Práctica 12: Estrategias para la construcción de algoritmos (segunda parte)

Estructuras de Datos y Algoritmos I

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

El alumno aprenderá a identificar la estrategia de recursión en retroceso (*backtracking*) para la construcción de algoritmos.

Opcionalmente subirá de nivel e incrementará su *ki* por encima de 9000, demostrándose capaz de desarrollar sus propios programas usando *backtracking*.

2. Material

Se asume que el alumno cuenta con un una computadora con arquitectura Intel x86 o compatible con interprete de python instalado (versión 3.5 o posterior).

2.1. Instalación de Python 3 en Ubuntu Linux

Para instalar Python3 en Ubuntu Linux ejecute el siguiente comando en la terminal (se requieren privilegios de superusuario).

```
sudo apt-get update
sudo apt-get -y install python3-all
```

O bien, si se desea instalar sólo los paquetes mínimos necesarios

```
sudo apt-get -y install python3 python3-pip python3-numpy python3-matplotlib
```

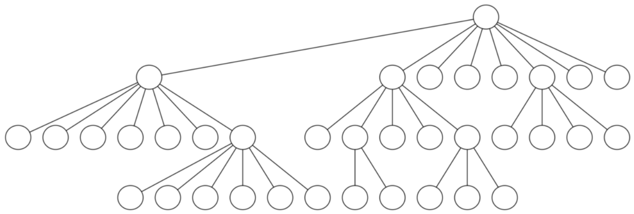
3. Antecedentes

La recursión hacia atrás o *backtracking* es una técnica para resolver problemas de forma recursiva e incremental (i.e. un paso a la vez), pero evitando los caminos que no cumplen con las restricciones que impone la solución del problema. A primera vista, la recursión hacia atrás o *backtracking* podría parecer como una búsqueda exhaustiva o por fuerza bruta con optimizaciones. Esta impresión no está muy lejos de la verdad, pero elimina la importancia fundamental de dichas restricciones, las cuales tienen la facultad de reducir la complejidad del cómputo uno o más órdenes de magnitud.

Al igual que los algoritmos de búsqueda exhaustiva o fuerza bruta, *backtracking* explora el espacio-solución de un problema, pero lo hace modelándolo como un árbol de decisión y recorriendo recursivamente sólo las ramas viables. En cada nuevo paso se examina un nodo o camino que es evaluado con tres posibles resultados:

- Si el nodo es una solución, ésta se devuelve y la exploración se termina.
- Si el nodo no es una solución pero las restricciones se cumplen, la exploración continúa.
- Si las restricciones no se cumplen el nodo se reconoce como inviable y el algoritmo retrocede al último nodo viable (hace *backtrack*) continuando la exploración desde allí.

Así, el algoritmo devolverá la primer solución viable que satizfaga todas las restricciones. Sin embargo, si se alcanza el último nodo del árbol sin que el algoritmo arroje una solución, se tratará de un problema que no puede ser resuelto con recursión hacia atrás, muy probablemente porque la solución no existe.



(a) Árbol de búsqueda del recorrido del caballo

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

(b) Número de posibles movimientos desde cada casilla

Figura 2

En una ejecución normal, un algoritmo de backtracking irá explorando el espacio-solución como si éste fuera un árbol pero, en lugar de explorar cada rama hasta la última hoja, explorará sólo hasta el punto donde se verifica que la rama no ofrece soluciones posibles antes de pasar a la siguiente, acortando así el tiempo de búsqueda.

Recorrido del caballo

Se desea que un caballo recorra las 64 casillas del tablero del ajedréz a partir de una posición dada pero con la condición de que niguna casilla podrá ser visitada más de una vez. Éste es el problema del recorrido del caballo y en general puede resolverse para cualquier tablero de tamaño $n \times n$.

La solución trivial del problema consiste en colocar al caballo en una casilla inicial y probar cada una de los movimientos posibles de forma recursiva hasta encontrar una solución, explorando todo el árbol de opciones.

Esta búsqueda exhaustiva corresponde al algoritmo de bpusqueda en profundidad *Depth First Search* o DFS, cuya complejidad algirítmica en tiempo es de k^n donde k es en número de movimientos válidos y n el tamaño del tablero, tal como lo muestra la Figura 2a.

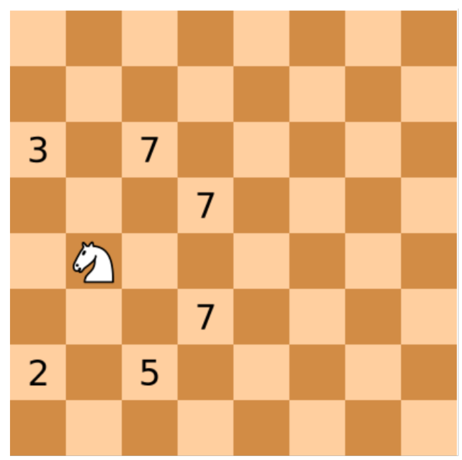


Figura 1: El problema del recorrido del caballo

Sabemos que el número de nodos de un árbol de órden k y altura n es $k^{n+1} - 1$, o $2^{n+1} - 1$ en el caso de un árbol binario. Con esta fórmila es posible estimar el tamaño del árbol que representa el espacio-solución del problema del recorrido del caballo (el número de nodos a visitar) simplemente tomando k igual al número promedio de subnodos.[1] Por ejemplo para un tablero de 5×5 (altura 25) el factor de ramaje k es 3.8, por lo que se tendrá $3.8^{25+1} - 1 \approx 3.12 \times 10^{14}$ posibles caminos. En el caso de un tablero de 6×6, $k = 4.4$ que produce aproximadamente 1.5×10^{23} posibles solucicones, mientras que el tablero tradicional de 8×8 tiene una $k = 5.25$ lo que implica un espacio solución del orden de 1.3×10^{46} elementos.

Con espacios de solución de este tamaño es claro que se debe buscar una forma más rápida de encontrar la solución.[2] Por fortuna el problema puede ser resuelto de forma relativamente más rápida si se modela con recursión hacia atrás. El algoritmo sería similar al siguiente:

- Si se han visitado todas las casillas:
    - devolver la solución.
- Si NO han visitado todas las casillas:

[1] Con base en la Figura 2b se puede calcular k_8 sumando todos los posibles movimientos de cada casilla y dividiendo por 64, es decir:

$$k_8 = \frac{16 \times 8 + 16 \times 6 + 20 \times 4 + 8 \times 3 + 4 \times 2}{64} = 5.25$$

[2] El DFS de un tablero de 8×8 tardaría aproximadamente 10 años si se probasen un millon de soluciones por segundo.

- Generar un vector $\bar{v}$ con todos los posibles movimientos
- Para cada movimiento $m \in \bar{v}$
  - Agregar el movimiento al vector-solución y probar recursivamente
  - Si el movimiento no es solución, eliminarlo del vector y continuar con el siguiente
- Si $\nexists m \in \bar{v}$ que sea solución, devolver falso para regresar al nivel superior de la solución.
- Si se alcanzó el primer nivel de recursión, la solución no existe.

Formalizando el algoritmo anterior y suponiendo que el tablero es inicializado con n^2 valores igual a -1 para indicar que las casillas no han sido visitadas, podríamos tener:

Algoritmo 1 Algoritmo del recorrido del caballo

```
procedure RecursiveKnightsTour(n, x0, y0, step)
    if step > n^2 then
        return True                          ▷ Se llegó al final del árbol. Solución encontrada.
    else
        for (x1, y1) ∈ ViableMoves(n, x0, y0) do
            board[x1][y1] ← step + 1         ▷ Suponemos que el movimiento (x1, y1) es solución.

            if RecursiveKnightsTour(n, x1, y1, step + 1) then
                return True                  ▷ La suposición es correcta y encontramos una solución
            end if
            board[x1][y1] ← −1               ▷ BackTrack: la suposición fue incorrecta.
        end for
        return False                         ▷ La rama explorada no conduce a una solución
    end if
end procedure
```

4. Desarrollo de la práctica

Lea cuidadosamente los problemas descritos a continuación y desarrolle los programas propuestos.

4.1. Actividad 1

El programa del Apéndice A.1 Obtiene el recorrido del caballo para un tablero de $n \times n$. Ejecútelo con la línea:

```
python3 kt.py 5
```

Ejecute el programa del Apéndice A.1 16 veces y complete el tablero mostrado a continuación. En cada casilla anote 1 (uno) si el programa pudo encontrar una solución, o 0 (cero) si dicha solución no fue encontrada. Modifique el programa de ser necesario. [3 puntos]:

4.2. Actividad 2

El programa del Apéndice A.2 mide e imprime el tiempo que tarda en ejecutarse la función `foo()`. Ejecútelo con la línea:

```
python3 time.py
```

Combine los programa de los Apéndices A.1 y A.2 para poder medir el tiempo que tarda el primero en calcular una solución al problema del recorrido del caballo. Con la información obtenida llene la siguiente tabla, anotando el promedio de 10 ejecuciones para tiempos de ejecución menores a 30 segundos y de 100 ejecuciones para tiempos menores a 3 segundos.

n		**Tiempo**	
	$(0,0)$	$(0,\frac{n}{2})$	$(\frac{n}{2},\frac{n}{2})$
5			
6			
7			
8			
9			
10			

¿Existe alguna diferencia significativa en el tiempo de ejecución al variar las condiciones iniciales? ¿Alguno es más rápido? ¿Es posible llenar la tabla? Explique y justifique su respuesta: [3 puntos] ____________________

Con base en la información de la tabla, genere una gráfica de dispersión $n, t(n, x, y)$ donde compare los tiempos de ejecución en función del tamaño del tablero (n). [4 puntos]: ____________________

4.3. Actividad 3 (Opcional)

[5 puntos extra] Modifique el programa del Apéndice A.1 para que en lugar de devolver la primer solución encontrada cuente el número de soluciones existentes en un tablero de $n \times n$ partiendo de una configuración inicial (x, y). Presente su código, explique y utilícelo para completar la siguiente tabla.

n	**Número de soluciones**		
	$(0,0)$	$(0, \frac{n}{2})$	$(\frac{n}{2}, \frac{n}{2})$
4			
5			
6			
7			
8			

4.4. Actividad 4 (Opcional)

[10 puntos extra] Tomando como base el programa del Apéndice A.1 desarrolle un programa que pueda resolver sudokus hexadecimales con números de `0x00` a `0x0f` usando una rejilla de 16×16 con particiones de 4×4. Presente su código y explique.

A. Código de ejemplo

A.1. Archivo **src/kt.py**

src/kt.py

```
import sys

def knight_tour(n, x, y):
  init_board(n)            # Initialize the N×N board
  board[x][y] = 1          # Start in x0, y0
  step = 1                 # This is the first step

  # Solve recursively with BackTracking
  return recursive_knight_tour(n, [x, y], step+1)
# end def

def recursive_knight_tour(n, move, step):
  if step > n**2:
    return True          # Solution found!

  # Iterate over all valid moves
  for x, y in get_moves(n, move[0], move[1]):
    board[x][y] = step   # Assume this is a solution
    # print(f'Trying {step} on ({x}, {y})')
    if recursive_knight_tour(n, [x, y], step+1):
      return True      # Assumption was correct, solution found
    board[x][y] = -1     # Assumption was NOT correct, BACKTRACK!
  return False             # No solution was found in this branch.
# end def

def init_board(n):
  global board
  board = [ [ -1 for i in range(n) ] for i in range(n) ]
# end def

def get_moves(n, x0, y0):
  kmoves = [ # All 8 knight moves
    (-2,  1), (-1,  2), ( 1,  2), ( 2,  1),
    ( 2, -1), ( 1, -2), (-1, -2), (-2, -1)
  ]
  for kmove in kmoves:
    x1 = x0 + kmove[0]
    y1 = y0 + kmove[1]
    # Discard moves out of the board
    if x1 < 0 or x1 >= n or y1 < 0 or y1 >= n:
      continue
    # Skip visited cells
    if board[x1][y1] != -1:
      continue
    # Return this (and only this) move
    yield x1, y1
# end def

def print_board():
  for row in board:
    print( ' '.join([ str(cell).rjust(3) for cell in row]) )
# end def
```

src/kt.py

```python
def main(argv):
  n = 8
  x = 0
  y = 0
  try:
    n = int(argv[1])
    x = int(argv[2])
    y = int(argv[3])
  except:
    pass
  if x < 0 or x >= n:
    x = 0
  if y < 0 or y >= n:
    y = 0

  if knight_tour(n, x, y):
    print_board()
  else:
    print(f'Can\'t find a solution for a {n}x{n} board.')
# end def



if __name__ == "__main__":
  main(sys.argv)
```

A.2. Archivo `src/time.py`

src/time.py

```python
from time import perf_counter, sleep

def foo():
  sleep(1)

def main():
  start = perf_counter()
  foo()
  elapsed = perf_counter() - start
  print("Foo took {:.2f}ms".format(elapsed*1000))


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

B. Reporte Escrito

El reporte de la práctica deberá ser entregada en un archivo en formato `PDF` siguiendo las siguientes especificaciones:

- El nombre del documento `PDF` deberá ser `nn-XXXX-L12.pdf`, donde:
  - `nn` es el número de lista del alumno a dos dígitos forzosos (ej. 01, 02, etc.).
  - `XXXX` corresponden a las dos primeras letras del apellido paterno seguidas de la primera letra del apellido materno y la primera letra del nombre, en mayúsculas y evitando cacofonías; es decir, los cuatro primeros caracteres de su RFC o CURP.
- El reporte consiste en un documento de redacción propia donde el alumno explica de forma concisa y a detalle las actividades realizadas en la práctica, y reportando los resultados obtenidos.
- La longitud del reporte no deberá exceder las 3 cuartillas, sin considerar imágenes, tablas, código y gráficas.
- El reporte deberá seguir todos los lineamientos para documentos escritos establecidos al inicio del curso.
- Todas las referencias deberán estar debidamente citadas.

IMPORTANTE: No se aceptan archivos en otros formatos ni con nombres distintos a los especificados.