Programación Declarativa

Caso de Estudio

Desarrollo de una interfaz gráfica para el problema de las cuatro reinas con SWI-Prolog

Grupo ET2
Tercer curso
Computación
2018/2019

Pedregal Hidalgo, Diego
Velasco Mata, Alberto

Índice

1. Conceptos teóricos y prácticos

1.1. Definición del problema

Nuestro caso de estudio trata de trabajar con el desarrollo de interfaces gráficas dentro de la herramienta de programación lógica *Prolog*, más concretamente utilizando su librería específica para estos aspectos: *XPCE*. Para ello, se nos ha planteado un problema, que consiste en desarrollar una interfaz gráfica para resolver el **problema de las cuatro reinas**.

Aunque es muy conocido, consideramos conveniente describirlo brevemente: el problema de las cuatro reinas consiste en colocar cuatro reinas, en un tablero de dimensiones 4 x 4, de tal manera que ninguna de las reinas se ataque con otra. Obviamente, este problema se puede extender a cualquier número de reinas, digamos n, de tal manera que, dispuestas en un tablero de n x n dimensiones, tampoco se ataque ninguna de ellas.

Una vez planteado el problema, nosotros hemos decidido asumir el reto de resolver el problema de n reinas, en vez de limitarnos simplemente a cuatro. Así pues, a continuación, presentamos la resolución y procedimientos desarrollados para dar solución a este caso de estudio.

1.2. XPCE

XPCE es la herramienta que *SWI-Prolog* usa para la creación de aplicaciones gráficas. Así, a lo largo de la siguiente sección vamos a explicar lo mejor posible los conocimientos básicos utilizados para el desarrollo de nuestro caso de estudio.

XPCE, al formar parte de *Prolog*, funciona básicamente con predicados, de los cuales, los más importantes en nuestro caso son cuatro: `new/2`, `send/2`, `get/3` y `free/1`.

1.2.1. new/2

`new/2` es un predicado que sigue la siguiente estructura: `new(?Referencia, +Elemento)`, de forma que se crea un objeto `Elemento` al que le asigna la referencia dada, o lo unifica con una referencia ya existente, según el caso.

`Elemento` será un término en el que el funtor indicará la clase de objeto que se creará, y como argumentos recibirá los parámetros necesarios para inicializarla por primera vez. En nuestro caso, podemos observar varios ejemplos de uso extraídos del programa desarrollado:

```
new(Dialog, dialog('Problema de N Reinas'))
new(@txtStep, int_item(solución, 1, low:=1, high:=10000))
new(TotalImage, image(ImageName))
```

1.2.2. send/2

Este predicado se usa para realizar modificaciones en los estados de los distintos objetos instanciados. También consta de dos argumentos, que se corresponden respectivamente con la referencia al objeto cuyo estado se pretende alterar, y el método que se desea invocar en el objeto al que se hace referencia, con los argumentos necesarios. Igualmente, tenemos numerosos ejemplos a lo largo de nuestro programa:

```
send(Dialog, open)
```

Abriría la ventana `Dialog`.

```
send(@resultWindow, width(NReinas*32))
```

Modificaría la anchura de `@resultWindow`.

```
send(@resultWindow, display, @cb)
```

Haría que `@cb` se mostrase en la pantalla `@resultWindow`.

1.2.3. get/3

Por medio de este predicado podemos obtener la información deseada de los objetos instanciados en ese momento. Los dos primeros argumentos son exactamente iguales a los de `send/2`, mientras que el último es el argumento que *Prolog* unifica con el valor de retorno, que suele ser una referencia a algún objeto. Algunos ejemplos son:

```
get(Bitmap, square_colour, SquareColour)
```

Para obtener el color del cuadrado en el que vamos a colocar una reina.

1.2.4. free/1

Por último, este predicado es el más sencillo de los cuatro, ya que simplemente se utiliza para eliminar el objeto de la base de objetos de *XPCE*. Básicamente, consiste en eliminar el objeto, quedando la referencia que tenía anteriormente libre para nuevo uso. En nuestro código tenemos ejemplos de ello también:

```
free(@cb)
```

Con esta orden podemos limpiar el tablero de ajedrez.

1.3. Problema de las N reinas

Como hemos indicado anteriormente, hemos decidido enfocar el problema de manera general, es decir, desarrollando una solución viable para ***n reinas***, en lugar de la propuesta inicial de *cuatro reinas*. Así pues, este caso de estudio se divide principalmente en dos áreas: la *resolución* del problema de ***n reinas***, y la integración de esta resolución en una *interfaz gráfica* con la que mostrarla al usuario.

En esta sección nos concentramos en explicar la resolución del problema de las *n reinas*, para lo cual hemos cogido como referencia la solución aplicada por *Pascual* en su libro *Programación Lógica. Teoría y Práctica*, en el ***Capítulo 7, sección 5.2***. Realmente lo que hemos hecho es utilizar su código, obviando la parte de representación de la solución imitando el tablero de ajedrez.

Lo que sí debemos reseñar es, que esta solución solamente contemplaba ocho reinas, por lo que hemos tenido que realizar algunas adaptaciones para hacerlo extensible a *n*. Siguiendo el orden de implementación del libro, procedemos a explicar los principales cambios realizados.

a. **`vector(Solucion)`**. En lugar de generar directamente una lista con 8 elementos, nosotros debemos crear una para cualquier número de elementos, por lo que introducimos un predicado nuevo, `creaListaC(N, L)`, donde `L` es una lista de `N` elementos con el formato `c(X, Y)`, que expresa la casilla donde se ubica cada reina (coordenada X, coordenada Y).
b. **`generar(L, N)`**. En este caso, el predicado se mantiene prácticamente igual, con dos excepciones mínimas: hay que introducir un nuevo argumento, `N`, para saber cuántas reinas tenemos que calcular; y hay que crear otro nuevo predicado `creaLista(N, L)`, que funciona exactamente igual que `creaListaC/2`, pero generando simplemente una lista `L` de `N` números del `1` a `N`. Este último predicado es necesario para sustituir la lista que el predicado generar original crea directamente de ocho elementos.

El último predicado necesario, `ataca/2` es utilizado tal cual aparece en la solución original del libro.

```
?- vector(Sol, 5).
Sol = [c(1, 5), c(2, 3), c(3, 1), c(4, 4), c(5, 2)] ;
Sol = [c(1, 5), c(2, 2), c(3, 4), c(4, 1), c(5, 3)] ;
Sol = [c(1, 4), c(2, 2), c(3, 5), c(4, 3), c(5, 1)] ;
Sol = [c(1, 4), c(2, 1), c(3, 3), c(4, 5), c(5, 2)] ;
Sol = [c(1, 3), c(2, 5), c(3, 2), c(4, 4), c(5, 1)] ;
Sol = [c(1, 3), c(2, 1), c(3, 4), c(4, 2), c(5, 5)] ;
Sol = [c(1, 2), c(2, 5), c(3, 3), c(4, 1), c(5, 4)] ;
Sol = [c(1, 2), c(2, 4), c(3, 1), c(4, 3), c(5, 5)] ;
Sol = [c(1, 1), c(2, 4), c(3, 2), c(4, 5), c(5, 3)] ;
Sol = [c(1, 1), c(2, 3), c(3, 5), c(4, 2), c(5, 4)] ;
false.
```

Ejemplo de solución para 5 reinas

2. Documentación

2.1. Planteamiento de la solución

El diseño que hemos propuesto para la interfaz del problema consiste en el manejo del programa mediante dos ventanas. Una de ellas mostraría únicamente el tablero con una solución, mientras que la otra permitiría controlar los parámetros de la solución que está siendo mostrada. Esta última ventana permitiría también poder iterar las soluciones posibles, mostrándolas en la ventana de resultados.

De esta manera, en la ventana de control tenemos los siguientes elementos:

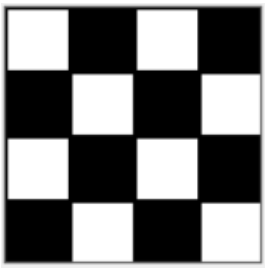
a. **Número de reinas**. Sencillamente una entrada numérica donde podemos indicar el número de reinas que queremos colocar, en un rango de 4 a 100. Es importante destacar que, a partir de un valor de aproximadamente 15, la solución consume demasiados recursos como para ser resuelta utilizando el código anterior. Sin embargo, y puesto que depende del computador de cada usuario, hemos decidido dejar ese rango de valores posibles.
b. **Número de solución**. Este número indica la solución que se quiere mostrar. Por ejemplo, al resolver el problema de las cuatro reinas, hay dos soluciones, por lo que escribiendo 1, dibujaría la primera solución, y escribiendo 2, la segunda y última solución.
c. **Calcular solución**. Presionando este botón, la solución deseada según los dos parámetros anteriormente mencionados es calculada, y mostrada en el tablero.
d. **Siguiente solución**. Este botón permite incrementar en una unidad el índice de solución y actualizar el tablero mostrándola, de forma que pulsándolo repetidamente se puede iterar visualmente sobre todas las soluciones posibles.

2.2. Interfaz gráfica procedural

2.2.1. Dibujo del tablero

Para dibujar el tablero hemos creado un predicado específico, `make_chess_boad/2`, que crea un objeto gráfico de *XPCE* y le va añadiendo tantos objetos de tipo box como celdas tendría un tablero de tamaño *NxN*.

El predicado `between/3` nos permite simular iteraciones en los ejes X e Y, en los cuales se van definiendo las coordenadas en las que se posicionarán los cuadrados o celdas del tablero. Posteriormente, se crea la propia celda como un objeto gráfico de tipo `box`. Este tipo primitivo es uno de los objetos predefinidos de *XPCE*. Una vez creada y añadida la celda al tablero, se le asigna un color que viene dado por otro predicado que hemos definido, `square_colour/3`.



Este predicado permite que `Colour` unifique con el color correspondiente a la casilla (`X`, `Y`).

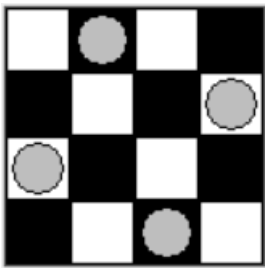
```
/* DIBUJO DE TABLERO */

make_chess_board(Board, N) :-
    new(Board, device),
    (   between(1, N, X),
        between(1, N, Y),
            GX is (X-1) * 32,
            GY is (N-Y) * 32,
            send(Board, display, new(Cell, box(32,32)), point(GX,GY)),
            /* Definimos el color de relleno de esta casilla dependiendo de su
posición */
            square_colour(X, Y, Colour),
            send(Cell, fill_pattern, colour(Colour)),
        fail ; true
    ).

%square_colour: Colour es el color correspondiente a la casilla (X,Y)
square_colour(X, Y, Colour) :-
    (X+Y) mod 2 =:= 0 ->
        Colour = black;
        Colour = white.
```

2.2.2. Dibujo de la solución

El primer paso para dibujar la solución en el tablero es poder dibujar una reina en una posición concreta del mismo. Para ello hemos definido el predicado `put/2`, que calcula las coordenadas del gráfico del tablero en las que ha de posicionar la reina, crea un objetivo primitivo de tipo `circle`, lo añade al gráfico y lo colorea de gris (para que sea visible tanto en las casillas negras como en las blancas).

Una vez definido esto, necesitamos otro predicado que, dada una solución en forma de lista con las posiciones de las reinas, las vaya colocando en el tablero. El predicado `pinta/1` realiza

esta función, extrayendo la cabeza de la lista y colocando esa reina en el gráfico usando el predicado anterior.

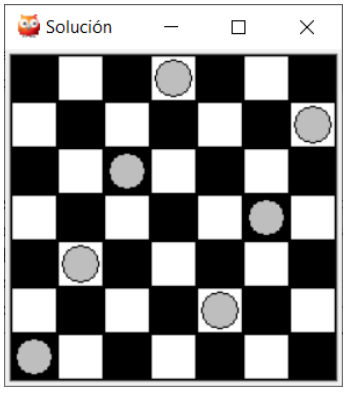


Por último, hemos considerado necesario definir otro predicado que interactúe con la ventana de la solución que se comentará posteriormente. Este predicado tendrá la función de eliminar el gráfico del tablero anterior, dibujar un nuevo tablero vacío y colocar todas las reinas de una solución concreta. Este predicado es `showSolution/2`, que será utilizado en el siguiente apartado.

```
/* DIBUJO DE LA SOLUCIÓN */

showSolution(NReinas, Sol) :-
        free(@cb),
        make_chess_board(@cb, NReinas),
        send(@resultWindow, display, @cb),
        pinta(Sol).

pinta([c(X, Y)|R]) :-
        put(@cb, [X,Y]),
        pinta(R).

put(Board, [X,Y]) :-
    GX is (X-1)*32+3,
    GY is (Y-1)*32+3,
    send(Board, display, new(Circle, circle(26)), point(GX,GY)),
    send(Circle, fill_pattern, colour(gray)).
```

2.2.3. Ventana de resultados

Teniendo en cuenta el diseño que propusimos para este problema, necesitamos manejar una ventana en la que se muestren en forma de tablero los resultados del problema. Hemos definido el predicado `resolver/2` con ese objetivo.



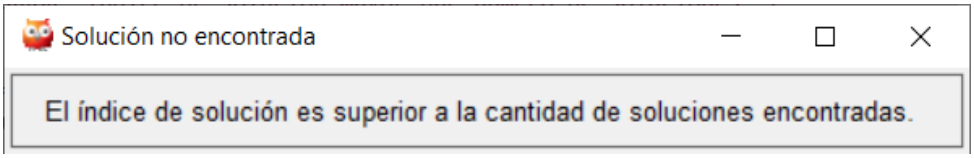
Lo primero que debe hacer este predicado es controlar si ya existe una ventana que muestre los resultados, con el objetivo de no crear una nueva ventana para cada solución. Posteriormente, generará la lista con todas las soluciones posibles y mostrará la que haya sido elegida en la ventana de control.

```
/* VENTANA DE RESULTADOS */

resolver(NReinas, SolutionIndex) :-
        /* Comprobamos si existe la ventana para los resultados */
        (   not(object(@resultWindow)) ->
            /* ResultWindow no existe, la creamos y la abrimos */
            new(@resultWindow, window('Solución', size(NReinas*32, NReinas*32))),
            send(@resultWindow, open);
```

```
            /* ResultWindow existe, reajustamos las dimensiones por si NReinas ha
cambiado */
            send(@resultWindow, width(NReinas*32)),
            send(@resultWindow, height(NReinas*32))
        ),
        /* Limpiamos el tablero*/
        free(@cb),
        make_chess_board(@cb, NReinas),
        send(@resultWindow, display, @cb),
        /* Buscamos la solución número SolutionIndex */
        findall(Sol, vector(Sol, NReinas), Solutions),
        length(Solutions, AllSolutionsCount),
        resolver(NReinas, SolutionIndex, Solutions, AllSolutionsCount).

%  resolver(NReinas, SolutionIndex, Solutions, NSolutions)
%         NReinas: Número de reinas (tamaño del tablero) a resolver
%         SolutionIndex: Solución que queremos mostrar
%         Solutions: Lista de soluciones que quedan
%         NSolutions: Número total de soluciones posibles
resolver(_, _, [], _) :- mensaje_error.
resolver(NReinas, SolutionIndex, [Sol|Sr], NSolutions) :-
        length(Sr, NSolsRestantes),
        Index is NSolutions - NSolsRestantes,
        (   Index = SolutionIndex ->
            showSolution(NReinas, Sol);
            resolver(NReinas, SolutionIndex, Sr, NSolutions)
        ).
```

Como puede darse el caso de que el usuario intente obtener una *n-ésima solución* sin existir tantas soluciones para esa dimensión del problema, hemos decidido mostrar un *mensaje de aviso o error* para notificarle de la imposibilidad de calcular esa solución.

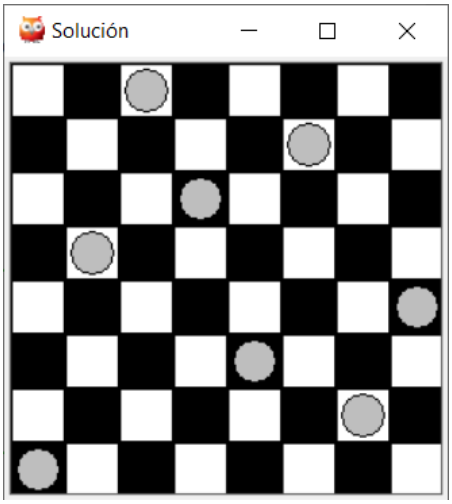
```
/* VENTANA DE ERROR: Índice de solución mayor que número de soluciones */

mensaje_error :-
        new(Dialog, dialog('Solución no encontrada')),
        send_list(Dialog, append, [
                    label(error_label, "El índice de solución es superior a la
cantidad de soluciones encontradas.")
                ]),
        send(Dialog, open).
```



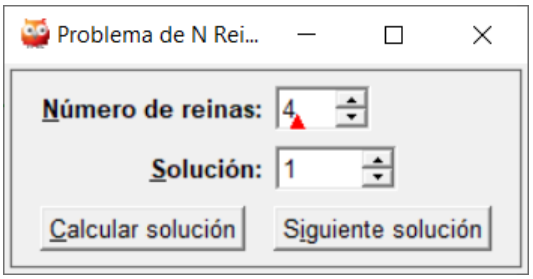
Es importante destacar que, aunque nuestro programa está pensado para ser manejado desde una ventana de control, este predicado puede utilizarse de forma directa para obtener una representación visual de una solución concreta. De esta manera,

```
?- resolver(8,1).
```

mostraría la siguiente ventana:



2.2.4. Ventana de control



Como hemos comentado en planteamiento de la solución, nuestro objetivo es poder controlar las soluciones mostradas desde una ventana de control. Para ello, hemos definido el predicado `n_reinas/0`, que crea una ventana con dos selectores de números. Uno de ellos permite la elección del número de reinas, equivalente al tamaño de un lateral del tablero. El otro permite elegir cuál de las posibles soluciones queremos que sea mostrada, en forma de índice de una lista que contendría todas las soluciones.

Como resulta evidente, necesitamos también un botón que permita calcular la solución seleccionada de forma acorde a los dos parámetros anteriores.

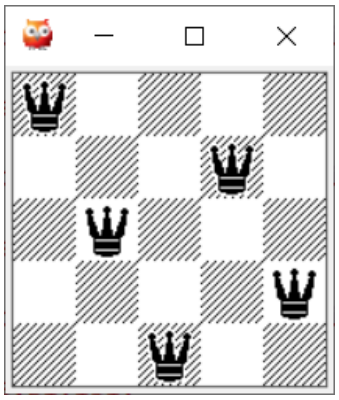
Hemos incluido también un botón extra que permite incrementar en una unidad el índice de la solución y recalcular la solución mostrada. La función sería equivalente a realizar esa secuencia de acciones con los otros componentes de la interfaz, pero creemos que puede ser útil para el usuario tener disponible esta función para mostrar de forma iterativa todas las soluciones.

Por último, abrimos la ventana que hemos creado y mostramos la ventana de solución con los parámetros por defecto.

```
/* VENTANA DE CONTROL */

n_reinas :-
        new(Dialog, dialog('Problema de N Reinas')),
        free(@txtStep),
        send_list(Dialog, append,
              [ new(TxtReinas, int_item(número_de_reinas, 4, low := 4, high := 100)),
                new(@txtStep, int_item(solución, 1, low:=1, high:=10000)),
```

```
                button(calcular_solución,
                       message(@prolog, resolver,
                               TxtReinas?selection, @txtStep?selection)),
                button(siguiente_solución,
                       message(@prolog, siguienteSolucion,
                               TxtReinas?selection, @txtStep?selection))
              ]),
        send(Dialog, open),
        resolver(4,1).

siguienteSolucion(NReinas, SolutionIndex) :-
        NextIndex is SolutionIndex + 1,
        send(@txtStep, selection(NextIndex)),
        resolver(NReinas, NextIndex).
```

2.3. Interfaz gráfica con carga de imágenes



La interfaz explicada hasta ahora es completamente funcional. Sin embargo, las reinas se muestran como puntos. XPCE permite cargar imágenes, por lo que pensamos que sería un buen añadido mostrar una pequeña imagen de la pieza de la reina en lugar de un punto.

Para ello, hemos aprovechado que la documentación de XPCE incluye un ejemplo en el que se crea una interfaz visual para el programa de ajedrez `gnuchessx`, que inicialmente se maneja mediante línea de comandos.

Hemos tomado de este el predicado `square_image/4`, que permite obtener un objeto gráfico de tipo `Bitmap` indicando la pieza de ajedrez que queremos y su color. XPCE no maneja la transparencia de las imágenes, por lo que si la superponemos directamente taparíamos el color de la casilla de debajo. Por ello, necesitamos proporcionarle también a este predicado el color del cuadrado del tablero, de forma que se carga una imagen de la pieza en un cuadrado de ese color.

Además de este añadido, hay que hacer algunas pequeñas modificaciones en los predicados `make_chess_board/2` y `put/2`, para que utilicen el objeto `Bitmap` proporcionado por este predicado en lugar de `box` y `circle`, respectivamente. Estas pequeñas modificaciones pueden verse en el código adjunto.

```
/* CARGA DE IMÁGENES */

%       square_image(+PieceName, +PieceColour, +SquareColour, -Image)
square_image(Piece, PieceColour, SquareColour, Image) :-
    computed_image(Piece, PieceColour, SquareColour, Image),
    !.
square_image(Piece, PieceColour, SquareColour, Image) :-
    image_name(Piece, ImageName),
    new(TotalImage, image(ImageName)),
    sub_area(PieceColour, SquareColour, Area),
    !,
    get(TotalImage, clip, Area, Image),
    send(Image, lock_object, @on),
    send(Image, attribute, attribute(piece, Piece)),
    send(Image, attribute, attribute(colour, PieceColour)),
    asserta(computed_image(Piece, PieceColour, SquareColour, Image)).

%       image_name(?PieceName, ?ChessProgram Id, ?ImageName)
image_name(empty, 'chesssquare.bm').
image_name(queen, 'queen.bm').

%       sub_area(+PieceColour, +SquareColour, -AreaTerm)
sub_area(white, white, area(32,  0, 32, 32)).
sub_area(white, black, area(0,   0, 32, 32)).
sub_area(black, white, area(32, 32, 32, 32)).
sub_area(black, black, area(0,  32, 32, 32)).
```

3. Bibliografía

1. *XPCE: the SWI-Prolog native GUI library*.
   De http://www.swi-prolog.org/packages/xpce/
2. Wielemaker, J & Anjewierden, A. *Programming in XPCE/Prolog*.
   De http://www.swi-prolog.org/packages/xpce/UserGuide/
3. Julián Iranzo, P. & Alpuente Frasnedo, M. (2007). Programación Lógica. Teoría y Práctica. Madrid: Pearson Prentice Hall. (pp. 265-267)