HEURÍSTICA Y OPTIMIZACIÓN

Satisfacción de restricciones y heurísticas

Carlos Contreras Sanz	100303562
Álvaro Gómez Ramos	100307009

14 DE DICIEMBRE DE 2014

UNIVERSIDAD CARLOS III DE MADRID

Contenido

En este documento se procederá a describir los modelos y explicar las decisiones tomadas a la hora de resolver el cuadrado mágico, el sudoku y el hidato, todos como satisfacción de restricciones y el último además como búsqueda heurística.

1. Cuadrado Mágico

Si tenemos un tablero de tamaño N filas por N columnas, se pide hacer que todas las filas, columnas y diagonales principales sumen lo mismo.

1.1 Modelado

- **Variables**: representan cada una de las casillas del tablero, total de NxN
  - $X = \{ x_{ij} \}_{i=1,j=1}^{n}$
- **Dominio**: representan los valores posibles de cada casilla, desde 1 hasta el número de casillas.
  - $D = \left\{ d_{ij} = y \ \in \{1, n^2\} \right\}_{i=1,j=1}^{n}$
- **Restricciones**: representan que cosas deben cumplirse.
  - $C = \{ C_1 = \{ \forall_i \ \sum_{j=1}^{n} x_{i,j} = t \},$ para las filas
  - $C_2 = \{ \forall_j \ \sum_{i=1}^{n} x_{i,j} = t \},$ para las columnas
  - $C_3 = \{ \sum_{i=1}^{n} x_{i,i} = t \},$ para la diagonal principal
  - $C_4 = \{ \sum_{i=n,j=1}^{i=1,j=n} x_{i,j} = t \}$ diagonal opuesta
  - $\}$

1.2 Implementación

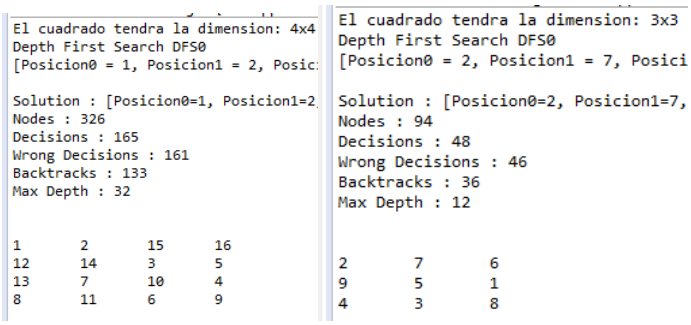
A la hora de implementar, lo primero ha sido comprobar el número de parámetros. Suponiendo que el parámetro es un número, se procede a la ejecución:

Se crea el tablero (array unidimensional de IntVar), y se inicializan las casillas, dando como dominio de las variables (cada una de las casillas) desde 1 hasta el número de casillas (NxN).

Se añade la restricción de que deben de ser todos los números distintos con un alldistinct (se usa array unidimensional en el tablero ya que es lo que hay que pasar al impose para crear las restricciones).

Después se crea un vector, que iremos reutilizando y rellenando con casillas sobre la que aplicar las restricciones modeladas. Cada vez que se rellena se hace un impose para añadir la restricción (restricción de suma de valores, todos iguales a una variable “suma”).

Tras esto se procede a resolver. Por ejemplo para los casos de N=4 y N=3:



Si analizamos los resultados, vemos que son correctos, en todas las ejecuciones del programa Además no se nos ha ocurrido plantear casos de prueba más allá de probar con 3 y 4 de dimensión, ya que el de 5 filas/columnas tardaba bastante y cancelamos la ejecución.

2. Sudoku

Se nos dará un tablero de 9x9, donde hay que rellenar filas, columnas y subcuadrados de 3x3 con números del 1 al 9 sin repetirse.

2.1 Modelado

- **Variables**: cada una de las casillas del tablero
  - $X = \{ x_{ij} \}_{i=1,j=1}^{9}$
- **Dominio**: valores de cada casilla van desde 1 a 9
  - $D = \left\{ d_{ij} = y \in \{1,9\} \right\}_{i=1,j=1}^{n}$
  - Excepto cuando nos pasan un número fijo del sudoku a resolver, entonces seria: $d_{ij} = y$
- **Restricciones**:
  - $C = \{ C_1 = \{ \forall_{i,j,k}\, x_{i,j} \neq x_{i,k} \},$ para las filas,
  - $C_2 = \{ \forall_{i,j,k}\, x_{i,j} \neq x_{k,j} \},$ para las columnas
  - $C_3 = \{ x_{i_j}, x_{i_{j+1}}, x_{i_{j+2}}, x_{i+1_j}, x_{i+1_{j+1}}, x_{i+1_{j+2}}, x_{i+2_j}, x_{i+2_{j+1}}, x_{i+2_{j+2}} \} \forall_{i,j}\ i,j = 0,3,6$ para los subcuadrados

2.2 Implementación

En este caso, al igual que en el anterior lo primero comprobar que tenemos un parámetro, y suponer que es una ruta válida a un fichero válido.

Tas eso se lee el fichero, y se crea un tablero unidimensional de IntVar (arrastramos la forma de representar el tablero del ejercicio anterior, no tiene ninguna ventaja reseñable). Se inicializan también las casillas, o bien con todo el dominio posible (1…9) o bien con solo un número en el dominio (ya nos proporcionaban un número en esa casilla).

Tras eso se imprime en tablero y se empieza con las restricciones de la misma forma que en el ejercicio anterior para las filas y columnas. La de los subcuadrados de 3x3 se hace con módulos de 3, y en cada iteración del bucle se añaden 3 casillas (las que serían consecutivas en fila en un tablero bidimensional). Todas estas restricciones serían de alldistinct.

Tras eso simplemente se llama al método para resolver. Por ejemplo para el sudoku dado:

```
5       3       {1..9}  {1..9}  7       {1..9}  {1..9}  {1..9}  {1..9}
6       {1..9}  {1..9}  1       9       5       {1..9}  {1..9}  {1..9}
{1..9}  9       8       {1..9}  {1..9}  {1..9}  {1..9}  6       {1..9}
8       {1..9}  {1..9}  {1..9}  6       {1..9}  {1..9}  {1..9}  3
4       {1..9}  {1..9}  8       {1..9}  3       {1..9}  {1..9}  1
7       {1..9}  {1..9}  {1..9}  2       {1..9}  {1..9}  {1..9}  6
{1..9}  6       {1..9}  {1..9}  {1..9}  {1..9}  2       8       {1..9}
{1..9}  {1..9}  {1..9}  4       1       9       {1..9}  {1..9}  5
{1..9}  {1..9}  {1..9}  {1..9}  8       {1..9}  {1..9}  7       9
```

Los casos de prueba analizados han sido distintos tableros iniciales, llegando en todos los casos a la solución de forma inmediata. No es posible adjuntar dichas pruebas por falta de espacio.

```
Depth First Search DFS0
[Posicion0 = 5, Posicion1

Solution : [Posicion0=5, P
Nodes : 0
Decisions : 0
Wrong Decisions : 0
Backtracks : 0
Max Depth : 0

5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

3. Hidato

Para el hidato se nos dará un archivo conteniendo las dimensiones de un tablero, y el tablero, con números y con casillas no rellenables representadas por #. Se debe encontrar un camino que, partiendo del número menor (siempre 1) llegue al mayor, transitando entre números consecutivos (todos diferentes) que estén en casillas consecutivas, ya sea en vertical, en horizontal o en diagonal

3.1 Modelado

- **Variables**: cada una de las casillas del tablero que no son #
  - $X = \{ x_{ij} \}_{i=1,j=1}^{9}$
- **Dominio**: valores de cada casilla van desde 1 al máximo
  - $D = \left\{ d_{ij} = y \in \{1, Max\} \right\}_{i=1,j=1}^{n}$
  - Excepto cuando nos pasan un número fijo del hidato a resolver, entonces seria: $d_{ij} = y$
- **Restricciones**:
  - $C = \{ C_1 = \{ \forall_{i,j,k,l}\, x_{i,j} \neq x_{k,l} \} \}$ todas las casilla desiguales

3.2 Implementación

Lo primero es comprobar número de argumentos, y suponer su validez. Después pasamos a leer el archivo que contiene dimensiones y tablero, primero dimensiones, creamos tableros y guardándolo en un tablero bidimensional de int (para poder ver cuál es el máximo y el mínimo).

Con ese tablero auxiliar vemos cual será el máximo y el mínimo, y será el que usemos para las comparaciones de números (si hay que añadir restricción o no). Tras eso creamos e inicializamos las variables del tablero de IntVar con el dominio adecuado.

Después recorremos el tablero diciendo para todas las posiciones, que (suponemos que se rellena desde el mínimo) en alguna de las casillas de alrededor debe haber un numero consecutivo a la casilla actual.

Por ultimo como hay que pasar al método para resolver un vector, pasamos la matriz a vector y se resuelve.

Cabe destacar que tarda varios segundos en hacerlo, y eso nos consta que no es lo óptimo. En cualquier caso halla la solución, por ejemplo del problema propuesto por los profesores:

```
{1..40} 33      35      {1..40} {1..40} 0       0       0
{1..40} {1..40} 24      22      {1..40} 0       0       0
{1..40} {1..40} {1..40} 21      {1..40} {1..40} 0       0
{1..40} 26      {1..40} 13      40      11      0       0
27      {1..40} {1..40} {1..40} 9       {1..40} 1       0
0       0       {1..40} {1..40} 18      {1..40} {1..40} 0
0       0       0       0       {1..40} 7       {1..40} {1..40}
0       0       0       0       0       0       5       {1..40}
```

Con la solución:

```
Solution : [Posicion0=32, Posicion0=33, Posicion0=35, Posic
Nodes : 15475878
Decisions : 7737947
Wrong Decisions : 7737931
Backtracks : 7737600
Max Depth : 425


32      33      35      36      37      0       0       0
31      34      24      22      38      0       0       0
30      25      23      21      12      39      0       0
29      26      20      13      40      11      0       0
27      28      14      19      9       10      1       0
0       0       15      16      18      8       2       0
0       0       0       0       17      7       6       3
0       0       0       0       0       0       5       4
```

A pesar de llegar a la solución, expande demasiados nodos, y llega a una profundidad demasiado grande. Sabemos que esto no es del todo correcto, pero por problemas de tiempo, una vez vimos que lograba sacar soluciones tuvimos que continuar, a pesar de saber que es muy mejorable.

En cualquier caso llega a la solución, tardando más o menos. Da igual donde estén las casillas vacías, donde estén las # y donde estén los números fijos. Llega a la solución de forma satisfactoria en cualquier caso.

4. Hidato (Búsqueda Heurística)

Toda búsqueda tiene un Espacio de problemas, formado por el Espacio de estados, el Estado inicial y el Estado final. Por lo que vamos a definir el estado de problemas que tenemos.

4.1 Espacio de Problemas.

4.1.1 Conjunto de estados.

4.1.1.1 Estados.

Los Estados nos muestran toda la información sobre cómo se encuentra el problema tras aplicarse una determinada acción sobre él.

Los modelamos de la siguiente forma:

<Tablero, Máximo, Números en tablero, Posición>

Tablero: array con el que conocemos la distribución de los números en el tablero.

Máximo: Número máximo del tablero, este número siempre lo obtenemos con el estado inicial, por lo que los números que podemos añadir en las casillas vacías del hidato deben de ser siempre menores que el máximo.

Números en tablero: ArrayList en el que introducimos los números que tenemos dentro del tablero, para que de una forma cómoda y poco costosa poder saber si un número está en el tablero o no.

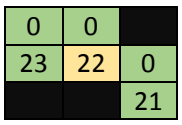
Posición: array de dos posiciones que nos indica la coordenada de la celda del tablero sobre la que vamos a comprobar que acciones aplicarle al estado.

4.1.1.2 Operadores.

Los operadores son el conjunto de acciones que se pueden aplicar a cualquier estado, para generar estados sucesores.

En este problema tenemos que buscar un camino que pase por todas las celdas disponibles del tablero, este camino empezara en el 1 (número mínimo en el tablero) y terminara en el número máximo que contiene el tablero.

Para buscar la solución aplicaremos acciones a los estados. Para saber que acción aplicar hacemos lo siguiente:



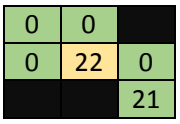
La celda amarilla indica la celda del tablero con las coordenadas que nos indica la variable posición del estado, es decir la celda sobre la que vamos a comprobar que acciones aplicar.

Las celdas negras son celdas que no pueden contener ningún número (Las representamos con el valor -1).

Las celdas verdes son las celdas sobre las que se aplicaran las acciones.

Primero, para ver si aplicamos la acción o no, comprobaremos si en alguna de las celdas de nuestro alrededor existe el valor de nuestra casilla +1, si existe generamos un nuevo estado <Tablero nuevo, Máximo, Números en tablero nuevo, Posición nueva>, teniendo la celda del tablero con el valor de la casilla anterior +1.

En el caso de que no tengamos el valor de nuestra casilla +1, miraremos en el ArrayList del estado si tiene el número que buscamos, si lo tiene, el estado que estábamos comprobando no es válido, ya que no podríamos conseguir el camino entre el mínimo y el máximo dando saltos de uno en uno. Si no lo tiene, crearemos tantas acciones como celdas tenemos alrededor que no tienen valor (son 0).



En este caso crearíamos 4 acciones, que crearán 4 estados.

Entonces las acciones disponibles para cualquier posición son:

Sumar arriba (tablero [posición [0]] [posición [1]]) -> Esto nos pone el valor de la celda del tablero +1 en la celda que está arriba.

Sumar arriba-izquierda (tablero [posición [0]] [posición [1]]) -> Esto nos pone el valor de la celda del tablero +1 en la celda que está en la diagonal arriba-izquierda.

Sumar izquierda (tablero [posición [0]] [posición [1]]) -> Esto nos pone el valor de la celda del tablero +1 en la celda que está en la izquierda.

Sumar abajo-izquierda (tablero [posición [0]] [posición [1]]) -> Esto nos pone el valor de la celda del tablero +1 en la celda que está en la diagonal abajo-izquierda.

Sumar abajo (tablero [posición [0]] [posición [1]]) -> Esto nos pone el valor de la celda del tablero +1 en la celda que está abajo.

Sumar abajo-derecha (tablero [posición [0]] [posición [1]]) -> Esto nos pone el valor de la celda del tablero +1 en la celda que está en la diagonal abajo-derecha.

Sumar derecha (tablero [posición [0]] [posición [1]]) -> Esto nos pone el valor de la celda del tablero +1 en la celda que está en la derecha.

Sumar arriba-derecha (tablero [posición [0]] [posición [1]]) -> Esto nos pone el valor de la celda del tablero +1 en la celda que está en la diagonal arriba-derecha.

Estas acciones se pueden añadir siempre que el valor de la celda donde se va a aplicar sea >-1 y este dentro de los límites del tablero, por lo que si la posición sobre la que queremos ver qué acciones aplicar esta en la primera fila, no podemos añadir ninguna acción de arriba, si está en la primera columna no podemos aplicar ninguna acción de izquierda, igual para la última fila que no se pueden añadir acciones de abajo y para la última columna que no se pueden añadir de derecha.

4.1.2. Estado Inicial.

El estado inicial es el estado con el que iniciamos la búsqueda, en este caso será el tablero que obtendremos por parámetros.

Por lo que el Estado Inicial quedaría así:

<Tablero inicial, Máximo, Números en tablero inicial, Posición inicial>

Siendo la Posición inicial las coordenadas donde se encuentra el número 1.

4.1.3. Estado Final.

El estado final es el estado que buscamos encontrar aplicando acciones sobre los estados.

Por lo que el Estado final quedaría así:

<Tablero final, Máximo, Números en tablero final, Posición final>

Siendo la Posición inicial las coordenadas donde se encuentra el máximo del tablero.

4.2. Funciones Heurísticas.

4.2.1. Heurística 1.

Esta Heurística cuenta los ceros que tiene el tablero, por lo que no es una heurística muy buena, como veremos en las pruebas que realizaremos.

La heurística es Admisible, ya que no sobrevalora el esfuerzo.

4.2.2. Heurística 2.

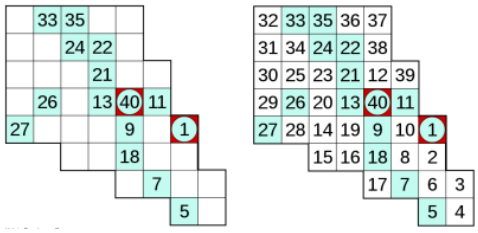
Esta Heurística calcula la distancia entre la celda que nos viene dada por las coordenadas de la posición del estado y las coordenadas de la celda donde se encuentra el máximo.

Comprobamos si ambas celdas se pueden unir con una diagonal, si se puede se devuelve el número de pasos que las separan, si no se pueden unir por una diagonal utilizamos la distancia de manhattan.

No podemos asegurar que esta heurística es Admisible, ya que puede que no se puedan unir dos celdas directamente con una diagonal, pero si movemos cualquier celda una fila o columna puede que se puedan unir mediante una diagonal, siendo el valor de la heurística menor que el valor de la distancia de Manhattan, por lo que puede que se sobrevalore el esfuerzo.

4.3. Pruebas.

Primero probaremos el siguiente hidato:



Este Hidato lo hacemos, obteniendo los siguientes resultados:

Amplitud:

```
Plan Length: 39
Time: 0.063 s
pathCost : 39.0
maxQueueSize : 159
queueSize : 0
nodesExpanded : 800
```

Profundidad:

```
Plan Length: 39
Time: 0.043 s
pathCost : 39.0
maxQueueSize : 21
queueSize : 19
nodesExpanded : 145
```

A* con Heurística 1:

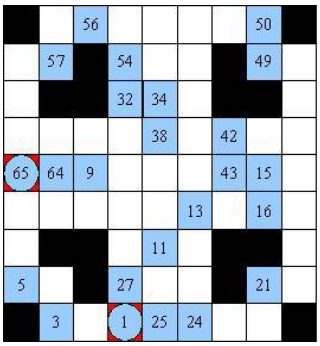
```
Plan Length: 39
Time: 0.121 s
pathCost : 39.0
maxQueueSize : 77
queueSize : 0
nodesExpanded : 800
```

A* con Heurística 2:

```
Plan Length: 39
Time: 0.105 s
pathCost : 39.0
maxQueueSize : 107
queueSize : 30
nodesExpanded : 770
```

Como Podemos ver el mejores resultado que obtenemos es con la búsqueda en profundidad, ya que expandimos 145 nodos, en cambio con el resto de búsquedas expandimos 800 nodos o 770 en el caso de la H2. Los Tiempos empeoran con las heuristicas, ya que añadimos calculos con cada Estado que creamos y comprobamos su heuristica, aunque no se pueden tener mucho en cuenta, porque imprimimos por pantalla algunas trazas para saber como avanza la solucion y puede añadir algo de tiempo. Antes de obtener estos resultados pensabamos que la H1 se pareceria a la busqueda en amplitud, pero tras obtener estos resultados nos dimos cuenta de que como la busqueda en A* no da la solucion hasta que expande el nodo final, tendra que recorrer todos los nodos creados para ver si existe una solucion mejor, por eso expande los mismos nodos que la busqueda en amplitud.

Ahora vamos a probar el siguiente hidato:



Este Hidato lo hacemos, obteniendo los siguientes resultados:

Amplitud:

```
Plan Length: 64
Time: 0.517 s
pathCost : 64.0
maxQueueSize : 939
queueSize : 0
nodesExpanded : 6802
```

Profundidad:

```
Plan Length: 64
Time: 0.535 s
pathCost : 64.0
maxQueueSize : 30
queueSize : 17
nodesExpanded : 5265
```

A* con Heurística 1:

```
Plan Length: 64
Time: 0.63 s
pathCost : 64.0
maxQueueSize : 284
queueSize : 0
nodesExpanded : 6802
```

A* con Heurística 2:

```
Plan Length: 64
Time: 0.529 s
pathCost : 64.0
maxQueueSize : 787
queueSize : 0
nodesExpanded : 6802
```

Como Podemos ver el mejores resultado vuelve a ser la búsqueda en profundidad, ya que expandimos 5265 nodos, en cambio con el resto de búsquedas expandimos 6802 nodos. El peor tiempo es el de la H1, ya que añadimos calculos con cada Estado que creamos y comprobamos su heuristica. En este ejemplo vemos que las heuristicas que hemos escogido en problemas mas grandes no nos aportan demasiada informacion util, siendo iguales a la busqueda en amplitud a la hora de fijarnos en los nodos expandidos.

5. Conclusiones

Esta práctica nos ha valido para ver las diferencias a la hora de implementar los distintos métodos de resolución, y no solo eso, sino que además hemos visto las grandes diferencias en rendimiento en función de cómo se implemente.

Para la implementación del hidato con búsqueda heurística hemos tenido que replantear el problema, ya que nuestra solución original, basada en parte en nuestro modelo e implementación del hidato de satisfacción de restricciones.