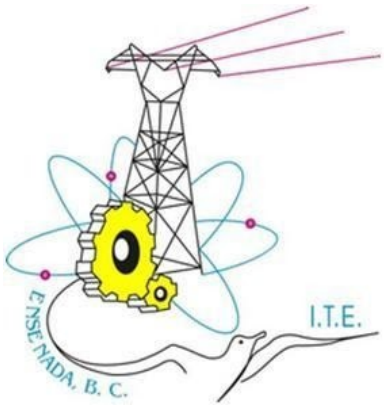
Tecnológico Nacional de México

Instituto Tecnológico de Ensenada

“Por la tecnología de hoy y del futuro”



El problema de las 8 reinas

Ingeniería en sistemas computacionales

Alumno: Jesus Luque Espinoza

Inteligencia artificial

Grupo 8SA

Profesor: Eddie Helbert Clemente Torres

Ensenada, B.C. 20 de octubre de 2020

Introducción

El problema de las 8 reinas, propuesto por Max Bezzel en 1848, consiste en situar 8 damas en un tablero de ajedrez de tal forma que no existan dos en la misma fila, columna o diagonal. Existen 92 soluciones, de las cuales 12 son únicas y las 80 restantes son obtenidas por giros y simetrías. El problema de las 8 reinas es utilizado para aplicar los algoritmos de búsqueda, que aunque no tiene utilidad alguna, es un buen ejemplo para entender sus aplicaciones.

A continuación se muestran los algoritmos BFS y DFS para resolver este problema.

Breadth First Search – Búsqueda en anchura

La clase ReinaBfs (Imagen 1) contiene un arreglo cola, que es en donde se guardará la solución, reinas, que es la cantidad de reinas a colocar y por tanto el tamaño del tablero, y coordenadas, que servirán para definir en que casillas irán las reinas.

```python
import copy
from time import time

class ReinaBfs:
    cola = []   #
    reinas = 0 # indica el tamaño del tablero
    coord = ['a', 'b', 'c', 'd', 'e', 'f', 'g','h']

    def __init__(self, n):
        self.reinas=n
```

Imagen 1: Clase ReinaBfs.

Además las clase ReinasBfs contiene las siguientes funciones para poder llegar a el resultado.

Función crearTablero(). Se crea un tablero en blanco, donde cada 0 en una posición indica que la casilla no está ocupada (Imagen 2).

```python
def crearTablero(self):
    tablero = []
    for fila in range(self.reinas):
        casillas = []
        for column in range(self.reinas):
            casillas.append(0)
        tablero.append(casillas)
    return tablero
```

Imagen 2: Función crearTablero().

Función colocarReina(). Se coloca un entero 1 en la matríz de tablero, indicando que existe una reina en esa casilla (Imagen 3).

```
def colocarReina(self, tablero, fila, columna):
    tablero = copy.deepcopy(tablero);
    tablero[fila][columna] = 1
    return tablero
```

Imagen 3: Función colocarReina().

Función colocar(). Se verifica si se puede colocar una reina en la posición recibida, si en el eje horizontal, vertical y diagonales del tablero existe una reina (entero 1), devuelve un False, de lo contrario devuelve un True (Imagen 4).

```
def colocar(self, tablero, fila, columna):
    # horizantal
    x = 0
    while(x<self.reinas):
        if tablero[x][columna]:
            return False
        x += 1

    # vertical
    y = 0
    while(y<self.reinas):
        if tablero[fila][y]:
            return False
        y += 1

    # diagonal superior derecha
    x = fila + 1
    y = columna + 1
    while x < self.reinas and y < self.reinas:
        if tablero[x][y]:
            return False
        x += 1
        y += 1

    # diagonal superior izquierda
    x = fila - 1
    y = columna - 1
    while x >= 0 and y >= 0:
        if tablero[x][y]:
            return False
        x -= 1
        y -= 1

    # diagonal inferior izquierda
    x = fila - 1
    y = columna + 1
    while x >= 0 and y < self.reinas:
        if tablero[x][y]:
            return False
        x -= 1
        y += 1

    # diagonal inferior derecha
    x = fila + 1
    y = columna - 1
    while x < self.reinas and y >= 0:
        if tablero[x][y]:
            return False
        x += 1
        y -= 1

    return True
```

Imagen 4: Función colocar().

Función conPosiciones(). De acuerdo a la cantidad de reinas, se crea un bucle que se repita esa cantidad, para determinar si se puede colocar una reina en la columna recibida, usando la función

colocar(). En caso de que sea válido, se añadirá este movimiento a la lista de posiciones posibles (Imagen 5).

```
def conPosiciones(self, tablero, columna):
    fila = 0
    posiciones = []
    while fila < self.reinas:
        if self.colocar(tablero, fila, columna):
            # se añade el tablero resultante al colocar una reina
            # en las posiciones
            posiciones.append(self.colocarReina(tablero, fila, columna))
        fila += 1
    return posiciones
```

Imagen 5: Función conPosiciones().

Función fin(). Se da fin a la ejecución del BFS, imprime la solución buscando un entero 1 en la matriz del tablero para componenr la notación algebraica (Imagen 6).

```
def fin(self, tablero, columna):
    if columna == self.reinas:
        solucion = []
        for x in range(self.reinas):
            for y in range(self.reinas):
                if (tablero[x][y] == 1):
                    solucion.append(self.coord[y]+str(x+1))
        print(solucion)
    else:
        print("Ha ocurrido un error!")
```

Imagen 6: Función fin().

Función inicio(). Se inicializa el tiempo de ejecución y se ejecuta el BFS (Imagen 7). Al terminar, se imprime el tiempo de ejecución y se llama a la función para que imprima el resultado.

```python
def inicio(self):
    tiempo_i = time()
    # se consiguen todos las posiciones iniciales
    columna = 0
    self.cola = self.conPosiciones(self.crearTablero(),columna)

    # posicionar en la columna 1
    columna +=1

    # ciclar hasta que todos los movimientos posibles se consigan
    while len(self.cola):
        temp_cola = []

        # recorrer cada etapa, para buscar en su respectiva columna(nodo vecino)
        # todos los posibles movimientos, cada etapa es un tablero al que no se
        # le han colocado todas las reinas
        for etapa in self.cola:
            # se consiguen los nuevos movimientos disponibles de esa etapa
            movimientos = self.conPosiciones(etapa, columna)

            # si se coloca la última reina, se rompe el ciclo
            if columna == self.reinas -1 and len(movimientos):
                self.cola = []
                etapa = movimientos[0]
                break;

            # si existen nuevos posibles movimientos,
            # se añaden a la cola temporal,
            if len(movimientos):
                temp_cola += movimientos

        # se asgina a cola la cola temporal, para conseguir los posibles
        # movimientos apartir de esta
        self.cola = temp_cola
        columna += 1

    tiempo = time() - tiempo_i
    print("Tiempo de ejecución: %0.10f segundos" % tiempo)

    # enviar la etapa final (tablero con todas fichas)
    # y la columna(cantidad de casillas en la fila)
    print(columna)
    self.fin(etapa, columna)
```

Imagen 7: Funcion inicio().

Función main(). Todo lo anterior está definido dentro de la clase ReinaBfs, fuera de esta se encuentra la función main(), en ella se instancia la clase ReinaBfs indicando la cantidad de reinas y se le da inicio al algoritmo (Imagen 8).

```python
def main():
    rbfs = ReinaBfs(reinas)
    rbfs.inicio()

print("Búsqueda por anchura, cantidad de reinas: 8")
reinas = 8

if __name__ == '__main__':
    main()
```

Imagen 8: Función main().

Ejecución. Al ejecutar el algoritmo, el tiempo de calculo es al rededor de 0.096 segundos. Dando como resultado las posiciones [a1, g2, e3, h4, b5, d6, f7, c8] (Imagen 9).

```
C:\Users\fidxl\ia>python bfs_reinas.py
Búsqueda por anchura, cantidad de reinas: 8
Tiempo de ejecución: 0.0960040092 segundos
Solución:  ['a1', 'g2', 'e3', 'h4', 'b5', 'd6', 'f7', 'c8']
```

Imagen 9: Ejecución del algoritmo.

Tablero. Las posiciones en el tablero son las siguientes (Imagen 10).

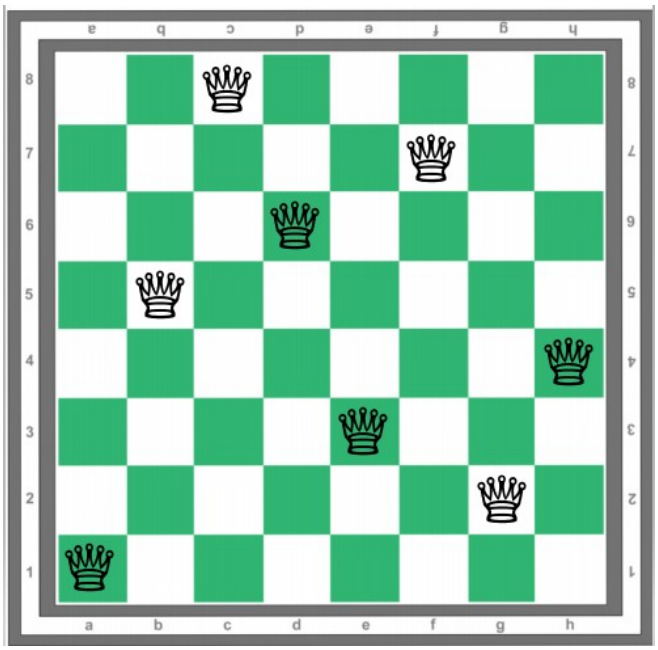


Imagen 10: Solución en tablero.

Deep First Search – Búsqueda en profundidad

Función ReinaDfs(). Aquí se recorre cada uno de los caminos posibles, colocando las piezas y evaluando si entra en conflicto en la posición colocada (Imagen 11).

```python
from time import time

def ReinaDfs(solucion,etapa,reinas):
    # si el nivel(etapa) que se evalua es mayor
    # a la cantidad de reinas se devuelve un False
    if etapa>=reinas:
        return False
    # inicializamos exito a False
    exito = False

    while True:

        # se verifica que el valor de la columna dentro de la fila
        # no supera el número de reinas (tamaño del tablero)
        if(solucion[etapa] < reinas):
            # se incrementa el valor de la columna a ser evaluada
            solucion[etapa] += 1

        # se verifica que etapa actual está en una posición válida,
        # es decir, si no se encuentra en conflicto con otra reina
        if (Valido(solucion,etapa)):

            # si la etapa en la que se encuentra no es la última,
            # se avanza a la siguiente etapa
            if etapa != reinas-1:
                exito = ReinaDfs(solucion, etapa+1,reinas)

                # si exito es false significa que el nivel siguiente
                # no es la solución,
                # por tanto se le asigna un 0 para no ser contado
                if exito==False:
                    solucion[etapa+1] = 0

            # de lo contrario, si ya se llegó al nodo(etapa) final
            # del camino correcto se devuelve un True
            else:
                exito = True

        # se sale del ciclo si el valor de la columna es
        # igual al número de reinas o si ya se ha llegado a la solución
        # y devuelve el valor de la operación
        if (solucion[etapa]==reinas or exito==True):
            return exito

    return exito
```

Imagen 11: Función ReinaDfs.

Función Valido(). Recibe un camino y la etapa del camino a evaluar (Imagen 12). Si la posición que contiene la etapa es igual a la posición de etapas anteriores, entra en conflicto por que estaría ubicado en la misma columna, por lo tanto devuelve un False de que la posición no es válida. En caso contrario devuelve un True.

```python
# Recibe un vector, este vector es la posible solución,
# para comprobarlo se verifica que se puede colocar una reina
# en la columna de esta etapa. Devuelve un True si es posible,
# un False en caso contrario

def Valido(solucion,etapa):
    for i in range(etapa):
        if(solucion[i] == solucion[etapa]) or (ValAbs(solucion[i],solucion[etapa])==ValAbs(i,etapa)):
            return False
    return True

def ValAbs(x,y):
    if x>y:
        return x - y
    else:
        return y - x
```

Imagen 12: Función Valido()

Rutina de ejecución. Aquí se manda a llamar a ReinaDfs, si devuelve un True, revisa el arreglo e imprime el resultado (Imagen 13).

```python
reinas = 8
print ("Búsqueda por profundidad, cantidad de reinas:", reinas)

coord = ['a', 'b', 'c', 'd', 'e', 'f', 'g','h']
solucion = []
notacion = []
tiempo_inicial = time()
for i in range(reinas):
    solucion.append(0)
etapa = 0

resultado = ReinaDfs(solucion, etapa, reinas)
tiempo = time() - tiempo_inicial

# se recorre la lista de resultados
# para sacar las posiciones de las damas
if resultado:
    for x in range(reinas):
        notacion.append(coord[x]+str(solucion[x]))
    print("Tiempo de ejecución: %0.10f segundos" % tiempo)
    print("Las posiciones de las reinas son: ", notacion)
else:
    print("Ha ocurrido un error")
```

Imagen 13: Rutina de ejecución

Ejecución. Al ejecutar el algoritmo (Imagen 14), el tiempo de calculo es al rededor de 0.001 segundos. Dando como resultado las posiciones [a1, b5, c8, d6, e3, f7, g2, h4] (Imagen 9).

```
C:\Users\fidxl\ia>python main.py
Búsqueda por profundidad, cantidad de reinas: 8
Tiempo de ejecución: 0.0010001659 segundos
Las posiciones de las reinas son:  ['a1', 'b5', 'c8', 'd6', 'e3', 'f7', 'g2', 'h4']
```

Imagen 14: Resultado de ejecución

Tablero. Las posiciones en el tablero son las siguientes (Imagen 15).

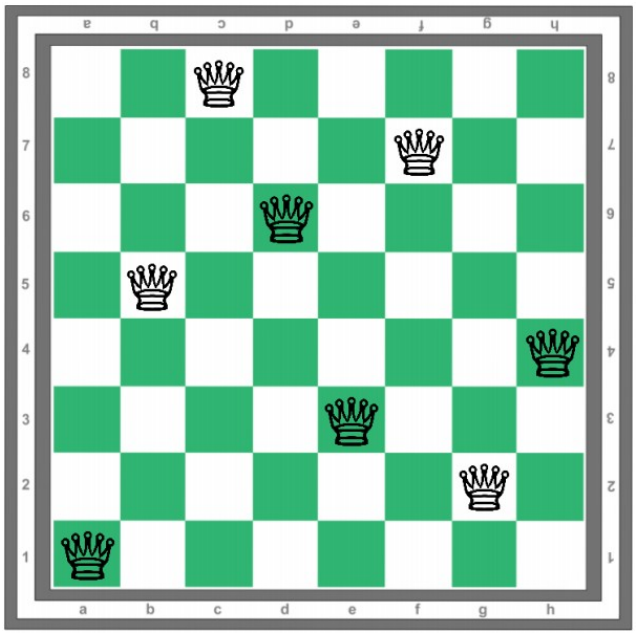


Imagen 15: Resultado en tablero