Documentación Nonogram

Materia: Lógica para ciencias de la computación

Etapa del Proyecto: Etapa 1

Fecha: 03/05

Comisión: 12

Integrante 1: Marcos Simón Fernandez - 142203

Integrante 2: Marcos Goizueta- 142071

ÍNDICE

Resolución de requerimientos funcionales (solo prolog):

Presentar al jugador una grilla interactiva con pistas:

Todo lo relacionado a esto ya estaba implementado en prolog por la cátedra.

Permitir elegir, en todo momento, el modo de marcado (checkbox / toggle), entre pintar o modo cruz:

Implementado por nosotros de forma total en react.

En todo momento, y para cada fila o columna, indicar visualmente si ésta satisface las pistas asociadas:

Utilizamos los predicados ***initClues/5*** e ***initCluesAux/5*** para inicializar por primera vez las pistas y así mostrar en pantalla si están satisfechas o no desde el inicio.

initClues/5:

Este predicado cuenta con los siguientes parámetros: initClues(+Grid,+RowsClues,+ColsClues,-NewSatisfiedRowClues,-NewSatisfiedColClues). Se encarga de inicializar todas las pistas apenas comienza el nivel, usando la grilla y las pistas recibidas por parámetros, derivando esta tarea a ***initCluesAux/3*** para el caso de las filas y columnas por separado.

```
initClues(Grid,RowsClues, ColsClues,NewSatisfiedRowClues,NewSatisfiedColClues):-
    initCluesAux(Grid,RowsClues,NewSatisfiedRowClues),
    transpose(Grid,TransposeGrid),
    initCluesAux(TransposeGrid,ColsClues,NewSatisfiedColClues).
```

initCluesAux/5:

Este predicado cuenta con los siguientes parámetros: initCluesAux(+Grid,+LineClues,-NewSatisfiedLineClues). Va verificando si se cumplen las pistas de la línea por medio del predicado antes visto ***satisfiedLine/3*** y asigna el valor obtenido a una nueva lista de pistas satisfechas que será devuelta para inicializarlas apenas comienza el juego.

```
initCluesAux([],_,[]).
%CB 1 sola linea por recorrer
initCluesAux([Line|[]],[ActualLineClues|[]],[IsSatisfied]):-
    satisfiedLine(ActualLineClues,Line,IsSatisfied).
%CR: más de 1 línea por recorrer
initCluesAux([Line|RestOfLine],[ActualLineClues|RestOfClues],[IsSatisfied|RestOfSatisfied]):-
    satisfiedLine(ActualLineClues,Line,IsSatisfied),
    initCluesAux(RestOfLine,RestOfClues,RestOfSatisfied).
```

Luego para la actualización de pistas ante cada movimiento del usuario extendimos la implementación de put como se explica a continuación:

put/8 (Extensión):

Este predicado cuenta con los siguientes parámetros: put(+Content, +Pos, +RowsClues, +ColsClues, +Grid, -NewGrid, -RowSat, -ColSat). El código que le agregamos primero recupera las pistas de la fila y columna correspondientes a la posición en la que se realizó el put, para el caso de la columna usamos la matriz traspuesta para así obtenerla más fácil, luego llamamos al predicado ***satisfiedLine/3*** que verifica si la línea(fila o columna) satisface sus pistas.

```
nth0(RowN, RowsClues, ActualRowClues),
%Chequeo si se cumplen las pistas de la fila actual
satisfiedLine(ActualRowClues,NewRow,RowSat),

transpose(NewGrid,TransposeNewGrid),
nth0(ColN,TransposeNewGrid,Col),
nth0(ColN, ColsClues, ActualColClues),
%Chequeo si se cumplen las pistas de la columna actual
satisfiedLine(ActualColClues,Col,ColSat).
```

satisfiedLine/3:

Estel predicado cuenta con los siguientes parámetros: satisfiedLine(+Clues, +Line, -IsSatisfied). Su propósito es verificar si la fila o columna pasada por parámetros cumple cada una de sus pistas asociadas. Devuelve 1 en caso de que las cumpla, 0 en caso contrario. Cuenta con un caso base que ocurre cuando no hay más pistas por verificar o solo hay una pista vacía, y un caso recursivo que ocurre cuando hay más de una. La estrategia es llamar al predicado ***satisfiedClue/4*** para comprobar si se verifica la pista, luego si quedan más pistas por verificar pasamos a la siguiente, y si no queda ninguna chequeamos que el resto de la línea esté vacía, lo que significa que no hay ninguna celda pintada que incumpla las pistas ya verificadas.

```
%IsSatisfied: 1 = TRUE, 0 = FALSE
%CB: Sin pistas(Mismo efecto que pista 0), me fijo que el resto de la línea esté sin pintar
satisfiedLine([],Line,IsSatisfied):-
    cleanLine(Line,IsSatisfied).

%CR: mas de 1 pista: chequeo pista actual y paso a la siguiente
satisfiedLine([HC|Clues],Line,IsSatisfied):-
    satisfiedClue(HC,Line,IsSatisfied,RestOfLine),
    satisfiedLine(Clues,RestOfLine,IsSatisfied).

%Entra en este caso y devuelve falso si no pudo satsifacer niguno de los anteriores
satisfiedLine(_,_,0).
```

cleanLine/3:

Comenzamos con ***cleanLine/3*** ya que es muy simple y no depende de otros predicados. Cuenta con los siguientes parámetros: cleanLine(+Line, -IsClean). Recorre la línea recibida por parámetros hasta llegar al final mientras no encuentre ninguna celda pintada. De lo contrario devuelve 0 (falso).

```
%Devuelve 1 si la línea está limpia (no tiene ninguna celda pintada), de lo contrario devuelve 0
cleanLine([],1).
cleanLine([H|_],0):- H == "#".
cleanLine([H|T],IsClean):-
    H \== "#",
    cleanLine(T,IsClean).
```

satisfiedClue/4:

Este predicado cuenta con los siguientes parámetros: satisfiedClue(+Clue, +Line, -IsSatisfied, -RestOfLine). Su propósito es verificar si una pista se satisface en una línea que recibe por parámetro. Tiene un caso base en el que si la pista es 0, devuelve verdadero. Luego para los casos generales la estrategia se basa en avanzar en la línea hasta llegar a una celda pintada y ahí llamar al predicado ***nConsecutive/3*** que chequea si hay N celdas consecutivas pintadas, siendo N el número de la pista. Un aspecto positivo de esta solución es que agregamos un caso en el que si el número de la pista es mayor a la longitud que queda por recorrer de la línea, directamente devolvemos que la pista no se satisface, lo que la hace más eficiente.

```
%Caso pista 0
satisfiedClue(0,Line,1,Line).
%La cantidad de celdas que faltan pintar es mayor a lo que queda por recorrer de la lista
satisfiedClue(N,Line,0,Line):-
    length(Line, Length),
    N > Length.

%Estoy en una celda sin pintar, sigo recorriendo
satisfiedClue(N,[H|T],IsSatisfied,RestOfLine):-
    H \== "#",
    satisfiedClue(N,T,IsSatisfied,RestOfLine).

%Se encontraron exáctamente N celdas pintadas consecutivas
satisfiedClue(N,Line,1,RestOfLine):-
    nConsecutive(N,Line,RestOfLine).
```

nConsecutive/3:

Este predicado tiene los siguientes parámetros: nConsecutive(+RequiredCells, +Line, -RestOfLine). Verifica que hayan exáctamente (ni más ni menos) N celdas consecutivas pintadas desde el inicio de la lista que recibe por parámetro.

```
%CB: No se necesitan pintar mas celdas.
nConsecutive(0,[],[]).
nConsecutive(0,[H|T],[H|T]):- H \== "#".
%CR
nConsecutive(N,[H|T],Ts):-
    H == "#",
    Nr is N -1,
    nConsecutive(Nr,T,Ts).
```

Esos fueron todos los predicados implicados en el chequeo de pistas de la fila y columnas correspondientes a la posición en la que se hizo el ***put/8***.

Detectar que se resolvió el nonograma: todas las líneas cumplen con las pistas:

Para esto utilizamos el predicado ***checkWin/3*** de la siguiente forma:

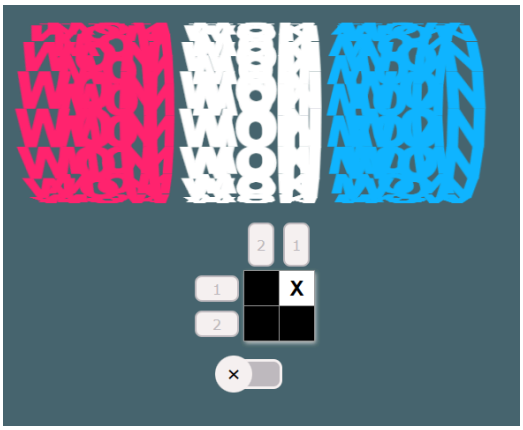
checkWin/3:

Este predicado cuenta con los siguientes parámetros: checkWin(+SatisfiedRowClues, +SatisfiedColClues, -IsAWin). Se utiliza para verificar si se ganó el nivel o se puede seguir jugando. Verifica si todas las pistas en las listas de pistas satisfechas lo están o no. En caso de que todas lo estén devuelve verdadero, y falso en caso contrario.

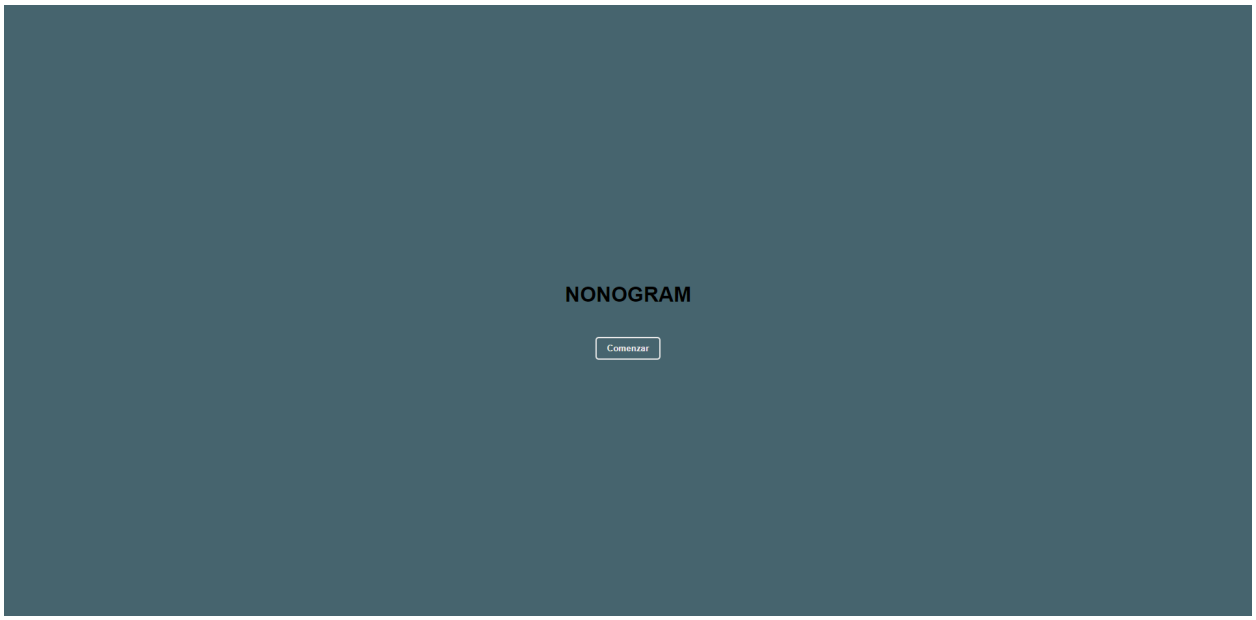
```
%Si todas las pistas de las filas y columnas son 1 (están satisfechas) estoy en condiciones de ganar
checkWin([],[],true).
checkWin(SatisfiedRowClues,SatisfiedColClues,true):-
    forall(member(RowClue,SatisfiedRowClues), 1 is RowClue),
    forall(member(ColClue,SatisfiedColClues), 1 is ColClue).
checkWin(_,_,false).
```

Funcionalidades extra:

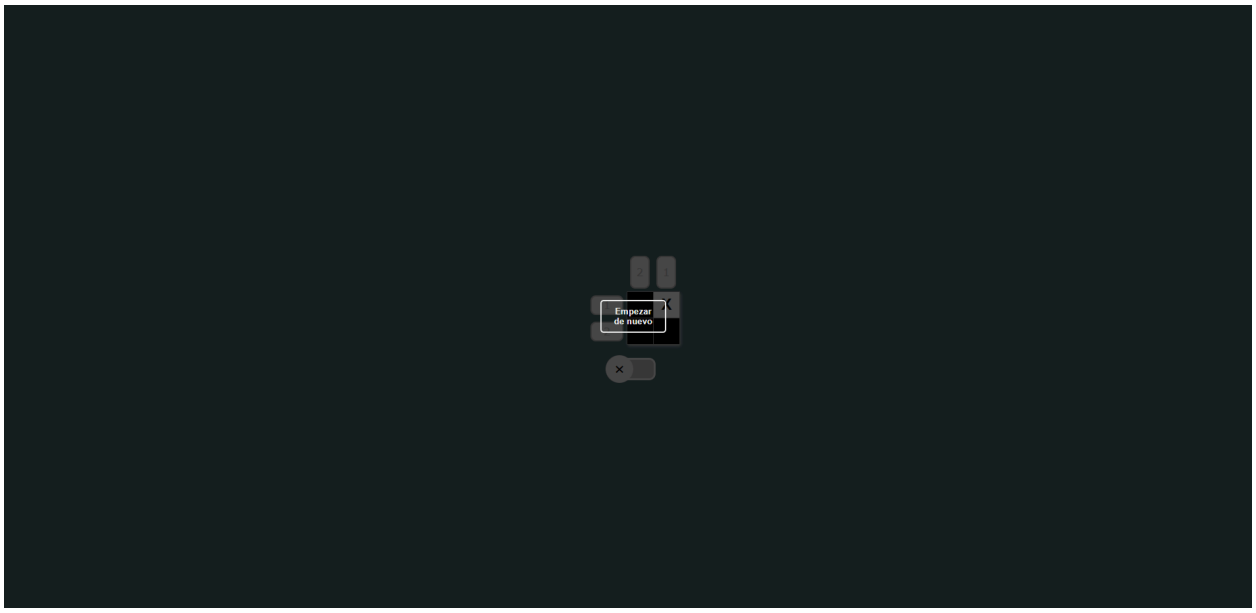
Animación al ganar el nivel:



Menú de inicio:



Botón para volver a jugar el nivel:



Desafíos superados:

Warning al tratar de inicializar las pistas: la idea era a través de un useEffect() que ejecutaba su código solo al principio del juego cuando el estado gameStauts se

seteaba en verdadero. Así si el juego inicializa las pistas, y hace el chequeo para ver si se ganó el nivel sin necesidad de que el usuario realice ningún click.

```
useEffect(() => {
  const handleEffect = () => {
    if (gameStatus) {
      const rowsCluesS = JSON.stringify(rowsClues);
      const colsCluesS = JSON.stringify(colsClues);
      const gridS = JSON.stringify(grid);
      const queryI = `initClues(${gridS},${rowsCluesS},${colsCluesS},NewSatisfiedRowClues,NewSatisfiedColClues)`;
      pengine.query(queryI, (success, response) => {
        if (success) {
          console.log(response['NewSatisfiedRowClues']);
          console.log(response['NewSatisfiedColClues']);
        }
      });
    }
  };

  handleEffect(); // Ejecutar la función al principio para no depender de las dependencias

  // No incluir ninguna dependencia en el array de dependencias
}, [gameStatus]); // Solo dependencia gameStatus
```

```
src\Game.js
  Line 18:21:  'setGameStatus' is assigned a value but never used                                                    no-unused-vars
  Line 79:6:   React Hook useEffect has missing dependencies: 'colsClues', 'grid', and 'rowsClues'. Either include them or remove the dependency array  react-hooks/exhaust
ive-deps

webpack compiled with 1 warning
```

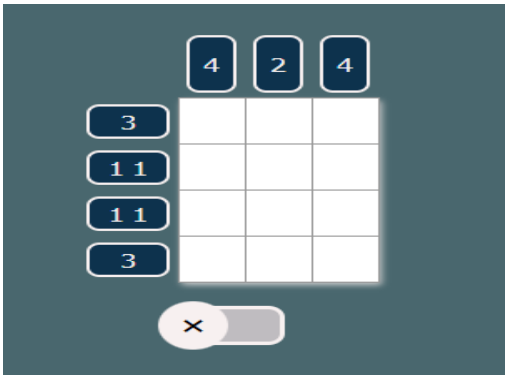
La solución que encontramos fue pedir los estados que necesitábamos usar para inicializar las pistas desde prolog y con eso realizar otra query para luego hacer un set de las pistas.

```
function handleServerReady(instance) {
  pengine = instance;
  const queryS = 'init(RowClues, ColumClues, Grid)';
  pengine.query(queryS, (success, response) => {
    if (success) {
      let gridS = response['Grid'];
      let rowCluesS = response['RowClues'];
      let colCluesS = response['ColumClues'];
      setGrid(gridS);
      setRowsClues(rowCluesS);
      setColsClues(colCluesS);
      setSatisfiedRowClues(Array(rowCluesS.length).fill(0));
      setSatisfiedColClues(Array(colCluesS.length).fill(0));
      //Modifico las variables para poder usarlas en la siguiente query
      gridS = JSON.stringify(response['Grid']);
      rowCluesS = JSON.stringify(response['RowClues']);
      colCluesS = JSON.stringify(response['ColumClues']);
      //Inicializo las listas que chequean si las pistas están satisfechas
      const queryI = `initClues(${gridS},${rowCluesS},${colCluesS},NewSatisfiedRowClues,NewSatisfiedColClues)`;
      pengine.query(queryI, (success, response) => {
        if (success) {
          setSatisfiedRowClues(response['NewSatisfiedRowClues']);
          setSatisfiedColClues(response['NewSatisfiedColClues']);
          //Borrar
          console.log("Row "+response['NewSatisfiedRowClues']);
          console.log("Col "+response['NewSatisfiedColClues']);
        }
      })

    }
  });
}
```

Casos de test significativos:

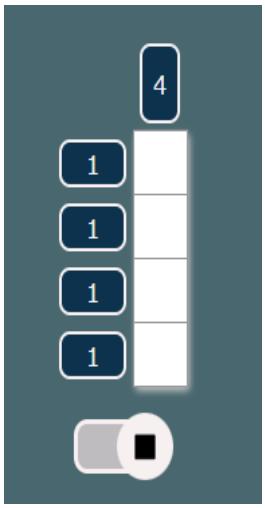
Matriz rectangular:

```
init(
[[3],[1,1],[1,1],[3]],  % RowsClues

[[4],[2],[4]],  % ColsClues

[[_,_,_],
 [_,_,_],
 [_,_,_],
 [_,_,_]
]
).
```

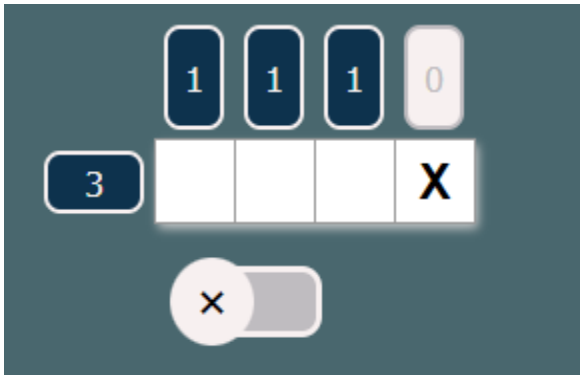


Matriz fila o columna:

```
init(
[[1],[1],[1],[1]],  % RowsClues

[[4]],  % ColsClues

[[_],
 [_],
 [_],
 [_]
]
).
```

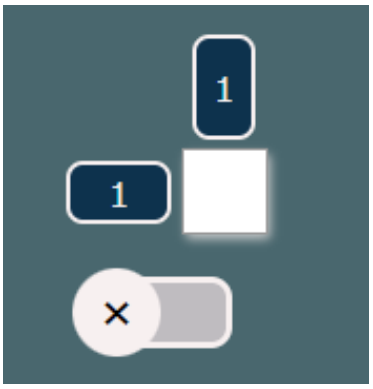



```
init(
[[3]],  % RowsClues

[[1],[1],[1]],  % ColsClues

[[_,_,_,"X"]
]
).
```



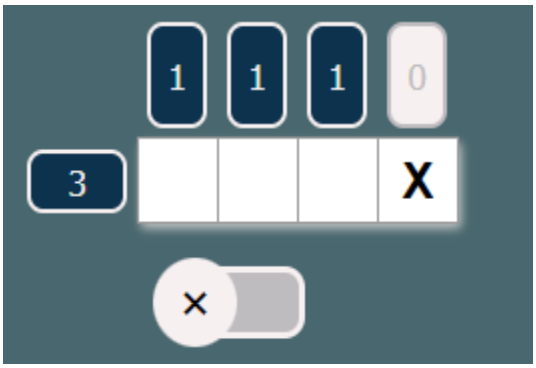
Matriz 1x1:

```
init(
[[1]],  % RowsClues

[[1]],  % ColsClues

[[_]
]
).
```



Pista 0:

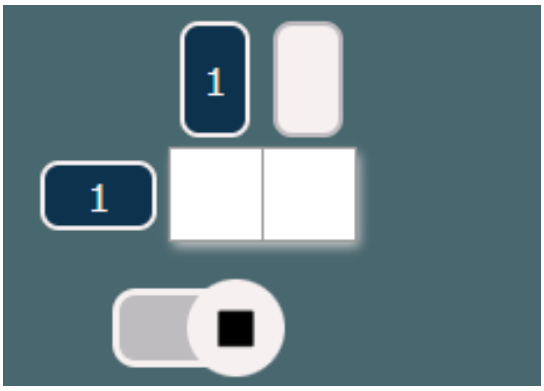
```
init(
[[3]],  % RowsClues

[[1],[1],[1],[0]],  % ColsClues

[[_,_,_,"X"]
]
).
```



Se satisface si no hay ninguna celda pintada en la fila o columna correspondiente a la pista.

Pista vacía:

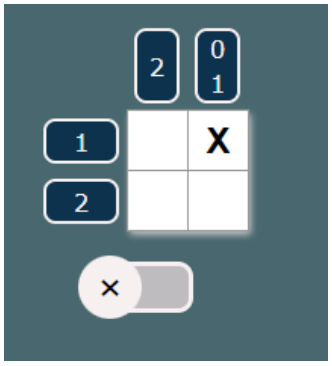
```
init(
[[1]],  % RowsClues

[[1],[]],   % ColsClues

[[_,"X"]
]
).
```

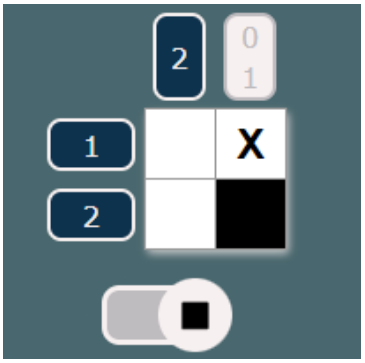


Se satisface si no hay ninguna celda pintada en la fila o columna correspondiente a la pista.

Fila o columna con más de 1 pista, dónde una de estas es 0:

```
init(
[[1],[2]],  % RowsClues

[[2],[0,1]],    % ColsClues

[[_,"X"],
 [_,"_"]
]
).
```

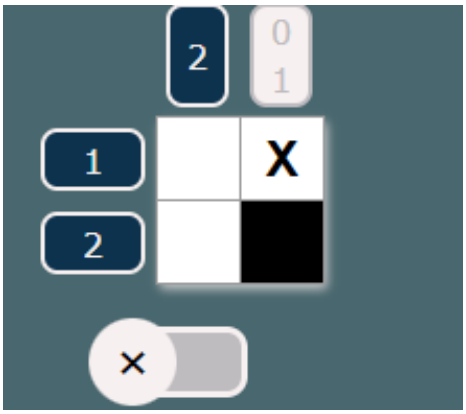




Tomamos la convención de que cuando una de las pistas es 0, se la “ignora”, ya que el predicado en prolog entra en un caso que devuelve verdadero y sigue chequeando las siguientes pistas desde la misma posición.

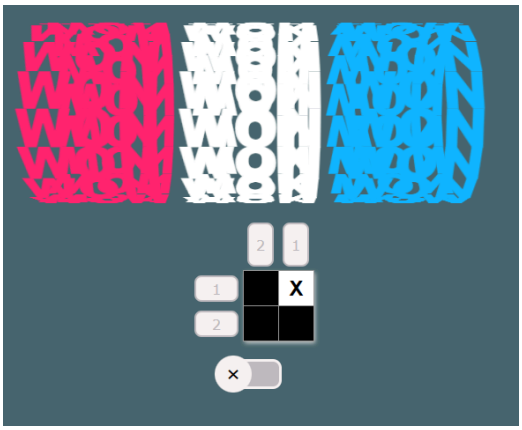
Juego comienza con pista satisfecha:

```
init(
[[1],[2]],  % RowsClues

[[2],[0,1]],    % ColsClues

[[_,"X"],
 [_,"#"]
]
).
```



Juego comienza con todas las pistas satisfechas:

```
init(
[[1],[2]],  % RowsClues

[[2],[0,1]],    % ColsClues

[[X,"X"],
 [X,"#"]
]
).
```



Automáticamente se gana el juego.

```
paintClue(0,[],[],[]).

paintClue(0,[H|T],T,[H]):- H \== "#".

paintClue(Clue,[H|T],RestOfLine,["#"|L]):-
Clue > 0,
H \== "X",
RC is Clue -1,
paintClue(RC,T,RestOfLine,L).
```