

Memoria Práctica 1

Sistemas inteligentes

Leopoldo Cadavid Piñerooooooooo

Febrero 2022

Índice

1. Introducción	3
2. Algoritmos impletmentados	3
2.1. Backtracking	3
2.1.1. Funciones y métodos	3
2.1.2. Problemas durante la implementación	4
2.2. Algoritmo AC3	5
2.2.1. Funciones y métodos	5
2.2.2. Problemas durante la implementación	7
2.3. Foward Checking	8
2.3.1. Funciones y métodos	8
2.3.2. Problemas durante la implementación	9
3. Estudio de los tiempos de ejecución y conclusiones	9

1. Introducción

En la siguiente memoria se procederá a explicar los diferentes algoritmos de resolución empleados para resolver distintos tableros del juego Futoshiki.

Además, se compararán los resultados de estos en cuanto a su coste temporal, Para poder concluir en cual método es el óptimo para el problema propuesto.

2. Algoritmos impletmentados

Para cada algoritmo comentaremos las funciones añadidas y la explicación general de su funcionamiento, añadiendo también un apartado comentando fallos o problemas durante la implementación. Posteriormente, en el apartado 3 se comentarán las diferencias temporales con los demás algoritmos.

2.1. Backtracking

La primera solución implementada para el problema Futoshiki ha sido el esquema backtracking. Primero, se comentará la explicación conceptuas de que cómo se ha hecho la implementación del algoritmo, repasando las funciones y estructuras utilizadas.

2.1.1. Funciones y métodos

- `bool bt_futoshiki()`: es la función donde se encuentra la estructura de resolución backtracking. Devolverá true si se ha llegado a la solución, o false en caso contrario.
- `bool factible()`: en esta función se comprueba si el valor seteado en la casilla cumple con las restricciones propias de las reglas de Futoshiki.
- `void ejecutarBT()`: función por defecto dada en la práctica. Se utilizará para obetener los valores iniciales de partida para la solución Backtracking y desde esta llamaremos a `bt_futoshiki()`. No se han añadido parámetros a esta ni se han modificado sus propiedades.

Entrando más en detalle con el funcionamiento de la función `bt_futoshiki()`, lo que se hace en el código es recorrer el tablero empezando por la superior iz-

quiera y para cada posición se comprueban los números de 1 a N. Si el número es factible en esa posición, lo colocamos en la casilla con `setCasilla()` y llamaremos a la recursividad para seguir comprobando. Si no es factible, comprobamos el siguiente valor posible.

Tras llamar a la recursividad debemos ver si ha devuelto un resultado un resultado verdadero, en cuyo caso se ha encontrado la solución y debemos salir con *true*.

Es importante tener en cuenta que si hemos comprobado todos los valores posibles y no hay ninguno factible, estamos siguiendo una instanciación errónea y debemos devolver *false*.

En el caso de que en la casilla ya haya un valor prefijado, debemos comprobar aún así si es factible, pues si no tendremos que volver hacia atrás. En ningún caso modificamos el valor de la casilla.

También es importante comentar el funcionamiento de `factible()`. En esta función comprobamos si, al poner el valor en la casilla deseada:

- Si el número se repite en la casilla o en la columna.
- Si el número cumple las restricciones de "*mayor que*" o "*menor que*" con respecto a la casilla anterior, pues al recorrer el tablero hacia abajo/derecha, comprobar con las casillas a nuestra derecha significa que estaríamos comprobando con casillas vacías, lo cual nos llevaría a resultados incorrectos.

2.1.2. Problemas durante la implementación

Durante la implementación del algoritmo se dieron algunos problemas, que consiguieron ser solucionados para cumplir con la tarea. Los más destacable:

- No se comprobaba si las llamadas recursivas devolvían la solución, con lo cual el algoritmo seguía aplicándose una vez habiendo encontrado la solución.
- Al principio no se comprobaba si los números prefijados eran factibles, lo cual daba problemas en el momento en que uno de estos tenía una restricción binaria con una casilla adyacente. Esto llevaba a resultados incorrectos en tableros como el de 6x6.

- No se devolvía falso cuando el algoritmo había descartado todos los números como soluciones factibles de una casilla, lo cual llevaba a una indeterminación en la solución.

2.2. Algoritmo AC3

2.2.1. Funciones y métodos

Para la implementación de este algoritmo, con el fin de seguir la estructura aprendida en clase, se implementaron nuevas clases al proyecto, además de añadir nuevos métodos y estructuras de datos.

Primero veamos las **clases** creadas:

- **Clase nodo:** representa la estructura de datos de un nodo. Se compone de dos enteros que son sus coordenadas X e Y.
- **Clase arista:** representa la estructura de datos de una arista. Está compuesta de dos nodos A y B, y en el algoritmo consideraremos que la arista tiene dirección de A a B, aunque realmente la arista no tiene una dirección.
- **Clase matdominios:** representa el dominio de las casillas del tablero. Se compone de una matriz de 3 dimensiones, donde la primera y segunda serán la fila y columna del tablero. La tercera dimensión representa los números que pueden estar en esa posición. El valor en sí de la casilla [X][Y][Z] es "1" o "0" para indicar si el valor está o no en el dominio de esa casilla. En el código, se definirá como una **variable global**.

Ahora hablaremos de las **funciones** y **métodos** más importantes:

- **ejecutarAC3():** función base dada en la práctica, en ella se arregla el dominio del tablero siguiendo el esquema AC3.
- **consistente():** en esta función se analiza la consistencia para un valor del nodo A con respecto a los valores del nodo B. Devolverá *true* si hay algún valor del nodo que hace consistente al nodo A.
- **enDominio():** método de la clase **matdominio** que analiza si un número se encuentra dentro del dominio de una casilla. Debemos especificar la fila, columna y valor a comprobar.

- **sacarDominio**: método de la clase **matdominio** donde sacamos un valor dado del dominio de la casilla (fila, columna) que le indiquemos.
- **meterDominio**: método de la clase **matdominio** donde metemos un valor dado en el dominio de la casilla (fila, columna) que le indiquemos.
- **imprimirDominio()**: método de la clase **matdominio** donde imprimimos por pantalla el dominio de la matriz de dominios.
- También se utilizan los constructores, getters y setters de las clases.

En cuanto a **estructuras de datos** usadas, destacamos las siguientes:

- **Nodo**: punto con coordenadas X e Y.
- **Arista**: conexión entre dos nodos.
- **Matriz de dominios**: para saber que valores son accesibles para una casilla del tablero.
- **Colas de doble frente**: Las utilizaremos para guardar todas las aristas del tablero, y poder ir haciendo el análisis en el algoritmo arista a arista. También utilizaremos otra para guardar aristas descartadas.

Entrando en detalle con el funcionamiento del algoritmo, lo primero que haremos será sacar del dominio todos los valores de casillas prefijadas (dejando también solo esos valores en el dominio de su casilla). Tras esto crearemos una matriz de nodos con todos los nodos del tablero, y con ella iremos creando todas las aristas necesarias. Serán dos aristas (para cada dirección) por cada par de nodos que estén en la misma fila o columna. Todas las aristas creadas se meten dentro de una cola.

Ahora, mientras la cola de descartes no quede vacía (haber analizado todas las aristas). Repetiremos el siguiente proceso:

- Sacamos una arista del frente de la cola.
- De 1 a N comprobaremos si los valores están en el dominio del nodo A.
- Si esta en el dominio, comprobaremos si el valor es consistente con el nodo B, usando **consistente()**.
- Si **no** es consistente, sacaremos el valor del dominio del nodo A, usando **sacarDominio()**.

- En caso de haber cambiado el dominio, debemos comprobar las aristas que se han analizado previamente que apunten al nodo **A** actual. Esto lo haremos recurriendo a una cola de descartes, donde se encuentran todas las aristas analizadas previamente.
- Si se queda vacío el dominio de un nodo, entonces no hay solución y salimos con `false` de la función.

Ahora comentaremos la función `consistente()`, que es muy importante en cuanto a la resolución del algoritmo.

Esta función recibe un valor del nodo **A** y comprueba, para todos los valores del nodo **B**, las siguientes restricciones:

- Que no sean iguales.
- En caso de que la distancia entre nodos sea 1, usamos la función `getElement()` para ver la posición entre nodos y si hay una restricción. En este caso, comprobaremos si los valores cumplen esta restricción (de haberla).
- Solo necesitamos que un valor cumpla con las restricciones para que la función devuelva `true`.

Tras hacer los arreglos en el dominio con AC3, llamamos a `bt_futoshiki()`, que ha sido modificada para tener en cuenta la matriz de dominios (realizar comprobaciones sólo con números que se encuentren dentro del dominio de una casilla) y esta resolverá el tablero, pero partiendo con un problema simplificado, lo que a priori debería mejorar el tiempo de resolución.

2.2.2. Problemas durante la implementación

Durante la implementación de AC3, no hubo grandes problemas más allá de errores de compilación puntuales o problemas de *segmentation fault*. Aunque si es cierto que hicieron falta muchas pruebas para llegar a la idea de usar la cola de descartes o comprobar que las clases implementadas funcionaban bien con el código.

2.3. Foward Checking

2.3.1. Funciones y métodos

Para la implementación de Forward Checking, se reutilizaron varios métodos ya descritos en el apartado anterior 2.2.1, como por todos los métodos de la case **matdominio**. Las nuevas funciones y estructuras añadidas para este algoritmo son:

- **Matriz de podados:** es una matriz del tipo **matdominios**, análoga a la matriz de dominio pero que se usa para saber, en cada nivel, que valores han sido eliminados del dominio. Se recurrirá a esta en caso de que queramos reestablecer todos los valores eliminados en una determinada instanciación.
- **ejecutarFC():** función dada para la práctica donde sacaremos los números prefijados de los dominios correspondientes y donde rellenaremos el tablero una vez la función recursiva haya acabado.
- **fc_futoshiki():** función recursiva y la principal del esquema de Forward Checking.
- **forward():** función que realiza el razonamiento hacia adelante”.
- **instanciar():** función que instancia un valor a una posición del dominio, sacando los demás posibles valores del dominio de esa casilla.
- **restaurar():** en esta función usaremos la matriz de podados para volver a introducir todos los valores que fueron podados por la instanciación actual.
- **consistente_fc():** es prácticamente la misma función que usamos para AC3, pero esta solo realiza la comparación entre un solo par de valores.

Ahora, el funcionamiento del algortimo consiste en los siguientes pasos:

- Instanciamos uno de los valores posibles en la posición en la que nos encontramos.
- Razonamos hacia adelante, es decir, comprobamos si la instanciación que hemos propuesto es consistente con todas las demás casillas de la misma fila/columna. Si un valor no es consistente, lo eliminamos del

dominio.

- Si no hay valores que consigan hacer cumplir el razonamiento hacia adelante, debemos restaurar todos los números podados.
- Si el razonamiento no da problemas, llamamos a la recursividad para la siguiente casilla.
- En caso de que ningún valor de los probados de resultados, debemos restaurar todo lo podado a partir de la casilla actual y volver hacia atrás.
- Si hemos llegado a la última posición, consideraremos que tenemos el resultado correcto.

2.3.2. Problemas durante la implementación

El principal problema que dio la implementación del algoritmo surge de no haber tenido en cuenta, al principio, que se debían arreglar los dominios cuando realizamos la instanciación. Esto provocaba problemas a la hora de volver hacia atrás. Otro problema fue considerar que el código solo restauraba si `forward()` devolvía *false*, cuando debía restaurar también si la llamada recursiva devolvía *false*.

```
// ESTRUCTURA DE DATOS A UTILIZAR //  
priority_queue<Node> pq; // Creamos la cola de prioridad "pq"
```

Figura 1: Implementación de la cola de prioridad

3. Estudio de los tiempos de ejecución y conclusiones

En este apartado procederemos a comparar y comentar los distintos tiempos de ejecución y pasos entre las diferentes estrategias de resolución descritas durante la práctica. Además, a modo de experimentación, se han medido los tiempos resultantes de combinar las estrategias de AC3 y forward checking. Los resultados han sido los siguientes:

Tiempos de ejecución (en segundos)				
Tablero	Backtracking	AC3	Forward Checking	FC + AC3
4x4	0,000055	0,000472	0,000181	0,000772
5x5	0,005804	0,004638	0,011652	0,004686
6x6	0,000316	0,006238	0,002783	0,005765
7x7	0,118968	0,08557	0,518506	0,316627
9x9 (1)	??	1,73674	??	6,044
9x9 (2)	1,36732	0,173556	3,47478	0,406233

Tabla 1: Tiempos de ejecución para distintos tamaños de tablero

Número de Pasos dados por casa algoritmo				
Tablero	Backtracking	AC3	Forward Checking	FC + AC3
4x4	33	26	17	17
5x5	7.362	26	1902	25
6x6	181	37	110	36
7x7	232.989	183.504	120.765	76.465
9x9 (1)	??	4.719.952	??	774.559
9x9 (2)	3.005.483	357.409	455.114	47.251

Tabla 2: Pasos tomados para la solución distintos tamaños de tablero

Como podemos observar, el esquema que nos da un mejor resultado en promedio es la solución AC3. Aunque en los tableros de 4x4 y 5x5 vemos un tiempos mejores con backtracking, esto ocurre debido al coste fijo que tiene para AC3 recorrer todas las aristas. Para tamaños mayores observamos que el algoritmo AC3 nos ayuda a reducir en gran magnitud el tiempo. Un caso muy notorio es con los tableros de 9x9 donde el *V1B1F2_9* tarda casi 10 veces menos en ser resuelto o el *V1B1F2_9* que pasa de no poder ser resuelto en un tiempo menor a varios minutos, a ser resuelto en 1,73 segundos.

En cuanto a los pasos, vemos que AC3 siempre ayudará tanto a Forward Checking como a Backtracking, a realizar menos pasos, aunque sea a costa de un coste fijo inicial.