UNIVERSIDAD COMPLUTENSE
MADRID

Programación Declarativa Avanzada

Facultad de Informática

Samuel Méndez Galán
Arturo Pareja García

Junio 2014

Autorización de difusión

Los abajo firmantes autorizan a que se publique el proyecto que han realizado para la asignatura de PROGRAMACIÓN DECLARATIVA AVANZADA de forma que podrá estar online y accesible para terceros, bajo las siguientes licencias: el software liberado bajo la licencia BSD (*) y la memoria, documentación, gráficos u otro contenido liberado bajo la licencia Creative Commons Atribución 4.0 Internacional (**). La autoría será respetada en todos los casos.

(*): http://directory.fsf.org/wiki/License:BSD_3Clause
(**): https://creativecommons.org/licenses/by/4.0/deed.es

Nombre, DNI y firma: Samuel Méndez Galán. X1525018A

Nombre, DNI y firma: Arturo Pareja García. 52896380Z

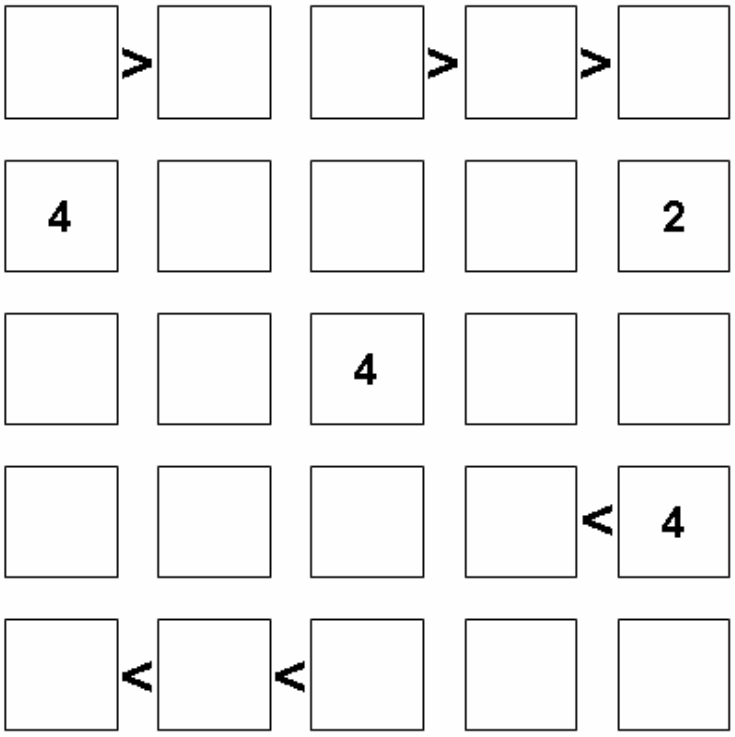
Madrid, 23 de Junio de 2014

Índice

Introducción

Futoshiki [1], también conocido como Unequal, es un juego de lógica tipo puzzle de origen japonés. Se juega en un tablero cuadrado de tamaño prefijado de, por ejemplo, 5x5 celdas.

El objetivo es colocar en cada celda un número entre el 1 y el tamaño del tablero. En cada fila y en cada columna aparece un dígito una sola vez. Así, cuando se soluciona el tablero, el resultado es un **cuadrado latino**.



Ejemplo de Futoshiki 5x5

En un principio, su resolución recuerda a la del famoso juego Sudoku, pues también se cumple la condición de cuadrado latino. Pero en este caso, también se han de cumplir las restricciones matemáticas que aparecen entre celdas del tablero. Éstas sirven de pistas al empezar a resolver el problema, junto con la posible aparición de algún número al inicio. Cada puzzle debe tener una única solución.

Motivación

Al pensar en qué tipo de práctica final realizar, lo primero por lo que empezamos a buscar fueron juegos. Los juegos suelen gustar a todo el mundo y en el ámbito de la programación se utilizan mucho para ver las capacidades de una máquina al resolver problemas complejos.

Encontramos una página web [2] con una gran variedad de juegos de lógica para los que poder desarrollar un programa que los resolviese. Entre todos ellos nos decidimos por 2:

- Net: juego de lógica cuyo objetivo es que un láser se comunique desde la fuente a todos los receptores destino, rotando cada reflector [3].
- Unequal: puzzle de estilo sudoku, cumpliendo además las restricciones de desigualdad.[4]

Empezamos la especificación del primero de ellos, pero enseguida nos dimos cuenta de que no era una tarea fácil su representación, además de carecer de heurísticas que, a priori, nos diese alguna información sobre la cercanía de la solución. Por otra parte, existían muchos estados posibles para cada celda, junto con los movimientos y la diferencia de la celda central. Después de pensarlo mucho y consultar al profesor decidimos elegir otro juego.

Y escogimos Futoshiki. Pero pensamos en un **resolutor** para el juego. Hicimos una breve investigación y vimos que, utilizando la unificación de variables mediante la propagación de restricciones de la librería *clpfd,* era muy sencillo resolverlo.

Fue entonces cuando el profesor nos sugirió diseñar un **generador** de tableros para el juego. Buscando información sobre resolutores no encontramos ninguno para este juego. Si que existen páginas que generan tableros para que un usuario los resuelva, pero era imposible acceder al código de cómo se generan esos tableros. Lo más parecido era un generador para Sudokus, que nos podía servir como base para los cuadrados latinos.

Metodología

Resolutor

El resolutor de Futoshiki (`futoshiki.pl`) se ha implementado mediante el uso de la librería *clpfd,* puesto que, el mismo juego viene dado por la resolución de restricciones (desigualdades y cuadrado latino) en un dominio finito de números enteros (1..N siendo N el tamaño del tablero).

Dada la eficiencia proporcionada por la propagación de restricciones en *clpfd* podemos conseguir que resuelva tableros de tamaño 10x10 de forma casi instantánea. Además, el código queda muy sencillo y legible, como se muestra a continuación:

Resolutor de Futoshiki

```
futoshiki(Rows, Lts) :-
        length(Rows, N), maplist(length_(N), Rows),
        append(Rows, Cells), Cells ins 1..N,
        unequals(Rows, Lts),
        latin_square(Rows),
        label(Cells).
```

En la restricción de "*unequals*", se impone que todas las restricciones de desigualdad (Less than o, abreviado, Lts) se cumplan en el tablero (Rows).

En la restricción de "*latin_square*", usaremos la función **transpose** de la librería *clpfd* para asegurar que se cumpla la restricción de cuadrado latino en todas las filas y columnas.

También hemos implementado predicados para la comprobación de solución única. Dichos predicados inicialmente eran muy ineficientes ya que utilizaban el predicado *findall,* pero al final hemos podido solventarlo gracias al predicado *findnsols,* cuyo código hemos tenido que incorporar al proyecto directamente [8] puesto que sólo está implementado en las últimas versiones de swi-prolog.

Estos predicados nos han ayudado a que la generación de tableros de Futoshiki sea mucho más eficiente, ya que su solución tiene que ser única.

Generador

La implementación de un generador de Futoshiki (`futoshiki_gen.pl`) tiene 2 posibles formas de ser abordada. Ambas partirán de un juego completamente resuelto y con todas las restricciones entre celdas establecidas, pero, en un primer enfoque, podemos ir eliminando celdas y restricciones, mientras que la segunda opción sería ir rellenando casillas y restricciones de un tablero vacío, con las de la solución de partida.

Si la comprobación de solución única tiene el mismo coste que la comprobación de no ser solución única, entonces, ambos enfoques tienen el mismo número de combinaciones. Esto es debido a que tiene el mismo coste asintótico quitar una celda que añadirla.

Nosotros hemos optado por la segunda opción al pensar que tardaríamos menos en llegar a la solución rellenando un problema "vacío" puesto que un tablero generado, por lo general, dispone inicialmente de muy pocas celdas y restricciones.

Con este enfoque hemos conseguido implementar hasta 4 versiones distintas:

- Versión de dominios finitos: predicado `gen_futoshiki1(N)`. Obtiene distintas permutaciones de celdas y restricciones mediante *clpfd,* pero, como hay muy pocas restricciones, resulta muy ineficiente.

- Versión combinatoria: predicado `prueba_gen(N, NC, NL, Rows1, Lts1).` Dado que la anterior versión generaba muchas posibilidades que nunca se iban a tomar, se decidió crear de antemano todas las combinaciones (sin repetición) de NC celdas y NL "less-thans". También resulta bastante ineficiente.

- Versión de selección aleatoria: predicado `gen_futoshiki(N)`. Esta es la versión más eficiente de todas, pero no es del todo determinista. La idea es insertar NC celdas y NL "less-thans", escogidos de forma totalmente aleatoria. NC y NL van a venir dados en función de la dificultad que escoja el usuario y, que encuentre o no solución vendrá dado en gran medida por el grado de dificultad.

- Versión de conjuntos críticos: predicado `genfutoshiki(N, NC, NL)`. Basado en la definición matemática de **conjunto crítico** [7], que no es más que el conjunto de celdas y restricciones más pequeño posible para que un tablero de tamaño N tenga solución única. Con ello, generamos rápidamente una lista de posibles posiciones de NC celdas y NL restricciones. Pero, la definición que hemos encontrado de conjunto crítico para Futoshiki no es lo suficientemente formal y eso nos introduce cierto grado de indeterminismo e ineficiencia al buscar entre más permutaciones después de haber fallado al encontrar solución única en unas cuantas vueltas del algoritmo.

Manual de uso

Descargar el proyecto desde Github [9]. Opción “Download ZIP”.

El proyecto consta de 3 archivos fuente: `futoshiki.pl`, `futoshiki_gen.pl` y `findnsols_lib.pl`.

Para ejecutar el resolutor, cargar el archivo `futoshiki.pl` y ejecutar los siguientes predicados:

- Los problemas de prueba unifican con `problem(+Index, +Rows, +Lts)`. Index va a identificar el número del problema, Rows unifica con el tablero y Lts unifica con la lista de restricciones. Ejemplo:

```
problem(2,
        [[_,_,_,2],  % problem grid
         [_,_,_,_],
         [_,_,_,_],
         [_,_,_,_]],

        [[0,0,1,0],    % [i1,j1, i2,j2] requires that values[i1,j1] < values[i2,j2]
         [0,1,1,1],
         [2,1,3,1],
         [2,3,3,3]]).
```

- Para resolver un problema, ejecutar `futoshiki_solve(+Index, -Rows)`. Index va a unificar con el identificador del problema a resolver y Rows será el Futoshiki resuelto.

Para ejecutar el generador, cargar el archivo futoshiki_gen.pl y ejecutar los siguientes predicados (N: tamaño del tablero; NC: número de celdas; NL: número de restricciones):

- `gen_futoshiki(N)`: el programa nos preguntará por la dificultad y mostrará el tablero generado y las restricciones debajo. Trabaja bien con N = {4,5,6}. Es la versión estable y completamente funcional de la práctica.
- `gen_futoshiki1(N)`: el programa preguntará al usuario el nivel de dificultad. Introducir ‘F’ (sin comillas) para nivel fácil y ‘D’ para el nivel difícil. Para tamaños de tablero mayores que 5 puede no terminar.
- `prueba_gen(N, NC, NL, Rows1, Lts1)`: devuelve la solución en Rows1 y Lts1. NC y NL servirán para gradar la dificultad. Ineficiente para N > 5.
- `genfutoshiki(N, NC, NL)`: la dificultad se puede configurar con NC y NL. Si no tiene las suficientes casillas o restricciones puede no terminar.

Tecnologías utilizadas

Para el desarrollo de Futoshiki hemos utilizado las siguientes tecnologías:

- **SWI-Prolog**[5]. Se trata de una implementación en código abierto de Prolog que incluye, entre otras características, librerías y un IDE con interfaz gráfica y debugger. Lo hemos utilizado tanto para editar el código como para compilarlo y ejecutarlo, ya que a través de su interfaz gráfica se le puede incluir una consola de Prolog.

- **Clpfd**. Librería para SWI-Prolog de programación lógica basada en restricciones (CLP). Se extiende la noción de una variable lógica permitiendo que las variables pertenezcan a un dominio en lugar de poseer un valor específico. Las variables también pueden ser restringidas, lo que significa que su valor debe respetar ciertas reglas especificadas por el programador. Sus características permiten resolver problemas combinatorios complejos con una cantidad mínima de código.

- **Sublime-Text**[6]. Es un editor de texto multiplataforma orientado principalmente a la programación. Se trata de un editor rápido, potente y con multitud de plugins que permiten añadir casi cualquier funcionalidad imaginable. La interfaz gráfica y visualización del código es mejor que el editor de SWI-Prolog.

Conclusiones

Para el desarrollo de esta práctica hemos utilizado la librería *clpfd* vista en clase y, gracias a ella, ha habido algunos apartados que han sido realmente fáciles de conseguir, como el resolutor del puzzle. además de proporcionar una gran eficiencia.

Pero también hemos encontrado problemas: quizás más acostumbrados a la programación imperativa que a la declarativa, hemos encontrado dificultades con Prolog al utilizar la recursión para verificar que un tablero tenía una solución única. De esto se deriva que solo tengamos tableros hasta 7x7 de tamaño.

En definitiva, hemos aprendido que lo más eficiente es usar dominios finitos siempre y cuando existan muchas restricciones que nos puedan acotar el problema. Si no, debemos siempre acudir a una definición formal matemática que nos restrinja al máximo el espacio de búsqueda, y si no es posible, tratar de encontrar soluciones “eficientes” como las aproximaciones por algoritmos probabilísticos totalmente aleatorios (pero restringiendo mucho los valores que puede tomar) para que el problema de la explosión combinatoria nos afecte lo menos posible.

Trabajo futuro

Actualmente el generador de tableros de Futoshiki está disponible para tamaños de 4x4, 5x5 y 6x6, debido a tener que añadir niveles de dificultad y también al tiempo que tardaba en encontrar un tablero que tuviese solución única. Una de las mejoras que podría tener sería que fuese capaz de generar tableros de NxN, optimizando más el método para encontrar un tablero único.

También se pensó en un principio en una interfaz gráfica más amigable que mostrar el tablero por consola, pero hubiese llevado bastante tiempo y nuestro objetivo era dejar el juego lo más avanzado posible. Aún así se ha intentado formatear la salida por consola para que se vea lo mejor posible y sea entendible.

Bibliografía

[1] 2006. Futoshiki - Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Futoshiki.

[2] 2004. Simon Tatham's Portable Puzzle Collection - Chiark.
http://www.chiark.greenend.org.uk/~sgtatham/puzzles/.

[3] 2013. Net, from Simon Tatham's Portable Puzzle Collection - Chiark.
http://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/net.html.

[4] 2013. Unequal, from Simon Tatham's Portable Puzzle Collection - Chiark.
http://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/unequal.html.

[5] 2005. SWI-Prolog - Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/SWI-Prolog.

[6] 2007. Sublime Text: The text editor you'll fall in love with.
http://www.sublimetext.com/.

[7] 2012. Critical Sets in Futoshiki Squares. Dan Katz.
http://www.math.brown.edu/~dkatz/futoshiki-talk.pdf.

[8] 2010. SWI-Prolog compatibility library. Find N-sols code.
https://lists.iai.uni-bonn.de/pipermail/swi-prolog/2010/003518.html

[9] Repositorio del proyecto (Download ZIP)
https://github.com/arturpareja/futoshiki