**SEBASTIAN GARCIA RIOS**

**TAREA 1.**

PROBLEMA DE PROGRAMACION DE TRABAJOS(JSSP)

**¿En qué consiste el Problema de Programación de Trabajos (JSSP)?**

El problema de programación de trabajos JSSP consiste en asignar un conjunto de trabajos a un número limitado de máquinas de manera que se minimice el tiempo total de producción cumpliendo ciertas restricciones.

Se trata de un problema clásico de optimización combinatoria que aparece en muchas industrias, como la manufactura, la logística y la planificación de recursos.

Dado un conjunto de J trabajos y M máquinas, cada trabajo está compuesto por una secuencia de operaciones que deben realizarse en un orden específico en las máquinas.

**El objetivo** es encontrar una programación óptima que:  
Minimice el makespan (el tiempo total de finalización).

Evite colisiones (una máquina solo puede ejecutar una operación a la vez).  
Respete las restricciones de orden (cada trabajo sigue una secuencia específica).

**Características del problema**

Cada trabajo tiene un conjunto de operaciones que deben ejecutarse en un orden específico.  
Cada operación solo puede ejecutarse en una máquina específica y tiene un tiempo de procesamiento fijo.  
Las máquinas solo pueden ejecutar una operación a la vez (no pueden procesar múltiples trabajos simultáneamente).

Las operaciones no pueden interrumpirse (una vez que comienzan, deben terminar sin interrupción).

El objetivo principal es minimizar el tiempo total de finalización (makespan).

**CONCLUSION:**

El problema de programación de trabajos (JSSP) es un problema complejo pero muy importante en la industria y la inteligencia artificial.

Existen múltiples formas de resolverlo, desde métodos exactos hasta heurísticas y metaheurísticas que permiten encontrar soluciones en problemas grandes.

La solución en Python con OR-Tools es una forma eficiente de abordar este problema usando programación por restricciones (CP), que permite optimizar la planificación de trabajos en fábricas, logística y más.

Código que lo resuelve:

import itertools

class JobShopScheduler:

    def \_\_init\_\_(self, jobs):

        self.jobs = jobs  # Lista de trabajos con sus máquinas y tiempos

        self.num\_jobs = len(jobs)

        self.num\_machines = max(machine for job in jobs for machine, \_ in job) + 1

        self.schedule = {}  # Almacena el tiempo de inicio de cada tarea

    def is\_valid\_schedule(self, schedule):

        """Verifica si un horario cumple con restricciones de precedencia."""

        for job\_id, job in enumerate(self.jobs):

            prev\_end\_time = 0  # Asegurar orden dentro del trabajo

            for machine, duration in job:

                start\_time = schedule[(job\_id, machine)]

                if start\_time < prev\_end\_time:

                    return False  # Restricción violada

                prev\_end\_time = start\_time + duration

        return True

    def find\_best\_schedule(self):

        """Busca la mejor manera de programar los trabajos en las máquinas."""

        all\_permutations = itertools.permutations(range(self.num\_jobs))  # Permutaciones de trabajos

        best\_schedule = None

        best\_time = float('inf')

        for perm in all\_permutations:

            current\_schedule = {}

            time\_tracker = [0] \* self.num\_machines  # Tiempo disponible en cada máquina

            for job\_id in perm:

                job = self.jobs[job\_id]

                job\_start\_time = 0

                for machine, duration in job:

                    start\_time = max(time\_tracker[machine], job\_start\_time)

                    current\_schedule[(job\_id, machine)] = start\_time

                    time\_tracker[machine] = start\_time + duration

                    job\_start\_time = start\_time + duration

            max\_time = max(time\_tracker)

            if max\_time < best\_time:

                best\_time = max\_time

                best\_schedule = current\_schedule

        return best\_schedule, best\_time

# 📌 \*\*Ejemplo de entrada (trabajos con secuencias de máquinas y tiempos)\*\*

jobs = [

    [(0, 3), (1, 2), (2, 2)],  # Trabajo 1: Máquina 0 (3 seg), luego Máquina 1 (2 seg), etc.

    [(0, 2), (2, 1), (1, 4)],  # Trabajo 2: Máquina 0 (2 seg), luego Máquina 2 (1 seg), etc.

    [(1, 4), (2, 3)]           # Trabajo 3: Máquina 1 (4 seg), luego Máquina 2 (3 seg)

]

# \*\*Ejecutar el solucionador\*\*

scheduler = JobShopScheduler(jobs)

best\_schedule, best\_time = scheduler.find\_best\_schedule()

# \*\*Mostrar resultados\*\*

print("Mejor horario encontrado:")

for task, start\_time in sorted(best\_schedule.items()):

    print(f"Tarea {task} comienza en {start\_time}s")

print(f"Tiempo total mínimo: {best\_time}s")

**PROBLEMA N REINAS**

**EN QUE CONSISTE?**

El problema de las N reinas es un problema clásico de ajedrez que consiste en colocar N reinas en un tablero de NxN de manera que ninguna se ataque entre sí.  
Es decir, no puede haber dos reinas en la misma fila, columna o diagonal.

**HISTORIA**

Fue propuesto por Max Bezzel en 1848 y más tarde generalizado a N reinas. Hoy en día, tiene aplicaciones en inteligencia artificial, teoría de grafos y optimización combinatoria.

**Métodos para resolverlo**

Existen varias formas de encontrar soluciones al problema de las N reinas:

**Fuerza bruta:**

Probar todas las combinaciones posibles (muy ineficiente para N grandes).

**Backtracking (Vuelta atrás):**

Técnica de búsqueda que coloca reinas de una en una y retrocede si hay conflictos.

**Algoritmos genéticos / heurísticos:**

Se utilizan técnicas como búsqueda local o recocido simulado.

**Programación con restricciones (CSP):**

Define restricciones y usa un solver de satisfacción de restricciones.

Código en Python (Backtracking):

def resolver\_n\_reinas(N):

    soluciones = []

    tablero = [-1] \* N  # Lista donde tablero[i] = columna donde está la reina en la fila i

    def es\_valido(fila, col):

        # Verifica si una reina puede colocarse en (fila, col)

        for i in range(fila):

            if tablero[i] == col or abs(tablero[i] - col) == abs(i - fila):

                return False

        return True

    def backtrack(fila):

        if fila == N:

            soluciones.append(tablero[:])  # Guardar una copia de la solución encontrada

            return

        for col in range(N):

            if es\_valido(fila, col):

                tablero[fila] = col

                backtrack(fila + 1)

                tablero[fila] = -1  # Retroceder

    backtrack(0)

    return soluciones

# 🔹 Ejecutar para N=8 y mostrar soluciones

N = 8

soluciones = resolver\_n\_reinas(N)

# Mostrar una solución de ejemplo

def imprimir\_tablero(solucion):

    N = len(solucion)

    for fila in range(N):

        linea = ['.'] \* N

        linea[solucion[fila]] = '♛'  # Colocar la reina en la posición correcta

        print(" ".join(linea))

    print("\n")

# Mostrar la primera solución encontrada

if soluciones:

    print(f"Se encontraron {len(soluciones)} soluciones para {N} reinas.\n")

    imprimir\_tablero(soluciones[0])

else:

    print("No se encontraron soluciones.")

Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

**ARBOL DE EXPANSION MINIMA**

El problema del Árbol de Expansión Mínima (MST, por sus siglas en inglés) se refiere a encontrar un subconjunto de las aristas de un grafo no dirigido y ponderado que conecta todos los vértices sin ciclos, y cuya suma de pesos es mínima.

Los dos algoritmos más comunes para resolver este problema son:

1. **Algoritmo de Prim**
2. **Algoritmo de Kruskal**

CODIGO:

Texto

El contenido generado por IA puede ser incorrecto.

**PROBLEMA DEL AGENTE VIAJERO**

es un problema de optimización combinatoria en el que un agente debe visitar un conjunto de ciudades exactamente una vez y regresar al punto de inicio, minimizando la distancia total o el costo total del viaje.

**Enfoque de Fuerza Bruta (Exploración de todas las rutas posibles):**

La forma más directa (pero no eficiente para un gran número de ciudades) de resolver este problema es usando una búsqueda por **fuerza bruta**, es decir, generando todas las posibles permutaciones de ciudades y evaluando cuál tiene el costo más bajo.

A continuación, te muestro una implementación en Python usando **fuerza bruta**:



**Explicación del código:**

1. **Matriz de distancias**: ciudades[i][j] representa la distancia entre la ciudad i y la ciudad j. Es una matriz simétrica, ya que la distancia entre las ciudades es la misma en ambas direcciones.
2. **itertools.permutations(range(n))**: Genera todas las permutaciones posibles de las ciudades. Por ejemplo, si tenemos 4 ciudades, las permutaciones generadas serán todas las formas posibles de visitar esas 4 ciudades exactamente una vez.
3. **calcular\_distancia(ciudades, ruta)**: Esta función calcula la distancia total de una ruta dada, sumando las distancias entre las ciudades consecutivas y sumando también el regreso al punto de inicio.
4. **agente\_viajero(ciudades)**: Este es el algoritmo principal que genera todas las rutas posibles, las evalúa y selecciona la de menor distancia.