El primo raro del Sudoku

Francisco Batista Núñez
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
frabatnun@alum.es.es

Adrián Fernández Fernández
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
adrferfer@alum.us.es

Resolución óptima de puzzles "hitori" de varias dimensiones haciendo uso de algoritmos de búsqueda y explorar el rendimiento de cada uno.

El tiempo empleado en una búsqueda estándar crece exponencialmente con la magnitud del problema, por lo que pronto nos vemos obligados a hacer uso de algoritmos de búsqueda informada.

Hitori, Búsqueda Informada, Inteligencia Artificial.

I. Introducción

Los primeros algoritmos de búsqueda desarrollados (anchura, profundidad) pronto quedaron obsoletos debido al crecimiento exponencial de algunos problemas, viéndonos obligados a idear alguna forma de optimizar dichas búsquedas.

En el puzzle "hitori" puede observarse rápidamente esta necesidad, y es por eso que lo usaremos como punto de partida en la investigación, usando inicialmente los de tamaño mínimo e incrementando sus dimensiones a medida que mejora el tiempo empleado en resolverlo.

Cabe mencionar que, ya que la naturaleza del problema se amolda a una búsqueda de estados, la calidad de la representación del estado es tan importante como la calidad de la búsqueda en sí.

Finalmente, la investigación relacionada al trabajo nos indicará la calidad de la representación del estado definido para el problema.

II. Preliminares

A. Métodos empleados

- Algoritmos de búsqueda

  Se han usado dos tipos de algoritmos, búsqueda estándar y búsqueda informada, siendo búsqueda en anchura y búsqueda en profundidad (estándar, acotada e iterativa) del primer tipo y A* del segundo.

- Técnicas de Resolución del puzzle Hitori: [4]

  Se ha hecho uso diversas técnicas para la resolución del puzzle, siendo alguna de ellas "valor entre pares", por ejemplo.

III. Metodología

Para empezar a trabajar fue necesario comprender el funcionamiento del puzzle. Para familiarizarnos con el mismo, nos dispusimos a resolver algunos a mano con la ayuda de una aplicación online [5].

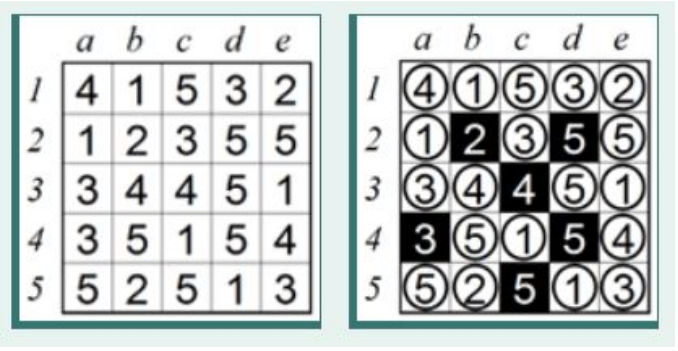

Fig. 1. Ejemplo de un puzzle Hitori resuelto. [4]

Una vez teníamos comprensión de cómo resolver el puzzle y qué técnicas eran eficientes, nos dispusimos a implementarlas:

1) Casilla entre pares: consiste en fijar una casilla situada entre un par de casillas con el mismo valor. De esta técnica se puede deducir otra técnica cuando hay tres valores consecutivos, la cual puede ser útil al resolverlo a mano, pero no es necesaria programáticamente.

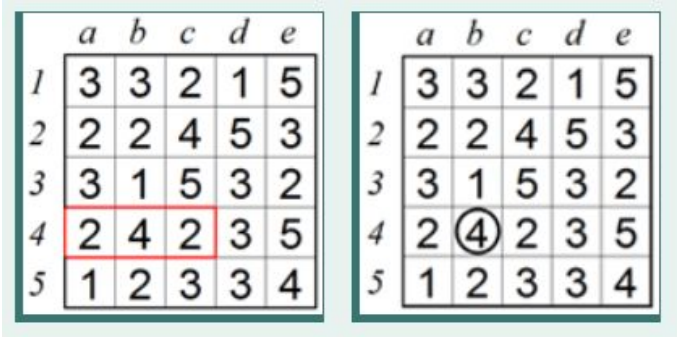

Fig. 2. Casilla entre pares. [4]

2) Suposición de conectividad: esta técnica puede ser algo confusa al principio, pero sin duda es de importancia general a la hora de resolver el puzzle. Dada la restricción de conectividad que impone las normas de Hitori, si al tachar una casilla esta rompe dicha conectividad, entonces debe fijarse.

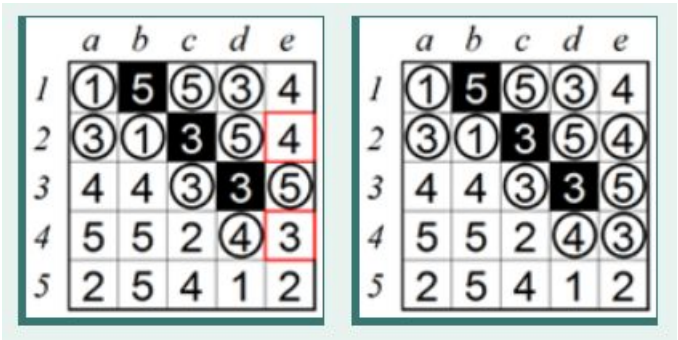

Fig. 3. Suposición de conectividad. [4]

3) Inducción de pares: si exclusivamente dos casillas adyacentes tienen el mismo valor, entonces se deberán tachar las casillas con el mismo valor en la fila o columna de la pareja
("exclusivamente" quiere decir que no es aplicable si la pareja está contenida en un conjunto mayor, como por ejemplo tres casillas consecutivas con el mismo valor).

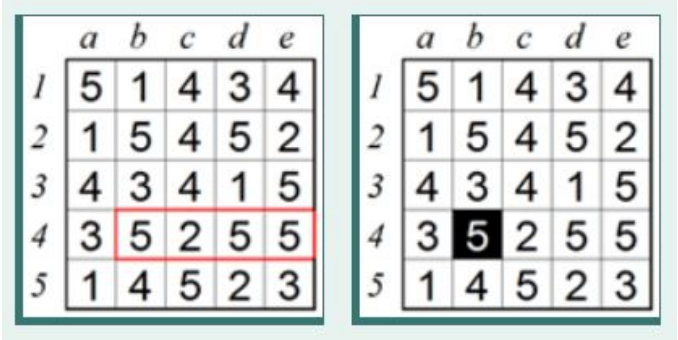

Fig. 4. Inducción de pares. [4]

Hemos mencionado varias técnicas para resolver el puzzle que implican *fijar* o *tachar* una casilla, pero ¿a qué se refieren concretamente? A estas dos acciones se les llama "técnicas básicas", y el problema gira en torno a ellas dos. Además, serán las acciones posibles en la definición de nuestro estado.

4) Fijar una casilla: fijar una casilla implica que todas las casillas con el mismo valor en la misma fila y columna deben tacharse.

5) Tachar una casilla: tachar una casilla implica que todas las casillas adyacentes a la misma deben fijarse.

Una vez aclaradas las técnicas, usadas en la implementación del problema, ilustraremos las partes más relevantes del código desarrollado para el algoritmo, partiendo de la idea de que el estado inicial es la matriz que define al puzzle junto a su número de casillas evaluadas.

Conforme la búsqueda vaya avanzado y analizando la matriz, este número se hará más pequeño. Aunque originalmente esto iba a ser parte del estado final , ya que si es cero entonces a evaluado completamente la matriz, al desarrollar y madurar la idea se detectó que esto no era necesario.

La idea no se ha descartado, pues este valor aún puede sernos de utilidad al calcular la heurística asociada a un estado.

Por tanto, la primera componente del estado (estado[0]) será la matriz, mientras que la segunda (estado[1]) será el número de casillas evaluadas.

1. **FUNCIÓN:** *Contigua No Tachada*

La función devuelve un valor booleano (verdadero o falso) si una de sus casillas adyacentes es cero, lo que indica que está tachada.

Entreda:

- Un **ESTADO** del problema
- Un puntero para **FILA**
- Un puntero para **COLUMNA**

Salida:

- **VERDADERO** o **FALSO**

Algoritmo:

- si **FILA**+1 < longitud(**ESTADO**[0]) y **ESTADO**[0][**FILA**+1][**COLUMNA**] == 0:
  devuelve **FALSO**
- si **FILA**-1 >= 0 y **ESTADO**[0][**FILA**-1][**COLUMNA**] == 0:
  devuelve **FALSO**
- si **COLUMNA**+1 < longitud(**ESTADO**[0][0]) y **ESTADO**[0][**FILA**][**COLUMNA+1**] == 0:
  devuelve **FALSO**
- si **COLUMNA-1** >= 0 y **ESTADO**[0][**FILA**+1][**COLUMNA**] == 0:
  devuelve **FALSO**

- si no: devuelve **VERDADERO**

Fundamental para cumplir las restricciones del puzzle.

2. **FUNCIÓN:** *Valor Repetido*

La función devuelve un valor booleano (verdadero o falso) dependiendo de si alguna casilla en la misma fila o columna tiene el mismo valor.

Entreda:

- Un **ESTADO** del problema
- Un puntero para **FILA**
- Un puntero para **COLUMNA**

Salida:

- **VERDADERO** o **FALSO**

Algoritmo:

- por cada i entre 0 y max_i:
  si **FILA** != i y
  ESTADO[0][i][**COLUMNA**] ==
  ESTADO[0][**FILA**][**COLUMNA**] y != 0:
  devuelve **VERDADERO**
- por cada j entre 0 y max_j:
  si **COLUMNA** != j y
  ESTADO[0][**FILA**][j] ==
  ESTADO[0][**FILA**][**COLUMNA**] y != 0:
  devuelve **VERDADERO**
- si no:
  devuelve **FALSO**

NOTA: max_i y max_j son los tamaños de fila y columna de la matriz

Fundamental para cumplir las restricciones del puzzle.

3. **FUNCIÓN:** *Es Continuo*

La función devuelve un valor booleano (verdadero o falso) dependiendo de si al tachar una casilla se forman ciclos, es decir, el número de componentes conexas de casillas no tachadas es mayor que 1

Entreda:

- Un **ESTADO** del problema
- Un puntero para **FILA**
- Un puntero para **COLUMNA**

Salida:

- **VERDADERO** o **FALSO**

Algoritmo:

- copia en nuevo_estado el **ESTADO**
- nuevo_estado[0][**FILA**][**COLUMNA**] = 0
- si el número de componentes conexas de nuevo_estado[0] == 1
  devuelve **VERDADERO**
- si no:
  devuelve **FALSO**

NOTA: para el cálculo de la componentes conexas se ha hecho uso de la librería externa: scipy, concretamente la función "label"

Fundamental para cumplir las restricciones del puzzle.

4. **FUNCIÓN:** *Pares Inducidos*

La función devuelve un nuevo estado en el cual se ha aplicado la técnica que da nombre a la función.

Entreda:

- El **ESTADO** inicial del problema

Salida:

- Un nuevo **ESTADO** del problema

Algoritmo:

- declarar una variable contador = 0
- por cada valor entre 1 y longitud(**ESTADO**[0]):
  por cada fila entre 0 y longitud(**ESTADO**[0]):
  por cada columna entre 0 y longitud(**ESTADO**[0][0]):
- si columna+1 <= longitud(**ESTADO**[0][0])
  y columna-1 >= 0:
- si **ESTADO**[0][fila][columna] ==
  ESTADO[0][fila][columna+1] !=
  ESTADO[0][fila][columna-1] y
  ESTADO[0][fila][columna] == valor:
- por cada columna2 entre 0 y
  longitud(**ESTADO**[0][0]):
- si **ESTADO**[0][fila][columna2] == valor
  y columna != columna2 != columna2+1 !=
  columna2-1:
  contador = contador+1
  ESTADO[0][fila][columna2] = 0

- transponer la matriz del estado para repetir el proceso, pero para las columnas
- volver a transponer para retomar el orden original
- devolver el nuevo estado compuesto de la nueva matriz y de [**ESTADO**][1]-contador

Fundamental para algunos problemas 6x6 y 9x9

5. FUNCIÓN: *Quitar casilla*

La función pone a cero el valor de la casilla indicada por los punteros de la acción. Aunque al principio es anti-instintivo, está función es usada en ambas acciones del problema, fijar y quitar.

Entreda:

- Un **ESTADO** del problema
- Un puntero para **FILA**
- Un puntero para **COLUMNA**

Salida:

- Un nuevo **ESTADO** del problema

Algoritmo:

- **ESTADO**[0][**FILA**][**COLUMNA**] = 0
- evaluadas = **ESTADO**[1]
- construimos el nuevo estado con las partes definidas anteriormente
- devolvemos el nuevo estado

Fundamental para algunos problemas 6x6 y 9x9

IV. Resultados

Para contrastar los resultados y observar de forma directa la mejora que supone añadir una función de heurística y funciones de coste relacionadas a cada a acción de nuestra representación, hemos medido y comparado los tiempos de algunas matrices del fichero de texto: "ejemplos_prueba.txt".

Tabla.1. Búsqueda Anchura vs. A*

Fila del txt	**Tiempo (en segundos)**		
	Búsqueda en Anchura	***Algoritmo A****	***Mejor***
1	0.010..	0.008..	empate
2	0.011..	0.007..	empate
3	0.009..	0.008..	empate
4	0.021..	0.007..	empate
5	0.023..	0.008..	empate
6	0.032..	0.041..	empate
7	0.219..	0.007..	empate
8	0.189..	0.033..	A*
9	0.548..	0.022..	empate
10	0.675..	0.017..	A*
11	> 60	0.121..	A*
12	> 60	0.141..	A*
13	> 60	0.155..	A*
14	> 60	0.156	A*
15	> 60	0.065..	A*
16	> 60	0.110..	A*
17	> 60	0.321..	A*
18	> 60	0.075..	A*
19	> 60	0.241..	A*
20	> 60	0.462..	A*

Como puede observarse en los tiempos, aún dando la ventaja de empate si ambos son menores a 1 segundo, la búsqueda informada queda obsoleta a partir de tamaño 5x5 del puzzle. Esto ocurre también con las demás búsquedas no informadas, destacando la importancia de una buena heurística y función de coste.

V. Afinando la Heurística y el Coste:

Una vez ya hemos dejado clara la gran ventaja de un algoritmo de búsqueda informada, vamos a centrarnos en explicar el algoritmo que nos a resultado más eficiente, A*, su heurística y sus funciones de coste.

Antes de empezar sería conveniente dejar claro lo que significan "heurística" y "coste" en este contexto.

Definiremos como heurística a la calidad de un nodo, mientras que coste será la penalización que aplicaremos dependiendo de la acción seleccionada.

El algoritmo minimiza la suma de ambos valores, por lo que a mayor heurística y menor coste, mejor será el estado asociado.

Dicho esto y haciendo uso del número de casillas evaluadas (estado[1]), nuestra heurística se ha basado en premiar un mayor número de casillas evaluadas y en penalizar las ramificaciones desde ese estado, es decir, a más casillas evaluadas (menor número en el estado) y menos acciones disponibles tenga el nodo, mejor será el estado.

Respecto a las funciones de coste, se ha definido una para cada acción disponible para el algoritmo, y es la forma en la que este se mueve entre un estado u otro.

Mientras que el coste de tachar siempre va a ser superior al de fijar bajo cualquier circunstancia, hemos usado la función de coste de fijar para distinguir entre posibles acciones.

Por ejemplo, aunque fijar por "casilla entre pares" y fijar una casilla adyacente a una tachada actúan de la misma forma, es mejor que primero use técnicas más seguras. Dicho esto, el orden de prioridad escogido para fijar casillas según sus costes es:

casilla entre pares < contigua no tachada < suposición de conectividad

Esta combinación de heurística y costes ha resultado ser bastante efectiva. Aunque nuestra representación no termina de ser perfecta, es capaz de encontrar la solución de algunas matrices 9x9 sacadas del fichero de texto: "ejemplos_reto.txt" con bastante soltura.

Tabla.2. Tiempos de algunas matrices 9x9

Fila del txt	**Tiempo (en segundos)**	
	Algoritmo A*	***Velocidad***
1	4.0788	alta
2	1.7356	muy alta
3	3.3048	alta
4	indefinido	muy baja
5	2.0805	alta
6	2.7633	alta
7	2.3242	alta
8	3.1373	alta
9	3.5344	alta
10	indefinido	muy baja
11	3.2266	alta
12	3.7670	alta
13	3.0441	alta
14	56.0329	muy baja
15	3.3900	alta
16	4.4538	media
17	4.3235	media
18	3.094	alta
50	4.6845	media
100	2.7753	alta

* *Redondeo al cuarto decimal*

Estudiando detenidamente las matrices que el algoritmo no es capaz de resolver en un tiempo aceptable y buscando el motivo de ello es un buen método para mejorar el algoritmo o encontrar fallas en el mismo.

VI. Conclusiones

Primero haremos un repaso del proyecto en general y del trabajo realizado. Recordemos que, tras una primera fase incial en la que se estaba tomando búsqueda no informada como algoritmo de búsqueda de estados y se estaba haciendo toma de contacto con algoritmos, pronto nos topamos con un bloqueo en la eficiencia.

Tras descartar la representación del estado o la implementación del problema como el motivo o causa de esto, se divisó la posibilidad de haber alcanzado un límite tecnológico con este tipo de búsqueda.

Con esto en mente, se idearon una heurística y unas funciones de coste que al principio no dieron muy buen resultado inicialmente. Después de entender un poco mejor el algoritmo e intentar varias combinaciones, se consiguió la heurística actual del proyecto, la cual parece eficiente.

Tabla.3. Cambios en los rendimientos usando matrices de prueba

Fila del txt	**Tiempo (cualitativo) de A***	
	Heurística inicial	***Heurística final***
1	bajo	bajo
2	bajo	bajo
3	bajo	bajo
4	bajo	bajo
5	bajo	bajo
6	medio	bajo
7	medio	bajo
8	bajo	bajo
9	medio	bajo
10	alto	bajo
11	alto	bajo
12	alto	bajo
13	alto	bajo
14	alto	bajo
15	alto	bajo
16	indefinido	bajo
17	indefinido	bajo
18	indefinido	bajo
19	indefinido	bajo
20	indefinido	bajo

Finalmente, podemos deducir la importancia de definir una buena representación del estado del problema y de un buen algoritmo de búsqueda, incluyendo su heurística y coste.

Además, es necesario destacar que, aunque es necesario emplear bastante tiempo en definir adecuadamente cada una de ellas, si ambas partes no alcanzan cierto nivel de calidad la eficiencia se ver realmente penalizada.

VII. Mejoras

Tal como se ha mencionado anteriormente, una buena idea para empezar a buscar cómo mejorar el algoritmo es fijarse en la matrices que no es capaz de hacer, es decir, cuya velocidad es lenta o muy lenta.

Una vez detectados las dificultades por la que pasa el algoritmo, el siguiente paso sería buscar una nueva técnica o mejorar la implementación de alguna ya existente que optimizará aún más la búsqueda.

Otra opción podría ser intentar mejorar la heurística de cada nodo o los costes de las acciones.

Referencias

[1] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed, Pearson, 2010.

[2] Y. LeCun, Y. Bengio, G. Hinton. “Deep Learning”, Nature, vol. 521, 2015, pp. 436-444.

[3] Página web del curso IA de Ingeniería del Software. https://www.cs.us.es/cursos/iais. Consultada el 24/03/2018.

[4] Técnicas de Resolución del puzzle Hitori https://www.conceptispuzzles.com/index.aspx?uri=puzzle/hitori/techniques

[5] Resolver Hitori http://www.hitoriconquest.com/