INSTITUTO TECNOLÓGICO DE CULIACÁN

Materia:

Tópicos de IA

Tarea:

Tarea#2 Unidad 2

Profesor:

Zuriel Dathan Mora Felix

Alumna:

Anette Leticia Robles Zamora

1. Descripción del Problema

El problema de las 8 reinas consiste en ubicar 8 reinas en un tablero de ajedrez de 8*8 de forma que ninguna se ataque entre sí. Dado que cada reina puede atacar en filas, columnas y diagonales, la solución debe garantizar que: No haya dos reinas en la misma fila (lo que se logra representando la solución como una permutación). No haya dos reinas en la misma columna (también garantizado por la permutación). No haya dos reinas en la misma diagonal (se verifica evaluando la diferencia de posiciones). El objetivo es encontrar una solución (o una aproximación) sin conflictos. En este caso se emplea el algoritmo de búsqueda tabú, una metaheurística que explora el espacio de soluciones, evitando ciclos mediante una lista tabú.

2. Representación de P y S

Problema: El conjunto de estados del problema es el conjunto de todas las permutaciones de [0,1,2,…,7] [0,1,2,…,7]. Cada estado es una solución candidata donde el índice representa la fila y el valor en esa posición la columna en la que se ubica la reina en esa fila.

Solución: Una solución es una permutación en la que no existan dos reinas en la misma diagonal. El estado óptimo tiene 0 conflictos.

3. Propuesta de Algoritmo en Pseudocódigo / Diagrama de Flujo

Pseudocódigo

INICIO

// Inicialización

Leer opción de configuración inicial:

Configuración aleatoria

Configuración ingresada por el usuario

Si opción == 2:

Solicitar y validar la configuración inicial

Sino:

Generar configuración aleatoria (permutación de 0 a 7)

```
Inicializar:
    solucion_actual ← configuración inicial
    mejor_solucion ← solucion_actual
    mejor_conflicto ← calcular_conflictos(solucion_actual)
    movimientos ← 0
    lista_tabu ← {} // Diccionario vacío
    iteración ← 0
    iniciar temporizador

// Búsqueda Tabu
Mientras (iteración < MAX_ITERACIONES y mejor_conflicto > 0) hacer:
    iteración ← iteración + 1
    generar vecinos intercambiando dos elementos de solucion_actual
    para cada vecino:
        calcular número de conflictos
        Si movimiento está en lista_tabu y no cumple criterio de aspiración:
            continuar con el siguiente vecino
        Si número de conflictos es menor que el mejor encontrado en vecinos:
            actualizar mejor vecino y movimiento asociado
    Fin para

    Si no se encontró vecino:
        Romper el ciclo

    Actualizar:
        solucion_actual ← mejor vecino encontrado
```

Agregar movimiento a lista_tabu con duración TENURE

movimientos ← movimientos + 1

Si conflictos del vecino < mejor_conflicto:

mejor_conflicto ← conflictos del vecino

mejor_solucion ← copia de mejor vecino

Mostrar estado de la iteración (número de conflictos, solución)

Fin Mientras

detener temporizador

Calcular tiempo transcurrido

Imprimir:

mejor_solucion, mejor_conflicto, número de iteraciones, tiempo transcurrido, movimientos

FIN

4. Implementación en Python

```
import random

def calcular_conflictos(solucion):
    """
    Calcula el número de conflictos en la solución.
    Dos reinas están en conflicto si se encuentran en la misma diagonal.

    Parámetros:
        solucion (list): Lista que representa la posición de las reinas.
                         El índice representa la fila y el valor la columna.

```

```
    Retorna:
        int: Número total de conflictos (pares de reinas que se atacan).
    """
    conflictos = 0
    n = len(solucion)
    for i in range(n):
        for j in range(i+1, n):
            # Si la diferencia absoluta entre columnas es igual
            # a la diferencia absoluta entre filas, están en la misma diagonal.
            if abs(solucion[i] - solucion[j]) == abs(i - j):
                conflictos += 1
    return conflictos

def generar_vecinos(solucion):
    """
    Genera los vecinos de la solución actual intercambiando dos reinas.

    Parámetros:
        solucion (list): Solución actual.

    Retorna:
        list: Lista de tuplas (movimiento, vecino), donde 'movimiento' es el par de índices intercambiados.
    """
    vecinos = []
    n = len(solucion)
    # Se intercambian pares de posiciones (i, j) para generar un vecino
    for i in range(n - 1):
        for j in range(i + 1, n):
            vecino = solucion.copy()
            # Intercambiar las reinas en las filas i y j
            vecino[i], vecino[j] = vecino[j], vecino[i]
            vecinos.append(((i, j), vecino))
    return vecinos

def tabu_search(n, max_iter, tenure):
    """
    Implementa el algoritmo de búsqueda tabu para el problema de las n reinas.

    Parámetros:
        n (int): Número de reinas.
        max_iter (int): Número máximo de iteraciones.
        tenure (int): Duración durante la cual un movimiento permanece en la lista tabu.

```

```
    Retorna:
        tuple: (mejor_solucion, mejor_conflicto, iteracion_final)
    """
    # Inicialización: solución aleatoria (permutación de 0 a n-1)
    solucion_actual = list(range(n))
    random.shuffle(solucion_actual)
    mejor_solucion = solucion_actual.copy()
    mejor_conflicto = calcular_conflictos(solucion_actual)

    # Diccionario que funciona como lista tabu, donde la clave es el movimiento (i,j)
    # y el valor es la iteración hasta la cual se considera tabu.
    lista_tabu = {}

    iteracion = 0
    # Se ejecuta hasta alcanzar el máximo de iteraciones o hasta encontrar una solución sin conflictos
    while iteracion < max_iter and mejor_conflicto > 0:
        iteracion += 1
        vecinos = generar_vecinos(solucion_actual)

        # Variables para almacenar el mejor vecino de la iteración
        mejor_vecino = None
        mejor_vecino_conflictos = float('inf')
        mejor_movimiento = None

        # Evaluar todos los vecinos generados
        for movimiento, vecino in vecinos:
            conflictos = calcular_conflictos(vecino)

            # Verificar si el movimiento está en la lista tabu
            # Se permite el movimiento si mejora la mejor solución (aspiración)
            if movimiento in lista_tabu and lista_tabu[movimiento] > iteracion and conflictos >= mejor_conflicto:
                continue

            # Seleccionar el vecino con menos conflictos
            if conflictos < mejor_vecino_conflictos:
                mejor_vecino_conflictos = conflictos
                mejor_vecino = vecino
                mejor_movimiento = movimiento

        # Si no se encontró un vecino (caso raro), se interrumpe el ciclo
        if mejor_vecino is None:
            break

```

```
            # Actualizar la solución actual con el mejor vecino encontrado
            solucion_actual = mejor_vecino

            # Registrar el movimiento en la lista tabu con una duración determinada
            lista_tabu[mejor_movimiento] = iteracion + tenure

            # Actualizar la mejor solución encontrada si se mejora
            if mejor_vecino_conflictos < mejor_conflicto:
                mejor_conflicto = mejor_vecino_conflictos
                mejor_solucion = mejor_vecino.copy()

            # Mostrar el progreso de la búsqueda en cada iteración
            print(f"Iteración {iteracion}: Conflictos = {mejor_vecino_conflictos} | Solución: {solucion_actual}")

    return mejor_solucion, mejor_conflicto, iteracion

if __name__ == '__main__':
    # Parámetros del problema
    N = 8            # Número de reinas (problema de las 8 reinas)
    MAX_ITER = 10000  # Número máximo de iteraciones permitidas
    TENURE = 5       # Tenencia para la lista tabu (duración de prohibición de movimientos)

    # Ejecutar el algoritmo de búsqueda tabu
    solucion, conflictos, iteraciones = tabu_search(N, MAX_ITER, TENURE)

    # Mostrar el resultado final obtenido
    print("\nResultado final:")
    print("Solución:", solucion)
    print("Número de conflictos:", conflictos)
    print("Número de iteraciones:", iteraciones)
```