Instituto tecnológico de Culiacán

Unidad 2

Actividad:

Tarea 2: Resolver el problema 8 reinas con el algoritmo de búsqueda tabú

Alumno:

Ivan Eduardo Ramírez moreno

Docente:

ZURIEL DATHAN MORA FELIX

Materia:

Tópicos de IA

Numero de control:

20170787

Semestre:

10

Descripción detallada del problema: 8 reinas

Introducción

El problema de las 8 reinas es un desafío clásico en el campo de la computación, la inteligencia artificial y la optimización combinatoria. Su objetivo es colocar 8 reinas en un tablero de ajedrez de 8x8, asegurando que ninguna de ellas pueda atacar a otra según las reglas del ajedrez.

Reglas y restricciones

Cada reina en el tablero tiene la capacidad de moverse en líneas horizontales, verticales y diagonales. Para que una configuración sea válida, deben cumplirse las siguientes condiciones:

1. No debe haber dos reinas en la misma fila.
2. No debe haber dos reinas en la misma columna.
3. No debe haber dos reinas en la misma diagonal ascendente o descendente.

Dado que un tablero de ajedrez tiene 8 filas y 8 columnas, cada solución debe distribuir las 8 reinas de manera que cumpla con estas restricciones.

Dificultades del problema

Este problema es no trivial, ya que el número total de configuraciones posibles en un tablero de 8x8 es 4,426,165,368 combinaciones. Sin embargo, el número de soluciones válidas es solo 92, de las cuales 12 son únicas si se excluyen rotaciones y reflejos.

Métodos de solución

Dado que el problema tiene una gran cantidad de combinaciones posibles, se utilizan algoritmos de optimización y búsqueda para encontrar soluciones de manera eficiente. Entre los métodos más comunes están:

- **Backtracking (retroceso):** explora todas las combinaciones posibles hasta encontrar una solución válida.
- **Algoritmos genéticos:** utilizan procesos evolutivos para mejorar iterativamente las configuraciones.
- **Búsqueda heurística (Hill Climbing, Simulated Annealing, Búsqueda Tabú):** buscan mejorar iterativamente las soluciones explorando vecinos cercanos.

En este caso, utilizaremos el algoritmo de búsqueda Tabú, que evita caer en ciclos y permite encontrar soluciones óptimas explorando diferentes configuraciones y evitando estados ya visitados.

Representación de P (Problema) y S (Solución) en el problema de las 8 reinas

Representación del Problema (P)

El problema de las **8 reinas** se puede representar como un estado en el que cada reina está ubicada en una fila distinta del tablero de ajedrez de 8x8. Para modelarlo, se usa una lista de tamaño 8, donde cada posición de la lista representa una fila y el valor almacenado en cada posición indica la **columna** en la que está la reina.

Ejemplo de Representación

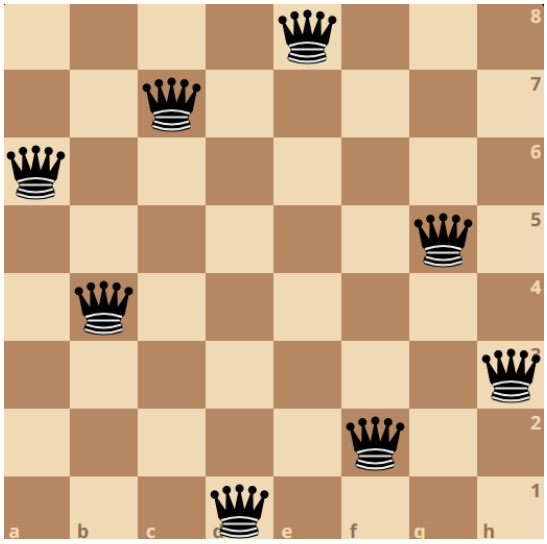
Si tenemos la lista:

tablero = [4, 2, 0, 6, 1, 7, 5, 3]

Esto significa que:

- La **primera reina** está en la fila 0, columna **4**.
- La **segunda reina** está en la fila 1, columna **2**.
- La **tercera reina** está en la fila 2, columna **0**.
- …
- La **octava reina** está en la fila 7, columna **3**.

Visualmente, este tablero se vería así:



Representación de una Solución (S)

Una **solución válida** es una configuración de la lista en la que **ninguna reina se ataque** siguiendo las reglas del ajedrez.

Por ejemplo, la siguiente lista representa una solución válida:

solucion = [4, 6, 0, 2, 7, 5, 3, 1]

Esto significa que cada reina está posicionada en una fila diferente y ninguna está en la misma columna ni en la misma diagonal que otra.

Si el algoritmo encuentra un tablero donde todas las reinas están ubicadas sin conflictos, se considera que ha encontrado una solución óptima.

Propuesta de Algoritmo en Pseudocódigo: Búsqueda Tabú para el problema de las 8 reinas

El algoritmo de **búsqueda Tabú** es un método heurístico que explora soluciones candidatas, evita ciclos utilizando una lista de estados prohibidos (*lista tabú*) y busca mejorar la solución actual iterativamente.

```
Inicio
  Definir N = 8  // Tamaño del tablero
  Definir MAX_ITER = 100  // Máximo de iteraciones
  Definir TABU_TAM = 10  // Tamaño de la lista tabú

  // Inicializar estado aleatorio
  tablero_actual ← Generar_tablero_aleatorio(N)
  mejor_solucion ← tablero_actual
  mejor_conflicto ← Calcular_conflictos(tablero_actual)
  lista_tabu ← []
  movimientos ← 0

  Para i desde 1 hasta MAX_ITER hacer
    vecinos ← Generar_vecinos(tablero_actual)
    Ordenar vecinos por número de conflictos (menor primero)
```

```
    Para cada vecino en vecinos hacer
      Si vecino NO está en lista_tabu entonces
        tablero_actual ← vecino
        conflictos_actual ← Calcular_conflictos(tablero_actual)
        Agregar tablero_actual a lista_tabu
        Incrementar movimientos

        Si tamaño de lista_tabu > TABU_TAM entonces
          Eliminar el estado más antiguo

        Si conflictos_actual < mejor_conflicto entonces
          mejor_solucion ← tablero_actual
          mejor_conflicto ← conflictos_actual
        Fin Si

        Romper (salir del bucle de vecinos)
      Fin Si
    Fin Para

    Si mejor_conflicto = 0 entonces
      Romper (solución encontrada)
    Fin Si
  Fin Para

  Retornar mejor_solucion, mejor_conflicto, movimientos
Fin
```

Explicación del Algoritmo

1. **Inicialización:**
    - Se genera un tablero aleatorio con **8 reinas colocadas al azar**.
    - Se almacena esta configuración como la mejor solución inicial.
    - Se define una lista tabú para evitar ciclos en la búsqueda.
2. **Búsqueda iterativa:**
    - Se generan **vecinos**, es decir, estados en los que una reina se mueve a otra posición en su fila.
    - Se selecciona el **vecino con menor número de conflictos** que **no esté en la lista tabú**.
    - Se actualiza la solución actual y, si es mejor que la anterior, se guarda como la mejor solución.
    - Se controla el tamaño de la lista tabú eliminando los elementos más antiguos.
3. **Criterio de terminación:**
    - Si se encuentra un estado **sin conflictos**, el algoritmo termina.
    - Si se alcanzan las **100 iteraciones**, se devuelve la mejor solución encontrada.

Ventajas del Algoritmo de Búsqueda Tabú

1.- Evita ciclos y estancamiento gracias a la lista tabú.
2.- Encuentra soluciones óptimas más rápido que métodos como backtracking.
3.- Explora soluciones vecinas sin necesidad de evaluar todas las posibilidades.

Este pseudocódigo define claramente la estrategia utilizada para resolver el problema de las **8 reinas** de manera eficiente.

Implementación en Python:

```
import random

import time


class TabuSearch:

  def __init__(self, initial_state, tabu_tenure, max_iterations):

    self.n = len(initial_state)  # Tamaño del tablero (n x n)

    self.tabu_tenure = tabu_tenure  # Duración del tabu

    self.max_iterations = max_iterations  # Número máximo de iteraciones

    self.board = initial_state[:]  # Estado inicial del tablero

    self.tabu_list = [[0 for _ in range(self.n)] for _ in range(self.n)]  # Lista tabu para evitar movimientos recientes

    self.best_solution = None  # Mejor solución encontrada

    self.best_cost = float('inf')  # Costo de la mejor solución (inicialmente infinito)

    self.total_moves = 0  # Contador de movimientos totales


  # Calcula el número de ataques entre reinas en el tablero.

  def cost(self, board):

    attacks = 0

    for i in range(self.n):

      for j in range(i + 1, self.n):

        if abs(board[i] - board[j]) == abs(i - j):

          attacks += 1

    return attacks


  # Genera un vecino del tablero actual intercambiando dos reinas.

  def generate_neighbor(self, board):

    neighbor = board[:]
```

```
        i, j = random.sample(range(self.n), 2)
        neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
        return neighbor, (i, j)

    # Realiza la búsqueda tabú para encontrar una solución al problema.
    def search(self):
        current_solution = self.board[:]
        current_cost = self.cost(current_solution)
        total_moves = 0

        for iteration in range(self.max_iterations):
            best_neighbor = None
            best_neighbor_cost = float('inf')
            best_move = None

            # Genera vecinos y selecciona el mejor vecino no tabú o que mejore la mejor solución global.
            for _ in range(self.n * self.n):
                neighbor, move = self.generate_neighbor(current_solution)
                neighbor_cost = self.cost(neighbor)

                if self.tabu_list[move[0]][move[1]] == 0 or neighbor_cost < self.best_cost:
                    if neighbor_cost < best_neighbor_cost:
                        best_neighbor = neighbor
                        best_neighbor_cost = neighbor_cost
                        best_move = move

            if best_neighbor is None:
```

```
            break

        current_solution = best_neighbor
        current_cost = best_neighbor_cost
        self.tabu_list[best_move[0]][best_move[1]] = self.tabu_tenure
        total_moves += 1  # Incrementa el contador de movimientos

        # Reduce el tiempo tabú de todos los movimientos en la lista tabú.
        for i in range(self.n):
            for j in range(self.n):
                if self.tabu_list[i][j] > 0:
                    self.tabu_list[i][j] -= 1

        # Actualiza la mejor solución si se encuentra una mejor.
        if current_cost < self.best_cost:
            self.best_solution = current_solution
            self.best_cost = current_cost

        # Si se encuentra una solución óptima (costo 0), se termina la búsqueda.
        if self.best_cost == 0:
            break

    return self.best_solution, self.best_cost, total_moves

# Función para imprimir el tablero con las reinas colocadas.
def imprimir_tablero(tablero):
    n = len(tablero)
    for i in range(n):
```

```
        row = ['.'] * n
        row[tablero[i]] = 'Q'
        print(' '.join(row))
    print()

# Función principal para resolver el problema de las N reinas utilizando búsqueda tabú.
def problema_n_reinas_tabu(initial_state, tabu_tenure=5, max_iterations=1000, num_executions=10):
    best_overall_solution = None
    best_overall_cost = float('inf')
    best_overall_moves = float('inf')
    start_time = time.time()  # Tiempo de inicio

    for _ in range(num_executions):
        tabu_search = TabuSearch(initial_state, tabu_tenure, max_iterations)
        solution, cost, moves = tabu_search.search()

        if cost < best_overall_cost or (cost == best_overall_cost and moves < best_overall_moves):
            best_overall_solution = solution
            best_overall_cost = cost
            best_overall_moves = moves

        if best_overall_cost == 0:
            break

    end_time = time.time()  # Tiempo de finalización
    total_time = end_time - start_time  # Tiempo total
```

```
    if best_overall_cost == 0:
        print("Solución óptima encontrada:")
        imprimir_tablero(best_overall_solution)
    else:
        print("No se encontró una solución óptima.")
        if best_overall_solution:
            imprimir_tablero(best_overall_solution)

    print(f"Cantidad total de movimientos: {best_overall_moves}")
    print(f"Tiempo total de ejecución: {total_time:.2f} segundos")

# Ejemplo de uso
initial_state = [0, 3, 2, 1, 6, 5, 4, 7]
problema_n_reinas_tabu(initial_state)
```

Referencias:

Chakraborty, M., & Basu, A. (2016). *Artificial intelligence and soft computing: Behavioral and cognitive modeling of the human brain*. CRC Press.

Glover, F. (1989). Tabu Search—Part I. *INFORMS Journal on Computing, 1*(3), 190-206. https://doi.org/10.1287/ijoc.1.3.190

Glover, F. (1990). Tabu Search—Part II. *INFORMS Journal on Computing, 2*(1), 4-32. https://doi.org/10.1287/ijoc.2.1.4

Kendall, G., Parkes, A. J., & Spoerer, K. (2008). A survey of NP-complete puzzles. *International Computer Games Association Journal, 31*(1), 13-34.

Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

Sörensen, K. (2013). Metaheuristics—the metaphor exposed. *International Transactions in Operational Research, 22*(1), 3-18. https://doi.org/10.1111/itor.12001