Backtracking – IV

Clase #29

IIC1103 – Introducción a la Programación

Marcos Sepúlveda (marcos@ing.puc.cl)

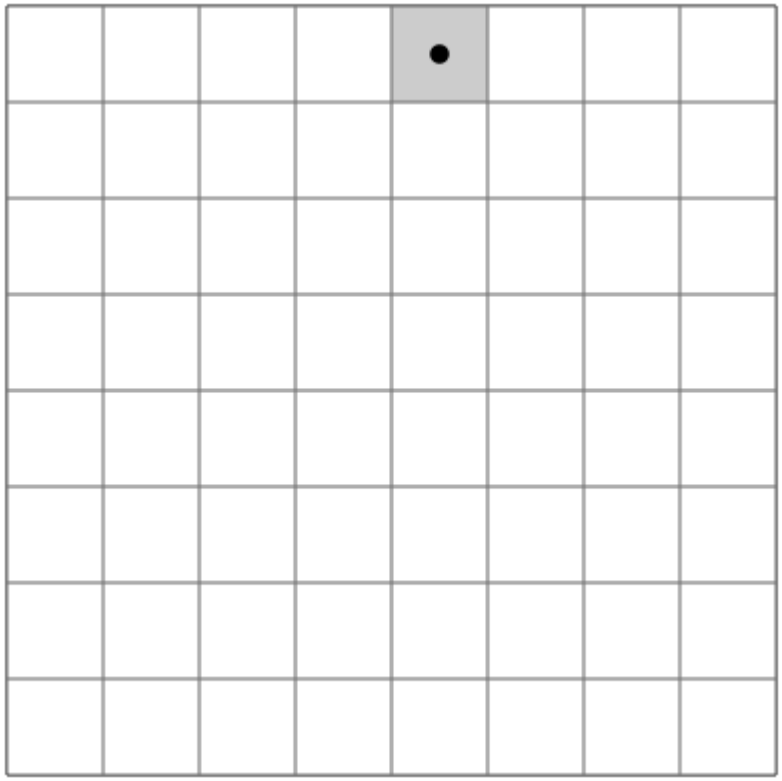
Veremos hoy …

- Ejemplo Problema del caballo

Problema del caballo

- “El problema del caballo es un antiguo problema matemático en el que se pide que, teniendo una cuadrícula de **n** x **n** casillas y un caballo de ajedrez colocado en una posición cualquiera **(x, y)**, el caballo pase por todas las casillas y una sola vez. Lo que resulta en $\mathbf{n^2-1}$ movimientos.”

Fuente: https://es.wikipedia.org/wiki/Problema_del_caballo



Fuente: https://es.wikipedia.org/wiki/Archivo:Knight%27s_tour_anim_2.gif

Backtracking – estructura genérica

```
# Retorna True si vale la pena explorar la solución candidata
def SirveComoSolucionParcial(solucion_candidata):
    pass

# Retorna True si la solución candidata resuelve el problema
def SirveComoSolucionFinal(solucion_candidata):
    pass

# Usa la solución candidata como una solución valida al problema
def MostrarSolucionFinal(solucion_candidata):
    pass

# Genera una lista de soluciones extendidas, derivadas de la solución candidata
def ObtenerSolucionesCandidatas(solucion_candidata):
    pass

# Retorna una solución candidata inicial
def SolucionInicial():
    pass

# Backtracking genérico
def Backtracking(solucion_candidata):
    if not SirveComoSolucionParcial(solucion_candidata):
        return False
    if SirveComoSolucionFinal(solucion_candidata):
        MostrarSolucionFinal(solucion_candidata)
        return True
    lista_candidatas = ObtenerSolucionesCandidatas(solucion_candidata)
    for solucion in lista_candidatas:
        if Backtracking(solucion):
            return True

Backtracking(SolucionInicial())
```

Problema del caballo – descripción general de la solución

- **Solución candidata parcial:**
  - **Casilla** se representa por una lista de coordenadas. Ej: [2,1]
  - **Solución parcial** corresponde a una lista de coordenadas de casillas que visita el caballo.
    Ej: [[0, 0], [2, 1], [4, 0], [3, 2], [4, 4]]

- **Solución inicial:**
  - Lista que contiene la **posición inicial**, definida por el usuario.

- **Soluciones candidatas:**
  - Solución parcial se extiende al **agregar una casilla adicional**, que representa un **movimiento válido del caballo** desde la última casilla ocupada.

- **Solución parcial sirve cuando:**
  - Solución parcial sirve cuando la **última casilla no ha sido visitadas** previamente.

- **Solución parcial es final cuando:**
  - La **solución final** contiene todas las casillas posibles dentro del tablero.

Implementación de clase Tablero – general

```
class Tablero:
    dimension = 0
    pos_inicial = []
    # tablero = [[]] # tablero dimension x dimension; se crea sólo al mostrarlo
    recorrido = [[]] # posiciones que ha ocupado el caballo

    def __init__(self, dimension=8, f_inicial=0, c_inicial=0):
        self.dimension = dimension
        self.pos_inicial = [f_inicial,c_inicial]
        self.recorrido = []

    def DentroTablero(self, f, c):
        return f >= 0 and f < self.dimension and c >=0 and c < self.dimension

    def YaVisitado(self, f, c):
        return [f,c] in self.recorrido

    def AgregarSalto(self, f, c):
        self.recorrido.append([f,c])

    def EliminarSalto(self):
        self.recorrido.pop()
```

Implementación de clase Tablero – general

```
def MovimientosPosibles(self, f, c):
    movimientos = []
    movimientos.append([f+2,c-1])
    movimientos.append([f+2,c+1])
    movimientos.append([f+1,c+2])
    movimientos.append([f-1,c+2])
    movimientos.append([f-2,c+1])
    movimientos.append([f-2,c-1])
    movimientos.append([f-1,c-2])
    movimientos.append([f+1,c-2])
    return movimientos

def MovimientosValidos(self, f, c):
    movimientos = self.MovimientosPosibles(f, c)
    validos = []
    for mov in movimientos:
        if self.DentroTablero(mov[0], mov[1]) and \
           not self.YaVisitado(mov[0], mov[1]):
            validos.append(mov)
    return validos
```

Implementación de clase Tablero – *backtracking*

```
# Retorna True si vale la pena explorar la solución candidata
def SirveComoSolucionParcial(self, solucion_candidata):
    f = solucion_candidata[-1][0]
    c = solucion_candidata[-1][1]
    return not self.YaVisitado(f, c)

# Retorna True si la solución candidata resuelve el problema
def SirveComoSolucionFinal(self, solucion_candidata):
    if len(solucion_candidata) < self.dimension * self.dimension:
        return False
    else:
        return self.SirveComoSolucionParcial(solucion_candidata)

# Usa la solución candidata como una solución valida al problema
def MostrarSolucionFinal(self, solucion_candidata):
    # solo creamos el tablero al momento de mostrarlo en pantalla
    tablero = []
    for i in range(0,self.dimension):
        f = []
        for j in range(0,self.dimension):
            f.append('')
        tablero.append(f)

    num = 1
    for mov in solucion_candidata:
        f = mov[0]; c = mov[1]
        tablero[f][c] = str(num).center(4)
        num += 1

    for f in tablero:
        print(f)
```

Implementación de clase Tablero – *backtracking*

```
# Genera una lista de soluciones extendidas, derivadas de solución candidata
def ObtenerSolucionesCandidatas(self, solucion_candidata):
    f = solucion_candidata[-1][0]
    c = solucion_candidata[-1][1]

    listaCandidatas = []
    for mov in self.MovimientosValidos(f, c):
        listaCandidatas.append(solucion_candidata + [mov])
    return listaCandidatas


# Retorna una solución candidata inicial
def SolucionInicial(self):
    return [self.pos_inicial]
```

Implementación de clase Tablero – *backtracking*

```
# Backtracking genérico
def Backtracking(self, solucion_candidata):
    if not self.SirveComoSolucionParcial(solucion_candidata):
        return False

    if self.SirveComoSolucionFinal(solucion_candidata):
        self.MostrarSolucionFinal(solucion_candidata)
        return True

    f = solucion_candidata[-1][0]
    c = solucion_candidata[-1][1]
    self.AgregarSalto(f, c)

    lista_candidatas = self.ObtenerSolucionesCandidatas(solucion_candidata)

    for solucion in lista_candidatas:
        if self.Backtracking(solucion):
            return True

    self.EliminarSalto()
    return False
```

Llamado a objeto de la clase Tablero

```
#Principal

lado = int(input("Indicame el tamaño del tablero: (5 o más): "))

print("Indicame la posición inicial:")
f_inicial = int(input("fila: "))
c_inicial = int(input("columna: "))

t = Tablero(lado, f_inicial, c_inicial)
if t.Backtracking(t.SolucionInicial()):
    print("Felicitaciones; encontramos una solución")
else:
    print("Ups!, no hay solución")
```

Problema del caballo – mejor implementación

- Se puede mejorar la implementación utilizando una técnica heurística conocida como la regla de Warnsdorf (“*Warnsdorf's rule*”).

  - *“Warnsdorf's rule is a heuristic for finding a knight's tour. We move the knight so that we always proceed to the square from which the knight will have the fewest onward moves. When calculating the number of onward moves for each candidate square, we do not count moves that revisit any square already visited. It is, of course, possible to have two or more choices for which the number of onward moves is equal; there are various methods for breaking such ties, including one devised by Pohl [14] and another by Squirrel and Cull.[15]*
    This rule may also more generally be applied to any graph. In graph-theoretic terms, each move is made to the adjacent vertex with the least degree. Although the Hamiltonian path problem is NP-hard in general, on many graphs that occur in practice this heuristic is able to successfully locate a solution in linear time.[14] The knight's tour is a special case.[16]”

    Fuente: https://en.wikipedia.org/wiki/Knight%27s_tour

- Implementación: http://rosettacode.org/wiki/Knight's_tour#Python