Universidad del País Vasco

Facultad de Ciencia y Tecnología

Diseño de Algoritmos
Práctica II

Grupo 11

Ander Cano

Mikel Lamela

Aitor Larrinoa

Febrero-Marzo 2021

Índice

1. Introducción

El presente informe resolverá un único problema mediante dos tipos de algoritmos, con el objeto de realizar una comparación tanto teórica como práctica de ambos. El primero de los algoritmos es un algoritmo de búsqueda en profundidad y el segundo de búsqueda en anchura. El problema a resolver es un juego clásico denominado "Problema del salto del caballo". Se plantea lo siguiente:

Se trata de mover un caballo dentro de un tablero de ajedrez $n \times n$ de forma que partiendo de una posición inicial dada (x_1,x_2) y realizando solamente los movimientos permitidos al caballo en el juego del ajedrez recorra exactamente una vez todas las casillas del tablero. Se pide buscar todas las soluciones posibles. Calcular la complejidad temporal y espacial del algoritmo.

Expondremos los algoritmos generales que resuelven cada problema siguiendo la estructura de cada familia, posteriormente analizaremos la complejidad temporal y espacial de dichos algoritmos propuestos. Por otro lado, mostraremos las implementaciones en Python 3 de los algoritmos y unas ejecuciones representativas, que ayudarán a la compresión de los resultados teóricos.

En siguiente lugar, discutiremos los resultados obtenidos presentando las conclusiones principales del estudio realizado.

Finalmente, en la sección 6, se podrán ver los distintos códigos utilizados en Python así como una breve explicación de cada uno de ellos.

2. Preludio

2.1. Notación

Antes de comenzar con los algortimos aclaremos aspectos generales de notación y representación. Consideremos el tablero de ajedrez n×n como una matriz, esto es, cada casilla estará identificada por una tupla (i,j) donde i representa la fila y j la columna. Se toma el origen (0,0) en la esquina noroeste al igual que en la notación habitual de matrices. Los índices i,j variarán de 0 a n-1, y un camino será un conjunto ordenado de tuplas. El camino estará formado por los movimientos de un caballo de ajedrez.

El salto del caballo tiene forma de L, y se compone de un desplazamiento de dos casillas en dirección horizontal o vertical, y otra casilla más en ángulo recto (ver figura [1]). Cuando el caballo se encuentre en posiciones centrales del tablero, podrá realizar 8 movimientos en total, pero este número de movimientos variará en función de la casilla en la que se encuentre en cada momento.

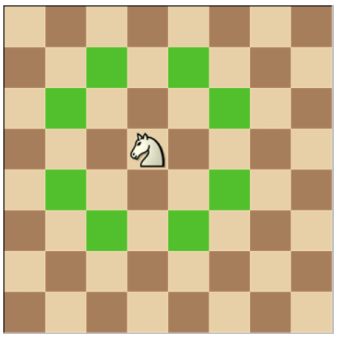

Figura 1: Movimiento de un caballo en ajedrez

2.2. Caso práctico

Para ilustrar el estudio, hemos decidido realizar distintas ejecuaciones en Python. Para ello, se tomará un n que corresponderá al tamaño del problema y se ejecutará el algoritmo implementado (Consultar la implementacion en [6]). Debemos tener en cuenta que para $n = 1$ tenemos una única solución, la trivial, en la que el caballo está en la única casilla del tablero, luego ya ha pasado una única vez por todas las casillas del mismo. Además, para los casos en los que $n = 2, 3, 4$ no existe solución. Por lo tanto, nos centraremos en los casos en los que $n \geq 5$. Como veremos en los apartado de complejidad temporal y espacial, el conjunto solución del algoritmo del caballo crece muy rápidamente con el tamaño del problema. Tanto es así que para $n = 6$ Python tarda varias horas en ejecutar el código (Ver [4]). Por lo tanto, tomaremos como ejemplo experimental el caso $n = 5$ (tablero 5x5) y las casillas iniciales (0,0), (2,2) y (2,4), mostrando los casos principales del problema. La casilla (0,0) expone el caso de encontrarse en una de las 4 esquinas, la situación es simétrica para cualquiera de las restantes, la casilla (2,2) muestra la situación del bloque central con 8 movimientos iniciales por los 2 del (0,0), y por último la casilla (2,4) ejemplifica las casillas exteriores (obviamente excluyendo las 4 esquinas).

3. Algoritmo en profundidad

Los algoritmos en profundidad son un tipo de algoritmo que permiten recorrer todos los nodos de un árbol de manera ordenada, con posibilidad de *Backtraking*. Esto ocurre cuando el algoritmo está recorriendo un camino del árbol y ya no hay posibilidad de seguir bajando más nodos en dicho camino, entonces el algoritmo vuelve a un nodo anterior, denominado este fenómeno como *Backtraking*. Son algoritmos que encuentran todas las soluciones a un problema propuesto y que construyen las soluciones de forma incremental. Además, estos tienen la peculiaridad de que están dados de forma recursiva, es decir, el algoritmo expresa la solución del problema en términos de una llamada a sí mismo.

3.1. Algoritmo

El algoritmo propuesto en esta sección consiste en un algoritmo principal, que se encargará de calcular todas las soluciones, y una función auxiliar que nos ayudará a saber a qué casillas se puede desplazar el caballo conociendo la casilla en la que se encuentra y las ya visitadas. La función principal tiene el nombre de *caballo_profundidad* y la función auxiliar tiene el nombre de *saltos_posibles*. A continuación se muestra el pseudocódigo de la función principal.

Función: ***Caballo_profundidad***(sol_parcial,n)
Entrada: n: Longitud del tablero; *sol_parcial*: conjunto que guarda la solución de un camino hasta el momento.
Salida: *sol*: conjunto de todas las soluciones posibles.

```
sol ← ∅
si longitud(sol_parcial) = n² entonces
    sol ← sol ∪ sol_parcial
en otro caso
    para cada salto en Saltos_posibles(sol_parcial, n) hacer
        sol_parcial ← sol_parcial ∪ salto
        sol ← sol ∪ Caballo_profundidad(sol_parcial, n)
        eliminar salto de sol_parcial
    fin
fin
devolver sol
```

Algoritmo 1: Algoritmo en profundidad del caballo

El objeto de este algoritmo es encontrar todas las posibles soluciones del problema del caballo [1] recorriendo el árbol de exploración en el orden que se establablece en los algoritmo de "Búsqueda en profundidad". La estructura es similar a la clásica de los algoritmos en porfundidad, a excepción que la función que ramifica también verifica si el nodo a vistar es posible. En cada nuevo nodo ramificado se realiza una llamada recusiva. En el backtrancking se van devolviendo las soluciones encontradas por dicho camino, de esta forma se acaban obteniendo todas. La implementación del algoritmo en Python 3, se puede ver en [6].

En este momento, abordaremos el problema de la funcíon auxiliar de ramificación. El pseudocódigo es el que se muestra a continuación:

Función: ***Saltos_posibles***(sol_parcial, n)
Entrada: n: Longitud del tablero; *sol_parcial*: conjunto que guarda la solución de un camino hasta el momento.
Salida: *Lista*: Conjunto de todas los posibles saltos que puede realizar el caballo, o bien $\emptyset$.

```
(x,y) ← última posición de sol_parcial
para cada (i,j) en {(2,1),(1,2),(-1,2),(-2,1),(-2,-1),(-1,-2),(1,-2),(2,-1)} hacer
    si 0 ≤ x+i < n & 0 ≤ y+j < n & (x + i,y + j) no en sol_parcial entonces
        Lista ←Lista ∪ (x + i, y + j)
    fin
fin
devolver Lista
```

Algoritmo 2: Algoritmo Saltos_posibles

La idea es clara, se comprueba si los 8 movimientos posibles del caballo se encuentran en los límites del tablero n×n y si la casilla ha sido visitada con anterioridad. Se puede consultar la implentación en [6].

Por ejemplo, tomando un tablero 5×5 y casilla de inicio (0,0), una solución obtenida es:

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

Figura 2: Ejemplo 1

3.2. Complejidad temporal y espacial

En lo referente a la complejidad temporal, como el número de llamadas recursivas dependerá de como se expanda cada nodo, tendremos un mejor y peor caso.

Por un lado, el mejor caso es el caso trivial, en el cúal en la casilla de inicio no se pueden realizar movimientos. Este caso sucede por ejemplo, en un tablero 3×3 y la casilla central.

Por otro lado, el peor caso se realizarían todas las ramificaciones posibles, siendo en las 4 esquinas los 2^4 movimientos. Analizando las casillas laterales, se deben diferenciar dos casos. Estos casos son, por un lado, cuando la casilla está pegada a una de las esquinas, o la casilla no está pegada a una de las esquinas. En el primero de los casos, se tienen 12 casillas (contando las que están en diagonal), en las cuales solo se pueden realizar 3 movimientos, esto es, 12^3 movimientos. En el segundo de los casos, se tienen $4 \cdot (n-4)$ casillas laterales exteriores en los que se pueden realizar 4 movimientos, esto es, $4^{4(n-4)}$ movimentos totales. En las $4 \cdot (n-4)$

casillas subexteriores se realizan $6^{4(n-4)}$ y finalmente en las $(n-4)^2$ casillas interiores tenemos $8^{(n-4)^2-1}$ saltos. Ahora como salvo en el primer movimiento en el resto habrá un salto que no será posible realizar, dado que es por el cual se ha llegado a dicha casilla actual, podemos quitar un movimiento en cada casilla. Luego, $t(n) \in O((3k)^{4(n-4)} \cdot (5k)^{4(n-4)} \cdot (7k)^{(n-4)^2-1})$.

La complejidad espacial depende del tamaño del problema, es decir, del tamaño del tablero. Las soluciones serán de longitud n^2, como veremos en el caso práctico, no siempre se tiene el mismo número de soluciones. Por lo tanto, la complejidad espacial $s(n) \in O(n^2)$.

Ejecutando el caso práctico para este algoritmo obtenemos estos resultados:

n=5 Profundidad		
Casilla de inicio	Número de soluciones	Tiempo de ejecución (segundos)
(0,0)	304	3.65625
(2,2)	64	1.28125
(2,4)	56	3.046875

Figura 3: Caso práctico Algoritmo en profundidad

Un claro ejemplo de este crecimiento en términos de tiempo y espacio es el caso de un tablero 6×6 y casilla de inicio (0,0). Veamos la ejecuación:

```
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\ander\OneDrive\Documentos\Universidad\4° Matemáticas\Diseño de algoritmos\Práctica 2\Problema del caballo P II.py
Práctica II - Grupo 11 - Ander Cano, Mikel Lamela y Aitor Larrinoa


Introducir valor de "n" : 6

NOTA: Se considera la casilla (0,0) del tablero aquella de la esquina noroeste
1ª coord casilla inicio: 0
2ª coord casilla inicio: 0
 1- Alg. Profundidad
 2- Alg. Anchura
 Introducir n°: 1
Nombre archivo resultados: profundidad_n=6
N° soluciones:  524486
Tiempo aproximado de ejecución:  52226.90625
Squeezed text (1914374 lines).
Para las soluciones ver profundidad_n=6.txt
>>>
```

Figura 4: Tablero 6×6 - Casilla de inicio (0,0)

Como podemos observar, el tiempo de ejecución sobrepasa las 14h y se encuentran 524486 soluciones, esto es, 524486 listas de tamaño 36. Un aumento más que considerable respecto a los casos de un tablero 5×5.

4. Algoritmo en anchura

Los algoritmos de búsqueda en anchura a diferencia de los algoritmos de búsqueda en profundidad, recorren el árbol por niveles, es decir se visitan los descendientes de cada nodo y a continuación, los descendientes de la siguiente generación, y así sucesivamente. Anteriormente hemos visto que los algoritmos de búsqueda en profundidad eran algoritmos recursivos, los algoritmos de búsqueda en anchura, en cambio, no son algoritmos naturalmente recursivos.

4.1. Algoritmo

El cambio fundamental respecto a los algoritmos de búsqueda en porfundidad, es el orden de visita de los diferentes nodos del árbol de representación del problema. En este caso se produce un avance por niveles, en los cuales se va desarrollando las soluciones, y previo paso a avanzar al siguiente nivel se realiza una poda. Este concepto quiere decir que se descartan soluciones parciales. En nuestro caso se podará cuando el nodo a visitar se haya visitado previamente. Al llegar al último nivel, se han hallado todas las soluciones del problema. Cabe destacar también que este algoritmo es no recursivo, otra diferncia significativo respecto a los algortimos en profundidad. El pseudocódigo del algoritmo propuesto es el siguiente:

```
Solucion ← ∅
V ← casilla
mientras V ≠ ∅ hacer
    x ← 1º elemento de V
    si longitud(x) = n^2 entonces
        Solucion ← Solucion ∪ x
    en otro caso
        para rama in saltos_posibles (x, n) hacer
            V ← V ∪ (x ∪ rama) nose si se ve claro la idea de expander la x
            eliminar x de V
        fin
    fin
fin
devolver Solucion
```

Algoritmo 3: Algoritmo búsqueda en anchura del caballo

4.2. Complejidad temporal y espacial

La complejidad temporal dependerá de la poda que se haga en cada nivel, por lo tanto, tendremos un mejor y peor caso. El mejor caso es obvio que será cuando de la casilla inicial no se pueda realizar ningún movimiento, como sucede para un tablero 3×3 en la casilla central. El peor se dará cuando se ramifique totalmente cada nodo, este caso concuerda con el peor caso del algoritmo en profundidad, así pues, $t(n) \in O((3k)^{4(n-4)} \cdot (5k)^{4(n-4)} \cdot (7k)^{(n-4)^2-1})$.

La complejidad espacial vendrá dada por la expansión de los nodos en cada nivel, dado que es casi seguro que se tendrán más soluciones de tamaño b, con $b \in 1,...,n^2$, que soluciones finales. Este fenómeno se puede ver en [4.2], donde se desarrollará más esta idea. Por lo tanto, la complejidad espacial del algoritmo de anchura es del orden $O(b \cdot 7^{b-1})$. Lo habitual será una b cercana a n^2, con una gran cantidad de nodos en ese b-ésimo nivel de exploración, por lo que la cantidad de datos a almacenar será enorme.

Profundicemos más, se requiere una gran cantidad de memoria, pudiendo darse el caso de un colapso del ordenador. Antes de este colpaso, el programa utilizaría la memoria de disco de más costoso acceso por parte del sistema ralentizando la ejecución. A continuación mostraremos unas gráficas que ilustran ese aumento del número de nodos a cada nivel.

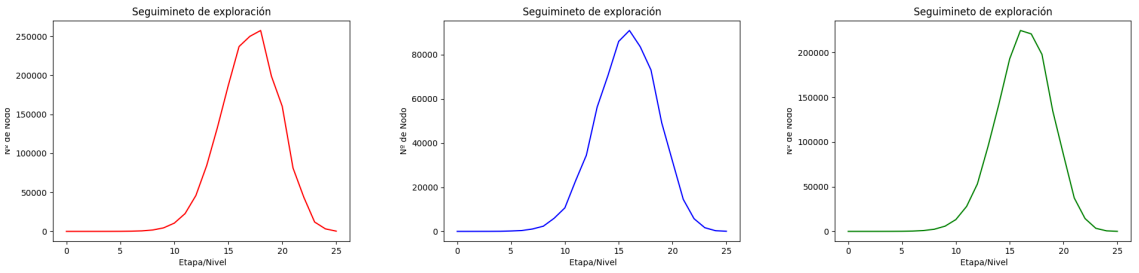

Figura 5: Casillas de inicio (0,0), (2,2) y (2,4) para n=5

Como vemos por ejemplo, empezando en la casilla (0,0) y en un tablero 5×5, se llega a manejar un lista de 25000 elemntos, que a su vez son listas de longitud 18. Si ejecutaramos para un tablero 6×6 es posible que llegaramos a no tener suficiente memoria.

Una vez estudiado el marco teórico, veamos que resultados se han obtenido en el caso práctico:

n=5 Anchura		
Casilla de inicio	Número de soluciones	Tiempo de ejecución
(0,0)	304	61.6875
(2,2)	64	8.828125
(2,4)	56	45.78125

Figura 6: Caso práctico algoritmo en anchura

Como era de esperar se han obtenido el mismo número de soluciones que para el algoritmo en profundidad, pero el tiempos de ejecución es abrumadoramente mayor.

5. Comparación de algoritmos y Conclusiones

La primera comparción es en términos temporales, en este caso, claramente es más rápido el algoritmo en profundidad. Por ejemplificar, tomando la casilla de inicio la (0,0), vemos como para profundidad obtenemos un tiempo de unos 4 segundos, mientras que para anchura se excede el minuto de ejecución. No obstante, cabe destacar que para este problema la mayor diferencia se encuentra en el aspecto del almacenamiento en memoria, donde el algoritmo en anchura necesita de una cantidad considerable de memoria pudiendo darse el caso de no tener suficiente el PC en el que se esté trabajando. Este último aspecto ya ha sido discutido en [4.2].

Brevemente mencionemos diferencias en el aspecto de implementación en Python. Pues como se verá en el Anexo I, son implemntaciones sencillas sin gran cantidad de sentencias y variables auxiliares, luego no se puede hablar de grandes diferencias a la hora de implementar.

En resumen, para este problema del salto del caballo ofrece mejores prestaciones el algoritmo en profundidad tanto en términos temporales como de almacenamiento.

6. Anexo I

La implementación del Algoritmo 2 es relativamente sencilla, la única sentencia a explicar en este caso es, " not (a,b) in S ", línea 7. Se ha optado esta forma dado que la versión más sencilla " (a,b) not in S " es más costosa dado que en cualquier caso hay que recorrer toda la lista. En cambio, con la sentencia utilizada, una vez que encuentre la casilla se detiene la búsqueda, devolviendo un " True " que con el " not " se convierte a " False ", evitando así que ese salto se añada a la lista de saltos posibles. El código implementado en lenguaje Python se muestra a continuación:

```
1  def saltos_posibles(S,n): # FUNCION AUXILIAR
2      (x,y) = S[-1]
3      lista = list()
4      for (i,j) in [(2,1),(1,2),(-1,2),(-2,1),(-2,-1),(-1,-2),(1,-2),(2,-1)]:
5          a = x + i
6          b = y + j
7          if 0 ≤ a < n and 0 ≤ b < n and not (a,b) in S :
8              lista.append((a,b))
9      return lista
```

Expongamos brevemente las ideas de implementación del Algoritmo 1, se ha utilizado una implementación con listas, donde la variable " sol " alamacenará las soluciones y la variable " sol_parcial " guardará el camino recorrido hasta ese momento. El procemiento de Backtracking se realiza al añadir el nodo con un "append" y despues eliminarlo con un "remove". El código es el siguiente:

```
1  def caballo_profundidad(sol_parcial,n): # ALG. EN PROFUNDIDAD
2      sol = list()
3      if len(sol_parcial) == n*n:
4          sol.append([sol_parcial[:]])
5      else:
6          for salto in saltos_posibles(sol_parcial,n):
7              sol_parcial.append(salto)
8              sol = sol + caballo_profundidad(sol_parcial,n)
9              del sol_parcial[-1]
10     return sol
```

Centremonos en la implentación del Algoritmo 3. Se ha optado como en los casos anteriores, por una implementación por listas. Se ha utilizado una única lista para almacenar tamto las soluciones parciales como las soluciones finales. La variable "activo" indica el índice de posición en cual se evalua si es solución o si se ramifica dicho nodo. Dicha variable solo aumenta al encontrarse una solución, esto es en las n^2-1 primeras etapas permanecerá a 0.

```
1  def caballo_anchura(casilla,n): # ALG. EN ANCHURA
2      activo = 0
3      V = [[casilla]]
4      n2 = n*n
5      while activo < len(V):
6          if len(V[activo]) == n2:
7              activo += 1
8          else:
9              for rama in saltos_posibles(V[activo],n):
10                 V.append(V[activo] + [rama])
```

```
            del V[activo]
    return V
```

El programa principal que permite ejecutar el algoritmo estableciendo el usuario el tamaño del tablero, casilla de inicio y algoritmo a utilizar, ha sido escrito en el mismo file, que las funciones.

```
# Programa Principal

import time

print('Pr ctica II - Grupo 11 - Ander Cano, Mikel Lamela y Aitor Larrinoa ...
    \n\n')
n = int(input('Introducir valor de "n" : '))
print('\nNOTA: Se considera la casilla (0,0) del tablero aquella de la ...
    esquina noroeste')
x = int(input('1   coord casilla inicio: '))
y = int(input('2   coord casilla inicio: '))
alg = input(' 1- Alg. Profundidad\n 2- Alg. Anchura\n Introducir  n  : ')
file = input('Nombre archivo resultados: ')
tiempo1 = time.process_time()
if alg == '1':
    SOLUCIONES = caballo_profundidad([(x,y)],n)
else:
    SOLUCIONES = caballo_anchura((x,y),n)
tiempo2 = time.process_time() - tiempo1

print(' N  soluciones: ',len(SOLUCIONES))
print('Tiempo aproximado de ejecuci n: ',tiempo2)
print(SOLUCIONES)
print('Para las soluciones ver ' + file + '.txt')

fp = open(file + '.txt','w')
fp.write('\nSOLUCIONES PROBLEMA CABALLO ALG. PROFUNDIDAD \n\n')
fp.write('Casilla de inicio: '+str((x,y))+'\n\n')
fp.write('Tama o del tablero: '+ str(n))
fp.write('\n\nTiempo aproximado de ejecuci n: '+ str(tiempo2)+'s \n')
fp.write('\ nN  soluciones: '+ str(len(SOLUCIONES))+'\n')
fp.write('\nSoluciones: \n\n')
for c in SOLUCIONES:
    fp.write(str(c) + '\n')
fp.close()
```

Por último, se muestra la variación del algortimo de anchura para realizar un seguimiento de los nodos en cada nivel de exploración.

```
from matplotlib.pyplot import *

def saltos_posibles(S,n): # FUNCI N AUXILIAR
    (x,y) = S[-1]
    lista = list()

    for (i,j) in [(2,1),(1,2),(-1,2),(-2,1),(-2,-1),(-1,-2),(1,-2),(2,-1)]:
        a = x + i
        b = y + j
        if 0 ≤ a < n and 0 ≤ b < n and not (a,b) in S :
            lista.append((a,b))
```

```
12
13      return lista
14
15  def caballo_anchura(casilla,n): # ALG. EN ANCHURA
16      activo = 0
17      V = [[casilla]]
18      n2 = n*n
19      Nlistas = [0,1]
20      m = 0
21      tama o = len(V[activo])
22      while activo < len(V):
23          if len(V[activo]) != tama o:
24              tama o = len(V[activo])
25              Nlistas.append(m)
26              m = 0
27
28          if len(V[activo]) == n2:
29              activo += 1
30          else:
31              for rama in saltos_posibles(V[activo],n):
32                  V.append(V[activo] + [rama])
33                  m = m + 1
34              del V[activo]
35      return Nlistas
36
37  print('GR FICA ALG. EN ANCHURA\n\n')
38  n = int(input('Introducir valor de "n" : '))
39  x = int(input('1   coord casilla inicio: '))
40  y = int(input('2   coord casilla inicio: '))
41  file = input('Nombre de la gr fica: ')
42
43  levelmax = (n*n) + 1
44  plot(list(range(levelmax)),caballo_anchura((x,y),n),'green')
45  ylabel(' N   de Nodo ')
46  xlabel(' Etapa/Nivel')
47  title(' Seguimineto de exploraci n ')
48  savefig(file + '.png')
```