Backtracking

Andrés Becerra Sandoval

29 de agosto de 2007

Resumen

Esta es una técnica fácil de implementar que permite diseñar algoritmos para resolver problemas de búsqueda y optimización.

1. Motivación

Hasta este punto del curso hemos visto técnicas de análisis de algoritmos que nos permiten hacer el análisis en el caso mejor, peor y el promedio. Para esto aplicamos conteo de instrucciones, planteamiento y solución de recurrencias, y análisis probabilístico del valor esperado de la complejidad por medio de variables aleatorias indicadoras.

Ahora empezaremos a ver *técnicas de diseño* de algoritmos. Estas son maneras de conceptualizar y resolver muchos problemas de acuerdo a su estructura. Vamos a empezar con la técnica de backtracking (vuelta atrás), que, sorprendentemente resulta fácil de implementar. Para ilustrar ésta técnica vamos a resolver el problema de situar N Reinas en un tablero de ajedrez de N filas y N columnas, sin que se ataquen entre si.

2. N-Reinas

Las reinas en el ajedrez se pueden atacar horizontalmente, verticalmente y en diagonal. El siguiente tablero ilustra una solución al problema de N-Reinas cuando n=8:

La pregunta que planteamos es ¿Como solucionar este problema por medio de un algoritmo?. Nada de lo que hemos visto hasta ahora (técnicas de análisis de algoritmos) nos sirve para solucionar este tipo de problemas.

Lo que necesitamos es técnicas de diseño, que permitan construir algoritmos Para situar las N reinas en el tablero, parece que necesitamos buscar en un conjunto grande de alternativas a solución, cuales respetan la condición del problema de que no se ataquen las reinas.

¿Cuantas alternativas tenemos que explorar?. Por conteo, tenemos 64 casillas en un tablero de ajedrez de 8×8, y 8 reinas que ubicar en posiciones distintas, asi que hay $\binom{64}{8}$ alternativas de solución (tableros con las 8 reinas situadas en posiciones distintas), esto es, 4.426.165.368. Al número de alternativas que tenemos que considerar se le denomina *tamaño del espacio de búsqueda,* y siempre que tengamos un problema de búsqueda como éste de las N-Reinas es conveniente estimarlo mediante técnicas de conteo.

2.1. Cambio de codificación

Una conveniente codificación permite reducir el tamaño del espacio de búsquda. Una mejora sencilla que viene a la mente de un programador consiste en codificar un tablero de ajedrez en un arreglo A, en el que A[k] contiene la columna en que se debe situar una reina en la fila k. Esto se puede visualizar con el siguiente ejemplo de arreglo en el que fila es el índice o posición del arreglo:

columna	4	2	5	8	6	1	3	7
fila	1	2	3	4	5	6	7	8

2.2. Búsqueda exhaustiva (ingenua)

Con este cambio de representación tenemos que buscar todas las posibles asignaciones de columnas para las 8 filas. Por conteo, esto es $8^8 = 16{,}777{,}216$, que es mucho mejor que 4.426.165.368. Además, esta codificación nos sugiere un algoritmo que busca exhaustivamente la asignación de columnas para cada fila:

```
REINAS()
    for a ← 1 to 8
        do for b ← 1 to 8
                do for c ← 1 to 8
                    ...
                        do for g ← 1 to 8
                                do for h ← 1 to 8
                                    A ← (a,b,c,...,g,h)
                                    if SOLUCION(A)
                                      then print A
```

Aquí a contendrá la columna en la que hay que ubicar la primera reina, c contendrá la columna en la que hay que ubcar la segunda reina, y asi sucesivamente

hasta h. Todo lo que resta es diseñar un algoritmo que chequee si un arreglo que contenga los valores (a,b,c,d,e,f,g,h) representa una solución válida (las reinas no se atacan entre si). Si generalizamos para un tablero de cualquier tamaño, obtenemos el siguiente algoritmo, cuya complejidad es $\Theta(n^n)$:

```
REINAS(n)
    for a ← 1 to n
        do for b ← 1 to n
               do for c ← 1 to n
                  ⋱
                      do for g ← 1 to n
                             do for h ← 1 to n
                                A ← (a, b, c, ..., g, h)
                                if SOLUCION(A)
                                  then print A
```

Pero nos falta averiguar si un arreglo contiene la solución a un problema de N-Reinas. Pero esto es un problema algorítmico mas sencillo que el original de buscar las soluciones. Para obtener el algoritmo basta con codificar matemáticamente la condición de validez: las reinas en las posiciones $(1,A[1]),(2,A[2]),\ldots,(8,A[8])$ no se atacan entre sí, lo cual puede especificarse así:

$$\begin{aligned} A[i] &\neq A[j] \quad i \neq j \quad i,j \in [1,8] \\ |A[i]-A[j]| &\neq |i-j| \quad i \neq j \quad i,j \in [1,8] \end{aligned}$$

La primera condición dice que no puede haber un par de reinas en la misma columna, y la segunda dice que no puede haber un par de reinas en la misma diagonal —como todas las diagonales son de 45^0, si la distancia horizontal $|A[i]-A[j]|$ es igual a la vertical $|i-j|$, entonces la reina que está en la fila i, columna A[i] está en la misma diagonal que la reina que está en la fila j, columna A[j]—. No hay necesidad de chequear que las reinas estén en filas diferentes porque la codificación de las filas como posiciones del arreglo se encarga de eso implícitamente. Las relaciones anteriores nos sugieren el siguiente algoritmo:

```
SOLUCION(A)
    for i ← 1 to 7
        do for j ← i + 1 to 8
                do if A[j] = A[i] or |A[j] − A[i]| = |i − j|
                    return FALSE
    return TRUE
```

Que puede generalizarze de la siguiente forma:

```
SOLUCION(A)
    for i ← 1 to n − 1
        do for j ← i + 1 to n
                do if A[j] = A[i] or |A[j] − A[i]| = |i − j|
                    return FALSE
    return TRUE
```

Con una complejidad de $\Theta(n^2)$ como usted puede verificar.

2.3. Solución por permutaciones

Otra forma de resolver el problema de las N-Reinas consiste en aprovechar el concepto de permutación. Sabemos que cada solución al problema es un arreglo de números de 1 a 8, donde no pueden haber repeticiones, y esto es exactamente una **permutación** del arreglo [1,2,3,4,5,6,7,8]. Asi que nuestro espacio de búsqueda podría ser el conjunto de todas las permutaciones del arreglo [1,2,3,4,5,6,7,8], cuyo tamaño viene dado por $n!$. Para $n = 8$, tenemos $8! = 40{,}320$, y esto es mucho mejor que 16.777.216 !. Aquí no hemos cambiado la representación del problema, pero si la forma de conceptualizar el espacio de búsqueda, y, debido a esto, hemos reducido el número de alternativas a considerar.

Esto se puede implementar en un algoritmo como el siguiente:

```
REINAS()
    A ← permutacion_inicial
    while A ≠ permutacion_inicial
        do A ← siguiente_permutacion
            if SOLUCION(A)
              then print A
```

Que nos plantea una pregunta natural¿Como permutar un arreglo? La solución consiste en pensar recursivamente. Por ejemplo, permutar:

a	b	c
1	2	3

Consiste en generar todas las posibilidades siguientes:

a	b	c
1	2	3

b	a	c
2	1	3

c	a	b
3	1	2

a	c	b
1	3	2

b	c	a
2	3	1

c	b	a
3	2	1

Lo que se puede generalizar de la siguiente forma. Si A tiene n elementos, las permutaciones se obtienen:

- Con $A[1]$, seguido de permutar $A[2..n]$
- Con $A[2]$, seguido de permutar $A[1] \cup A[3..n]$
- Con $A[3]$, seguido de permutar $A[1.,2] \cup A[3..n]$
- ...
- Con $A[n]$, seguido de permutar $A[1..n-1]$

Esta observación nos permite explicar el siguiente algoritmo para hallar una permutación del arreglo A, con tamaño n:

```
PERM(i,n,A)
    if i = n
       then print A
       else
            for j ← i to n
                do exchange A[i] ↔ A[j]
                   PERM(i+1,n,A)
                   exchange A[i] ↔ A[j]
```

Aquí ***i*** lleva la posición en el arreglo del elemento que se va a dejar fijo para permutar el resto del arreglo. El intercambio antes del llamado recursivo de perm nos permite colocar todos los elementos de A en la posición ***i*** a medida que el ciclo for avanza. El llamado recursivo permuta el resto del arreglo desde la posición i+1 y el último intercambio restablece el orden que el arreglo tenía antes de empezar la siguiente iteración del for. La complejidad de este algoritmo puede calcularse con la recurrencia:

$$T(n) = nT(n-1) + \Theta(1)$$

Cuya solución es $\Theta(n!)$, como es de esperarse.
Con éste algoritmo podemos refinar el cálculo de las posiciones de las reinas:

```
REINAS()
    A ← permutacion_inicial
    i ← 0
    while A ≠ permutacion_inicial
        do i ← i + 1
           PERM(i, n, A)
           if solucion(A)
             then print A
```

2.4. Solución por medio de backtracking

Observemos que las dos soluciones planteadas

- Ingenua
- Basada en permutaciones

Tienen el mismo defecto, solo prueban la validez de las posiciones cuando se han colocado todas las n reinas!. La primera idea que se puede tener es **hacer una revisión parcial** del arreglo A para evitar alternativas que tengan inconsistencias antes de situar todas las n reinas. Esto puede especificarse como garantizar que las reinas en las posiciones $(1,A[1]),(2,A[2]),\ldots,(k,A[k])$ no se atacan entre sí. Y donde el número k puede ser menor que n. Esta condición puede escribirse matemáticamente así:

$$A[i] \neq A[j] \quad i \neq j \quad i,j \in [1,k]$$
$$|A[i]-A[j]| \neq |i-j| \quad i \neq j \quad i,j \in [1,k]$$

Y podríamos decir que un erreglo A es k-Prometedor si cumple la condición anterior. De esta forma podemos escribir un algoritmo que chequee si un arreglo es k-Prometedor con una complejidad $\Theta(k)$:

```
PROMETEDOR(A,k)
    ▷ PRE: A ya es k-1 prometedor
    for j ← 1 to k − 1
        do if A[j] = A[k] or |A[j] − A[k]| = |i − k|
            return FALSE
    return TRUE
```

Observe que el algoritmo asume que el arreglo A ya era k-1 prometedor como precondición, esto permite bajar la complejidad que tenía **solucion** de $\Theta(n^2)$ a $\Theta(k)$. Con esta herramienta podemos plantear la búsqueda de soluciones de otra forma:

- Ir colocando las reinas una a una en columnas
    - Si el vector A es k prometedor (las primeras k reinas colocadas no se atacan)
    - Avanzar a colocar la reina k+1

Si tenemos éxito k irá aumentando de 1 hasta n y habremos situado las n reinas en columnas donde no se ataquen entre sí y obtendremos una solución. Si no tenemos exito, es necesario hacer una ***vuelta atrás*** para cambiar la columna en la que se colocó la última reina. Todo esto puede ilustrarse graficamente:

Una búsqueda recursiva que tenga como parámetro el número de variables instanciadas k, nos permitiría ir atrás automáticamente:

```
REINAS(A,k,n)
    if k = n
      then if PROMETEDOR(A,k)
            ▷ Se encontró la solución!
              then print A
              else return
      else
            for j ← 1 to n
                do if PROMETEDOR(A,k)
                      then A[k+1] ← j
                           REINAS(A,k+1,n)
```

El llamado inicial para iniciar puede ser:

REINAS$([0,0,0,0,0,0,0,0],0,8)$

Y esperaríamos que la complejidad del algoritmo sea mejor que $\Theta(n!)$. De hecho si utilizamos un contador para el número de llamados recursivos que se hacen a **reinas** podemos estimar la complejidad aproximadamente —analiticamente esto es muy difícil—. Para $n = 8$, reinas hace 15721 llamados recursivos que es mejor que $8! = 40320$.

Si escribimos éste algoritmo en un lenguaje de programación como python obtendríamos algo así:

```
global c
c = 0

def reinas(A,k,n):
    global c
    c = c+1
    if k==n:
        if prometedor(A,k):

            print A
        else: return
    else:
        for j in range(1,n+1):
            if prometedor(A,k):
```

```
                A[k+1]= j
                reinas(A, k+1,n)

L = 9*[0]
reinas(L,0,8)
print c
```

Con la siguiente implementación de la función prometedor:

```
def prometedor(A,k):
    #PRE: A ya es k-1 prometedor
    for j in range(1,k):
        if A[j]==A[k] or abs(A[j]-A[k])== abs(k-j):
            return False
    return True
```

3. Algoritmos ingenuos

Como vimos en la sección 2.2, una forma viable de resolver un problema de búsqueda consiste en explorar sistematicamente todos los valores que se le pueden asignar a las variables. Esto puede generalizarse a cualquier problema de búsqueda que tenga:

- n variables: $v_1, v_2, \ldots, v_n$
- con n dominios finitos: $D_1, D_2, \ldots, D_n$

La solución puede representarse mediante un arreglo A de n elementos que contenga en $A[k]$ un valor del dominio D_k asignado para la variable k. Mediante esta codificación puede construirse un algoritmo ingenuo para **cualquier** problema de búsqueda:

INGENUO()

foreach e_1 in D_1
  do foreach e_2 in D_2
    do foreach e_3 in D_3
      $\ddots$
        do foreach e_n in D_n
          $A \leftarrow (e_1, e_2, \ldots, e_n)$
          if SOLUCION(A)
            then print A

Cuya complejidad sería $\Theta(|D_1| \times |D_2| \ldots \times |D_n| \times S(n))$. En donde S sería la complejidad del algoritmo para chequear una **solucion.**

4. Backtracking

Como en la sección anterior, cualquier problema de n variables con n dominios puede codificarse en un arreglo A de n elementos

A =

1	2	3	4		...	n

En las que en en $A[k]$ se puede poner un valor del dominio D_k asignado para la variable k. Decimos que el arreglo A es **k-prometedor** si desde la posición 1 hasta la k tiene asinaciones de valores para las k primeras variables que no violan las condiciones del problema.

A =

A[1]	A[2]	A[3]	...	A[k]		
1	2	3	...	k		

Bactracking significa ir aumentando el k en cada paso hasta llegar a n y asignar un valor posible para la última variable. Si esto no tiene éxito, hay que dar *vuelta atrás* (backtrack) en un árbol de búsqueda implícito.

4.1. Implementación recursiva

La forma mas sencilla de escribir dicha vuelta atrás consiste en aprovecharse de la recursión:

```
INGENUO()
    foreach e_1 in D_1
        do foreach e_2 in D_2
                do foreach e_3 in D_3
                    ...
                        do foreach e_n in D_n
                            A ← (e_1, e_2, ..., e_n)
                            if solucion(A)
                              then print A
```

La mas antigua referencia a backtracking es la historia del hilo de Ariadna en la mitología griega. Su padre, el rey Minos de Creta detentaba su tiránico poder sobre su isla y, por conquista, sobre Atenas. El tributo que pedía de los atenienses consistía en grupos de jovenes que entraban a su laberinto donde habitaba el Minotauro (midat hombre, mitad toro) para encontrar la muerte a manos de éste. El héroe ateniense Teseo se ofreció voluntariamente a acompañar a un grupo de jovenes que se iban a sacrificar para salvar a Atenas de ésta cruel tradición. Ariadna se enamoró de Teseo y le regaló un ovillo de hilo dorado para que no se perdiera en el laberinto. La idea era que Teseo desenrrollara el ovillo a medida que avanzara por el laberinto, si llegaba a a un callejón sin salida tenía que *volver atrás* y enrrollar el hilo hasta llegar al punto donde había escogido dicho camino para intentar una ruta alternativa. El backtracking funcionó, Teseo asesinó al Minotauro y escapó del laberinto, y de la isla de Creta con los jovenes y con Ariadna.

4.2. Implementación Iterativa

La recursión puede evitarse si pensamos en la estrategia de backtracking como una búsqueda en profundidad en un árbol implícito en el que cada nodo interno es un arreglo A con una asignación parcial de valores para k variables. El parámetro k es el nivel de cada nodo y los nodos hoja consisten en asignaciones completas de variables para A, esto es $k = n = length[A]$. Una búsqueda iterativa puede hacerse bajando a través de los nodos siempre que sean prometedores, si nos topamos con uno que no sea prometedor subimos de nivel restando uno a k. Esto se puede especificar así:

```
BACKTRACK(A,k,n)
    k = 1
    while k > 0 and not ULTIMAOPCION(k)
        do
            while not ULTIMAOPCION(A,k) and PROMETEDOR(A,k)
                do A[k] ← PROXIMOELEMENTO(A,k)
                    if k=n and PROMETEDOR(A,k)
                      then print A
                      else  k ← k+1
            ▷ Vuelta atrás al nivel superior (Backtrack)
            k ← k−1
```

4.3. Como usar la técnica

Para usar el backtracking para resolver un problema determinado se hace necesario definir las funciones:

- ULTIMAOPCION
- PROMETEDOR
- PROXIMOELEMENTO

Cada una de estas debe pensarse para el problema específico que se este resolviendo. En el libro de Skiena [2] hay buenos ejemplos desarrollados de aplicación de backtracking, en el libro de Brassard encontrará una explicación de backtracking como técnica de exploración de arboles implícitos. [1]

Referencias

[1] Gilles Brassard and Paul Bratley. *Algorithmics: theory & practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[2] Steven S. Skiena. *The algorithm design manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.