INSTITUTO TECNOLOGICO DE CULIACAN

“Tópicos de IA”

Zuriel Dathan Mora Félix

Tarea 1 U2

12:00pm – 01:00pm

Ing. Sistemas Computacionales

Isaac Alejandro Pacheco Ruiz

07/03/2024

Problemas de Programación con la IA

* Problema de programación de trabajos (JSSP)
* Problema de las N reynas
* Árbol de expansión mínima (MST)
* Problema del agente viajero (TSP)
  + Descripción detallada del problema
  + Representación

Problema de Programación de Trabajos (JSSP)

El Job Shop Scheduling Problem (JSSP) o Problema de Programación de Trabajos es un problema clásico de optimización en el que se busca programar un conjunto de trabajos en múltiples máquinas de manera eficiente.

Descripción

* Se tienen n trabajos (jobs) y m máquinas.
* Cada trabajo consta de una secuencia de operaciones que deben procesarse en un orden específico.
* Cada operación debe ejecutarse en una máquina particular y tiene una duración fija.
* Una máquina solo puede procesar una operación a la vez.
* Un trabajo no puede ejecutar múltiples operaciones simultáneamente.
* El objetivo suele ser minimizar el makespan (tiempo total requerido para completar todos los trabajos).

Ejemplo

Este código define un problema de programación de trabajos con tres trabajos y diferentes tareas asignadas a máquinas específicas. Utiliza el modelo de programación con restricciones de OR-Tools para minimizar el makespan, que es el tiempo total necesario para completar todos los trabajos.

from ortools.sat.python import cp\_model

def main():

# Datos del problema: cada trabajo tiene una lista de tareas (máquina, duración)

jobs\_data = [

[(0, 3), (1, 2), (2, 2)], # Trabajo 0

[(0, 2), (2, 1), (1, 4)], # Trabajo 1

[(1, 4), (2, 3)] # Trabajo 2

]

# Crear el modelo CP-SAT

model = cp\_model.CpModel()

# Variables para almacenar las tareas

all\_tasks = {}

machine\_to\_intervals = {}

# Horizonte: suma de todas las duraciones

horizon = sum(task[1] for job in jobs\_data for task in job)

# Crear variables de inicio y fin para cada tarea

for job\_id, job in enumerate(jobs\_data):

for task\_id, (machine, duration) in enumerate(job):

suffix = f'\_j{job\_id}\_t{task\_id}'

start\_var = model.NewIntVar(0, horizon, 'start' + suffix)

end\_var = model.NewIntVar(0, horizon, 'end' + suffix)

interval\_var = model.NewIntervalVar(start\_var, duration, end\_var, 'interval' + suffix)

all\_tasks[job\_id, task\_id] = (start\_var, end\_var, interval\_var)

if machine not in machine\_to\_intervals:

machine\_to\_intervals[machine] = []

machine\_to\_intervals[machine].append(interval\_var)

# Añadir restricciones de precedencia

for job\_id, job in enumerate(jobs\_data):

for task\_id in range(len(job) - 1):

model.Add(all\_tasks[job\_id, task\_id][1] <= all\_tasks[job\_id, task\_id + 1][0])

# Añadir restricciones de no solapamiento en máquinas

for machine, intervals in machine\_to\_intervals.items():

model.AddNoOverlap(intervals)

# Variable para el makespan (tiempo total de finalización)

makespan = model.NewIntVar(0, horizon, 'makespan')

model.AddMaxEquality(makespan, [

all\_tasks[job\_id, len(job) - 1][1] for job\_id, job in enumerate(jobs\_data)

])

model.Minimize(makespan)

# Crear el solucionador y resolver el modelo

solver = cp\_model.CpSolver()

status = solver.Solve(model)

# Mostrar resultados

if status == cp\_model.OPTIMAL or status == cp\_model.FEASIBLE:

print(f'Makespan óptimo: {solver.Value(makespan)}')

for job\_id, job in enumerate(jobs\_data):

for task\_id, (machine, duration) in enumerate(job):

start = solver.Value(all\_tasks[job\_id, task\_id][0])

print(f'Trabajo {job\_id} Tarea {task\_id} (Máquina {machine}) inicia en {start} y dura {duration}')

else:

print('No se encontró una solución factible.')

if \_\_name\_\_ == '\_\_main\_\_':

main()

Problema de las N reynas

El Problema de las N-Reinas es un desafío clásico en matemáticas y ciencias de la computación que consiste en colocar N reinas en un tablero de ajedrez de N×N casillas de manera que ninguna reina pueda atacar a otra. Esto implica que no puede haber dos reinas en la misma fila, columna o diagonal.

Ejemplo

Para N=8, el objetivo es situar 8 reinas en un tablero de 8×8 sin que se amenacen entre sí. Una de las posibles soluciones es:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 |  |  |  |  | QUEEN |  |  |  |
| 2 | QUEEN |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  | QUEEN |  |
| 4 |  |  |  |  |  |  |  | QUEEN |
| 5 |  | QUEEN |  |  |  |  |  |  |
| 6 |  |  |  | QUEEN |  |  |  |  |
| 7 |  |  |  |  |  | QUEEN |  |  |
| 8 |  |  | QUEEN |  |  |  |  |  |

Una forma de implementarlo en Python es esta:

def es\_seguro(tablero, fila, col, N):

# Verificar si hay una reina en la misma columna

for i in range(fila):

if tablero[i] == col:

return False

# Verificar la diagonal superior izquierda

for i, j in zip(range(fila-1, -1, -1), range(col-1, -1, -1)):

if tablero[i] == j:

return False

# Verificar la diagonal superior derecha

for i, j in zip(range(fila-1, -1, -1), range(col+1, N)):

if tablero[i] == j:

return False

return True

def resolver\_n\_reinas(N):

def backtrack(fila):

if fila == N:

solucion = []

for i in range(N):

fila\_str = ['.'] \* N

fila\_str[tablero[i]] = 'Q'

solucion.append(''.join(fila\_str))

soluciones.append(solucion)

return

for col in range(N):

if es\_seguro(tablero, fila, col, N):

tablero[fila] = col

backtrack(fila + 1)

tablero[fila] = -1

soluciones = []

tablero = [-1] \* N

backtrack(0)

return soluciones

# Ejemplo de uso

N = 8

soluciones = resolver\_n\_reinas(N)

for sol in soluciones:

for fila in sol:

print(fila)

print()

Árbol de Expansión Mínima (MST)

Un Árbol de Expansión Mínima (AEM), también conocido como Árbol Recubridor Mínimo, es un subgrafo de un grafo conexo y no dirigido que conecta todos los vértices con el menor peso total posible y sin formar ciclos. Este concepto es fundamental en la teoría de grafos y tiene aplicaciones en áreas como el diseño de redes, optimización de rutas y planificación de infraestructuras.

Ejemplo

Consideremos un grafo con los siguientes vértices y aristas ponderadas:

* Vértices: A, B, C, D, E
* Aristas y sus pesos:
  + A-B: 4
  + A-C: 3
  + B-C: 1
  + B-D: 2
  + C-D: 4
  + C-E: 2
  + D-E: 3

Para encontrar el AEM, utilizaremos el Algoritmo de Kruskal, que es un método voraz para este propósito. El algoritmo funciona de la siguiente manera:

1. Ordena todas las aristas del grafo en orden ascendente según su peso.
2. Inicializa un conjunto vacío para las aristas del AEM.
3. Recorre las aristas ordenadas y añade cada una al conjunto si no forma un ciclo con las aristas ya añadidas.
4. Repite el paso 3 hasta que el conjunto tenga (n-1) aristas, donde n es el número de vértices del grafo.

Aplicando el Algoritmo de Kruskal al grafo dado:

1. Ordenamos las aristas por peso:
   * B-C: 1
   * B-D: 2
   * C-E: 2
   * A-C: 3
   * D-E: 3
   * A-B: 4
   * C-D: 4
2. Seleccionamos las aristas en este orden, asegurándonos de no formar ciclos:
   * Añadimos B-C (1)
   * Añadimos B-D (2)
   * Añadimos C-E (2)
   * Añadimos A-C (3)

Ahora en código se vería así.

class Grafo:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.aristas = []

def agregar\_arista(self, u, v, peso):

self.aristas.append([u, v, peso])

def encontrar(self, parent, i):

if parent[i] == i:

return i

return self.encontrar(parent, parent[i])

def union(self, parent, rank, x, y):

xroot = self.encontrar(parent, x)

yroot = self.encontrar(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

def kruskal(self):

resultado = []

i, e = 0, 0

self.aristas = sorted(self.aristas, key=lambda item: item[2])

parent = []

rank = []

for nodo in range(self.V):

parent.append(nodo)

rank.append(0)

while e < self.V - 1:

u, v, peso = self.aristas[i]

i = i + 1

x = self.encontrar(parent, u)

y = self.encontrar(parent, v)

if x != y:

e = e + 1

resultado.append([u, v, peso])

self.union(parent, rank, x, y)

print("Aristas en el Árbol de Expansión Mínima:")

for u, v, peso in resultado:

print(f"{u} -- {v} == {peso}")

Problema del Agente Viajero (TSP)

El Problema del Agente Viajero (TSP, por sus siglas en inglés de Travelling Salesman Problem) es un desafío clásico en la optimización combinatoria y la teoría de grafos. Consiste en encontrar la ruta más corta posible que permita a un vendedor visitar una lista de ciudades exactamente una vez y regresar a la ciudad de origen. Este problema es de gran importancia en investigación operativa y ciencias de la computación debido a su complejidad y amplia gama de aplicaciones prácticas, como la planificación, la logística y la fabricación de circuitos electrónicos.

Ejemplo:

Consideremos un conjunto de cuatro ciudades: A, B, C y D, con las siguientes distancias entre ellas (en kilómetros):

A-B: 10

A-C: 15

A-D: 20

B-C: 35

B-D: 25

C-D: 30

El objetivo es determinar la ruta más corta que visite cada ciudad una vez y regrese al punto de partida. Las posibles rutas y sus distancias totales son:

A → B → C → D → A = 10 + 35 + 30 + 20 = 95 km

A → B → D → C → A = 10 + 25 + 30 + 15 = 80 km

A → C → B → D → A = 15 + 35 + 25 + 20 = 95 km

A → C → D → B → A = 15 + 30 + 25 + 10 = 80 km

A → D → B → C → A = 20 + 25 + 35 + 15 = 95 km

A → D → C → B → A = 20 + 30 + 35 + 10 = 95 km

Las rutas más cortas son A → B → D → C → A y A → C → D → B → A, ambas con una distancia total de 80 km.

Ahora la implementación en Python se vería así:

import random

import numpy as np

# Definir las ciudades y sus coordenadas

ciudades = {

'A': (0, 0),

'B': (10, 0),

'C': (10, 10),

'D': (0, 10)

}

# Calcular la distancia euclidiana entre dos ciudades

def distancia(ciudad1, ciudad2):

return np.sqrt((ciudad1[0] - ciudad2[0])\*\*2 + (ciudad1[1] - ciudad2[1])\*\*2)

# Crear una ruta aleatoria

def crear\_ruta(ciudades):

return random.sample(list(ciudades.keys()), len(ciudades))

# Calcular la distancia total de una ruta

def distancia\_ruta(ruta):

distancia\_total = 0

for i in range(len(ruta)):

ciudad\_actual = ciudades[ruta[i]]

ciudad\_siguiente = ciudades[ruta[(i + 1) % len(ruta)]]

distancia\_total += distancia(ciudad\_actual, ciudad\_siguiente)

return distancia\_total

# Crear una población inicial de rutas

def crear\_poblacion(ciudades, tamano\_poblacion):

return [crear\_ruta(ciudades) for \_ in range(tamano\_poblacion)]

# Seleccionar las mejores rutas de la población

def seleccionar\_mejores(poblacion, tamano\_seleccion):

rutas\_ordenadas = sorted(poblacion, key=lambda ruta: distancia\_ruta(ruta))

return rutas\_ordenadas[:tamano\_seleccion]

# Cruzar dos rutas para generar una nueva ruta

def cruzar(ruta1, ruta2):

punto\_corte = random.randint(0, len(ruta1) - 1)

hijo = ruta1[:punto\_corte] + [ciudad for ciudad in ruta2 if ciudad not in ruta1[:punto\_corte]]

return hijo

# Mutar una ruta intercambiando dos ciudades

def mutar(ruta, tasa\_mutacion):

if random.random() < tasa\_mutacion:

i, j = random.sample(range(len(ruta)), 2)

ruta[i], ruta[j] = ruta[j], ruta[i]

return ruta

# Evolucionar la población

def evolucionar\_poblacion(poblacion, tamano\_seleccion, tasa\_mutacion):

mejores = seleccionar\_mejores(poblacion, tamano\_seleccion)

hijos = []

while len(hijos) < len(poblacion):

padre1, padre2 = random.sample(mejores, 2)

hijo = cruzar(padre1, padre2)

hijo = mutar(hijo, tasa\_mutacion)

hijos.append(hijo)

return hijos

# Parámetros del algoritmo genético

tamano\_poblacion = 100

tamano\_seleccion = 20

tasa\_mutacion = 0.01

generaciones = 500

# Inicializar la población

poblacion = crear\_poblacion(ciudades, tamano\_poblacion)

# Evolucionar la población a lo largo de las generaciones

for \_ in range(generaciones):

poblacion = evolucionar\_poblacion(poblacion, tamano\_seleccion, tasa\_mutacion)

# Obtener la mejor ruta encontrada

mejor\_ruta = seleccionar\_mejores(poblacion, 1)[0]

print(f"Mejor ruta: {mejor\_ruta}")

print(f"Distancia: {distancia\_ruta(mejor\_ruta)}")

Referencias:

* Problema de Programación de Trabajos (JSSP)
  + Pinedo, M. L. (2016). Scheduling: Theory, Algorithms, and Systems (5th ed.). Springer.
  + Blazewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., & Weglarz, J. (2001). Scheduling Computer and Manufacturing Processes (2nd ed.). Springer.
  + Perron, L. (2023). El problema del taller. Recuperado de <https://developers.google.com/optimization/scheduling/job_shop?hl=es-419>
* Problema de las N-Reinas
  + Zacarías, F. (s.f.). El Problema de las N-Reinas. Recuperado de https://www.cs.buap.mx/~zacarias/FZF/nreinas3.pdf
  + Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.
* Árbol de Expansión Mínima (AEM)
  + Wikipedia. (2023). Árbol recubridor mínimo. Recuperado de https://es.wikipedia.org/wiki/Árbol\_recubridor\_mínimo
  + Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
* Problema del Agente Viajero (TSP)
  + Wikipedia. (2023). Problema del viajante. Recuperado de https://es.wikipedia.org/wiki/Problema\_del\_viajante
  + Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). The Traveling Salesman Problem: A Computational Study. Princeton University Press.