Universitat d'Alacant
Universidad de Alicante

Sistemas inteligentes
Grado Ingeniería Robótica

Práctica 1

Búsqueda y satisfacción de restricciones

Carmen Ballester Bernabeu

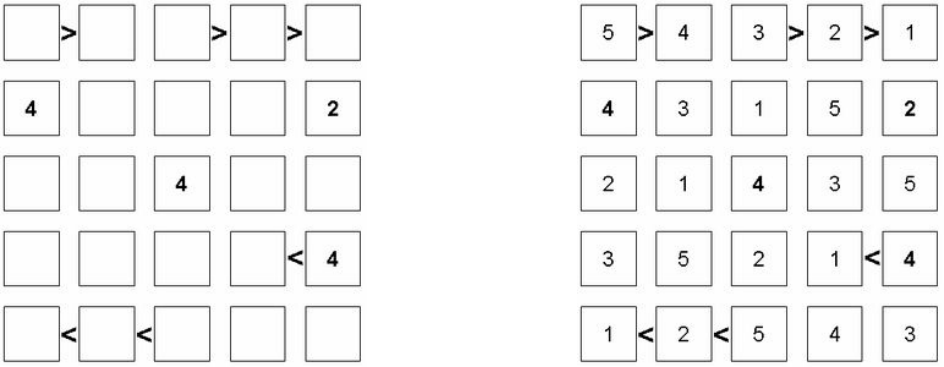
Enlace al github de la práctica

Índice

Introducción al problema

El problema que se plantea es un juego típico japonés, parecido al sudoku, llamado ***futoshiki.*** Consiste en un tablero de tamaño nxn donde en cada casilla podemos poner un valor desde 1 hasta n. Al igual que en los sudokus, normalmente hay casillas que ya están definidas con un valor concreto. Para definir el resto de casillas correctamente y resolver el juego, los valores tienen que cumplir tres restricciones:

- Un mismo valor no se puede repetir en una fila.
- Un mismo valor no se puede repetir en una columna.
- En el caso de que entre dos casillas adyacentes haya una restricción binaria, los valores tienen que cumplirla.



Figuras 1 y 2. Ejemplo de un tablero Futoshiki y su solución

Backtracking

La primera forma de resolver el problema es utilizando directamente backtracking, que es un algoritmo de búsqueda tentativa no informada. Construye la solución de forma parcial, comprobando que se cumplen las restricciones para expandir un nodo. Si no se cumplen, no sigue explorando esa rama y cambia los valores de los estados anteriores para expandir otras ramas, hasta llegar a una hoja con la solución.

Código del algoritmo

Para implementar este algoritmo se utiliza como base un método de la clase *Solver* proporcionada por el profesor para el desarrollo de la práctica. Sin embargo, para poder resolver correctamente la recursión, lo mejor es hacer una función a parte, además de otras funciones auxiliares para modular el código, incluidas también como métodos en la clase *Solver*.

Para trabajar con mayor facilidad, lo primero es decidir cómo almacenar los dominios de las casillas. Un dominio se define como un vector de enteros, por lo que cada casilla del tablero tiene asociado su dominio. Aprovechando las facilidades de C++, nos definimos dos tipos

para que la codificación de esta representación espacial quede más clara. Primero, definimos *Dominio* como un vector de enteros, y luego definimos *TDominios* como una matriz de Dominio, utilizando la clase *vector* de C++.

```
typedef vector<int> Dominio;
typedef vector<vector<Dominio>> TDominios;
```

También añadimos a la clase dos atributos. El primero es la matriz tridimensional *tDom*, que se declara como un atributo de la clase para que sea accesible por los distintos métodos de resolución. Como solo se inicializa el dominio una vez, hay que tener en cuenta cada vez que se llama a un método de resolución si la matriz de dominios está inicializada o no. Para ello se crea otro atributo, un booleano que indica si el dominio ha sido inicializado ya o no.

```
private:
  int backTrackingJumps;
  TDominios tDom;
  bool domInc = false;
```

Solver::cumpleRestricciones

Es uno de los métodos auxiliares que se utilizan para resolver backtracking. Recibe como parámetros el tablero y la fila y la columna de la casilla de la que queremos comprobar las restricciones, y devuelve en un booleano si las cumple o no.

Simplemente llama a los métodos de la clase tablero que comprueban las restricciones de fila, columna y las binarias de las casillas siguientes. Tras la primera implementación, el algoritmo no funcionaba. Esto se debía a que al poner una casilla no tenemos que comprobar únicamente las restricciones binarias siguientes, sino las anteriores también. Para ello, comprobamos si existe la casilla de arriba y la de la izquierda, y si existen se llaman a los métodos de la clase tablero anteriores pero con estas casillas, de forma que comprueban las restricciones anteriores a la casilla actual.

Solver::inicializarDominio

Es otra función auxiliar para crear la matriz tridimensional que va a almacenar los valores del dominio e inicializar los mismos, atributo de la clase. Vamos a utilizar los dos tipos definidos en el fichero de cabecera y los métodos estándar de C++ para la clase vector. Recibe como único parámetro el tablero, y no devuelve nada.

Cómo vamos a usar el método *push_back*, tenemos que crearnos una variable de tipo Dominio, un vector de dominios y finalmente una variable de TDominios. El algoritmo es sencillo, se recorre cada casilla del tablero, si está inicializada sólo se añade ese valor al dominio, y si no, se añaden los valores [1..s] a la variable de tipo Dominio. Al terminar ese bucle, añadimos esta variable a la variable de tipo vector<Dominio>, y después del siguiente bucle añadimos el vector de Dominios a la variable TDominios. Hay que tener en cuenta

que después de cada *push_back* hay que limpiar esa variable para que no se encadenen los datos.

Solver::backtracking

Es la función recursiva principal que resuelve el problema utilizando backtracking. Recibe como parámetros el tablero, y la fila y la columna de la casilla actual, y devuelve un booleano que puede ser true si encuentras solución por esa rama, o false si no.

El caso base de la recursión es cuando hemos completado el tablero, y por tanto, tenemos una solución del problema. Si no estamos ante el caso base, tenemos dos posibilidades, que la casilla actual esté vacía o que tenga un valor asignado. Si tiene un valor asignado llamamos a la función con la siguiente casilla.

Si no lo tiene, como en la mayoría de casillas, hay que recorrer el dominio de esa casilla e ir asignándole dichos valores del dominio. Al asignar un valor del dominio, comprobamos si se cumplen las restricciones y si se cumplen, llamamos a la función con la siguiente casilla. En el caso de que no se cumplan las restricciones, probamos con otro valor del dominio de la casilla actual, y si para ningún valor del dominio se cumplen, la función devuelve false, se descarta esa rama y se modifican valores de estados anteriores.

Hay que tener en cuenta que cuando se llama a la casilla siguiente tenemos dos posibilidades: la primera es que queden columnas dentro de la fila actual y la segunda es que sea la última columna de la fila y haya que saltar a la primera columna de la siguiente fila.

Solver::ejecutarBT

Es el método de la clase *Solver* con el que vamos a resolver el problema utilizando backtracking. No devuelve ningún valor, por lo que lo más cómodo es desde aquí llamar a la función recursiva anterior. Recibe el tablero y no devuelve ningún valor.

Simplemente llama a dos funciones, inicializarDominio, si no ha sido inicializado anteriormente por AC3, y backtracking. También muestra por pantalla el tiempo que ha tardado en resolverse el problema y los saltos hacia atrás que ha tenido que realizar la función recursiva.

Tests

Para comprobar el funcionamiento del algoritmo, almacenamos el tiempo que tarda en ejecutarse y el número de llamadas recursivas. Se ha ejecutado el código con tableros de distintos tamaños, desde 4x4 hasta 9x9. Si se quieren realizar más pruebas, se pueden generar más tableros en este enlace.

Como era esperado, sólo el algoritmo backtracking es lento, y puede ser que para tamaños grandes no resuelva cierto tipo de tableros en un tiempo aceptable, como sucede con el tablero 8 y el 9.1.

Para el resto de tableros sí obtenemos una solución en un tiempo adecuado. Podemos comprobar en los dos ejemplos de tamaño n que el tamaño no es el único indicativo del tiempo que va a tardar en resolver el algoritmo el problema, ya que un tablero de tamaño 8 no se resuelve por su dificultad.

Backtraking		
Tablero	**Saltos BT**	**Tiempo (ms)**
4x4	16	0,019
5x5	7362	4,975
6x6	129	0,168
7x7 (1)	399936	311,115
7x7 (2)	975083	751,97
8	-	-
9x9 (1)	-	-
9x9 (2)	1812689	1745,96

Tabla 1. Resultados backtracking

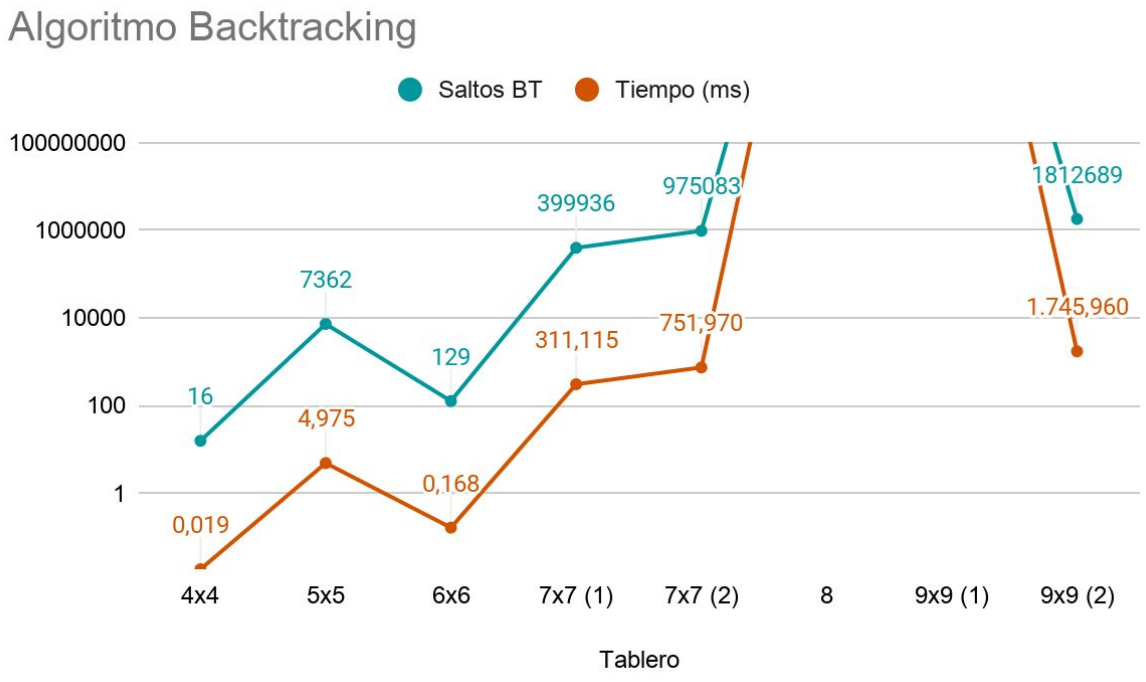


Figura 3. Gráfica de resultados backtracking

Propagación de restricciones - AC3

El problema de backtracking es que es una estrategia de búsqueda que en el peor de los casos expande todo el árbol de soluciones. Para ello, antes de aplicar backtracking se aplica un método de propagación de restricciones, en este caso, AC3. Este método analiza las aristas del problema, es decir, todas las restricciones binarias que tienen que cumplirse entre las casillas, y elimina los valores de los dominios de una casilla que causan inconsistencias de arista con la otra casilla que forma la arista.

Una arista es consistente si para cada valor del dominio del nodo inicial existe al menos un valor en el dominio del nodo final que cumple las restricciones. Por lo tanto, si eliminamos los valores de los dominios que crean inconsistencias de arista estamos transformando el problema en una red consistente que es mucho más fácil de resolver.

Código del algoritmo

El algoritmo se implementa directamente en el método de la clase *Solver* proporcionado, aunque necesita de otras funciones auxiliares para que el código quede más claro y modulado.

Cómo en backtracking, lo primero es decidir de qué forma vamos a codificar ciertas estructuras. Este método necesita almacenar en un conjunto las aristas, por lo que tenemos que codificar el conjunto y las aristas.

Para codificar el concepto de arista, primero creamos un *struct* Casilla, para poder identificar las casillas por su fila y su columna. Luego, definimos otro *struct* Arista como un valor tipo Casilla inicial y otro final.

```
struct Casilla {
    int fil;
    int col;
};

struct Arista {
    Casilla inicial;
    Casilla final;
};
```

De esta forma, ya podemos crear una lista con la clase estándar de C++ *list* de variables tipo Arista.

Solver::inicializarQ

Es un método auxiliar que añade a un conjunto todas las aristas del problema. Recibe como parámetros una lista de tipo Arista y el tablero del problema. El conjunto en el que se almacenan las aristas del problema es una lista de C++ del tipo Arista que se ha creado anteriormente.

Básicamente, recorre todo el tablero, y asigna cada casilla como el nodo inicial de una arista. Luego se recorre la fila y la columna de la casilla actual, se establece cada una de esas casillas como nodo final y se añade la arista al conjunto Q. No hace falta recorrer todo el tablero porque una casilla cualquiera solo tiene restricciones con el resto de casillas de su fila y de su columna. Hay que tener en cuenta que las restricciones son binarias, por lo que hay que comprobar que no estamos añadiendo una arista cuyo nodo inicial y final son el mismo.

Solver::comprobarRestriccionesBinarias

Este método auxiliar comprueba si entre el nodo inicial y final de una arista existe una restricción binaria y si se cumple. Recibe como parámetros la arista en cuestión y el tablero del problema, y devuelve un booleano a true si cumple las restricciones.

Hay que comprobar si los nodos que forman la arista son adyacentes. La función realiza cuatro comprobaciones: si el nodo final está arriba del inicial, llama al método de tablero que comprueba las restricciones binarias verticales con la casilla del nodo final; si está abajo, con la casilla del nodo inicial; si está a la izquierda, llama al método de tablero que comprueba las restricciones binarias horizontales con la casilla del nodo final; y si está a la derecha, con la casilla del nodo inicial. Para cada uno de estos casos se crea una variable booleana a true, de forma que para el caso en el que no haya restricción binaria o las casillas no sean adyacentes se devuelve true. Sólo se modifican estas variables si hay una restricción binaria que no se cumple. Para que la función devuelva true, todos estos booleanos tienen que estar a true.

Solver::comprobarConsist

Este método comprueba la consistencia de arista para un valor a del dominio del nodo inicial. Recibe como parámetros la arista, el valor del dominio del nodo inicial y el tablero, y devuelve true si la arista es consistente para el valor del dominio del nodo inicial.

Vamos a comprobar la consistencia llamando a la función anterior, que utiliza métodos de la clase tablero. Por ello, primero almacenamos los valores que tenían las casillas iniciales y finales en el tablero antes de poner los valores del dominio y comprobar la consistencia. En la casilla inicial simplemente ponemos el parámetro a, y en la casilla final se pone cada valor del dominio de dicha casilla.

Para cada valor del dominio de la casilla final se comprueba que el valor de la casilla inicial y final sean distintos para cumplir las restricciones de fila y columna, y si las cumplen se

llama a la función anterior para comprobar las restricciones binarias. Se restauran los valores iniciales del tablero y se devuelve true en el caso de que la arista sea consistente para ese valor del dominio del nodo inicial.

Solver::ejecutarAC3

En este método de la clase solver ejecutamos el algoritmo principal de propagación de restricciones. Únicamente recibe como parámetro el tablero del problema.

En primer lugar, llamamos a las funciones que inicializan el dominio de cada casilla del tablero y el conjunto de aristas del problema. Después, empieza el bucle principal de resolución. Mientras el conjunto de aristas *Q* no esté vacío, extraemos una arista y la almacenamos en una lista auxiliar, *elimQ*, para buscar aristas que puedan ser restauradas más adelante.

Una vez ya hemos eliminado la arista de *Q* y la hemos añadido a *elimQ*, comprobamos primero si es una casilla vacía o no. Si no lo es, simplemente se extrae otra arista de Q sin hacer nada, ya que ese dominio no se puede modificar. En el caso de que no lo sea, recorremos el dominio del nodo inicial de la arista y comprobamos su consistencia con la función anterior. Si para un valor del dominio del nodo inicial no hay consistencia de arista, ese valor se elimina del dominio y ponemos a true una variable booleana que indica si hemos modificado el dominio de un nodo. En el bucle que recorre el dominio hay que tener en cuenta que si eliminamos un valor del dominio, hay que restarle una unidad a la variable que estemos utilizando para recorrer el dominio para no saltarnos ningún valor del dominio.

Cuando terminamos el bucle que recorre el dominio del nodo inicial, primero comprobamos si el dominio se ha quedado vacío. Si esto sucede, significa que el problema es inconsistente y no tiene solución, por lo que se acaba la función. Después, comprobamos con la variable booleana si hemos modificado el dominio del nodo inicial. Si lo hemos modificado puede repercutir en otras aristas, de forma que dejen de ser consistentes, y hay que volver a comprobarlas. Por lo tanto, recorremos la lista *elimQ* y añadimos a *Q* las aristas diferentes a la actual cuyo nodo final sea el nodo inicial actual.

Cuando se acaba el bucle principal y el conjunto *Q* se queda vacío ya hemos transformado el problema en una red consistente de aristas y se puede resolver de forma más rápida por alguna estrategia de búsqueda, en este caso, backtracking. Esta función no llama al método *ejecutarBT*, sino que se tiene que llamar desde la ventana de la aplicación primero a resolver con AC3 y luego con backtracking.

Tests

Para comprobar el funcionamiento del algoritmo de propagación de restricciones AC3, vamos a resolver los mismos tableros que antes. Justo después de cargar un tablero tenemos que darle a la opción de ejecutar AC3, y obtendremos como salida por el terminal la matriz de dominios resultante una vez que se han eliminado los valores que crean inconsistencia de arista. Sin embargo, AC3, no resuelve el problema, por lo que una vez

que hemos modificado los dominios tenemos que ejecutar otra vez backtracking. Podemos comprobar que la propagación de restricciones reduce los dominios hasta resolver el problema en algunos casos, como el tablero de tamaño 5 o el de tamaño 6. En el resto de casos, reduce considerablemente el número de saltos hacia atrás que hacía backtracking cuando existían inconsistencias de arista. Para una comparación más detallada de los resultados, véase el siguiente punto. Aplicando AC3 podemos resolver tableros que únicamente por backtracking no podíamos, como el tablero 8 o el 9.1.

AC3 + BT		
Tablero	**Saltos BT**	**Tiempo (ms)**
4x4	9	0,051
5x5	0	0,012
6x6	0	0,015
7x7 (1)	294194	161,997
7x7 (2)	27353	15,011
8	947427584	623832
9x9 (1)	4206432	2809,25
9x9 (2)	315767	204,378

Tabla 2. Resultados AC3 + backtracking

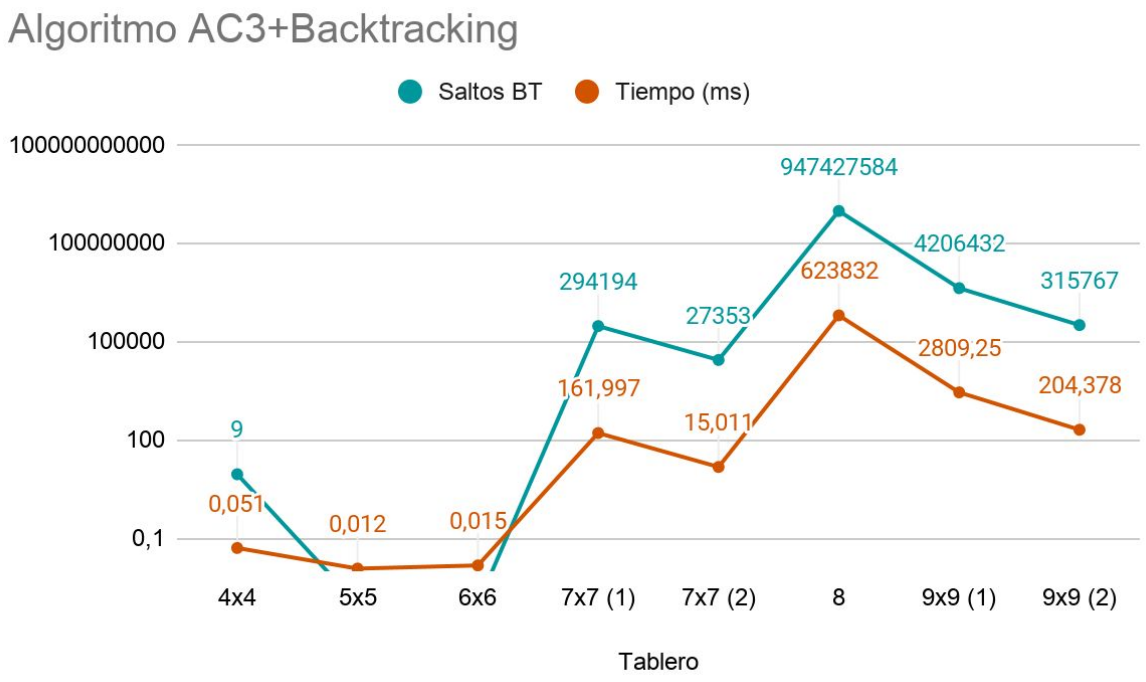


Figura 3. Gráfica de resultados AC3 + backtracking

Comparación de algoritmos

En las siguientes imágenes se muestran las capturas de pantalla de las soluciones obtenidas de los tableros anteriores:

Figura 4. Solución tablero 4

Figura 5. Solución tablero 5

Figura 6. Solución tablero 6

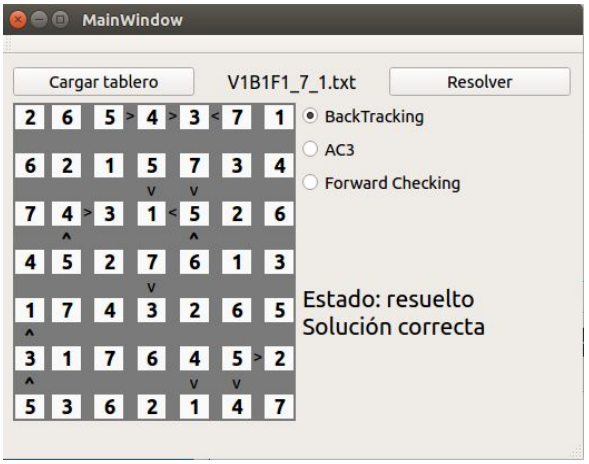


Figura 7. Solución tablero 7.1

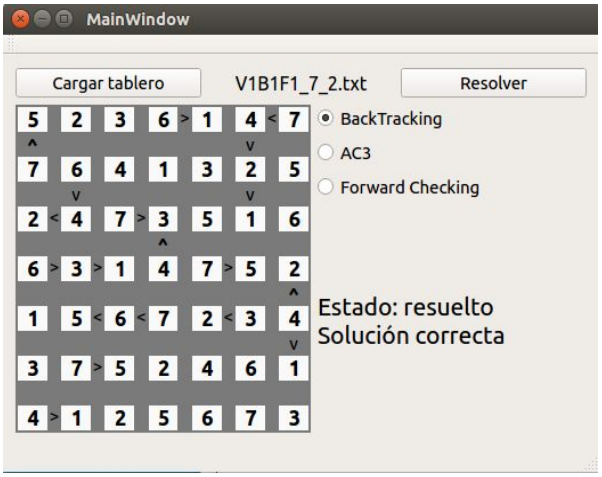


Figura 8. Solución tablero 7.2

Figura 9. Solución tablero 8

Figura 10. Solución tablero 9.1

Figura 11. Solución tablero 9.2

Acorde a los resultados obtenidos anteriormente, que se recogen en las dos gráficas siguientes, podemos observar que utilizar un algoritmo de propagación de restricciones cómo AC3 mejora el funcionamiento de un algoritmo de estrategia de búsqueda cómo backtracking, ya que en sí la ejecución de AC3 no supone un gran coste temporal, y más si se utiliza una implementación por listas antes que utilizando la clase vector de C++, como en este caso.

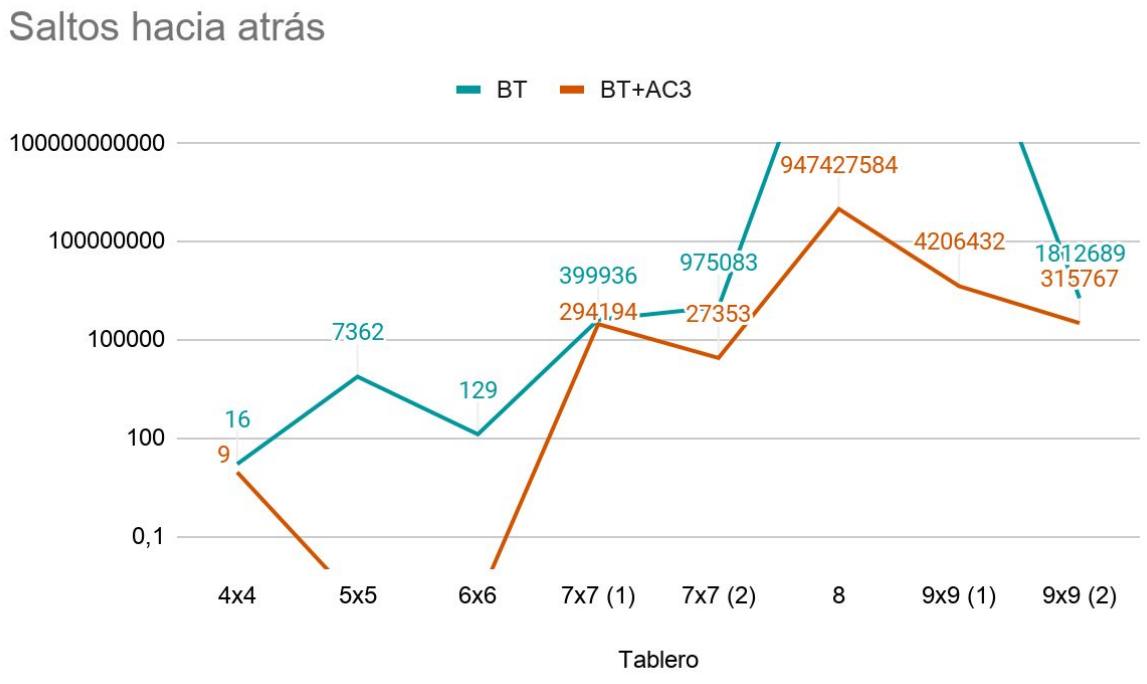


Figura 12. Gráfica de comparación de saltos hacia atrás de los algoritmos

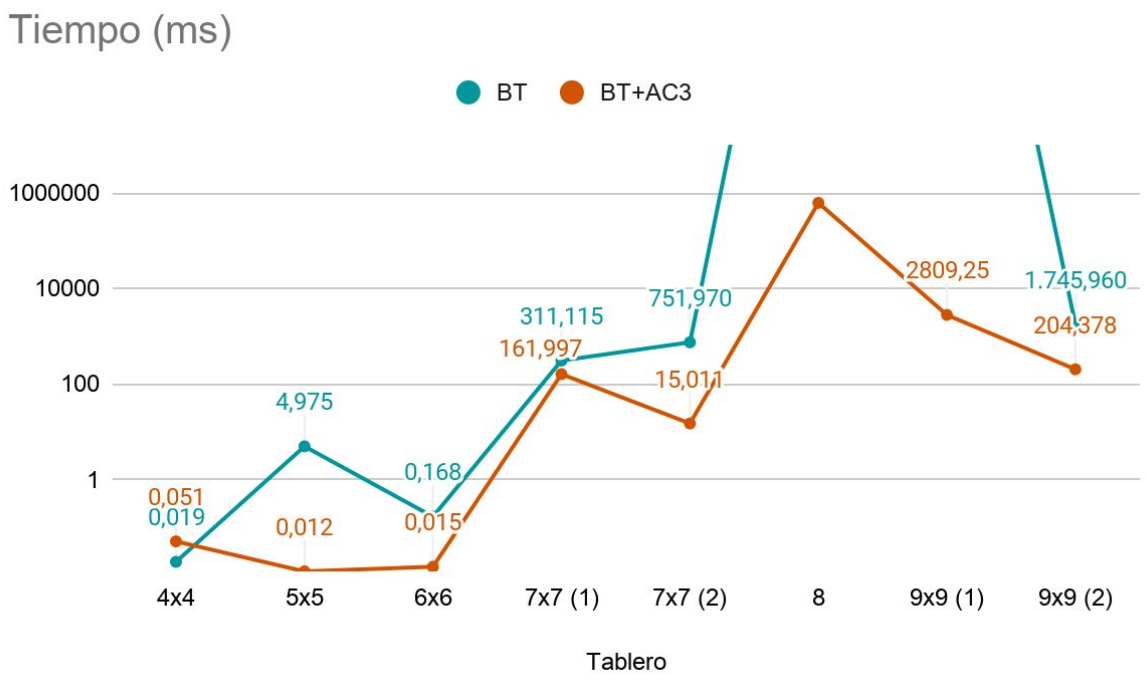


Figura 13. Gráfica de comparación de tiempo de ejecución de los algoritmos

Conclusiones

Al comparar los resultados obtenidos hemos podido observar que para mejorar el funcionamiento de una estrategia de búsqueda cómo backtracking, que como se ha mencionado antes explora todo el árbol de soluciones en el peor de los casos, implementar un algoritmo de propagación de restricciones disminuye los dominios y por tanto reduce el número de saltos hacia atrás de backtracking, ya que elimina las inconsistencias de arista.

Sin embargo, en estas implementaciones seguimos teniendo un componente que se establece de forma manual y que puede ser determinante en la resolución del problema, la permutación a que define el orden de selección para resolver el problema, en otras palabras, en qué orden recorremos las casillas del tablero para encontrar la solución. Depende de la instancia concreta del problema, unas permutaciones funcionan mejor que otras.