Práctica 3

Lógica

Junco de las Heras Valenzuela
y
Marta Vaquerizo Núñez

Índice

Problema de las 8 reinas

Descripción del problema

El problema de las ocho reinas consiste en encontrar una forma de colocar esas ocho reinas en un tablero de ajedrez tal que no se ataquen entre ellas. Se dice que una reina puede atacar a otra si esta se encuentra en una fila, columna o diagonal en la que está la primera. Entonces, se busca una forma de colocar las reinas en el tablero sin que estén en la misma fila, columna o diagonal. Esta información se puede encontrar en el link donde está el código Prolog que se va a analizar posteriormente:

`https://www.ic.unicamp.br/~meidanis/courses/mc336/2009s2/prolog/problemas/`

y de forma un poco más extensiva en el siguiente link:
`https://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas`.

Explicación del algoritmo de la solución

El algoritmo recibe como input N, que es la dimensión del tablero a resolver el problema de las N damas, pero nosotros vamos a comentar el caso donde N es igual a 8.

Para el algoritmo, se representan las posiciones de las reinas en una lista de 8 elementos de números del 1 al 8, ya que hay 8 filas y 8 columnas. Por ejemplo, si la solución propuesta es $[2, 4, 6, 8, 3, 1, 7, 5]$, esto significa que la reina de la primera columna está en la segunda fila, la reina de la segunda columna está en la cuarta fila, y así con todas las columnas.

Primero inicializa la lista resultante con los números del 1 al 8, satisfaciendo la condicion de que en cada fila y columna no haya dos damas atacándose.

Luego, el algoritmo permuta las posiciones de la lista garantizando que no haya dos reinas en la misma fila ni columna, faltando garantizar que no haya dos en la misma diagonal. Sabiendo que una reina que está la posición (X,Y) ataca dos diagonales, y si nombramos a esas diagonales $C = X - Y$ (la que va de abajo a la izquierda a arriba a la derecha) y $D = X + Y$ (la que va de arriba a la izquierda a abajo a la derecha), podemos llevar la cuenta de que reinas están en las diagonales y de esa manera comprobar si esa combinación se puede dar.

Después de permutar el algoritmo le aplica una función test que comprueba que no haya dos damas en la misma diagonal. En caso de que las haya devuelve un False y hace backtracking, para proceder a encontrar la siguiente solución.

La información ha sido sacada de las webs anteriores y del código `p90.pl`.

Descripción del código Prolog

Para ejecutar el código se ejecuta el comando: `queens_1(N, Q)`, donde `N` es el número de reinas a colocar y Q es la solución al problema. En nuestro caso `queens_1(8, Q)`.

Lo primero que hace el código es generar una lista con los número del 1 al `N` en ese orden, para esto se usa una recursión con el predicado: `range(A,B,L)`, donde A es el número mínimo del rango, B es el número máximo del rango, y L es la lista con los números del A al B. Una vez generada esta lista, se van permutando los elementos de la misma con los predicados: `permu(Xs,Zs)`, donde Xs es la lista original, y Zs es el resultado de permutar Xs; y `del(X,[Y|Ys],[Y|Zs])`, donde X es el elemento a eliminar de la lista $[Y|Ys]$, y el resultado de la eliminación se devuelve por $[Y|Zs]$.

Tras la permutación, se comprueba que en las diagonales no haya más de una reina. Esto se hace con los predicados: `test(Qs)`, donde Qs es una solución al problema a testear; y `test(Qs,X,Cs,Ds)`, donde Qs es una lista de las filas en las que están las reinas desde la columna X a la columna N (en nuestro caso 8) de la solución propuesta, Cs es una lista con los numeros que representan las diagonales que van de abajo a la izquierda a arriba a la derecha, y Ds es una lista con los numeros que representan las diagonales que van de arriba a la izquierda a abajo a la derecha. Al llevar la cuenta de las diagonales, cada vez que se chequea si una reina (posicionada en (X, Y)) ataca alguna diagonal en la que hay alguna reina, solo hace falta ver si C = X − Y está en Cs, y si D = X + Y está en Ds. Si ocurre que C está en Cs o D en Ds, esa solución no vale, y se vuelve a permutar la lista original.

Ejecución del código

A continuación se va a mostrar la ejecución de la primera iteración del código para ver mejor lo que se ha contado anteriormente.

```
[trace]  ?- queens_1(8,Q).
   Call: (10) queens_1(8, _15516) ? creep
   Call: (11) range(1, 8, _15950) ? creep
   Call: (12) 1<8 ? creep
   Exit: (12) 1<8 ? creep
   Call: (12) _16108 is 1+1 ? creep
   Exit: (12) 2 is 1+1 ? creep
   Call: (12) range(2, 8, _16004) ? creep
   Call: (13) 2<8 ? creep
   Exit: (13) 2<8 ? creep
   Call: (13) _16362 is 2+1 ? creep
   Exit: (13) 3 is 2+1 ? creep
   Call: (13) range(3, 8, _16258) ? creep
   Call: (14) 3<8 ? creep
   Exit: (14) 3<8 ? creep
   Call: (14) _16616 is 3+1 ? creep
   Exit: (14) 4 is 3+1 ? creep
   Call: (14) range(4, 8, _16512) ? creep
   Call: (15) 4<8 ? creep
   Exit: (15) 4<8 ? creep
   Call: (15) _16870 is 4+1 ? creep
   Exit: (15) 5 is 4+1 ? creep
   Call: (15) range(5, 8, _16766) ? creep
   Call: (16) 5<8 ? creep
   Exit: (16) 5<8 ? creep
   Call: (16) _17124 is 5+1 ? creep
   Exit: (16) 6 is 5+1 ? creep
   Call: (16) range(6, 8, _17020) ? creep
   Call: (17) 6<8 ? creep
   Exit: (17) 6<8 ? creep
   Call: (17) _17378 is 6+1 ? creep
   Exit: (17) 7 is 6+1 ? creep
   Call: (17) range(7, 8, _17274) ? creep
   Call: (18) 7<8 ? creep
   Exit: (18) 7<8 ? creep
   Call: (18) _17632 is 7+1 ? creep
   Exit: (18) 8 is 7+1 ? creep
   Call: (18) range(8, 8, _17528) ? creep
   Exit: (18) range(8, 8, [8]) ? creep
   Exit: (17) range(7, 8, [7, 8]) ? creep
   Exit: (16) range(6, 8, [6, 7, 8]) ? creep
   Exit: (15) range(5, 8, [5, 6, 7, 8]) ? creep
   Exit: (14) range(4, 8, [4, 5, 6, 7, 8]) ? creep
   Exit: (13) range(3, 8, [3, 4, 5, 6, 7, 8]) ? creep
   Exit: (12) range(2, 8, [2, 3, 4, 5, 6, 7, 8]) ? creep
   Exit: (11) range(1, 8, [1, 2, 3, 4, 5, 6, 7, 8]) ? creep
```

Figura 1: Creación de la lista de números del 1 al 8.

Como se puede observar en la imagen, se llama recursivamente al predicado `range(X,8,L)`, donde X empieza siendo 1, y va aumentando en una unidad hasta que llega a 8, donde acaba la recursión, y al volver de la recursión, se van añadiendo los números que se han ido obteniendo al sumar de 1 en 1. A continuación, se permuta esta lista recién generada:

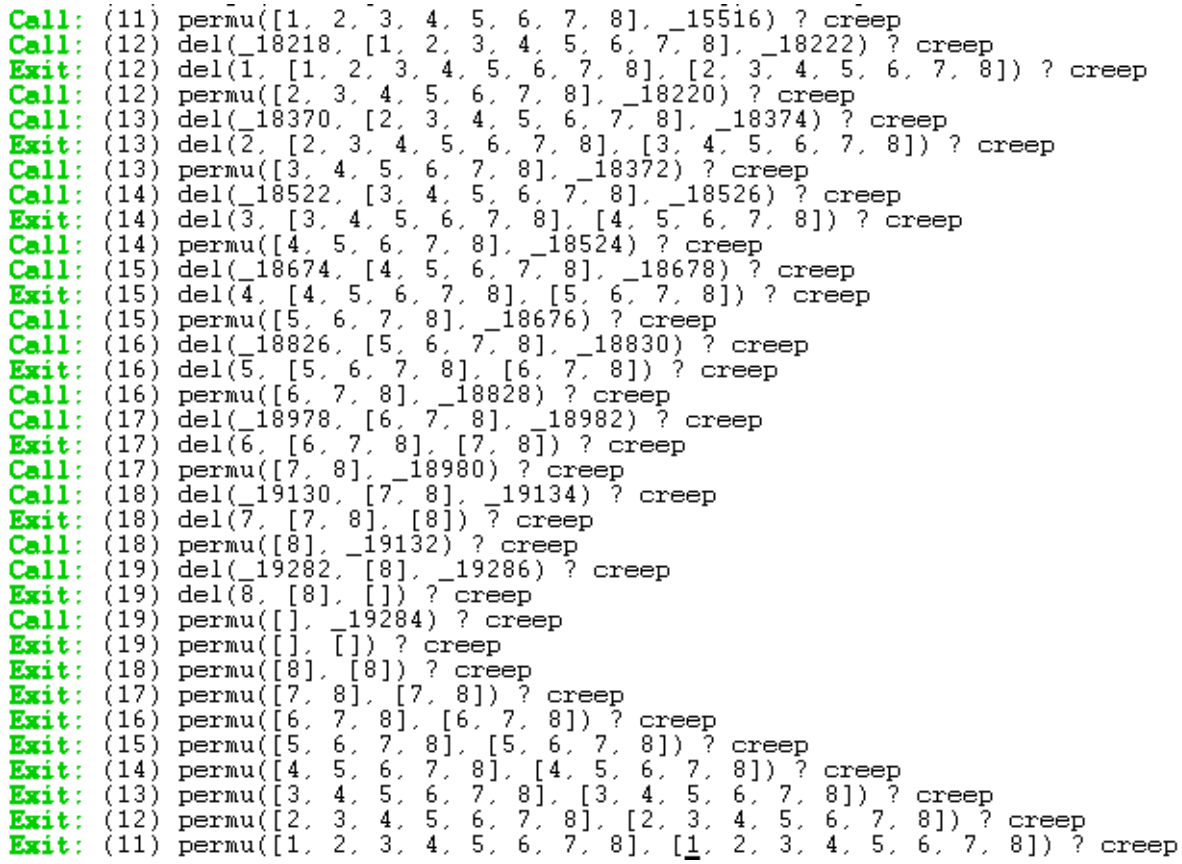

Figura 2: Generación de una permutación de la lista original.

En este caso, como es la primera iteración, no se produce ninguna permutación como tal, ya que se tiene que probar si la lista generada funciona, pero para observar las llamadas y el proceso de permutación nos sirve.

Se observa en la imagen que primero se llama a permutación con la lista original, y después se intercalan las llamadas de `del(X,L1,L2)` y de `permu(L2,R)`, donde L1 es la lista a la que se le quita el elemento X, y L2 es el resultado de esa eliminación, hasta que L2 se vacia. En la vuelta de la recursión se forma la permutación, que en este caso, como se ha comentado es la misma lista que la original, generada al principio.

De todas maneras, se muestra a continuación el mismo proceso de generación de una permutación, pero de la segunda iteración, en la que ya hay una permutación real.

```
Redo: (19) permu([], _19284) ? creep
Call: (20) del(_21014, [], _21018) ? creep
Fail: (20) del(_21014, [], _21018) ? creep
Fail: (19) permu([], _19284) ? creep
Redo: (19) del(_19282, [8], _19286) ? creep
Call: (20) del(_19282, [], _21216) ? creep
Fail: (20) del(_19282, [], _21216) ? creep
Fail: (19) del(_19282, [8], _19286) ? creep
Fail: (18) permu([8], _19132) ? creep
Redo: (18) del(_19130, [7, 8], _19134) ? creep
Call: (19) del(_19130, [8], _21462) ? creep
Exit: (19) del(8, [8], []) ? creep
Exit: (18) del(8, [7, 8], [7]) ? creep
Call: (18) permu([7], _19132) ? creep
Call: (19) del(_21658, [7], _21662) ? creep
Exit: (19) del(7, [7], []) ? creep
Call: (19) permu([], _21660) ? creep
Exit: (19) permu([], []) ? creep
Exit: (18) permu([7], [7]) ? creep
Exit: (17) permu([7, 8], [8, 7]) ? creep
Exit: (16) permu([6, 7, 8], [6, 8, 7]) ? creep
Exit: (15) permu([5, 6, 7, 8], [5, 6, 8, 7]) ? creep
Exit: (14) permu([4, 5, 6, 7, 8], [4, 5, 6, 8, 7]) ? creep
Exit: (13) permu([3, 4, 5, 6, 7, 8], [3, 4, 5, 6, 8, 7]) ? creep
Exit: (12) permu([2, 3, 4, 5, 6, 7, 8], [2, 3, 4, 5, 6, 8, 7]) ? creep
Exit: (11) permu([1, 2, 3, 4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 8, 7]) ? creep
```

Figura 3: Generación de una permutación de la lista original, segunda iteración.

En este caso, se puede observar que el no se llama a permutar con la lista original, sino que se hace un **Redo** de la llamada a `permu`, para generar una permutación nueva. Y por último se comprueba que las diagonales también son atacadas por una sola reina:

```
Call: (12) test([1, 2, 3, 4, 5, 6, 7, 8], 1, [], []) ? creep
Call: (13) _19960 is 1-1 ? creep
Exit: (13) 0 is 1-1 ? creep
Call: (13) memberchk(0, []) ? creep
Fail: (13) memberchk(0, []) ? creep
Redo: (12) test([1, 2, 3, 4, 5, 6, 7, 8], 1, [], []) ? creep
Call: (13) _20208 is 1+1 ? creep
Exit: (13) 2 is 1+1 ? creep
Call: (13) memberchk(2, []) ? creep
Fail: (13) memberchk(2, []) ? creep
Redo: (12) test([1, 2, 3, 4, 5, 6, 7, 8], 1, [], []) ? creep
Call: (13) _20462 is 1+1 ? creep
Exit: (13) 2 is 1+1 ? creep
Call: (13) test([2, 3, 4, 5, 6, 7, 8], 2, [0], [2]) ? creep
Call: (14) _20620 is 2-2 ? creep
Exit: (14) 0 is 2-2 ? creep
Call: (14) memberchk(0, [0]) ? creep
Exit: (14) memberchk(0, [0]) ? creep
Fail: (13) test([2, 3, 4, 5, 6, 7, 8], 2, [0], [2]) ? creep
Fail: (12) test([1, 2, 3, 4, 5, 6, 7, 8], 1, [], []) ? creep
Fail: (11) test([1, 2, 3, 4, 5, 6, 7, 8]) ? creep
Redo: (19) permu([], _19284) ? creep _
```

Figura 4: Comprobación de que en las diagonales no hay más que una reina.

En esta parte de la ejecución, se puede ver que tras llamar a `test(L, 1, Cs,Ds`, donde L es la permutación propuesta como solución, y donde inicialmente Cs y Ds son [ ], se calcula C de la primera reina, y se comprueba si C (en este caso con valor 0) está en Cs, pero como es la primera reina y Cs es [ ], pasa a comprobar si D (en este caso con valor 2) está en Ds, y como Ds es [ ], se pasa a la segunda reina. Antes de eso, a Cs se añade 0, pasando a ser Cs = [0], y a Ds se añade 2, pasando a ser Ds = [2]. Entonces pasa a la segunda reina, y vuelve a repetir el proceso. Primero calcula C, que en este caso tiene el valor 0, y se comprueba si C está en Cs = [0], como está, esa solución ya no es valida, entonces se vuelve de la recursión para repetir el proceso de generar otra solución con otra permutación. A continuación, se muestra una captura de una ejecución sin la opción `trace`, para ver algunos resultados.

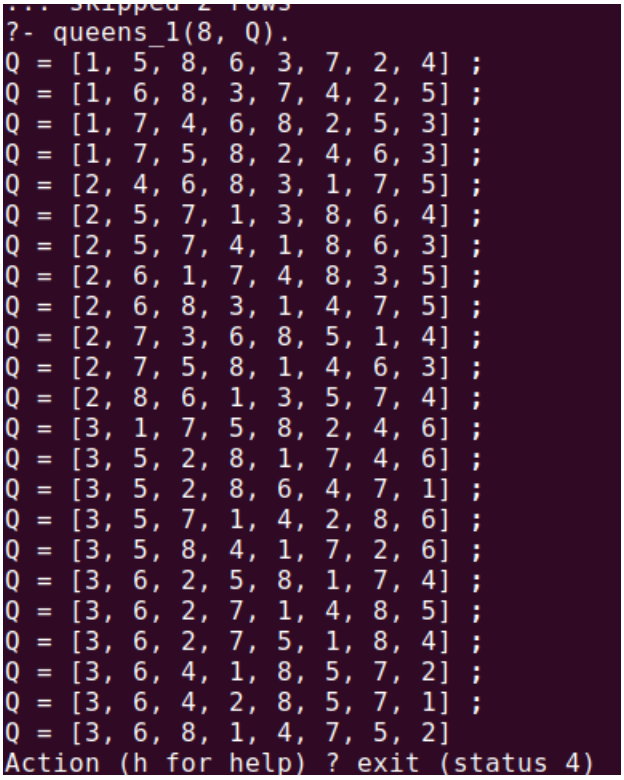

Figura 5: Ejemplo de ejecución con N = 8 y algunos primeros resultados.

Comentario personal sobre la solución

Nos parece que es una solución adecuada, ya que es capaz de encontrar todas las soluciones al problema. Sin embargo, hay algunas posibles mejoras al algoritmo. El algoritmo tiene tres fases: la primera es escribir los números del 1 al 8, la segunda es permutar esos números, y la tercera consiste en aplicar a la permutación la función test para comprobar si la solución es valida, pero esto no es óptimo, ya que si la primera dama está en la casilla $(1, 1)$ y la segunda en la casilla $(2, 2)$ entonces se están atacando por la diagonal, pero el algoritmo efectúa 6! operaciones permutando las columnas de la 3 a la 8.

Lo óptimo sería ir haciendo las comprobaciones de la diagonal al mismo tiempo que se va generando la permutación. Nótese que para ir comprobando una diagonal en la posición (i, j) no hace falta recorrer una por una todas las casillas hasta llegar al borde del tablero (que tendría complejidad $O(\mathrm{N})$), sino que se podría guardar una lista de visitados donde visitados[x] es `True` si hay una dama en la diagonal x, y como está en la casilla (i, j), la diagonal es la i+j, lo que permite tener una complejidad de $O(1)$ en la comprobación de las diagonales.