Benémerita Universidad Autónoma de Puebla

Facultad de Ciencias de La Computación

Reporte Práctica 01

El problema de las 8 reinas

Autores: Equipo 10 (Marrón)
Cedillo Garcia Bryan Arturo,
201934964
Amador Ortega Christian Amauri,
201927821
Hernandez Perez Mario,
Garcia Sanchez Bruno, 202037617

Profesor:
Pedro Tecuanhuehue Vera

Nombre del curso: Técnicas de Inteligencia Artificial

Puebla, Pue.
15 de septiembre de 2023

Índice

1. Introducción

El problema de las 8 reinas es un enigma ajedrecístico que ha desafiado a entusiastas de las matemáticas y programadores durante décadas. Fue popularizado por Max Bezzel en 1848, aunque su historia se remonta a tiempos anteriores. La premisa del problema es aparentemente simple: colocar ocho reinas en un tablero de ajedrez de 8x8 de tal manera que ninguna reina pueda amenazar a otra. Esto significa que ninguna reina debe compartir fila, columna o diagonal con otra reina.
A pesar de su enunciado sencillo, el problema de las 8 reinas es un ejemplo clásico de un problema de búsqueda y optimización que ha intrigado a matemáticos y científicos de la computación a lo largo de los años. Su resolución requiere un enfoque sistemático y estratégico, y ha dado lugar a una serie de algoritmos y técnicas de programación interesantes.
A lo largo de esta práctica, se presentará una estrategia y se demostrará su aplicación práctica para encontrar soluciones válidas al problema de las 8 reinas. Al final, habremos adquirido una comprensión más profunda de la resolución de problemas y las técnicas algorítmicas, y estaremos mejor preparados para enfrentar problemas similares en el futuro.

2. Planteamiento del problema

El problema de las ocho reinas es un pasatiempo que consiste en poner ocho reinas en el tablero de ajedrez sin que se amenacen. Fue propuesto por el ajedrecista alemán Max Bezzel en 1848. Como cada reina puede amenazar a todas las reinas que estén en la misma fila, cada una ha de situarse en una fila diferente. Podemos representar las 8 reinas mediante un vector V=(3,1,6,2,8,6,4,7) teniendo en cuenta que cada índice del vector representa una fila y el valor una columna. Es decir, V=(3,1,6,2,8,6,4,7) visto como vector de parejas ordenadas, es: V(fila,columna)=((1,3),(2,1),(3,6),(4,2),(5,8),(6,6),(7,4),(8,7))

3. Metodología

3.1. Identificación de los estados inicial y final.

- **Estado Inicial:**
  El estado inicial se refiere a la configuración inicial del tablero de ajedrez antes de que comencemos a colocar las reinas. En este caso, el estado inicial es un tablero vacío de 8x8, donde no hay reinas colocadas en ninguna posición. Las coordenadas de todas las casillas en el tablero están inicialmente en blanco. Por ejemplo, el estado inicial puede representarse de la siguiente manera, donde representa una casilla vacía:

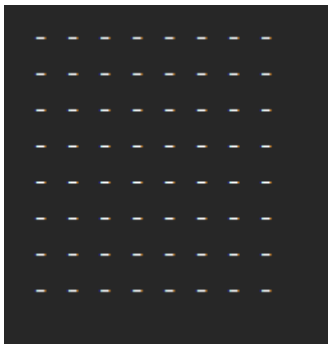

FIGURA 1. Estado inicial del problema de las 8 reinas.

- **Estado Final:**
  El estado final es el objetivo que deseamos alcanzar al resolver el problema de las 8 reinas. En el estado final, habremos colocado las 8 reinas en el tablero de tal manera que ninguna de ellas amenace a las demás. Esto significa que no habrá dos reinas en la misma fila, columna o diagonal. Un ejemplo de estado final válido podría ser el siguiente (donde Rrepresenta una reina y ünа casilla vacía):

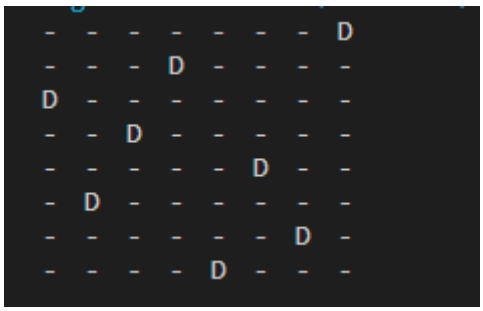

FIGURA 2. Estado final del problema de las 8 reinas.

El estado final es una solución válida para el problema de las 8 reinas, donde todas las reinas están colocadas correctamente y no se atacan mutuamente.
Identificar claramente el estado inicial y el estado final es fundamental para abordar cualquier problema de búsqueda o optimización, ya que establece los puntos de partida y llegada en la resolución del problema.

3.2. Diseño y descripción de la función objetivo. La función objetivo en el problema de las 8 reinas se utiliza para evaluar qué tan buena es una configuración particular del tablero en términos de cumplir con las restricciones del problema. En este caso, la función objetivo busca minimizar la cantidad de reinas que se amenazan mutuamente en el tablero. Una configuración óptima tendrá cero reinas amenazándose entre sí, lo que indica que hemos encontrado una solución válida.

Diseño de la función objetivo:

1. **Iniciar con un valor de puntuación igual a cero.**

- Al principio, no hemos colocado ninguna reina en el tablero, por lo que no hay amenazas entre reinas.

2. **Por cada reina en el tablero, evaluar las amenazas.**
   - Recorrer cada reina en el tablero y contar cuántas otras reinas la amenazan. Esto incluye reinas en la misma fila, columna o diagonal.
   - Para cada reina, agregamos la cantidad de reinas que la amenazan a la puntuación total.
3. **El valor de la función objetivo es la puntuación acumulada.**
   - La función objetivo se define como la suma de las amenazas de todas las reinas en el tablero.
4. **El objetivo es minimizar la función objetivo.**
   - Dado que estamos buscando una solución óptima, nuestro objetivo es minimizar la puntuación de la función objetivo. La solución óptima tendrá una puntuación de cero, lo que indica que no hay reinas amenazándose mutuamente.

Descripción de la Función Objetivo:
La función objetivo se encarga de cuantificar la calidad de una configuración dada del tablero en términos de cumplir con las restricciones del problema. Una puntuación más baja indica una configuración mejor, ya que significa que hay menos reinas amenazándose entre sí. El proceso de evaluación de la función objetivo implica identificar todas las amenazas entre reinas en el tablero y sumarlas. Cuantas menos amenazas haya, mejor será la configuración.

El objetivo final de la búsqueda es encontrar una configuración del tablero que minimice esta puntuación, es decir, una solución en la que ninguna reina amenace a otra. Una vez que la función objetivo alcanza un valor de cero, hemos encontrado una solución válida para el problema de las 8 reinas. La función objetivo desempeña un papel fundamental en la resolución del problema, ya que guía la búsqueda hacia soluciones cada vez más óptimas a medida que se aplican algoritmos de búsqueda y optimización.

3.3. Diseño y descripción de la función sucesor. La función sucesor en el contexto del problema de las 8 reinas se utiliza para generar configuraciones candidatas del tablero a partir de la configuración actual, con la esperanza de acercarnos a una solución óptima. Cada configuración candidata se obtiene al mover una reina a una nueva posición en su misma columna.

Diseño de la Función Sucesor:

1. Seleccionar una reina en el tablero.
   - En cada paso, seleccionamos una de las reinas en el tablero. Esto se puede hacer de manera aleatoria o siguiendo un orden específico, dependiendo del algoritmo que estemos utilizando.
2. Generar configuraciones candidatas.

   - Para la reina seleccionada, generamos configuraciones candidatas al moverla a todas las posibles casillas en su misma columna. Esto significa que solo estamos cambiando la fila de la reina, manteniendo su columna fija.
3. Evaluar cada configuración candidata.
   - Para cada configuración candidata, evaluamos su calidad utilizando la función objetivo. Es decir, calculamos la cantidad de amenazas que existen en la nueva configuración.
4. Seleccionar la mejor configuración candidata.
   - La función sucesor selecciona la configuración candidata que minimiza la puntuación de la función objetivo. Esto significa que estamos eligiendo el movimiento que reduce la cantidad de reinas que se amenazan mutuamente.
5. Actualizar el tablero con la mejor configuración candidata.
   - Una vez que se ha seleccionado la mejor configuración candidata, actualizamos el tablero moviendo la reina a la nueva posición. Esto nos da una nueva configuración del tablero que está más cerca de una solución óptima.

3.4. Diseño y orden de los operadores. En la función sucesor, primero se comprueba el operador çolumnaz luego se aplica el operador "diagonal.ªunque ambos son consultados al mismo tiempo dentro de un condicional (if). Si uno de los dos se cumple, descartamos el estado actual

- **Operador de Columna:** Verificamos si podemos colocar una reina en una columna determinada sin que choque con las demás reinas en la misma columna.
  Para este operador, se verifica si dos reinas están en la misma columna. Esto se hace comparando las columnas en las que se encuentran dos reinas diferentes. La comparación se realiza con la operación: nodo.estado[i] == columna.
  Donde nodo.estado[i] representa la columna en la que está ubicada la reina i, y columna es la columna en la que estamos considerando colocar una nueva reina. Si esta igualdad es verdadera para algún valor de i, significa que ya hay una reina en esa columna y no podemos colocar otra allí.
- **Operador de Diagonal:** Verificamos si podemos colocar una reina en una casilla de tal manera que no choque con las demás reinas en diagonales ascendentes y descendentes.
  Este operador verifica si dos reinas están en la misma diagonal, ya sea ascendente o descendente. Esto se hace comparando las diferencias entre las filas y columnas de las dos reinas.
  La comparación se realiza con la operación: abs(nodo.estado[i] - columna) == abs(i - fila_actual). Donde nodo.estado[i] representa la columna en la que está ubicada la reina i, columna es la columna en la que estamos considerando colocar una nueva reina, i es el índice de la reina en la que estamos iterando y fila_actual es la fila en la que estamos considerando colocar la nueva reina. Esta operación verifica si las reinas tienen la misma diferencia entre sus filas y columnas en valor absoluto, lo que significa que están en la misma diagonal.

4. Diseño y descripción del algoritmo primero en profundidad(Depth-First)

- Diseño del Algoritmo Primero en Profundidad (Depth-First Search - DFS)
  1. Nodo y Representación del Problema
     - El problema se representa mediante una clase Nodo, que contiene el estado actual del tablero (una lista que representa la ubicación de las reinas) y la cantidad de reinas_colocadas.
  2. Función Objetivo (funcion_objetivo)
     - La función objetivo verifica si hemos colocado todas las 8 reinas en el tablero. Si la cantidad de reinas colocadas es igual a 8, entonces hemos encontrado una solución válida.
  3. Función Sucesor (funcion_sucesor)
     - La función sucesor genera sucesores válidos a partir del estado actual del tablero. Itera a través de las columnas y verifica si es seguro colocar una reina en una fila y columna específicas. Si es seguro, crea un nuevo nodo con la reina agregada en esa posición.
  4. Algoritmo de Búsqueda DFS (Algoritmo_Busqueda_DFS)
     - Comienza con un estado inicial donde no se ha colocado ninguna reina (Nodo([], 0)).
     - Utiliza una pila (LIFO) para implementar la búsqueda en profundidad. Se agrega el estado inicial a la pila.
     - Utiliza un conjunto estados_visitados para registrar los estados ya visitados y evitar bucles infinitos.
     - El bucle principal se ejecuta mientras haya elementos en la pila.
     - En cada iteración, se saca el nodo superior de la pila y se verifica si su estado ya ha sido visitado.
     - Si el estado no ha sido visitado, se agrega a estados_visitados.
     - Si se alcanza la función objetivo (funcion_objetivo), significa que se ha encontrado una solución y el algoritmo termina. La solución se encuentra en nodo_actual.estado.
  5. Función de Costo (funcion_costo)
     - La función de costo devuelve la cantidad de estados visitados antes de encontrar la solución. Resta 1 para excluir el estado inicial.
  6. Main (if __name__ == "__main__":)
     - En el bloque principal, se configura el valor de n (en este caso, 8) y se llama a Algoritmo_Busqueda_DFS para encontrar una solución al problema de las 8 reinas.
     - Si se encuentra una solución, se imprime el tablero con las reinas colocadas y el costo (la cantidad de nodos visitados).
     - Si no se encuentra una solución, se muestra un mensaje indicando que no se encontró solución.
- Descripción del Algoritmo DFS El algoritmo de búsqueda en profundidad (DFS) es una técnica de búsqueda no informada que explora un árbol de estados de manera profunda antes de retroceder. En el contexto del problema de las 8 reinas, el algoritmo DFS comienza con un estado inicial (ninguna reina colocada) y avanza hacia estados sucesores válidos, explorando todas las posibilidades antes de retroceder cuando no es posible continuar.

  El algoritmo mantiene una pila de nodos (estados) a explorar, donde siempre se toma el

nodo más reciente de la pila (LIFO) para expandirlo. Se utiliza un conjunto para evitar visitar estados duplicados y asegurarse de que el algoritmo no se atasque en bucles infinitos.
El algoritmo continúa explorando estados hasta que se encuentra una solución (todas las reinas colocadas) o hasta que se hayan explorado todos los nodos posibles. Cada vez que se encuentra una solución, se almacena y se calcula el costo (la cantidad de nodos visitados).
En resumen, el algoritmo DFS es una forma sistemática de explorar el espacio de búsqueda del problema de las 8 reinas, asegurando que cada reina esté en una fila y columna diferentes y que no compartan diagonales ascendentes o descendentes.'

- ¿Porque se recomienda el uso del algoritmo búsqueda en profundidad (DFS) antes que el algortimo de búsqueda en anchura (BFS).?
  La elección entre el algoritmo de búsqueda en profundidad (DFS) y el algoritmo de búsqueda en anchura (BFS) depende de las características específicas del problema y de los objetivos que tengas en mente. En el caso del problema de las 8 reinas, el uso de DFS es más recomendado que BFS por varias razones:
  1. Eficiencia en Espacio El algoritmo DFS tiende a ser más eficiente en términos de uso de memoria en comparación con BFS. Esto se debe a que DFS utiliza una estructura de datos tipo pila (LIFO) para almacenar los nodos pendientes de exploración, lo que significa que solo se almacenan los nodos de un camino particular desde el nodo raíz hasta el nodo actual. En contraste, BFS utiliza una estructura tipo cola (FIFO) que almacena todos los nodos en el nivel actual antes de avanzar al siguiente nivel. En problemas con espacios de búsqueda grandes como el de las 8 reinas, DFS puede ser más eficiente en términos de espacio.
  2. Compleción más Rápida En el problema de las 8 reinas, la mayoría de las soluciones se encuentran a una profundidad menor en el árbol de búsqueda. DFS tiende a explorar rápidamente una rama del árbol hasta que encuentra una solución o una solución inválida, lo que suele ocurrir antes en el árbol de búsqueda en comparación con BFS. Esto significa que DFS tiende a encontrar soluciones más rápidamente en promedio.
  3. Optimización para Profundidad Dado que el problema de las 8 reinas se trata de encontrar una solución que cumpla con restricciones específicas (ninguna reina se amenaza mutuamente), DFS es una elección natural ya que se adapta bien a la profundidad del árbol de búsqueda. BFS, en cambio, podría requerir una cantidad significativa de memoria para almacenar todos los estados en cada nivel, lo que podría no ser práctico en términos de uso de memoria.
  4. No se Requiere la Búsqueda Completa En el problema de las 8 reinas, no es necesario explorar todas las posibles soluciones (nodos hoja del árbol de búsqueda). Una vez que se encuentra una solución válida, no se necesita continuar explorando. DFS es adecuado para este tipo de problema porque puede detenerse tan pronto como se encuentra una solución, mientras que BFS generalmente continuaría explorando todos los niveles antes de detenerse.

5. Reporte de la trayectoria de la solución encontrada mediante el algoritmo Depth-First

Nuestro algoritmo de búsqueda, implementado en la función "Algoritmo_Busqueda_DFS()"funciona de la siguiente manera:

- **Estado Inicial:** Comenzamos con un estado inicial vacío en el que no hemos colocado ninguna reina en el tablero.
- **Operadores:** Para generar sucesores a partir del estado actual, aplicamos los dos operadores mencionados en el apartado 3.4
- **Estado Objetivo:** El estado objetivo es aquel en el que hemos colocado las 8 reinas en el tablero de manera que no se amenacen mutuamente, es decir, cumpliendo las condiciones del problema.
- **Exploración:** Comenzamos la exploración desde el estado inicial y generamos sucesores aplicando los operadores. Cada sucesor representa un nuevo estado con una reina colocada en el tablero. Si el estado generado cumple con las condiciones del problema, es decir, no hay choques entre las reinas, lo consideramos como una solución.
- **Backtracking:** Si encontramos un estado en el que no es posible colocar una reina sin que se amenace con las demás, retrocedemos (backtracking) al estado anterior y exploramos otras posibilidades.
- **Estados Visitados:** Llevamos un registro de los estados que hemos visitado para evitar la exploración infinita de caminos repetidos. Esto lo hacemos utilizando un conjunto (estados_visitados) para asegurarnos de que no estamos repitiendo estados.
- **Costo:** El costo en este contexto se refiere a la cantidad de estados visitados antes de encontrar la solución. Se cuenta cada estado que exploramos, incluido el estado inicial.
- **Resultado:** Una vez que encontramos una solución (es decir, un estado en el que hemos colocado las 8 reinas sin conflictos), el algoritmo se detiene y muestra el tablero con las reinas colocadas y el costo (nodos visitados) que se ha incurrido durante la búsqueda.

1. Colocación de la Primera Reina: Colocamos la primera reina en la posición (0, 7) del tablero. El tablero se ve así:

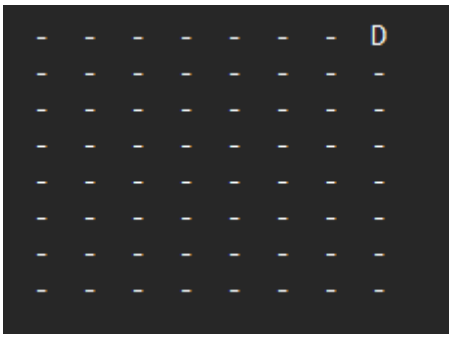

Figura 3. Primera reina colocada.

2. Colocación de la Segunda Reina: Continuamos colocando la segunda reina en la posición (1, 3) del tablero, evitando conflictos con la primera reina:

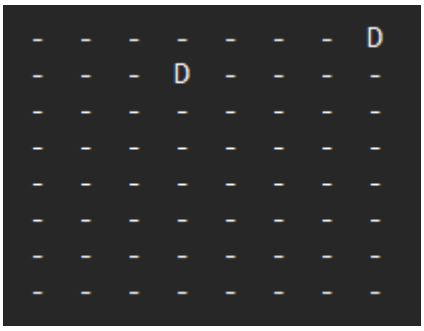

FIGURA 4. Segunda reina colocada

3. Colocación de la Tercera Reina: Proseguimos colocando la tercera reina en la posición (2, 0), evitando conflictos:

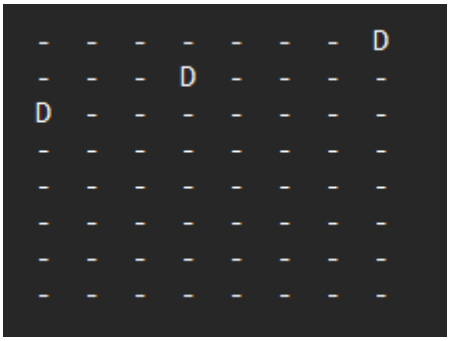

FIGURA 5. Tercera reina colocada

4. Colocación de las Reinas Restantes: Repetimos el proceso para colocar las reinas restantes, asegurando que ninguna se amenace mutuamente.
   - Resultado Final Después de completar la colocación de las 8 reinas, se obtiene una solución válida:

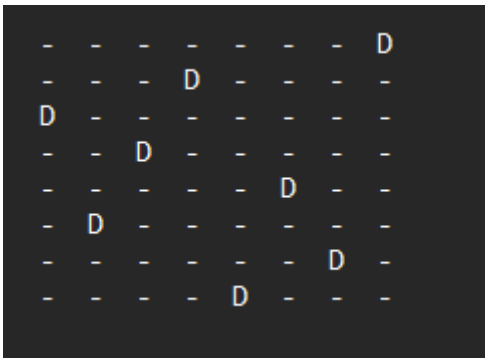

FIGURA 6. Solucion final con todas las reinas en el tablero

5. Costo de la Solución Se visitaron un total de [113] estados antes de encontrar esta solución.

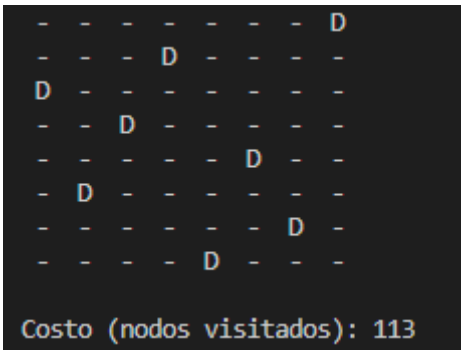

FIGURA 7. Resultado final junto a su costo total.

6. ANÁLISIS DE COMPLEJIDAD DEL ALGORITMO PRIMERO EN PROFUNDIDAD

El orden de complejidad temporal, también conocido como .ªnálisis de complejidad temporal.º "notación big O,.es una medida de cuánto tiempo tarda un algoritmo en ejecutarse en función del tamaño de la entrada. En el contexto del algoritmo de profundidad, esta medida se refiere a cuántos pasos u operaciones necesita el algoritmo para resolver un problema en función del número de nodos o elementos en una estructura de datos, como un grafo o un árbol. Para obtener el orden de complejidad temporal del algoritmo de profundidad, se realiza un análisis detallado de su ejecución. En el caso del recorrido en profundidad de un grafo o árbol, el algoritmo suele implementarse mediante una función recursiva o utilizando una pila. En general si se realiza un recorrido en profundidad de un grafo con n nodos y m aristas, su orden de complejidad suele ser $O(n + m)$, ya que en el peor de los casos se visita cada nodo una vez y se recorre cada arista una vez. Cabe recalcar que esta medida puede variar en función de cómo se pudiese estar representando el árbol en cuestión (lista de adyacencia, matriz de adyacencia, etc.), o bien de la forma específica de su implementación ya que la ejecución con métodos recursivos puede generar fácilmente desbordamientos de bits para grafos relativamente grandes.

Para calcular la complejidad temporal del algoritmo de búsqueda que implementamos en la función Algoritmo_Busqueda_DFS(n), primero debemos analizar su comportamiento en términos de tiempo en función de la entrada n, que representa el tamaño del tablero (tamaño del problema)

El algoritmo DFS explorará un árbol de búsqueda hasta encontrar una solución o agotar todas las posibilidades. En el peor caso, el algoritmo DFS explorará todos los nodos del árbol de búsqueda antes de encontrar una solución. Esto significa que visitará todos los estados posibles del tablero. La cantidad de estados posibles en el tablero de las 8 reinas es factorial(n), ya que en cada fila se puede colocar una reina en una de las n columnas disponibles, y hay n filas en total.

Por lo tanto, la complejidad temporal en el peor caso es O(n!), lo que significa que el tiempo requerido para encontrar una solución crecerá de manera factorial con respecto al tamaño del tablero. Esto hace que el algoritmo sea poco eficiente para tamaños de tablero grandes.

Sin embargo esta complejidad se refiere al peor caso, y en la práctica, el algoritmo DFS puede encontrar soluciones mucho más rápido para instancias típicas del problema (como n = 8, que es lo normal). Aunque para tableros grandes, esta complejidad puede hacer que el algoritmo sea prohibitivamente lento. Para mejorar la eficiencia en tableros grandes, se pueden explorar enfoques de búsqueda más avanzados o heurísticas específicas para el problema de las 8 reinas.

7. Apéndices

7.1. Código del programa.

```
1
2  class Nodo:
3      def __init__(self, estado, reinas_colocadas):
4          self.estado = estado  #Lista que representa la ubicacion de las reinas.
5          self.reinas_colocadas = reinas_colocadas
6
7
8  def funcion_objetivo(nodo, n):
9      #Si hemos colocado 8 reinas, terminamos (el algoritmo no permite agregar
           nuevas reinas
10     #si se amenazan unas a otras).
11     return nodo.reinas_colocadas == n
12
13
14 def funcion_sucesor(nodo, n):
15     sucesores = []
16
17     for columna in range(n): #Recorremos el tablero...
18         fila_actual = nodo.reinas_colocadas
19
20         #Verificar si es seguro colocar una reina en la fila_actual y columna
               actual.
21         seguro = True
22
23         #OPERADORES:
24         for i in range(nodo.reinas_colocadas):
25             if (
26                 nodo.estado[i] == columna or  #OPERADOR 1: COLUMNA
27                 abs(nodo.estado[i] - columna) == abs(i - fila_actual)  #OPERADOR
                       2: DIAGONAL
28             ):
29                 seguro = False
30                 break
31
32         if seguro:
33             #Si es seguro, agregamos la reina en la columna y creamos un nuevo
                   nodo sucesor.
34             nuevo_estado = nodo.estado[:] + [columna]
35             sucesores.append(Nodo(nuevo_estado, fila_actual + 1))
```

```
36
37      return sucesores
38
39
40  def Algoritmo_Busqueda_DFS(n):
41      GA = Nodo([], 0)  #ESTADO INICIAL: vector prometedor inicial con 0 reinas en
            el tablero
42      pila = [GA]  #Utilizamos una PILA (FIFO) para implementar DFS (Depth First
            Search).
43
44      estados_visitados = set()  #Conjunto para registrar estados visitados.
45
46      while pila:
47          nodo_actual = pila.pop()
48          estado_actual = tuple(nodo_actual.estado)  #Convertimos la lista de
                estado a una tupla para ser hashable.
49
50          if estado_actual in estados_visitados:
51              continue  #Si ya hemos visitado este estado, lo saltamos y
                    continuamos con el siguiente.
52
53          estados_visitados.add(estado_actual)  # Agregamos el estado a
                estados_visitados.
54
55          if funcion_objetivo(nodo_actual, n):
56              # nodo_actual.estado es el equivalente a Vk. Si en algun momento
                    retornamos
57              # nodo_actual.estado, entonces encontramos una solucion. Y termina
                    el algoritmo.
58              return nodo_actual.estado, len(estados_visitados)
59
60          #Si no, generamos mas sucesores
61          sucesores = funcion_sucesor(nodo_actual, n)
62          pila.extend(sucesores)  # PUSH
63
64      return None, len(estados_visitados)  # No se encontro una solucion.
65
66
67  def funcion_costo(estados_visitados):
68      # Retorna la cantidad de estados visitados antes de encontrar la solucion.
69      return estados_visitados - 1  # Restamos 1 para excluir el estado inicial.
70
71
72  if __name__ == "__main__":
73      n = 8
74      solucion, estados_visitados = Algoritmo_Busqueda_DFS(n)
75
76      if solucion:
77          for fila in range(n):
78              print(" . " * solucion[fila] + " D " + " . " * (n - solucion[fila] -
                    1))
79          costo = funcion_costo(estados_visitados)
80          print(f"\nCosto (nodos visitados): {costo}")
81      else:
82          print("No se encontro solucion.")
```

7.2. Manual de usuario.

- **Introducción y requisitos:** El código proporcionado resuelve el clásico problema de las 8 reinas del ajedrez utilizando el algoritmo Depth-First Search (DFS). El objetivo es colocar 8 reinas en un tablero de ajedrez de 8x8 de manera que ninguna reina amenace a otra. El código se ejecuta en Python 3.x. por lo que tu computadora debe tenerlo descargado.
- **Instrucciones:** Descarga el código fuente proporcionado en un archivo .py. Abre una terminal o consola de comandos. Navega hasta el directorio donde se encuentra el archivo .py descargado. Ejecuta el programa con el siguiente comando:
  bash
  Copy code
  main.py
  O si cuentas con un IDE que trabaje con python, crea un archivo.py nuevo, copia y pega el código fuente, y ejecútalo.
- **Resultados:**
  - a. Solución Encontrada: Si el programa encuentra una solución válida, mostrará el tablero con la ubicación de las 8 reinas. Las reinas se representan con "Dƶ las casillas vacías con ".".
    Junto a la solución, se mostrará el costo, que es la cantidad de estados visitados (nodos) antes de encontrar la solución.
  - b. Sin Solución: Si el programa "no puede encontrar una solución", mostrará el mensaje "No se encontró solución."(aunque por la naturaleza del código y del problema, este código siempre va a ecntrar una solución)(la primera solución encontrada tras aplicar el algoritmo DFS)(el fragmento "solucion no ecntrada"simplemente se presenta al usuario del código como una formalidad.)
- **Consideraciones Adicionales:** El programa utiliza el algoritmo DFS para explorar profundamente el espacio de búsqueda, lo que significa que puede tomar un tiempo considerable antes de encontrar una solución. El código se puede personalizar para cambiar el tamaño del tablero modificando el valor de la variable "n".