Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructuras de Datos y Algoritmos

Tarea 2

26 de mayo de 2018

1er semestre 2018 - Profesor F. Yadran

Ignacio Tomas Hermosilla Cornejo - 1362072J

Analisis empírico

Para realizar el análisis empírico de tiempo, cree un contador para ver cuantas veces hacía designaciónes, en particular si vemos en mi código, podemos apreciar que

```
1
2 bool backtracking(Board *board)
3 {
4   ...
5   board_set_status(board, cell->row, cell->col, 0); // backtrack
6   backtracks += 1;
7   return false;
8 }
```

justo después de retornar la llamada y deshacer la designación, aumentamos el contador backtrack en 1, esto es un buen indicador de la complejidad cunto con el tiempo, ya que nos dice cuantas veces nos equivocamos al asignar y será lo que estudiaremos principalmente, ya que lo considero una mejor métrica que el tiempo, el cual va a variar de computador en computador, en base a la velocidad del procesador.
Podemos revisar esta propiedad con un poco de trabajo. Para el archivo *easy_5.txt*, solo existe una posible solución y si sabemos el orden de asignación, podemos calcular cuantas veces se realiza el backtrack.
Ese archivo tiene un tamaño de 5x5, por lo que solo tenemo que hacer 2^9 asignaciones, es decir 512 asignaciones en el peor de los casos. Si partimos de arriba hacia abajo, de izquierda a derecha, asignando celdas leales (leales al imperio) primero, solo cometerá 2 equivocaciones al principio, por lo que tendremos recorreremos 3/4 del arbol, es decir realizaríamo 384 backtracks, y revisando nuestro contador, tenemos que realizamos 382 backtracks.

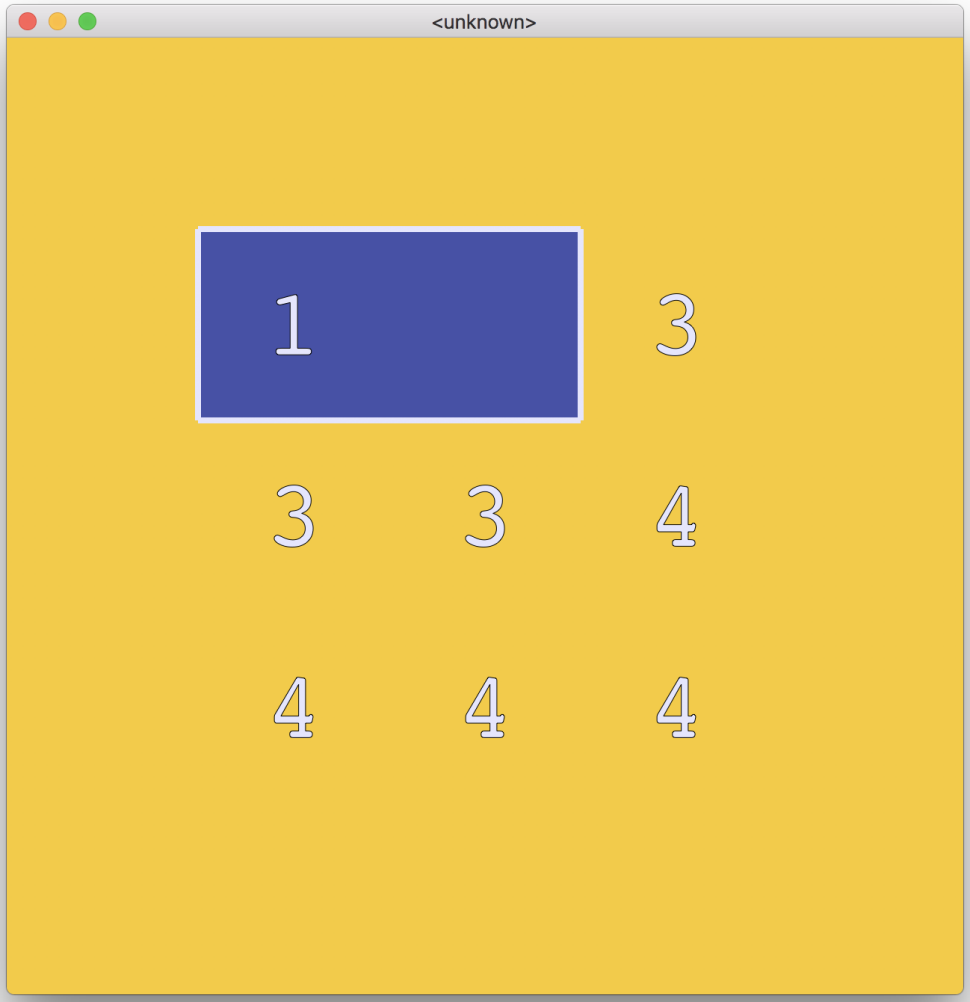

Figura 1: Solución del archivo easy_5.txt

Backtracking

Partiremos nuestro análisis émpirico midiendo los tiempos y performance de nuestro algoritmo para cuando no implementamos ninguna de las dos podas, lo que implica que las asignaciones la realizará bien, ya que podemos ir chequeando la restricción de los vecinos, pero debemos esparar a que todas las celdas esten asignadas para poder revisar si cumple con la restricción de los grupos, por lo que es sumamente ineficiente. Por eso mismo tuve que basar mis analisis en los ejemplos faciles, cosa que tuvieran tiempo de terminar. Ahora, para una cuadrilla de 7x7, estamos hablando de 2^{49} asignacione, lo que es demasiado y me daría timeout, por lo que

cree mis propios ejemplos para poder estudiar el comportamiento a medida que cambia el n. Utilicé el mismo ejemplo easy_5.txt pero realizando yo asignaciones cosa que el backtracking tuviera que buscar una celda menos, es decir, disminuir a la mitad el númeor de asignaciones. Por ejemplo, $n = 8$, tenemos el siguiente caso antes de comenzar:

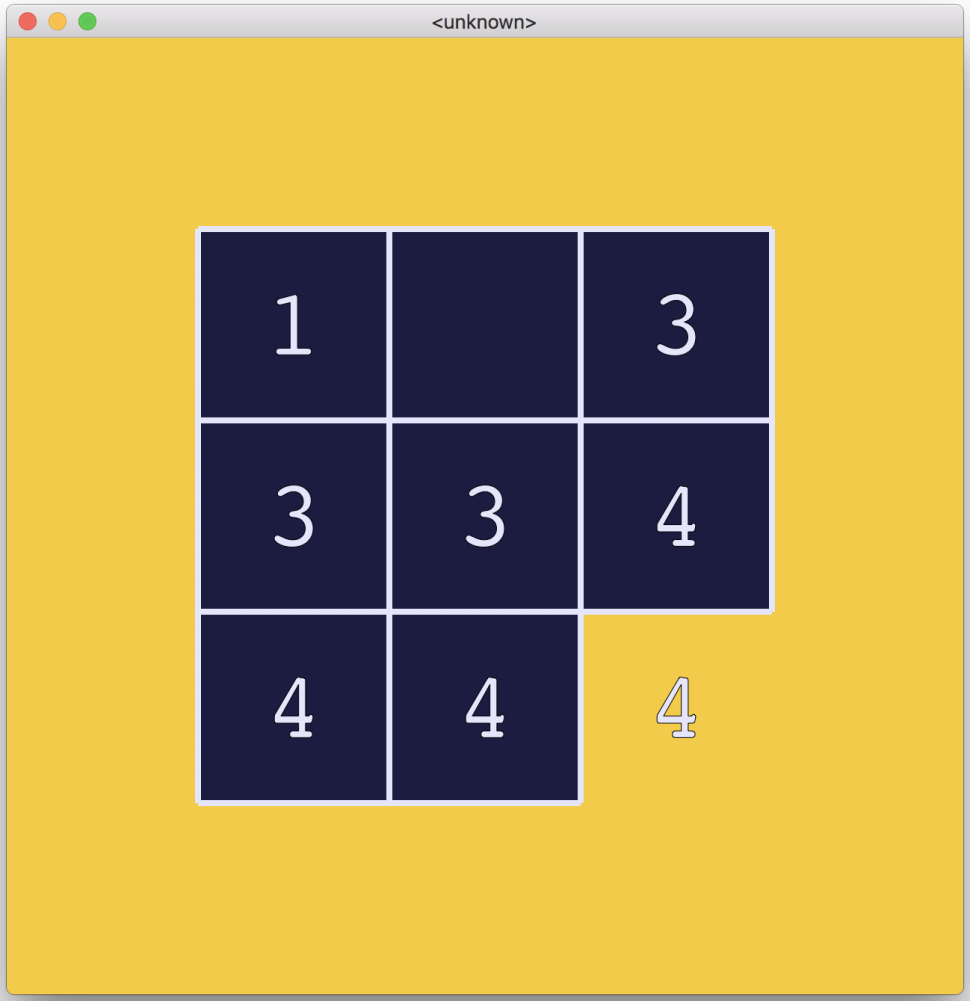

Figura 2: Archivo easy_5.txt con solo 8 celdas para asignar

Los resultados obtenidos son los siguientes y se muestran en la tabla y grafico a continuación

n	número de asignaciones	número de backtracks	tiempo (segundos)
9	512	382	0.19
8	256	190	0.16
7	128	94	0.13
6	64	46	0.10
5	32	10	0.9

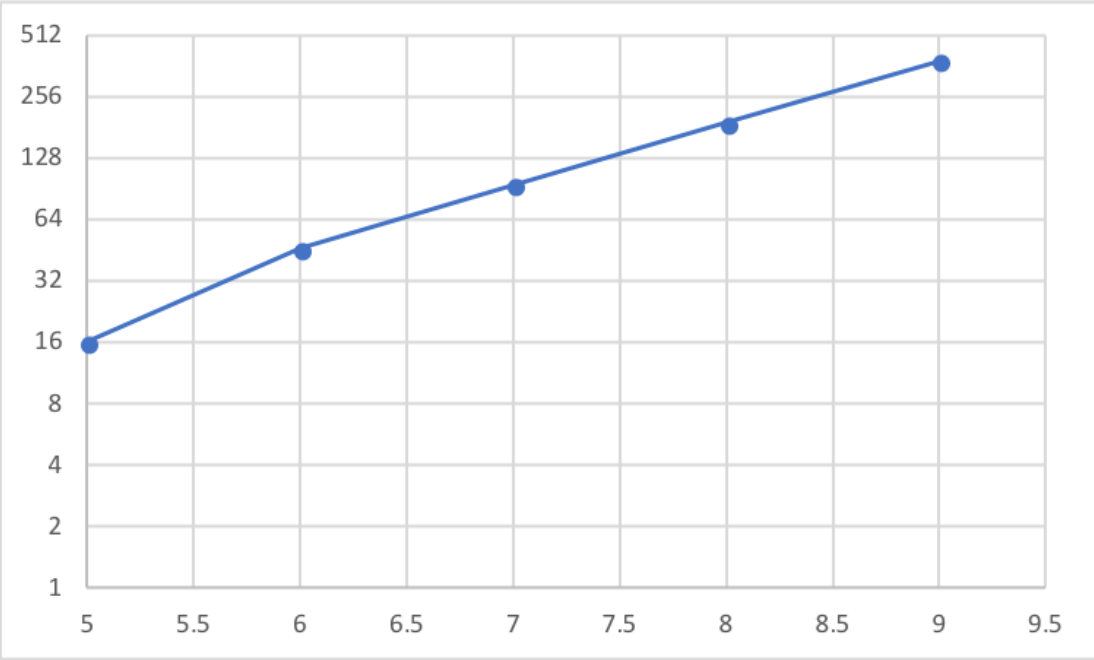

Figura 3: Crecimiento de backtracks vs n, escala logarítmica

Como podemos ver del grafico y la tabla, el crecimiento del numero de backtracks realizado es claramente exponencial, de hecho, la linea en la figura 3 esta ajustada al logarítmo de 2 y nos da casi una linea recta, lo que confirma nuestra teoría.

Backtracking con podas

Ahora con las podas, no tiene sentido utlizar el mismo ejemplo que antes, ya que nos da un tiempo constante, por lo que utlizaremos ejemplos más grandes

archivo	n	número de backtracks	tiempo (segundos)
easy_1.txt	2^{49}	125	0.20
normal_1.txt	2^{100}	3.231	0.76
hard_1.txt	2^{100}	11.224	1.96

Acá podemos ver que claramente el crecimiento del numero de backtracks disminuye exponencialmente. Interesantemente, podemos ver que dentro del mismo numero de asignaciones, el numero de backtracks parece cambiar, lo que tiene que ver mucho con la suerte ya que equivocarse en las primeras asignaciones afecta mucho más que en las últimas, ya que es mas caro volver. Entonces, para poder hacer un estudio más serio, vamos a repetir la técnica anterior, pero esta vez para el ejemplo *hard_1.txt*, cuya solución podemos ver acontinuación, en la figura 4

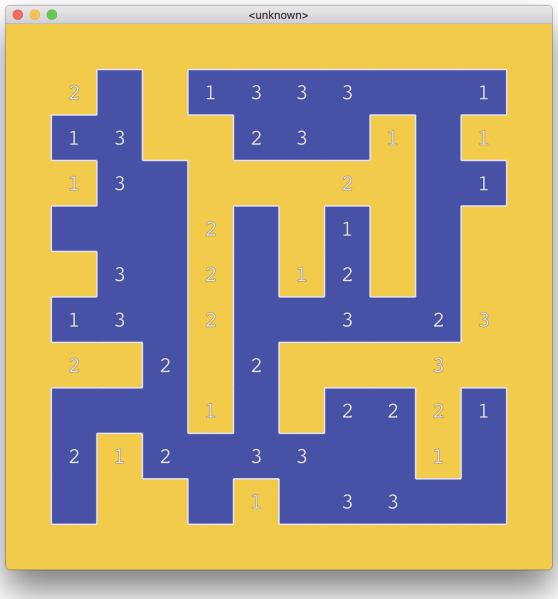

Figura 4: Solución del problema hard_1.txt

Para ir bajando el numero de n, iremos marcando celdas con la solución antes de iniciar el backtrack, como se muestra en la figura 5

Ahora, corriendo nuestro análisis, obtenemos los siguientes resultados

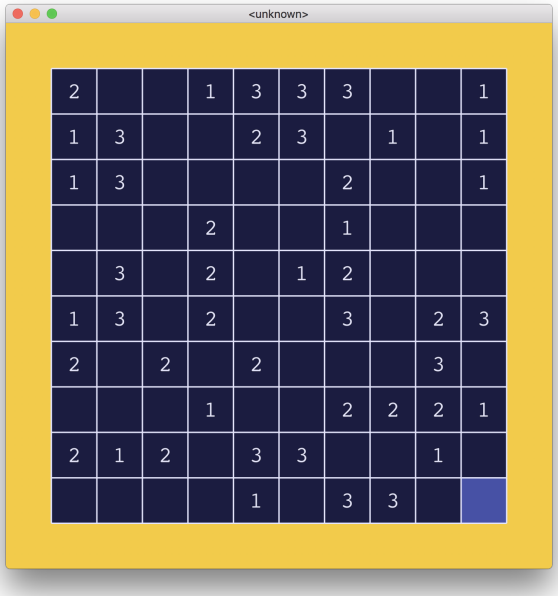

Figura 5: Problema hard_1.txt con una casilla menos

n	número de asignaciones	número de backtracks	tiempo (segundos)
49	2^{49}	11.224	2.83
48	2^{48}	11.222	2.31
47	2^{47}	11.184	2.11
46	2^{46}	10.926	2.07
45	2^{45}	10.667	1.84

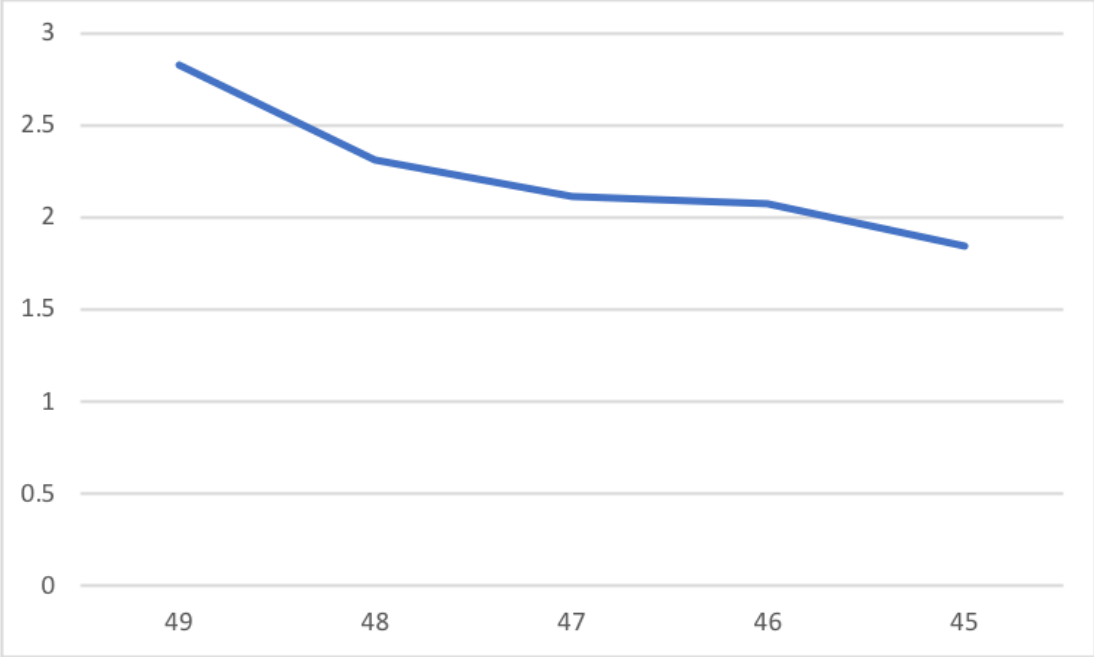

Figura 6: Tiempo vs numero de asignaciones

Vemos que con las podas, el comportamiento está mucho mas cerca de ser lineal segun la figura 6 que sin las podas, lo que tiene sentido, ya que le es más dificil al numero de backtracks explotar, al ser podadas la gran mayoriía de ellas. Si lo pensamos, si resolvieramos para $n = 49$, por fuerza bruta, estaríamos hablando de 2^{49} asignaciones, por lo que si tuvimos que devolvernos 11.224 veces, estamos hablando de un infima parte que se aproxima al 0 %

Poda adicional

Ahora, existen una infinidad de podas adicionales que se pueden pensar en, pero para cada una, hay que pensar en el tiempo de ejecución y cuantas podas realmente hacen como para ver si es que valen la pena implementar. Yo propongo la siguiente poda, si es que tenemos dos diagonales de colores opuestos, como se puede apreciar en la figura 7

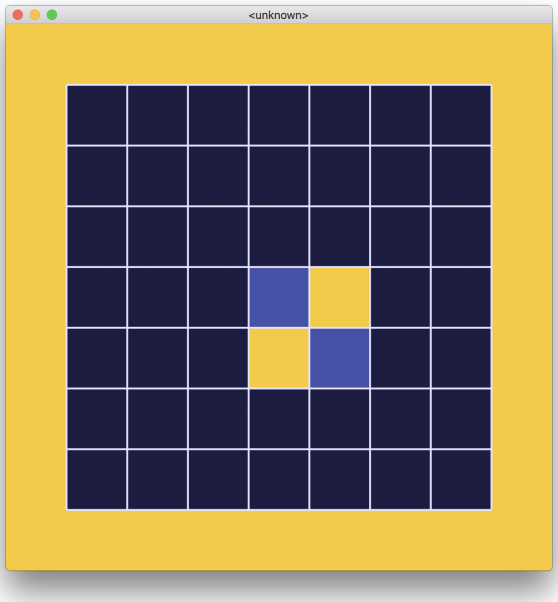

Figura 7: Ejemplo de una nueva poda

Esta poda, la cual viene de las restricción de los grupos nos permite darnos cuenta de que no será posible satisfacer la restricción más adelante y devolvernos. De hecho, si pensamos por que esto es así, podemos dar cuenta que para conectar las dos casillas azules, necesariamente vamos a tener que rodear una de las casillas amarillas, por lo que las casillas amarillas van a formar dos grupos separados, incumpliendo la restricción de los grupos. Esta poda es bastante facil de implentar, y eficiente y tambien, ya que no requiere ver a todo el tablero para revisarla despues de cada asignación, si no que solo mirar las 8 casillas vecinas de una después de que esta es asignada.

Propagación

Para propagar, hay varios ejemplos que saltan a la mente, pero mencionaré lo mas sencillos. Por ejemplo, si realizamo una asignación a una celda que es de grado 4, o vecina de una, necesariamente todos sus vecinos deben ser del mismo grado, por lo que podemos propagar inmediamente la asignación a sus vecinos. Por ejemplo, se puede ver en la figura que las 4 celdas deben tomar el mismo color

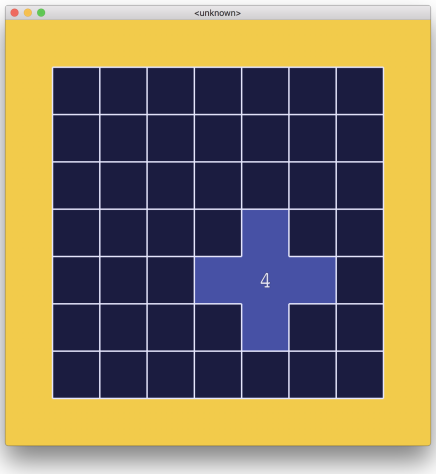

Figura 8: Ejemplo de propagación, todas las celdas vecinas deben ser azules

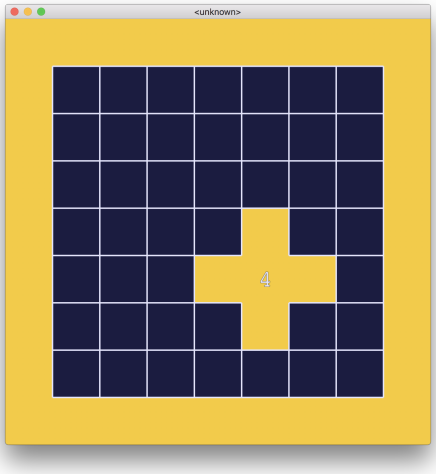

Figura 9: Ejemplo de propagación, todas las celdas vecinas deben ser amarillas

Así como es el caso, hay muchas otras propagaciónes posibles y no voy a entrar en detalles, ya que son muchos casos posibles, eso si, todas son producto de las restricción de los vecinos. Una posible propagación de la restricción de los grupos es la siguiente, vista en la figura 10

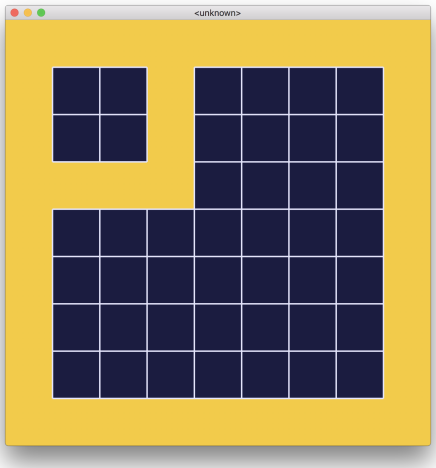

Figura 10: Ejemplo de propagació, uno de los dos grupos en blanco debe ser amarillo o azul

Ahora, en la práctica esta segunda propagación es muy dificil de lograr y cara de implementar, por lo que no es si sería recomendablem ya que no creo que pase muy seguido. La primera propagación es más práctica, ya que sucede mucho y disminuye enormemente la complejidad de la solución, cada asignación hecha a partir de una propagación disminuye por 2 el número de asignacione restantes a revisar, por lo que definitavamente vale la pena hacerla. Además, es implementable recursivamente, lo que tiene la ventaja que una propagación puede llevar a otra y a otra y así.

Heurística

Dado que estamos haciendo propagación, una heurítica que salta inmediatamente ordenar las casillas por el grado, de mayor grado a menor grado, lo que inmediatamente no daría las propagaciones más importantes que son las de grado 4 y 3 al inicio del arbol, disminuyendo en ordenes de magnitud la solución. Sin embargo, tambien vale la pena mencionar otra heuristica que es un poco mejor que es calcular para cada celda una prioridad, de 1 a 2, donde 1 es que esa celda solo se puede poner de una forma y 2 es que se puede poner de ambas formas. La ventaja de esta heurística es que tiene la propagación de los vecinos que mencionamos anteriormente implicita. Además, es facilmente implementable, ya que se pueden tener 2 stacks, uno por prioridad y al momento de cambiar una celda, es cosa de sacarlo de un stack y ponerlo en el otro, con nuestro algorimo siempre asignando primero a las celdas en el stack de prioridad 1 y despues en el stack de 2. Esto, a parte de ser muy facil dei implementar, tiene la ventaja de que no requiere como en la propagación que mencionamos en la sección anterior, recordar todas las asignaciones hechas producto de una propagación, en caso de que tengamos que desacerlas, ya que la recursión del backtrack se haría cargo esto.