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

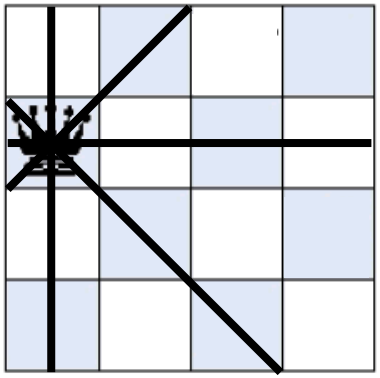
Alberto Velasco Mata
Diego Pedregal Hidalgo

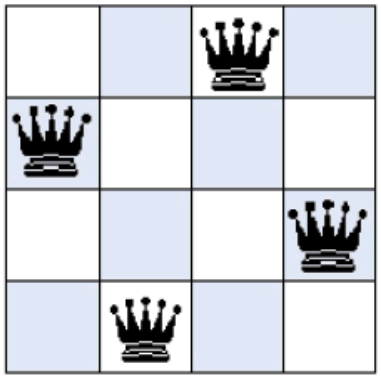
*Grupo ET2*


Escuela Superior de Informática

# Definición del problema

Problema de las 4 reinas → *N* reinas





SWI Prolog XPCE

# XPCE

*new/2*

- Crea objeto
- Primer argumento → referencia del objeto
- Segundo argumento → *elemento(arg1, arg2, …)*

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

*send/2*

- Modifica estados de objetos
- Primer argumento → referencia del objeto
- Segundo argumento → método a invocar sobre el objeto

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

# XPCE

*get/3*

- Obtener información de objetos
- Primer argumento → referencia del objeto
- Segundo argumento → método a invocar sobre el objeto
- Tercer argumento → unificación con valor de retorno

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

*free/1*

- Eliminar objetos
- Argumento → referencia del objeto

```
free(@cb)
```

# Resolución del problema de las n reinas

- Código → *Programación Lógica. Teoría y Práctica*

- Modificaciones 8 reinas → *n* reinas
  - *vector(Solucion)*
    - Añadimos *creaListaC(N, L)*
  - *generar(L, N)*
    - Añadimos *N*
    - Añadimos *creaLista(N, L)* ≈ *creaListaC(N, L)*

```
?- 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.
```

# Diseño de la interfaz

## Parámetros

- Número de reinas
  - Entrada numérica (4 – 100)
- Número de solución
  - Entrada numérica (1 – nº soluciones)

## Botones

- Calcular solución
  - Calcula solución indicada
- Siguiente solución
  - Muestra siguiente solución posible (si existe)

# Dibujo del tablero

*make_chess_board/2*

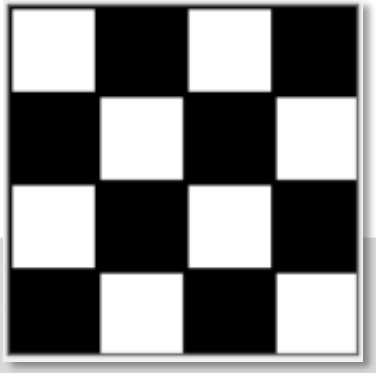
- Crea objeto gráfico *XPCE*

*between/3*

- Iteraciones ejes X e Y

*square_colour/3*

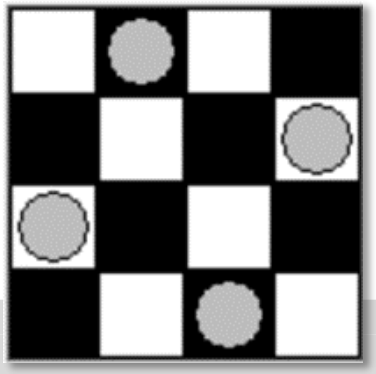
- Asigna color a casillas



```
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.
```

# Dibujo de la solución



*put/2*

- Coloca reina en la casilla correspondiente

*pinta/1*

- Pinta solución en el tablero

*showSolution(Nreinas, Sol)*

- Elimina tablero antiguo
- Dibuja nuevo tablero
- Coloca reinas según *Sol*

```
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)).
```

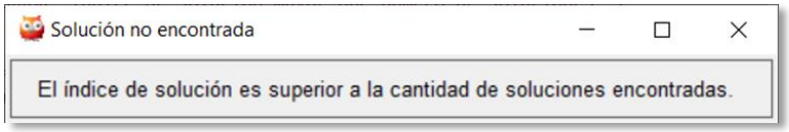
# Ventana de la solución

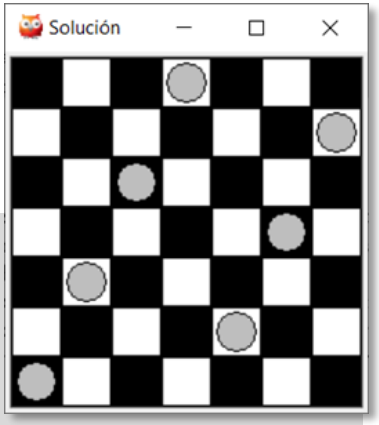
*resolver/2*

- Comprueba si ya existe ventana de resultados
- Genera lista con soluciones
- Muestra solución elegida

```
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(_, _, [], _) :- 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)
        ).
```

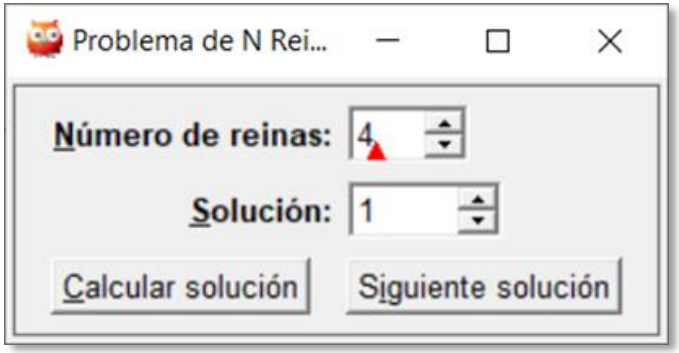
# Ventana de control

*n_reinas/0*

- Crea ventana principal

```
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).
```



# Mejora:
## Carga de imágenes

→ Ejemplo para *gnuchessx*

*square_image/4*

- Obtiene objeto Bitmap
  - Pieza de ajedrez
  - Cuadro del tablero

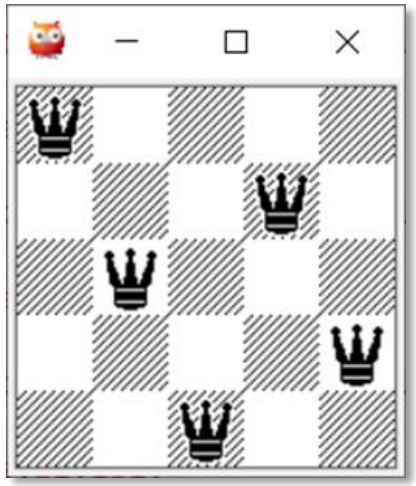
Modificaciones → para usar Bitmap

- *make_chess_board/2*
- *put/2*

```
%       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)).
```



# DEMOSTRACIÓN