Uso de 3 listas para representar el orden de llenado.
- Ayuda a garantizar factibilidad.
- Es la forma en la que llegarían los paquetes al operario.

| | | | | |
|---|---|---|---|---|
| Tipo | 2 | 1 | 4 | 3 |
| Rotación | 0 | 1 | 1 | 0 |
| Cantidad | 27 | 10 | 18 | 13 |

Ejemplo de codificación

# Algoritmo Genético

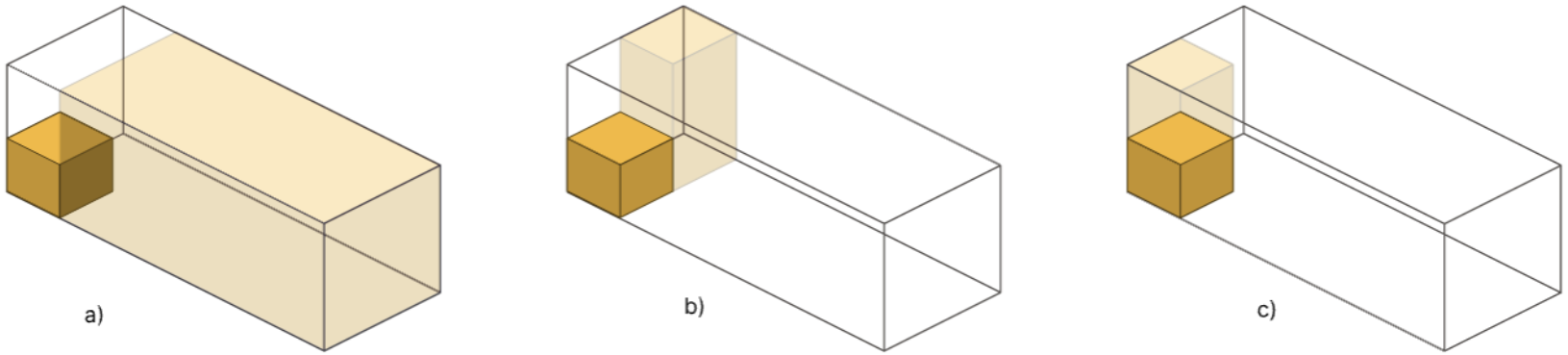
## Función de evaluación:

- Calcula el beneficio total de la carga.
- Verificar la factibilidad de la soluciones.
- Es necesario el desarrollo de un algoritmo que **imite el procedimiento de llenado manual**.

# Algoritmo Genético: Función de evaluación

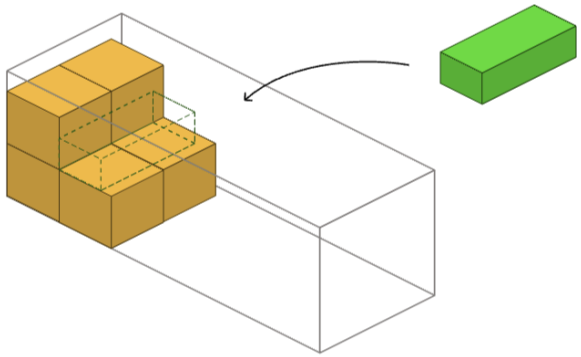
## Algoritmo de llenado manual:

Basado en un método conocido como Deepest Bottom Left with Fill (DBLF).



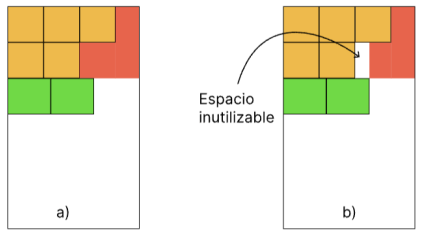

División del espacio restante en el contenedor luego de colocar un paquete

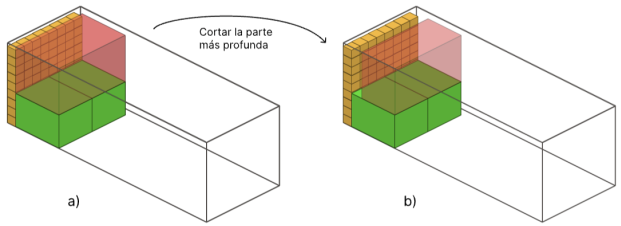
# Algoritmo Genético: Función de evaluación



## Adaptación del Algoritmo DBLF:

- Unión de subespacios
- Eliminación de subespacios inaccesibles
- Eliminación de subespacios profundos



# Algoritmo Genético: Función de evaluación

## Algoritmos adaptados

**Algoritmo 2** Algoritmo de unión de subespacios

```
1: Subespacios ← lista de subespacios disponibles
2: i ← longitud de Subespacios − 1
3: while i > 0 do
4:     if Subespacios[i].esSimilar(Subespacios[i − 1]) then
5:         Subespacios[i − 1].unir(Subespacios[i])
6:         Subespacios.remove(Subespacios[i])
7:     end if
8:     i ← i − 1
9: end while
10: return Subespacios
```

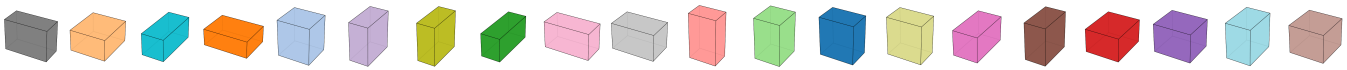
**Algoritmo 3** Algoritmo de eliminación de subespacios inaccesibles

```
1: Subespacios ← lista de subespacios disponibles
2: i ← longitud de Subespacios − 1
3: while i > 0 do
4:     if Subespacios[i].esInaccesibleParcialmente() then
5:         Subespacios[i].recortar()
6:     else if Subespacios[i].esInaccesibleTotalmente() then
7:         Subespacios.remove(Subespacios[i])
8:     end if
9:     i ← i − 1
10: end while
11: return Subespacios
```

**Algoritmo 4** Algoritmo de eliminación de subespacios profundos

```
1: Subespacios ← lista de subespacios disponibles
2: PosicionCajaMasCercana ← posición de la caja más cercana a la puerta
3: PosicionMaxima ← posición máxima que un operador puede alcanzar
4: i ← longitud de Subespacios − 1
5: while i > 0 do
6:     if Subespacios[i].esProfundoParcialmente(PosicionMaxima) then
7:         Subespacios[i].recortar()
8:     else if Subespacios[i].esProfundoTotalmente(PosicionMaxima) then
9:         Subespacios.remove(Subespacios[i])
10:    end if
11:    i ← i − 1
12: end while
13: return Subespacios
```

# Ejemplo de Llenado Manual



| 14 | 3 | 18 | 2 | 1 | 9 | 16 | 4 | 13 | 15 | 7 | 5 | 0 | 17 | 12 | 10 | 6 | 8 | 19 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 24 | 18 | 42 | 27 | 0 | 0 | 17 | 51 | 35 | 48 | 55 | 15 | 18 | 73 | 40 | 32 | 47 | 53 | 69 | 91 |

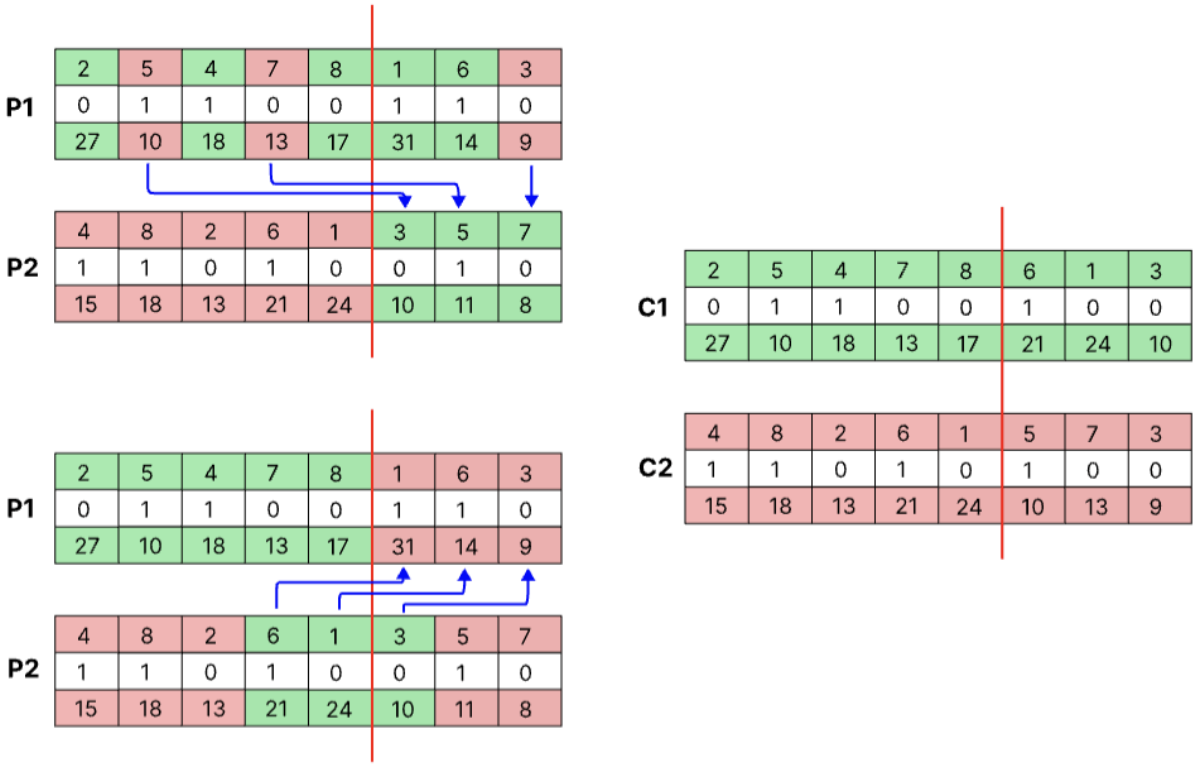
shorturl.at/6gatK

# Algoritmo Genético: Pasos

1. Generación aleatoria de la población inicial
2. Método de torneo binario para la selección

# Algoritmo Genético: Cruce

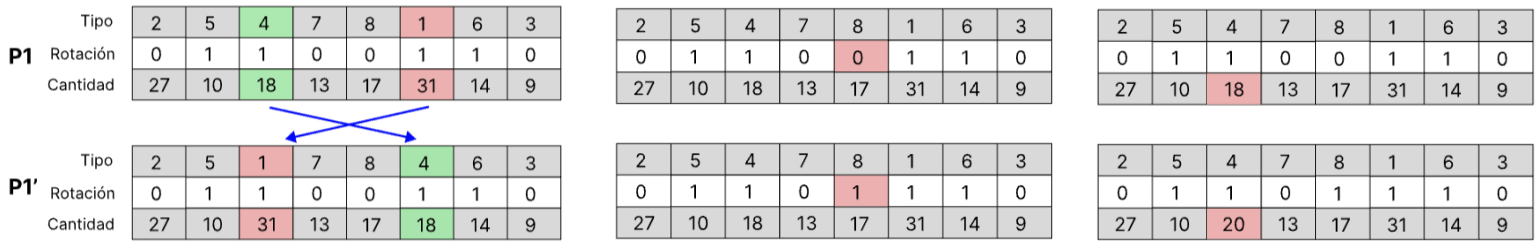
## 3 Adaptación del operador de cruce de un punto




Método de cruce adaptado para permutaciones que evita generar soluciones no factibles

# Algoritmo Genético: Mutación

4. Adaptación del operador de mutación de tres tipos

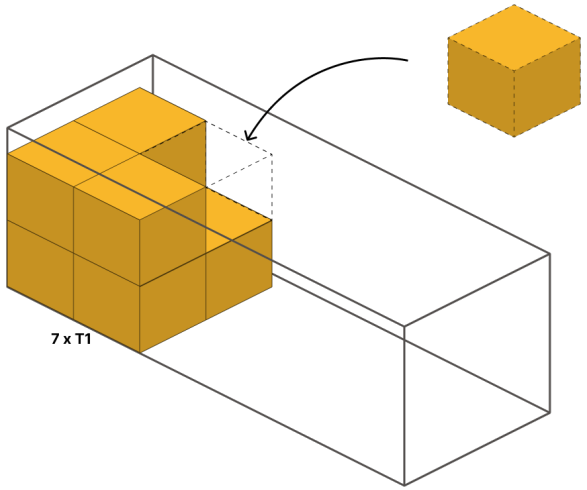


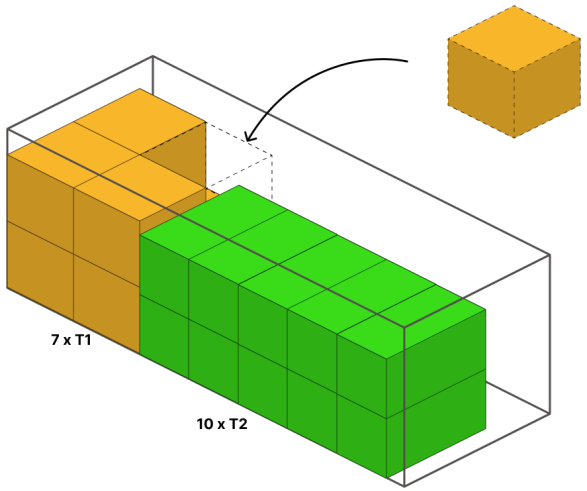

Métodos de mutación para
el **orden**, la **rotación** y la **cantidad** por tipo

# Diseño de procedimientos de mejora

## Procedimiento de mejora de las soluciones

- Ayuda a completar espacios vacíos si aún hay paquetes disponibles.
- Aplicarla durante el llenado podría empeorar una solución.
- Aplicarla luego del llenado garantiza que la solución no sea peor.
- Agregan tiempo adicional de cómputo.



# Estudio Computacional

## Generación de Datos de Prueba

- Un contenedor con dimensiones fijas $12010 \times 2330 \times 2380mm$.
- Instancias con 5, 10, 20, 30, 40 y 50 tipos de paquetes.
- Tamaños de paquetes aleatorios entre 250 y 750mm
- Asignación de beneficios aleatorios a cada paquete entre 10 y 100

# Estudio Computacional

## Configuración del Algoritmo

- Algoritmos implementados en Python 3.10.
- Población inicial de 100 individuos.
- $P_{cross} = 0.8$.
- $P_{mut} = 0.05$.

# Estudio Computacional

## Variantes del algoritmo

Comparar las distintas configuraciones del algoritmo

- **M0**: Sin mejora del algoritmo en el llenado.
- **M1**: Llenado adicional durante a toda la población.
- **M2**: Llenado adicional al final a toda la población.
- **M3**: De tipo M2 pero aplicado al 50% de la población.
- **M4**: De tipo M2, pero aplicado al mejor individuo de la población.

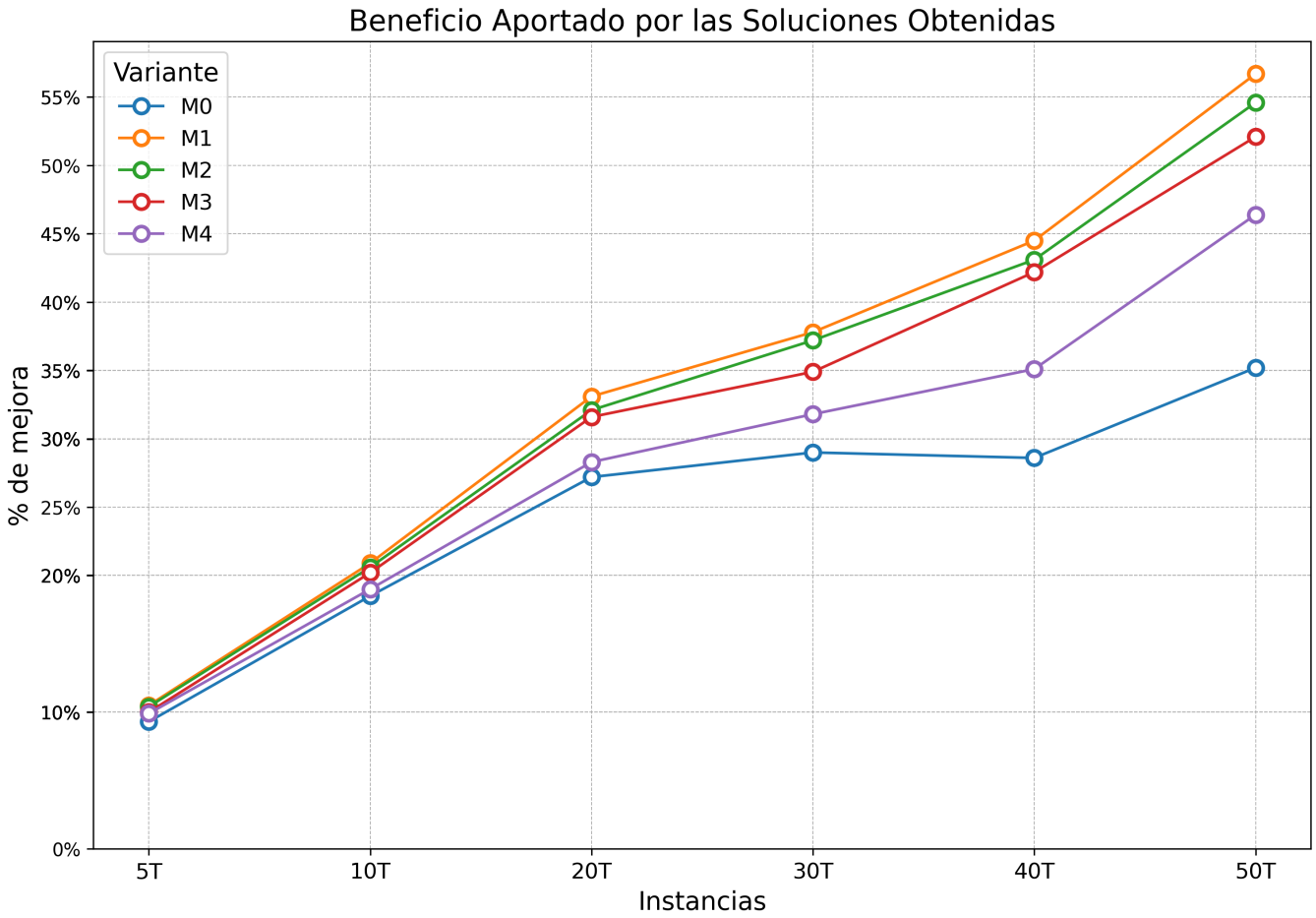
# Estudio Computacional

## Configuración del Experimento

- 25 problemas para cada tipo de instancia (5T, 10T, 20T, 30T, 40T y 50T), total 150 problemas.
- Tiempo fijo de 5 minutos para cada problema.
- Tiempo de ejecución total de 62.5 horas (2.6 días).
- Todas las variantes comienzan con la misma población inicial, para hacer una comparación justa.

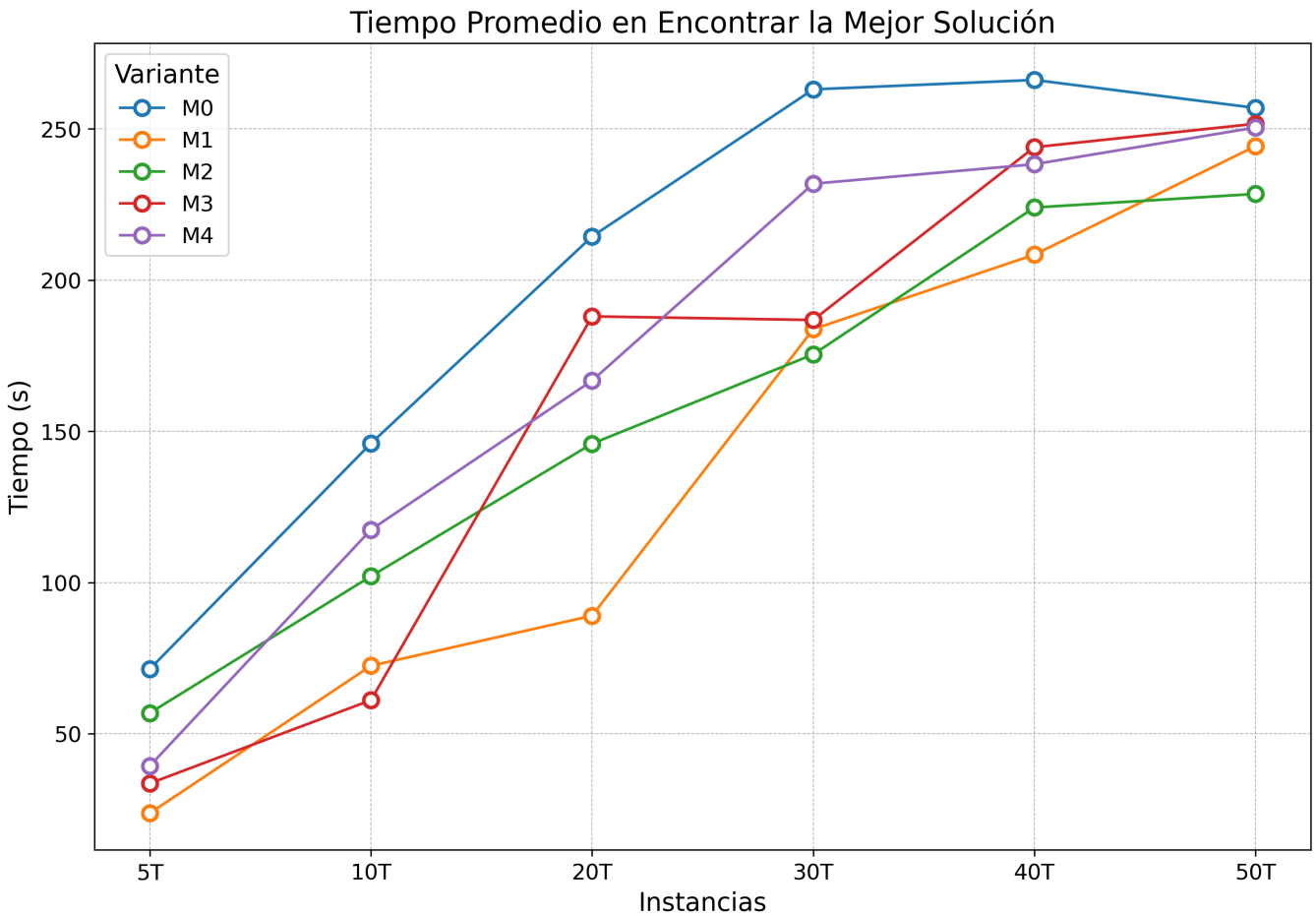
# Estudio Computacional: Resultados

## Beneficio aportado por las soluciones



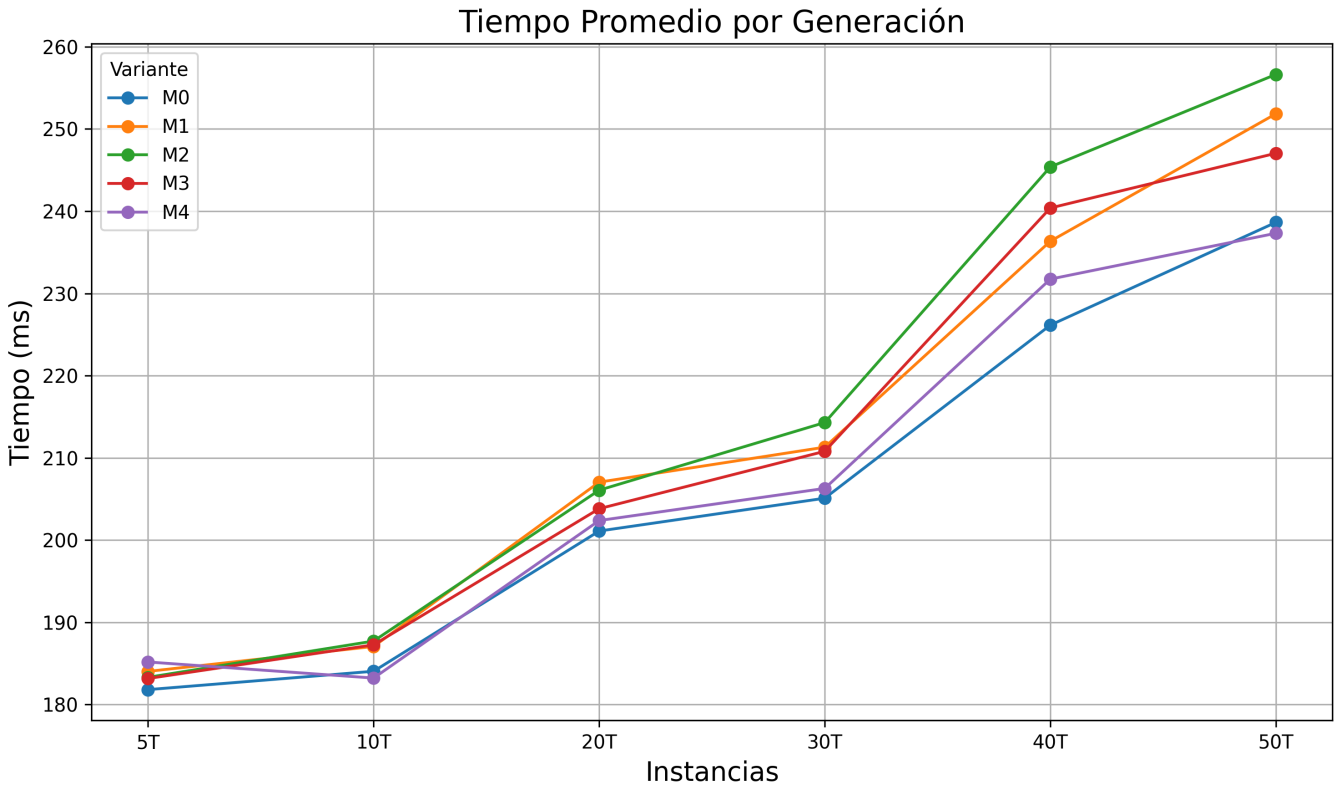
# Estudio Computacional: Resultados

## Tiempo promedio para encontrar la mejor solución de cada variante



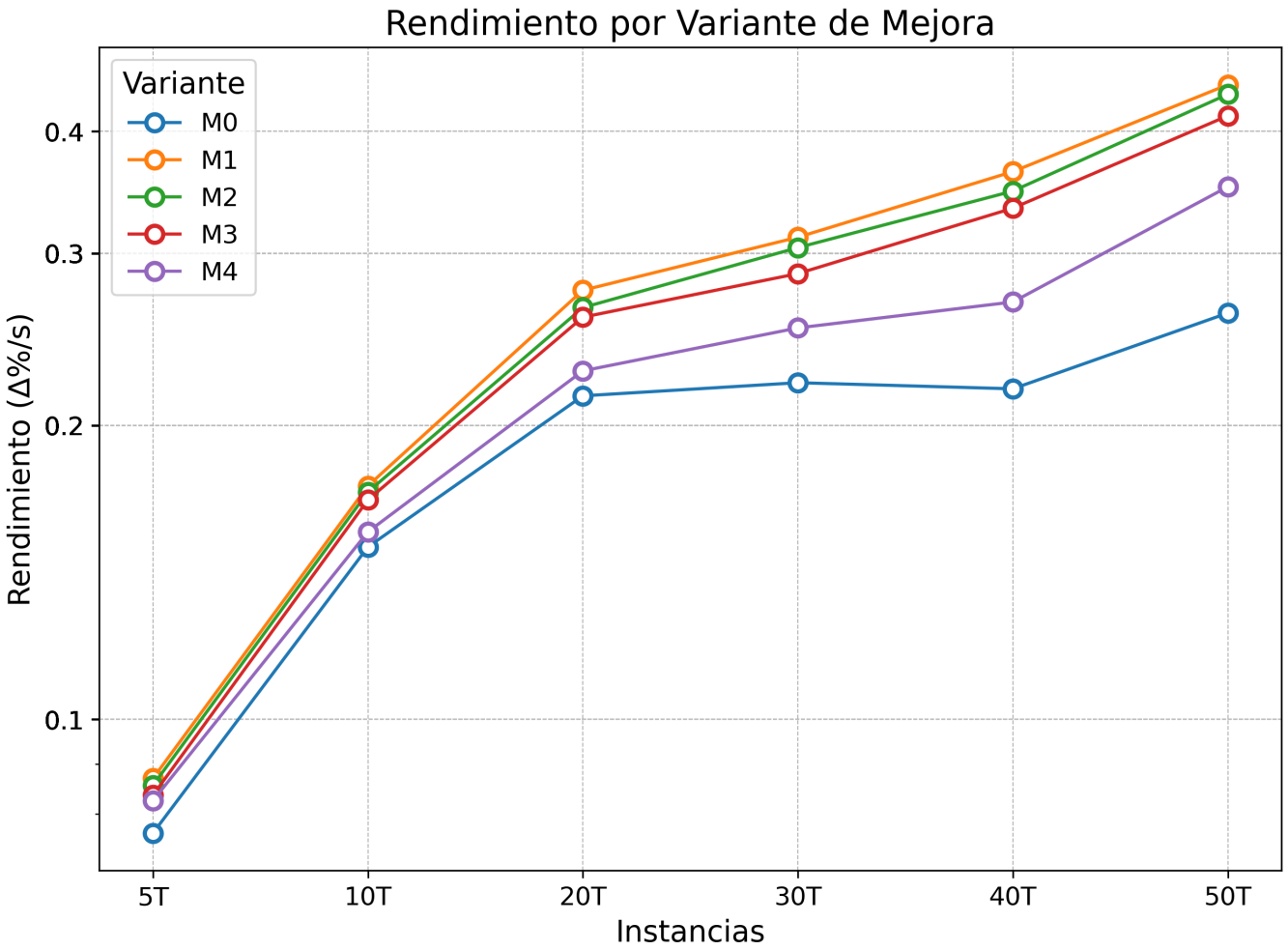
# Estudio Computacional: Resultados

## Tiempo promedio por generación



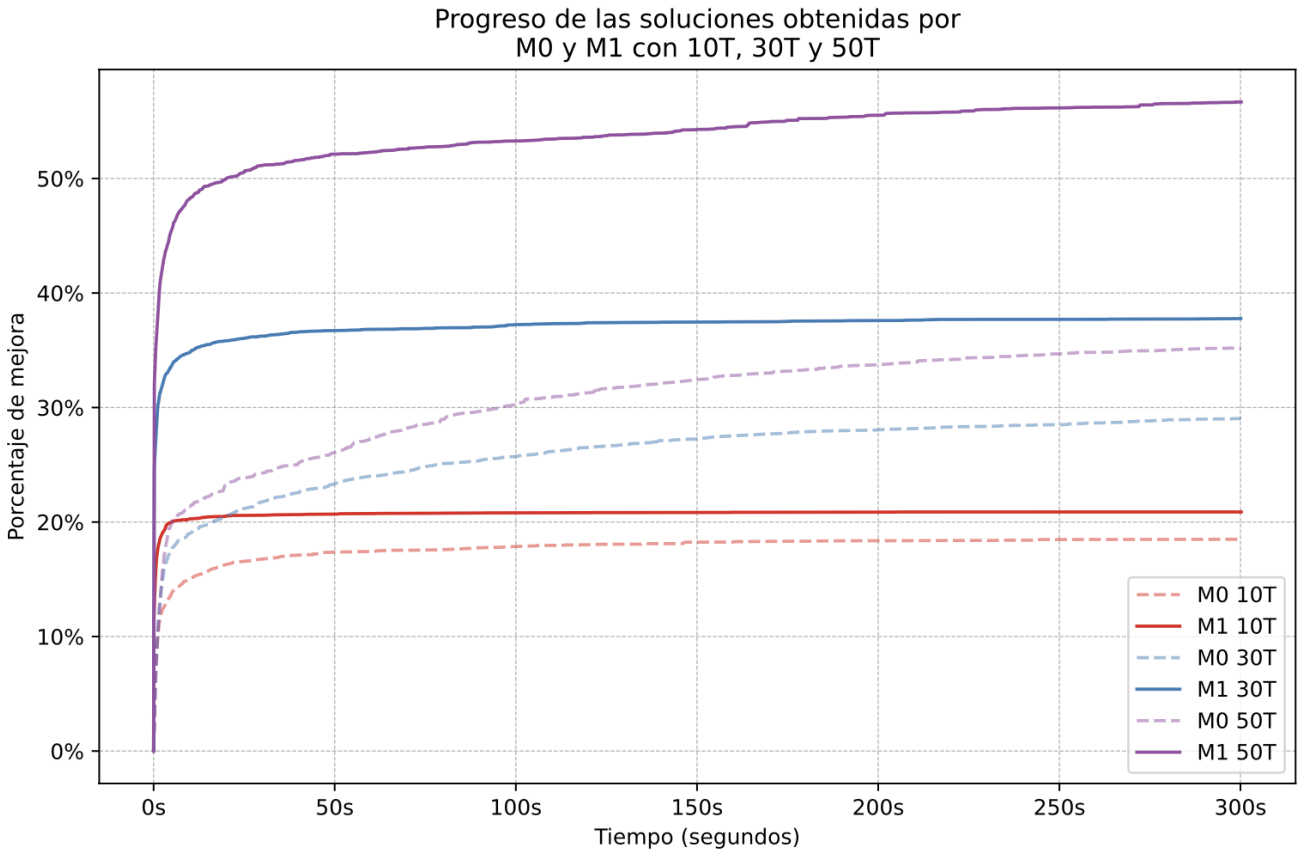
# Estudio Computacional: Resultados

## Rendimiento por variante de mejora



# Estudio Computacional: Resultados

## Progreso de las soluciones de la mejor variante elegida: M1



# Conclusiones

- El algoritmo ha demostrado ser un método eficiente para determinar la disposición de los paquetes en el contenedor.
- Los métodos de mejoras consiguen reducir el tiempo necesario para encontrar la mejor solución.
- También consiguen mejorar la calidad de las soluciones.
- Elevado número de tipos de paquetes necesitan más tiempo para converger.

# Trabajos Futuros

- Diseñar otros tipos de operadores de cruce y mutación.
- Considerar otras restricciones prácticas.
- Probar el algoritmo con datos reales.
- Incluir e algoritmo en un DSS utilizado en tiempo real.

*Gracias*