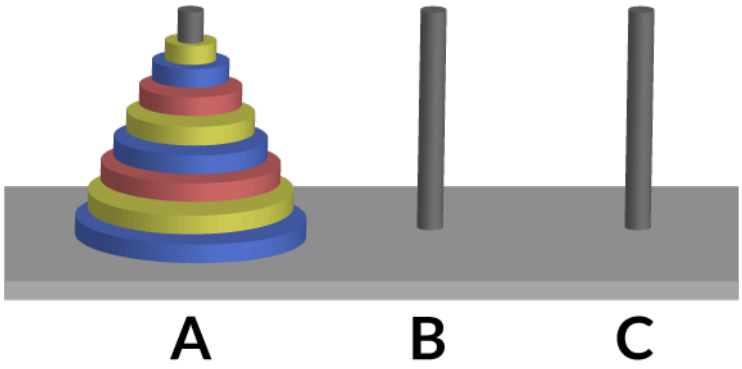
Técnicas de Búsqueda Ciega
Sistemas Inteligentes

February 19, 2018

1 Torres de Hanoi



1.1 Reglas

1. Solo puedes mover el disco superior de cada columna.
2. Solo puedes mover el disco superior de cada columna.
3. Ningún disco puede estar encima de un disco más pequeño.

1.2 Objetivo

Mover la torre entera a otra columna.

1.3 Resolución

1. Se efectua el primer movimiento según el numero de anillos.
   - Si n es impar, el primer movimiento será de A a C.
   - Si n es par, el primer movimiento será de A a B.
2. Ningún anillo par se deberá poner directamente encima de otro anillo par.
3. Ningún anillo impar se deberá poner directamente encima de otro anillo impar.
4. Si hay más de un movimiento posible, (siendo las opciones una columna vacía y otra no está vacia) pon el disco en la columna que no esta vacía.
5. Nunca muevas un disco el cual se ha movido en la rotación anterior.
6. Repetir desde el paso 2 hasta completar.

2 2n y 2n + 1

2.1 Enunciado

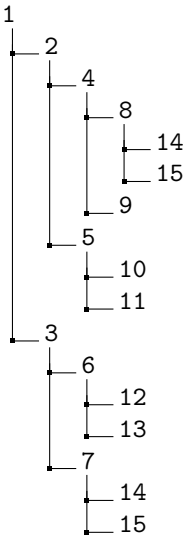
Considere un espacio de estados donde el estado comienzo es el número 1 y la fun- ción sucesor para el estado n devuelve 2 estados, los números 2n y 2n + 1.

1. Dibuje el trozo del espacio de estados para los estados del 1 al 15.
2. Supongamos que el estado objetivo es el 11. Enumere el orden en el que serán visitados los nodos por la búsqueda primero en anchura, búsqueda primero en profundidad con límite tres, y la búsqueda de profundidad iterativa.
3. ¿Será apropiada la búsqueda bidireccional para este problema? Si es así, describa con detalle cómo trabajaría.
4. ¿Qué es el factor de ramificación en cada dirección de la búsqueda bidireccional?
5. ¿La respuesta (3) sugiere una nueva formulación del problema que permitiría resolver el problema de salir del estado 1 para conseguir un estado objetivo dado, con casi ninguna búsqueda?

2.2 Resolución

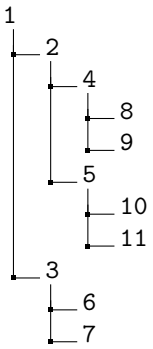
2.2.1 Dibuje el trozo del espacio de estados para los estados del 1 al 15.

El arbol resultante sería el siguiente.



2.2.2 Supongamos que el estado objetivo es el 11. Enumere el orden en el que serán visitados los nodos por la búsqueda primero en anchura, búsqueda primero en profundidad con límite tres, y la búsqueda de profundidad iterativa.

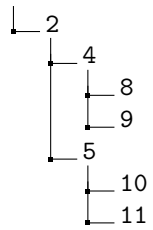
Por primero anchura, los nodos generados por el algoritmo de busqueda serian



El orden en el que accede sería:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10 y 11

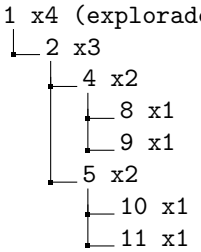
Y por primero profundidad con limite 3, los siguentes nodos serían explorados:



El orden en el que accede seria:

1, 2, 4, 8, 9, 5, 10 y 11

Finalmente por profundidad iterativa. En la cual se generarían los siguientes nodos:



El orden en el que accede seria el siguiente:

1, 1, 2, 1, 2, 4, 5, 1, 2, 4, 5, 8, 9, 10 y 11

2.2.3 ¿Será apropiada la búsqueda bidireccional para este problema? Si es así, descríba con detalle cómo trabajaría.

Si partimos la busqueda en dos, tendremos dos busquedas $b^{d/2}$, lo cual resultaría en $b^{d/2} + b^{d/2}$ que es consideráblemente menor que b^d

$$b^{d/2} + b^{d/2} \ll b^d \tag{1}$$

El problema es que esto no nos reduce el numero de nodos que tenemos que explorar, solo los reparte en dos busqedas, si fuera ordenación no sería así, pero al ser busqueda no mejora el orden

2.2.4 ¿Qué es el factor de ramificación en cada dirección de la búsqueda bidireccional?

El factor de ramificación se refiere a cuantas busquedas se realizan por separado en el mismo arbol o red de nodos.

2.2.5 ¿La respuesta (3) sugiere una nueva formulación del problema que permitiría resolver el problema de salir del estado 1 para conseguir un estado objetivo dado, con casi ninguna búsqueda?

No, por las mismas razones que se exponene en dicha respuesta

3 Arbol

3.1 Enunciado

Consideremos un árbol finito de profundidad d con un factor de ramificación b (un solo nodo raíz, con profundidad 0) y b sucesores por cada nodo, etc... Supongamos que el nodo objetivo menos profundo se encuentra a una profundidad g $\leq$ d.

1. ¿Cuál es el número mínimo de nodos generados por la búsqueda primero en profundidad con una profundidad límite d?. ¿Y el máximo?.

2. ¿Cuál es el número mínimo de nodos generados por la búsqueda primero en anchura?. ¿Y el máximo?.

3. ¿Cuál es el número mínimo de nodos generados por el descenso iterativo?. ¿Y el máximo?. (Considérese que comenzamos con una profundidad límite inicial de 1 y la vamos incrementando una unidad cada iteración).

3.2 Resolución

1. El numero de nodos mínimo generados por la función primero por profundidad será:

   $$(d) \tag{2}$$

   ya que, si el nodo que estamos buscando esta contenido en la primera rama que buscamos, solo hará falta llegar hasta él de una forma lineal.

Por otra parte, tenemos que el máximo será el numero totales de nodos menos la profundidad por debajo de d.

$$[n^b - (n - d)] \tag{3}$$

Debido a que en el peor de los casos, pasaremos por todas las ramas erróneas y en la ultima, encontraremos el nodo deseado en el nivel de profundidad d.

2. El numero mínimo de nodos generados utilizando primero en anchura será:

$$[(d - 1^b) + 1] \tag{4}$$

ya que habrá que explorar todos los nodos de los niveles anteriores más el nodo buscado.
En el caso contrario, el máximo sería que este sea el último nodo del nivel d, quedando el número de nodos generados por la búsqueda seria:

$$(d^b) \tag{5}$$

3. El número mínino de nodos generados por descenso iterativo será:

$$(\sum_{i=0}^{(d-1)} b^i) + 1 \tag{6}$$

siendo la totalidad de los niveles menores que d, (d-1) más el nodo buscado del nivel d.
El máximo sería exactamente igual, pero sumándole todos los del nivel de profundidad d, quedando la siguiente expresión:

$$\sum_{i=0}^{d} b^i \tag{7}$$

siendo el nodo buscado el último en ser accedido en el nivel d.

4 Sol y Sombra

1. Solo puedes mover un elemento a la vez.
2. Los elementos pueden saltar, pero solo a otro elemento contiguo en su dirección de movimiento.
3. Todos lo elementos solo se pueden mover hacia la dirección contraria del tablero, es decir, los de la derecha solo se pueden mover hacia la izquierda.

5 Objetivo

- Cambiar de posición ambos elementos (soles con sombras).

6 Resolución

El conjunto de pasos que da solución son dos, depende del primer movimiento. Exploramos una de las ramas.

iteración	p1	p2	p3	p4	p5	movimiento
iteración 0	o	o		x	x	posición inicial
iteración 1	o	o	x		x	se mueve x
iteración 2	o		x	o	x	se mueve o
iteración 3		o	x	o	x	se mueve o
iteración 4	x	o		o	x	se mueve x
iteración 5	x	o	x	o		se mueve x
iteración 6	x	o	x		o	se mueve o
iteración 7	x		x	o	o	se mueve o
iteración 8	x	x		o	o	se mueve x, posición final

6.1 Algoritmo

6.1.1 Directivas

1. El primer movimiento es libre.
2. Se mueven dos fichas (una antes y la otra después, en dos turnos) del símbolo contrario del símbolo de la ficha movida en el turno inicial, solo hay un movimiento legal por cada turno.
3. Repetir paso dos hasta completar.

Este algoritmo es valido n=[1,2] donde n ∈ "numero de fichas de cada signo".

6.1.2 Código del algoritmo

Código que resuelve este problema (hecho en python):

```
import sys

class Tabletop:
        def __init__(self,_n_elem):
                self.table=[]
                self.iteration=0
                self.n_elem=int(_n_elem)
                #asignation of default values
                for i in range(0,(self.n_elem*2)+1):
                        if i<self.n_elem:
                                self.table.append(SHADOW)
                        elif i==self.n_elem:
                                self.table.append(EMPTY)
                        else:
                                self.table.append(SUN)

        def print_table(self):
                for i in range(0,(self.n_elem*2)+1):
                        sys.stdout.write("____")
```

```
            sys.stdout.write("\n")
            sys.stdout.write("| ")
            for i in range(0,(self.n_elem*2)+1):
                    sys.stdout.write(self.table[i])
                    sys.stdout.write(" | ")
            print ("iteration no: %d"%self.iteration)
            for i in range(0,(self.n_elem*2)+1):
                    sys.stdout.write("---")
            sys.stdout.write("\n")
            return

    def move_sun(self):
            print("searching...")
            for i in range(1,(self.n_elem*2)+1):
                    if self.table[i]==SUN
                    and (self.table[i-1]==EMPTY
                    or (self.table[i-2]==EMPTY
                    and self.table[i-1]==SHADOW)):
                            print("bingo!!")
                            x=i
                            if self.table[i-2]==EMPTY:
                                    y=i-2
                            else:
                                    y=i-1
                            #swap!
                            aux=self.table[x]
                            self.table[x]=self.table[y]
                            self.table[y]=aux
                            #end swap :(
                            self.iteration+=1
                            return True
            print("nope...")
            return False

    def move_shadow(self):
            print("searching...")
            for i in range(0,(self.n_elem*2)):
                    if self.table[i]==SHADOW
                    and (self.table[i+1]==EMPTY
                    or (self.table[i+2]==EMPTY
                    and self.table[i+1]==SUN)):
                            print("bingo!!")
                            x=i
                            if self.table[i+2]==EMPTY:
                                    y=i+2
                            else:
                                    y=i+1
                            #swap!
                            aux=self.table[x]
                            self.table[x]=self.table[y]
```

```
                        self.table[y]=aux
                        #end swap :(
                        self.iteration+=1
                        return True
            print("nope...")
            return False

    def solve(self):
            if self.n_elem==1:
                    self.move_sun()
                    self.print_table()
                    self.move_shadow()
                    self.print_table()
                    self.move_sun()
                    self.print_table()
                    print("\tfinished!")
                    return
            else:
                    self.print_table()
                    solved=False
                    if not self.move_shadow():
                            return
                    else:
                            self.print_table()
                            while not solved:
                                    try:
                                            self.move_sun()
                                    except:
                                            print("\tfinished!")
                                            return
                                    self.print_table()

                                    try:
                                            self.move_sun()
                                    except:
                                            print("\tfinished!")
                                            return
                                    self.print_table()
                                    try:
                                            self.move_shadow()
                                    except:
                                            print("\tfinished!")
                                            return
                                    self.print_table()

                                    try:
                                            self.move_shadow()
                                    except:
                                            print("\tfinished!")
                                            return
```

```
                                        self.print_table()

                                    return

def main():
        #checks the argument
        if len(sys.argv)<2:
                print("the program needs a comand line")
                print("argument for the number of elemnts")
                print("sol_y_luna.py <n_elem [1/2]>")
                return

        n_elem=str(sys.argv[1])

        if int(n_elem)<1 or int(n_elem)>2:
                print("\tinvalid number of elements!")
                print("enter 1 or 2 elements\n")
                return

        greeting="\tThis program solves the sun "
        greeting+="and shadow game.\n"
        greeting+="\tThis is the initial tabletop"
        print(greeting)

        tabletop=Tabletop(n_elem)
        tabletop.solve()
        return

if __name__ == "__main__":
        EMPTY= " "
        SHADOW = "x"
        SUN = "o"
        main()
```