make.tol de Constraint.Queen

Solucionador del juego de las 8 reinas en un tablero de ajedrez como un sistema de restricciones con 3 componentes: a) restriccion que hay que cumplir, que es condicion logica, b) accion, que es una funcion correctora, c) reaccion, que es una funcion de backtrack frente a ciclos. Es un solucionador, programado de forma iterativa, desarrollado en un solo fichero Tol en el que se declaran: a) todas las funciones para las reinas, b) para sus restricciones de no estar ni en la misma fila ni en diagonal y c) que asume como punto de partida que cada reina esta en una columna diferente a las otras. Este metodo iterativo que: a) para guardar memoria del estado en curso utiliza la reasignacion := de Tol y b) ademas tiene una variable de memoria de texto, ver seccion blackboard, que conserva todos los estados del proceso de solucion. La memoria (QueMemory) permite detectar ciclos y ademas generar una traza para que un simulador Javascript reproduzca visualmente el proceso de resolucion en Tol. El contenido de la memoria a partir de un estado inicial INI podria ser el siguiente 11111111INIT| 12111111A2:2| 13111111A2:3| 13211111A3:2|... 13524111R6:1|... como resultado de aplicar acciones (A) y reacciones (R) que mueven una reina a una fila, por ejemplo, A2:3, es la accion de mover la reina 2 a la fila 3. Esta comprobado el funcionamiento de Constraint.Queen para las versiones de Tol 1.1.5, 1.1.6 y 2.0.1, pero funciona con la version 1.1.1. Una posible razon son los 5 usos de la reasignacion := que se realizadan en este programa Tol cuando lo usual, y recomendable, es ninguno.

Constraint.Queen es una version basica de programacion con restricciones, casi para uso formativo formativo, donde: a) Cada restriccion logica tiene una funcion correctora, que es siempre del mismo estilo, que consiste en que la reina que no cumple la restriccion avance un paso, esto es que suba a su fila superior. b) Los hechos son para cada reina son su posicion de fila y columna, si bien la columna no cambia en todo el proceso de resolucion. c) El dominio de los valores, de las posiciones 1 a 8, esta subsumido en la funcion que avanza a las reinas de fila 1 :- 2, 2 :- 3,..., 7 :- 8 y 8 :- 1. d) La restriccion, ademas de su accion correctora, incluye una funcion llamada de reaccion, que se ejecuta solo cuando se detecta que de aplicar la accion se entraria en un ciclo que se detecta porque se guarda memoria de todos los pasos dados en el proceso de resolucion. Notese que no es habitual encontrar este tipo de restricciones que incluyan una funcion correctora (accion) y una funcion equiparable a un backtrack (reaccion) como las que implementa el programa Constraint.Queen. Dentro de esta programacion se denomina reina esclava, slave queen, a la reina restringida por la condicion logica y reinas maestras, masters queen, a las que restringen. Para facilitar la declaracion de las restricciones cada reinas esclava es restringida por todas las que están a su izquierda, esto es, por sus reinas maestras.

El programa Constraint.Queen realiza 3 funciones principales: a) Resuelve el problema clasico con las 8 reinas en la primera fila como posicion de partida. b) Tambien se autoplantea problemas, generando distribuciones al azar de las 8 reinas en diferentes filas, aunque cada una en su columna, y busca una solucion a partir de esa distribucion al azar de las 8 reinas sobre el tablero de ajedrez. Con ello es capaz de encontrar las 92 posibles soluciones al problema, si bien en la literatura se consideran que son 12 las unicas, pues las 92 se pueden generar mediante rotaciones y simetrias de las 12 unicas. Ha de hacerse notar que desde posiciones al azar no todas las 92 soluciones parecen ocurrir de forma equiprobable, si bien, esto es solo una intuicion producto de la ejecucion reiterada de este solucionador. c) Finalmente, Constraint.Queen, es capaz de generar una traza en lenguaje

Javascript, en base a un array de pasos para cada caso resuelto y un array con todos los casos (hasta 92), para que un simulador programado en Javascript pueda reproducir el proceso de resolucion de cada caso desde una posicion al azar. Por lo que el numero de pasos de cada caso depende de la distancia de la posicion inicial al azar a la solucion alcanzable.

Árbol de ficheros

- Constraint.Queen solucionador de las 8 reinas en ajerez con restricciones y acciones
  - ← make.tol resuelve con restricciones y 2 clases de acciones las 8 reinas
  - ← make.bat mandato de ejecucion del programa de resolucion de las 8 reinas
  - simulator directorio del simulador del proceso de resolucion en Javascript
    - css directorio para css, Cascading Style Sheets, del simulador
      - ← simulator.css css para simular un tablero de ajedrez con 8 reinas
    - src directorio de codigo fuente Javascript del simulador de 8 reinas
      - ← simulator.js funciones para simular la resolucion del tablero y 8 reinas
      - → simulatorarray.js array metaprogramado con los 92 casos de solucion
    - solved directorio con las 92 posibles variantes del problema resueltas
      - → p46827135.js array con los pasos de resolucion de un caso con 8 reinas
  - → startlog.txt traza de solucion partiendo de las 8 reinas en la primera fila
  - → simulator.html simulador visual en Javascript de los 92 casos de las 8 reinas
  - → sim92chess.html las 92 soluciones juntas de 8 reinas en un tablero de ajedrez
  - → constraint_queen.pdf funciones del solucionador con restricciones de las 8 reinas

Declaraciones

Pizarra

- `Text QueMemory`

  Memoria de todos los estados de posiciones por los que han pasado las reinas, donde cada estado es una secuencia de 8 digitos, del 1 al 8, por las 8 filas del tablero de ajedrez, terminado por: a) un indicador de la regla de accion (A) o de reaccion (R) que se ha empleado para alcanzar ese estado b) el identificado de la reina que ha realizado el movimiento (digito del 1 al 8) y c) la fila en la que se ha posicionado dicha reina. La informacion de esta memoria permite detectar ciclos y generar trazas para que un simulador pueda reproducir el proceso de solucion.

- `Text QueLast`

  Memoria de la ultima reina movida 1 al 8.

Funciones de nombre corto

- `Text F(Real numDat)`

  Retorna un numero como un solo digito en texto.

- `Real R(Real maxDat)`
  Retorna un numero aleatorio enterio de 1 a maxDat.

Constantes

- `Set QueChess`
  Tablero con 8 reinas en sus posiciones iniciales todas en la primera fila. Notese que hay otras muchas posiciones de partida y, cada una de ellas, a diferentes numeros de pasos a una solucion. Se ha de tener en cuenta que ademas, existen diferentes soluciones a este problema de las 8 reinas en el tablero de ajedrez, en concreto 92.
- `Set QueConstraint`
  Conjunto de las ternas: a) restriccion de fila o de diagonal que ha de cumplir cada reina, cada una con las de su izquierda y b) la accion correctora en caso de que no se cumpla la restriccion y c) la reaccion para cuando por la aplicacion sistematica de la accion se entra en ciclo, puede considerarse una funcion de backtrack.
- `Real QueDim`
  Dimension del problema, usualmente 8.
- `Text QueIni`
  Mandato de inicializacion de la memoria.
- `Text QueSep`
  Separador de pasos en la memoria de estados.
- `Text QueSim`
  Directorio de traza para el simulador en Javascript.

Funciones

- `Real NotRow(Set tabSet, Real posSla, Set setMas)`
  Retorna cierto si la reina esclava posSla no esta en linea, misma fila, con ninguna de las reinas maestras, conjunto setMas, que la restringen dentro del tablero tabSet, que es la base de hechos.
- `Real NotDiag(Set tabSet, Real posSla, Set setMas)`
  Retorna cierto si la reina esclava posSla no forma diagonal con ninguna de las reinas maestras, conjunto setMas, que la restringen dentro del tablero tabSet, que es la base de hechos.
- `Set RowMove(Set tabSet, Real posQue)`
  Retorna un nuevo tablero, identico al tablero tabSet de entrada, pero con su reina de posicion posQue adelantada a la siguiente fila. Si la reina estaba ya en la ultima fila entonces la pasa a la primera fila.
- `Text QueStatus(Set tabSet)`
  Retorna un texto de 8 digitos del 1 al 8 que describe la disposicion de las reinas en el tablero segun tabSet, esto es, un determinado estado de la base de hechos. Posteriormente, a este texto se le añadira una terminacion para separarlo de otros estados y poder hacer busquedas rapidas con TextFind() sin preocuparse de que existan coincidencias de trozos parciales, finales e iniciales, de 2 estados consecutivos.
- `Real QueWrite(Set tabSet, Text actRec)`

Añade a la memoria las posiciones del tablero que son la base de hechos. Como efecto lateral saca una traza por pantalla del estado.

- `Real QueExist(Set tabSet)`
  Retorna cierto si ya se ha pasado por el estado descrito por tabSet, para ello lo busca en la memoria de estados previos. Es tan simple como usar la funcion TextFind() pues cada estado esta separado del anterior y del siguiente por el codigo de las funciones de accion o reaccion y un caracter separador.

- `Real QueTest(Set tabSet, Set conObj)`
  Retorna TRUE si en el tablero tabSet, base de hechos base, se cumple la restriccion conObj. Una restriccion conObj es una estructura [restriccion, accion, reaccion] y se extrae la restriccion y se comprueba. Esta funcion podria haberse programado en 1 sola linea, pero se ha realizado en 2 para ver cual es el codigo y su aplicacion.

- `Set QueChange(Set tabSet, Set conObj)`
  Retorna un nuevo tablero, una nueva base de hechos, resultado de aplicar la parte de la accion de la restriccion cobObj. Pero si el resultado de aplicar la accion es un estado por el que ya se ha pasado anteriormente, entonces en vez de aplicar la funcion de accion aplica la funcion de reaccion, backtrack, para no entrar en ciclo. Dependiendo del paso dado, accion o reaccion, guarda el estado, el tipo de paso dado, la reina afectada y la fila alcanzada en la memoria.

- `Set QueSolve(Set tabSet, Set conSet)`
  Ciclo iterativo del motor de aplicacion de restricciones y de resolucion del problema de las 8 reinas en un tablero. Retorna la base de hechos resuelta, esto es el tablero de ajedrez resuelto. El algoritmo es que mientras no se consiga dar una vuelta completa, While(), a todo el sistema de restricciones, conSet, sin que en el estado actual, tabNow, alguna reina incumpla alguna restriccion, se toma la primera restriccion incumplida, se aplican sus funciones correctoras, bien de accion o de reaccion, de ello se encarga QueChange() y se vuelven a comprobar todas las restricciones desde el principio, Real(conCnt:= 1). Dada la primera vuelta completa al sistema de restricciones sin incumplimientos, el estado actual, tabNow, es la solucion.

- `Real QueTrace(Text solMem)`
  Retorna el numero de pasos para alcanzar esta solucion y escribe en un fichero Javascript un array con todos los pasos y acciones y reacciones realizadas para alcanzarla. Solo guarda en un subdirectorio del directorio QueSim las soluciones: a) que son nuevas al problema de las 8 reinas en el tablero de ajedrez esto es, que su fichero de pasos de solucion no existe, o b) que es mas corta en pasos que la solucion ya existente, esto es, que ocupa menos bytes. El nombre de los ficheros esta formado por una letra p mas los 8 digitos de las filas en las que esta cada reina, el orden de estos digitos es de la columna 1 a la columna 8.

- `Real QueJavascript(Text dirSol, Text dirSrc)`

Reune todos los problemas de las 8 reinas en un tablero de ajedrez resueltos, como maximo son 92, y los prepara para que su resolucion pueda ser simulada por un simulador visual Javascript. Genera 2 ficheros en lenguaje Javascript: a) un fichero indice con un array que contiene todos los problemas resueltos ordenado de mas a menos pasos, para que los mas largos esten al principio y b) un fichero mas grande, con tantos arrays como casos resueltos, cada uno de estos arrays contiene todos los pasos de resolucion del caso, el orden de declaracion de estos arrays puede ser cualquiera, pero se emplea el anterior, de mas a menos pasos de resolucion. El nombre de los arrays esta formado por una letra p mas los 8 digitos de las filas en las que esta cada reina, el orden de estos digitos es de la columna 1 a la columna 8. Para la generacion de estos 2 ficheros emplea 2 ficheros semilla que tienen la cabecera de comentarios de los programas Javascript. Recibe como parametros a) el directorio de casos resueltos de entrada, que han sido generados por la funcion QueTrace() y b) el directorio de salida donde se escribira el codigo Javascript, asume que los ficheros semilla Javascript estan en este mismo directorio y los nombres de los ficheros para el array indice y el banco de datos son fijos. Retorna el numero de problemas resueltos que en el caso de las 8 reinas sin atacarse en un tablero de ajedrez son 92 posibilidades como máximo, si bien, de esas 92, unas puede obtenerse de otras mediante rotaciones y simetrias.

Proceso

- `Real solCla`

  Resolver el problema clasico de las 8 reinas.

- `Real solRnd`

  Resuelve varios problemas con las 8 reinas al azar.

- `Real solJsc`

  Prepara los ficheros para Javascript con los casos.

Text QueMemory

```
////////////////////////////////////////////////////////////////////////
Text QueMemory = ""; // Inicialmente vacia
////////////////////////////////////////////////////////////////////////
PutDescription(
"Memoria de todos los estados de posiciones por los que han pasado las reinas,
donde cada estado es una secuencia de 8 digitos, del 1 al 8, por las 8 filas
del tablero de ajedrez, terminado por:
a) un indicador de la regla de accion (A) o de reaccion (R) que se ha empleado
   para alcanzar ese estado
b) el identificado de la reina que ha realizado el movimiento (digito del 1 al
   8) y
c) la fila en la que se ha posicionado dicha reina.
La informacion de esta memoria permite detectar ciclos y generar trazas para
que un simulador pueda reproducir el proceso de solucion.",
QueMemory);
////////////////////////////////////////////////////////////////////////
```

Text QueLast

```
////////////////////////////////////////////////////////////////////////
Text QueLast = ""; // Inicialmente vacia
////////////////////////////////////////////////////////////////////////
PutDescription("Memoria de la ultima reina movida 1 al 8.",QueLast);
////////////////////////////////////////////////////////////////////////
```

Estructuras de datos

```
Struct Queen // Estructura para las reinas(col, row), aunque con row bastaria
{
  Real col, // Columna del 1 al 8
  Real row  // Fila del 1 al 8
};


Struct Constraint // Estructura para las restricciones
{
  Code con, // Restriccion a comprobar / constraint
  Code act, // Accion hacia adelante / action
  Code bck  // Reaccion hacia atras al ser inutil la accion / backtrack
};
```

Funciones de nombre corto

Text F()

```
////////////////////////////////////////////////////////////////////////////
Text F(Real numDat) // Un numero entero de 1 digito
////////////////////////////////////////////////////////////////////////////
{ FormatReal(numDat, "%1.0lf") };
////////////////////////////////////////////////////////////////////////////
PutDescription("Retorna un numero como un solo digito en texto.",F);
////////////////////////////////////////////////////////////////////////////
```

Real R()

```
////////////////////////////////////////////////////////////////////////////
Real R(Real maxDat) // Aleatorio entero de 1 a maxDat
////////////////////////////////////////////////////////////////////////////
{ 1+Floor(maxDat * Rand(0,1)) };
////////////////////////////////////////////////////////////////////////////
PutDescription("Retorna un numero aleatorio enterio de 1 a maxDat.",R);
////////////////////////////////////////////////////////////////////////////
```

Constantes

Set QueChess

```
////////////////////////////////////////////////////////////////////////////
Set QueChess =  // Tablero, base de hechos, 8 reinas [columna, fila]
[[ Queen(1, 1), // es el inicio mas clasico, con las 8 reinas en la fila 1
   Queen(2, 1), // En este programa resuelve otros tableros de ajedrez con
   Queen(3, 1), // sus 8 reinas puestas al azar
   Queen(4, 1),
   Queen(5, 1),
   Queen(6, 1),
   Queen(7, 1),
   Queen(8, 1)
]];
////////////////////////////////////////////////////////////////////////////
```

```
PutDescription(
"Tablero con 8 reinas en sus posiciones iniciales todas en la primera fila.
Notese que hay otras muchas posiciones de partida y, cada una de ellas,
a diferentes numeros de pasos a una solucion.
Se ha de tener en cuenta que ademas, existen diferentes soluciones a este
problema de las 8 reinas en el tablero de ajedrez, en concreto 92.",
QueChess);
////////////////////////////////////////////////////////////////////////
```

Set QueConstraint

```
////////////////////////////////////////////////////////////////////////
Set QueConstraint = // Restricciones a cumplir del tablero tab de 8 reinas
[[
  Constraint( // La reina 2 no puede estar en la misma fila que la 1
    Real(Set tab) { NotRow (tab, 2, [[1]]) },
    Set (Set tab) { RowMove(tab, 2) },  // si no se cumple la reina 2 se mueve
    Set (Set tab) { RowMove(tab, 1) }), // otra opcion

  Constraint( // La reina 2 no puede estar en diagonal con la 1
    Real(Set tab) { NotDiag(tab, 2, [[1]]) },
    Set (Set tab) { RowMove(tab, 2) },  // si no se cumple la reina 2 se mueve
    Set (Set tab) { RowMove(tab, 1) }), // otra opcion

  Constraint( // La reina 3 no puede estar en la misma fila que la 1 o la 2
    Real(Set tab) { NotRow (tab, 3, [[1,2]]) },
    Set (Set tab) { RowMove(tab, 3) },  // si no se cumple la reina 3 se mueve
    Set (Set tab) { RowMove(tab, 2) }), // otra opcion

  Constraint( // La reina 3 no puede estar en diagonal ni con la 1, ni la 2
    Real(Set tab) { NotDiag(tab, 3, [[1,2]]) },
    Set (Set tab) { RowMove(tab, 3) },  // si no se cumple la reina 3 se mueve
    Set (Set tab) { RowMove(tab, 2) }), // otra opcion

  Constraint( // La reina 4 no puede estar en la misma fila que la 1, la 2 o 3
    Real(Set tab) { NotRow (tab, 4, [[1,2,3]]) },
    Set (Set tab) { RowMove(tab, 4) },  // si no se cumple la reina 4 se mueve
    Set (Set tab) { RowMove(tab, 3) }), // otra opcion

  Constraint( // La reina 4 no puede estar en diagonal con las reinas 1, 2 o 3
    Real(Set tab) { NotDiag(tab, 4, [[1,2,3]]) },
    Set (Set tab) { RowMove(tab, 4) },  // si no se cumple la reina 4 se mueve
    Set (Set tab) { RowMove(tab, 3) }), // otra opcion

  Constraint( // La reina 5 no puede estar en la misma fila que la 1, 2, 3 o 4
    Real(Set tab) { NotRow (tab, 5, [[1,2,3,4]]) },
    Set (Set tab) { RowMove(tab, 5) },  // si no se cumple la reina 5 se mueve
    Set (Set tab) { RowMove(tab, 4) }), // otra opcion

  Constraint( // La reina 5 no puede estar en diagonal con la 1, 2, 3 o 4
    Real(Set tab) { NotDiag(tab, 5, [[1,2,3,4]]) },
    Set (Set tab) { RowMove(tab, 5) },  // si no se cumple la reina 5 se mueve
    Set (Set tab) { RowMove(tab, 4) }), // otra opcion

  Constraint( // La reina 6 no puede estar en la misma fila que la 1,2,3,4 o 5
    Real(Set tab) { NotRow (tab, 6, [[1,2,3,4,5]]) },
    Set (Set tab) { RowMove(tab, 6) },  // si no se cumple la reina 6 se mueve
    Set (Set tab) { RowMove(tab, 5) }), // otra opcion

  Constraint( // La reina 6 no puede estar en diagonal con la 1, 2, 3, 4 o 5
    Real(Set tab) { NotDiag(tab, 6, [[1,2,3,4,5]]) },
    Set (Set tab) { RowMove(tab, 6) },  // si no se cumple la reina 6 se mueve
    Set (Set tab) { RowMove(tab, 5) }), // otra opcion

  Constraint( // La reina 7 no puede estar en la misma fila que la 1,2,3,4,5,6
    Real(Set tab) { NotRow (tab, 7, [[1,2,3,4,5,6]]) },
    Set (Set tab) { RowMove(tab, 7) },  // si no se cumple la reina 7 se mueve
    Set (Set tab) { RowMove(tab, 6) }), // otra opcion

  Constraint( // La reina 7 no puede estar en diagonal con la 1, 2, 3, 4, 5 o
    Real(Set tab) { NotDiag(tab, 7, [[1,2,3,4,5,6]]) },
    Set (Set tab) { RowMove(tab, 7) },  // si no se cumple la reina 7 se mueve
```

```
    Set (Set tab) { RowMove(tab, 6) }), // otra opcion

  Constraint( // La reina 8 no puede estar en fila con la 1, 2, 3, 4, 5, 6 o 7
    Real(Set tab) { NotRow (tab, 8, [[1,2,3,4,5,6,7]]) },
    Set (Set tab) { RowMove(tab, 8) },  // si no se cumple la reina 8 se mueve
    Set (Set tab) { RowMove(tab, 7) }), // otra opcion

  Constraint( // La reina 8 no puede estar en diagonal con la 1,2,3,4,5,6 o 7
    Real(Set tab) { NotDiag(tab, 8, [[1,2,3,4,5,6,7]]) },
    Set (Set tab) { RowMove(tab, 8) },  // si no se cumple la reina 8 se mueve
    Set (Set tab) { RowMove(tab, 7) })  // otra opcion
]];
////////////////////////////////////////////////////////////////////////
PutDescription(
"Conjunto de las ternas:
a) restriccion de fila o de diagonal que ha de cumplir cada reina,
   cada una con las de su izquierda y
b) la accion correctora en caso de que no se cumpla la restriccion y
c) la reaccion para cuando por la aplicacion sistematica de la accion se
   entra en ciclo, puede considerarse una funcion de backtrack.",
QueConstraint);
////////////////////////////////////////////////////////////////////////
```

Real QueDim

```
////////////////////////////////////////////////////////////////////////
Real QueDim = Card(QueChess);
////////////////////////////////////////////////////////////////////////
PutDescription("Dimension del problema, usualmente 8.", QueDim);
////////////////////////////////////////////////////////////////////////
```

Text QueIni

```
////////////////////////////////////////////////////////////////////////
Text QueIni = "INIT";
////////////////////////////////////////////////////////////////////////
PutDescription("Mandato de inicializacion de la memoria.", QueIni);
////////////////////////////////////////////////////////////////////////
```

Text QueSep

```
////////////////////////////////////////////////////////////////////////
Text QueSep = "|";
////////////////////////////////////////////////////////////////////////
PutDescription("Separador de pasos en la memoria de estados.", QueSep);
////////////////////////////////////////////////////////////////////////
```

Text QueSim

```
////////////////////////////////////////////////////////////////////////
Text QueSim = "simulator";
////////////////////////////////////////////////////////////////////////
PutDescription("Directorio de traza para el simulador en Javascript.",QueSim);
////////////////////////////////////////////////////////////////////////
```

Real NotRow()

```
////////////////////////////////////////////////////////////////////////
Real NotRow(Set  tabSet, // Tablero, conjunto de reinas
```

```
            Real posSla, // Identificador de la reina esclava
            Set  setMas) // Conjunto de reinas maestras
////////////////////////////////////////////////////////////////////////////
{
  Set queSla = tabSet[posSla]; // Extraer la reina esclava

  Set tstSet = EvalSet(setMas, Real(Real posMas) // Para toda reina maestra
  {
    Set queMas = tabSet[posMas]; // Extraer la reina maestra
                                 // Comprobar que no estan en linea
    queMas->row != queSla->row   // Cierto si la fila no es la misma
  });
  SetSum(tstSet) == Card(tstSet) // Cierto si se cumplen todas
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna cierto si la reina esclava posSla no esta en linea, misma fila, con
ninguna de las reinas maestras, conjunto setMas, que la restringen dentro del
tablero tabSet, que es la base de hechos.",
NotRow);
////////////////////////////////////////////////////////////////////////////
```

Real NotDiag()

```
////////////////////////////////////////////////////////////////////////////
Real NotDiag(Set  tabSet, // Tablero, conjunto de reinas
             Real posSla, // Identificador de la reina esclava
             Set  setMas) // Conjunto de reinas maestras
////////////////////////////////////////////////////////////////////////////
{
  Set queSla = tabSet[posSla]; // Extraer la reina esclava

  Set tstSet = EvalSet(setMas, Real(Real posMas) // Para toda reina maestra
  {
    Set queMas = tabSet[posMas]; // Extraer la reina maestra
                                 // Comprobar que no estan en diagonal
    Abs(queMas->col - queSla->col) != Abs(queMas->row - queSla->row)
  });
  SetSum(tstSet) == Card(tstSet) // Cierto si se cumplen todas
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna cierto si la reina esclava posSla no forma diagonal con ninguna de
las reinas maestras, conjunto setMas, que la restringen dentro del tablero
tabSet, que es la base de hechos.",
NotDiag);
////////////////////////////////////////////////////////////////////////////
```

Set RowMove()

```
////////////////////////////////////////////////////////////////////////////
Set RowMove(Set  tabSet, // Tablero, conjunto de reinas
            Real posQue) // Identificador de una reina esclava
////////////////////////////////////////////////////////////////////////////
{
  Real tabCrd = Card(tabSet); // En un tablero normal es siempre 8

  For(1, tabCrd, Set(Real posTab) // Para todas las posiciones del tablero
  {
    Set  queObj = tabSet[posTab]; // Extraer la reina
    If(posTab != posQue, queObj,  // No hay que moverla
    {
      Real newRow = If(queObj->row == tabCrd, 1,            // Avance circular
                                              queObj->row+1); // Avance 1 fila
      Text(QueLast:=F(queObj->col)+":"+F(newRow));     // Recuerda para trazar
      Queen(queObj->col, newRow) // La reina en la nueva fila
```

```
    })
  })
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna un nuevo tablero, identico al tablero tabSet de entrada, pero con su
reina de posicion posQue adelantada a la siguiente fila.
Si la reina estaba ya en la ultima fila entonces la pasa a la primera fila.",
RowMove);
////////////////////////////////////////////////////////////////////////////
```

Text QueStatus()

```
////////////////////////////////////////////////////////////////////////////
Text QueStatus(Set  tabSet) // Tablero, conjunto de reinas
////////////////////////////////////////////////////////////////////////////
{
  SetSum(For(1, Card(tabSet), Text(Real tabPos)
         { F(tabSet[tabPos]->row) }))
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna un texto de 8 digitos del 1 al 8 que describe la disposicion de las
reinas en el tablero segun tabSet, esto es, un determinado estado de la base
de hechos.
Posteriormente, a este texto se le añadira una terminacion para separarlo de
otros estados y poder hacer busquedas rapidas con TextFind() sin preocuparse
de que existan coincidencias de trozos parciales, finales e iniciales, de
2 estados consecutivos.",
QueStatus);
////////////////////////////////////////////////////////////////////////////
```

Real QueWrite()

```
////////////////////////////////////////////////////////////////////////////
Real QueWrite(Set  tabSet, // Tablero, conjunto de reinas
              Text actRec) // Accion o reaccion realizada
////////////////////////////////////////////////////////////////////////////
{
  Text staTxt = QueStatus(tabSet)+actRec+QueSep;
  Text memNew = If(actRec == QueIni, staTxt,           // Es un nuevo problema
                                     QueMemory+staTxt); // Proceso de solucion
  Text(QueMemory:= memNew);

  Text linSep = If(actRec != QueIni, "",                // Proceso de solucion
                             Repeat("_",78)+"\n");      // Es un nuevo problema
  Text WriteLn(linSep+staTxt);

  TRUE
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Añade a la memoria las posiciones del tablero que son la base de hechos.
Como efecto lateral saca una traza por pantalla del estado.",
QueWrite);
////////////////////////////////////////////////////////////////////////////
```

Real QueExist()

```
////////////////////////////////////////////////////////////////////////////
Real QueExist(Set  tabSet) // Tablero, conjunto de reinas
////////////////////////////////////////////////////////////////////////////
{ TextFind(QueMemory, QueStatus(tabSet)) };
////////////////////////////////////////////////////////////////////////////
```

```
PutDescription(
"Retorna cierto si ya se ha pasado por el estado descrito por tabSet,
para ello lo busca en la memoria de estados previos.
Es tan simple como usar la funcion TextFind() pues cada estado esta separado
del anterior y del siguiente por el codigo de las funciones de accion o
reaccion y un caracter separador.",
QueExist);
////////////////////////////////////////////////////////////////////////
```

Real QueTest()

```
////////////////////////////////////////////////////////////////////////
Real QueTest(Set tabSet, // Tablero, conjunto de reinas
             Set conObj) // Una restriccion concreta [restriccion, accion]
////////////////////////////////////////////////////////////////////////
{
  Code conCod = conObj->con; // Extraer la restriccion
  conCod(tabSet)             // Comprobar si el tablero cumple la restriccion
};
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna TRUE si en el tablero tabSet, base de hechos base, se cumple la
restriccion conObj.
Una restriccion conObj es una estructura [restriccion, accion, reaccion] y
se extrae la restriccion y se comprueba.
Esta funcion podria haberse programado en 1 sola linea,
pero se ha realizado en 2 para ver cual es el codigo y su aplicacion.",
QueTest);
////////////////////////////////////////////////////////////////////////
```

Set QueChange()

```
////////////////////////////////////////////////////////////////////////
Set QueChange(Set tabSet, // Tablero, conjunto de reinas
              Set conObj) // Una restriccion concreta [restriccion, accion]
////////////////////////////////////////////////////////////////////////
{
  Code actCod = conObj->act;    // Extraer la accion
  Set  tabAct = actCod(tabSet); // Aplicar la accion

  If(! QueExist(tabAct),
  {
    Real QueWrite(tabAct,"A"+QueLast); // Nuevo estado + accion + movimiento
    tabAct                             // No hay ciclo proseguir con la accion
  },
  {
    Real QueWrite(tabAct,"R"+QueLast); // Nuevo estado + reaccion + movimiento
    Code bckCod = conObj->bck;         // Si ha ciclo -> aplicar reaccion
    bckCod(tabSet)
  })
};
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna un nuevo tablero, una nueva base de hechos, resultado de aplicar la
parte de la accion de la restriccion cobObj.
Pero si el resultado de aplicar la accion es un estado por el que ya se ha
pasado anteriormente, entonces en vez de aplicar la funcion de accion
aplica la funcion de reaccion, backtrack, para no entrar en ciclo.
Dependiendo del paso dado, accion o reaccion, guarda el estado, el tipo de
paso dado, la reina afectada y la fila alcanzada en la memoria.",
QueChange);
////////////////////////////////////////////////////////////////////////
```

Set QueSolve()

```
////////////////////////////////////////////////////////////////////////
```

```
Set QueSolve(Set tabSet, // Tablero, conjunto de reinas
             Set conSet) // Restricciones
////////////////////////////////////////////////////////////////////////////
{
  Real conCnt = 1;            // Contador de restricciones
  Real conTot = Card(conSet); // Total de restricciones
  Set  tabNow = Copy(tabSet); // Memoria local del estado

  Real QueWrite(tabSet, QueIni); // Inicia la memoria y guarda el problema

  Real While(conCnt <= conTot,
  {
    If(! QueTest(tabNow, conSet[conCnt]),
    {
      Set  tabNew = QueChange(tabNow, conSet[conCnt]); // Aplica la accion
      Set (tabNow:= Copy(tabNew));
      Real(conCnt:= 1);
      TRUE  // Con cambios
    },
    {
      Real(conCnt:= conCnt + 1);
      FALSE // Sin cambios
    })
  });

  Copy(tabNow) // Retorna la ultima solucion
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Ciclo iterativo del motor de aplicacion de restricciones y de resolucion
del problema de las 8 reinas en un tablero.
Retorna la base de hechos resuelta, esto es el tablero de ajedrez resuelto.
El algoritmo es que mientras
no se consiga dar una vuelta completa, While(),
a todo el sistema de restricciones, conSet,
sin que en el estado actual, tabNow, alguna reina incumpla alguna restriccion,
se toma la primera restriccion incumplida,
se aplican sus funciones correctoras, bien de accion o de reaccion,
de ello se encarga QueChange() y
se vuelven a comprobar todas las restricciones desde el principio,
Real(conCnt:= 1).
Dada la primera vuelta completa al sistema de restricciones sin
incumplimientos, el estado actual, tabNow, es la solucion.",
QueSolve);
////////////////////////////////////////////////////////////////////////////
```

Real QueTrace()

```
////////////////////////////////////////////////////////////////////////////
Real QueTrace(Text solMem) // Memoria del proceso de solucion
////////////////////////////////////////////////////////////////////////////
{
  Text iniTxt = ""+Char(34);     // Texto inicial sin margen
  Text midTxt = Char(34)+",\n"; // Texto final para pasos intermedios
  Text endTxt = Char(34);        // Texto final para el ultimo paso

  Set  rowRng = Range(QueDim, 1, -1);    // Filas de arriba a abajo
  Set  colRng = Range(1, QueDim,  1);    // Columnas de izquierda a derecha

  Text celCol(Real rowNum, Real colNum)  // Retorna el color de la casilla
  { If((rowNum+colNum) % 2, "·", "X") }; // Impar blanca ·, par negra X

  Text queBck(Text celTxt)               // Reina sobre blanco · -> O
  { If(celTxt == "·", "O", "Q") };       // Reina sobre negro  X -> Q

  Set  stpSet = Select(Tokenizer(solMem, QueSep), // Extrae todo los pasos
                       Real(Text stpTxt) { Compact(stpTxt) != "" });

  Real stpCrd = Card(stpSet); // Numero total de pasos

  Text stpFun(Text stpTxt, Real stpEnd)          // Retorna el texto de 1 paso
  {                                   // Añade , si intermedio y no si final
```

```
    Set  rowCic = EvalSet(rowRng, Text(Real rowNum)    // Para todas las filas
    {
      Set  colCic = EvalSet(colRng, Text(Real colNum)     // Para toda columna
      {
        Text celTxt = celCol(rowNum, colNum);          // Celda blanca o negra
        Real celQue = Text Sub(stpTxt, colNum, colNum)==F(rowNum); // Si reina
        If(celQue, queBck(celTxt), celTxt)        // Poner una reina si la hay
      });
      SetSum(colCic)+QueSep
    });     // Todo el tablero de ajedrez + 2 caracteres de la accion/reaccion
    Text actRec = Sub(stpTxt, QueDim+1, QueDim+4);  // Traza accion o reaccion
    iniTxt+SetSum(rowCic)+actRec+If(stpEnd, endTxt, midTxt) // Ajedrez + traza
  };

  Set  stpCic = For(1, stpCrd, Text(Real stpNum)    // Para todos los pasos
  { stpFun(stpSet[stpNum], stpCrd==stpNum) });      // el ultimo es diferente

  Text solNam = "p"+Sub(stpSet[stpCrd], 1, QueDim); // nombre del array

  Text scrArr = "var "+solNam     +    // Declara el array
                " = new Array(\n" +    // Abre un nuevo array
                SetSum(stpCic)    +    // Todos los pasos del primero al ultim
                ");\n";                // Cerrar el array

  Text filPth = QueSim+"/solved/"+solNam+".js"; // Ruta del fichero de salida

  Real If(And(FileExist(filPth),
              FileBytes(filPth) < TextLength(scrArr)), FALSE, // No escribe
  { // Solo guarda soluciones nuevas o mas corta que la ya existente
    Text WriteFile(filPth, scrArr); // Escribe el array en un fichero
    TRUE
  })
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna el numero de pasos para alcanzar esta solucion y escribe en un
fichero Javascript un array con todos los pasos y acciones y reacciones
realizadas para alcanzarla.
Solo guarda en un subdirectorio del directorio QueSim las soluciones:
a) que son nuevas al problema de las 8 reinas en el tablero de ajedrez
   esto es, que su fichero de pasos de solucion no existe, o
b) que es mas corta en pasos que la solucion ya existente,
   esto es, que ocupa menos bytes.
El nombre de los ficheros esta formado por una letra p mas los 8 digitos de
las filas en las que esta cada reina, el orden de estos digitos es de la
columna 1 a la columna 8.",
QueTrace);
////////////////////////////////////////////////////////////////////////////
```

Real QueJavascript()

```
////////////////////////////////////////////////////////////////////////////
Real QueJavascript(Text dirSol, // Directorio de problemas resueltos
                   Text dirSrc) // Directorio de Javascript
////////////////////////////////////////////////////////////////////////////
{
  Text dirSol = QueSim+"/solved"; // Directorio de problemas resueltos
  Text dirSrc = QueSim+"/src";    // Directorio de Javascript

  Set  filSet = GetDir(dirSol)[1]; // Ficheros de casos resueltos
  Set  filTab = EvalSet(filSet, Set(Text filNam)
  {
    Text filPth = dirSol+"/"+filNam;  // Ruta completa
    Real filByt = FileBytes(filPth);  // Numero de bytes
    [[ filPth, filByt ]]              // Par ruta y tamaño
  });

  // Ordenar los mas grandes, con mas bytes, al principio
  Set  filSor = Sort(filTab, Real(Set a, Set b) { Compare(b[2],a[2]) });

  // Contruir el array de casos, indice
  Set  arrSet = For(1, Card(filSor), Text(Real filPos) // Para los casos
```

```
  {
    "  "+GetFilePrefix(filSor[filPos][1])+If(Card(filSor)==filPos, "", ",\n")
  });
  Text arrTxt = SetSum(arrSet);
  Text arrSed = dirSrc+"/simulatorarray.sed"; // Fichero semilla
  Text arrPth = Replace(arrSed,".sed",".js"); // Fichero javascript
  Text WriteFile(arrPth, Replace(ReadFile(arrSed), "_ARR_", arrTxt));

  // Construir el banco de datos de pasos, steps, de los casos
  Text sdbSep = "\n"+Repeat("/", 78)+"\n\n";
  Set  sdbSet = For(1, Card(filSor), Text(Real filPos) // Para los casos
  {
    ReadFile(filSor[filPos][1])+If(Card(filSor)==filPos, "", sdbSep)
  });
  Text sdbTxt = SetSum(sdbSet);
  Text sdbSed = dirSrc+"/simulatordb.sed";
  Text sdbPth = Replace(sdbSed,".sed",".js");
  Text WriteFile(sdbPth, Replace(ReadFile(sdbSed), "_SDB_", sdbTxt));

  Card(filSor) // Numero de casos
};
////////////////////////////////////////////////////////////////////////
PutDescription(
"Reune todos los problemas de las 8 reinas en un tablero de ajedrez resueltos,
como maximo son 92, y los prepara para que su resolucion pueda ser simulada
por un simulador visual Javascript.
Genera 2 ficheros en lenguaje Javascript:
a) un fichero indice con un array que contiene todos los problemas resueltos
   ordenado de mas a menos pasos, para que los mas largos esten al principio y
b) un fichero mas grande, con tantos arrays como casos resueltos,
   cada uno de estos arrays contiene todos los pasos de resolucion del caso,
   el orden de declaracion de estos arrays puede ser cualquiera,
   pero se emplea el anterior, de mas a menos pasos de resolucion.
El nombre de los arrays esta formado por una letra p mas los 8 digitos de las
filas en las que esta cada reina, el orden de estos digitos es de la columna 1
a la columna 8.
Para la generacion de estos 2 ficheros emplea 2 ficheros semilla que tienen
la cabecera de comentarios de los programas Javascript.
Recibe como parametros
a) el directorio de casos resueltos de entrada, que han sido generados por la
   funcion QueTrace() y
b) el directorio de salida donde se escribira el codigo Javascript,
   asume que los ficheros semilla Javascript estan en este mismo directorio y
   los nombres de los ficheros para el array indice y
   el banco de datos son fijos.
Retorna el numero de problemas resueltos que en el caso de las 8 reinas sin
atacarse en un tablero de ajedrez son 92 posibilidades como máximo, si bien,
de esas 92, unas puede obtenerse de otras mediante rotaciones y simetrias.",
QueJavascript);
////////////////////////////////////////////////////////////////////////
```

Real solCla

```
////////////////////////////////////////////////////////////////////////
Real solCla = If(FALSE, FALSE, // Cambiar TRUE/FALSE para ejecutar
{
  Set  queSol = QueSolve(QueChess,       // Tablero inicial con 8 reinas
                         QueConstraint); // Restricciones que han de cumplir
  Real QueTrace(QueMemory);              // Guarda la traza del proceso
  Card(queSol)
});
////////////////////////////////////////////////////////////////////////
PutDescription("Resolver el problema clasico de las 8 reinas.", solCla);
////////////////////////////////////////////////////////////////////////
```

Real solRnd

```
////////////////////////////////////////////////////////////////////////
Real solRnd = If(TRUE, FALSE, // Cambiar TRUE/FALSE para ejecutar
```

```
{
  Real maxCic = 20;
  Set  simCic = For(1, maxCic, Real(Real numCic)
  {
    Set  tabSet = For(1, QueDim, Set(Real colPos)    // Crea un tablero con
                    { Queen(colPos, R(QueDim)) }); // las reinas al azar

    Set  queSol = QueSolve(tabSet,          // Tablero inicial al azar
                           QueConstraint); // Restricciones que han de cumplir
    Real QueTrace(QueMemory);               // Guarda la traza del proceso
    Card(queSol)
  });
  Card(simCic)
});
////////////////////////////////////////////////////////////////////////////
PutDescription("Resuelve varios problemas con las 8 reinas al azar.", solRnd);
////////////////////////////////////////////////////////////////////////////
```

Real solJsc

```
////////////////////////////////////////////////////////////////////////////
Real solJsc = If(TRUE, FALSE, // Cambiar TRUE/FALSE para ejecutar
{
  QueJavascript(QueSim+"/solved", // Directorio de problemas resueltos
                QueSim+"/src")    // Directorio de Javascript
});
////////////////////////////////////////////////////////////////////////////
PutDescription("Prepara los ficheros para Javascript con los casos.", solJsc);
////////////////////////////////////////////////////////////////////////////
```