Resolución del Hitori mediante búsqueda informada

Rubén Bueno Menéndez
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
rubbuemen@alum.us.es // ruben_077@msn.com

***Resumen*— Este trabajo presenta los resultados obtenidos mediante la búsqueda en espacios de estado sobre la resolución del pasatiempo lógico japonés Hitori usando el lenguaje de programación Python. La aplicación de la correcta implementación junto a la investigación de estrategias para resolverlo ha traído resultados positivos tanto para obtener la solución como para hacerlo en un tiempo razonable.**

Palabras Clave—Cuadrícula, puzle, casilla, fila, columna, repetido, marcar, contigua, búsqueda.

I. INTRODUCCIÓN

El pasatiempo lógico Hitori es un juego donde dada una cuadrícula con valores numéricos en sus casillas que varían entre 1 y el máximo entre el número de filas y el número de columnas de la cuadrícula, se tiene el objetivo de determinar cuáles hay que quitar (marcar en negro en la cuadrícula) para conseguir que no haya elementos repetidos ni en las filas ni en las columnas. Además, para llegar a este objetivo, se deben cumplir también las siguientes restricciones:

- No puede haber dos casillas marcadas en negro juntas horizontal o verticalmente
- Las casillas que no están marcadas en negro deben estar todas conectadas entre sí, esto es que dada una casilla no marcada en negro podemos llegar a cualquier otra moviéndonos horizontal o verticalmente sin pasar por ninguna casilla marcada en negro. Es decir, que la cuadrícula sea conexa. [1]

Para familiarizarme con la resolución de este pasatiempo, lo ideal fue resolver algunos puzles Hitori de diferentes dimensiones manualmente [2].

Una vez familiarizado, planteé las diferentes problemáticas a la hora de resolverlo, como el hecho de que no se incumplieran las restricciones descritas anteriormente, llevándome a diferentes estrategias que permiten una más fácil resolución del pasatiempo Hitori. Dichas estrategias serán comentadas en profundidad a lo largo del documento.

De esta manera, el documento quedará estructurado por secciones en donde se detallarán unas preliminares para la resolución, una metodología de cómo se ha resuelto, los resultados obtenidos y las conclusiones sobre éste.

II. PRELIMINARES

En esta sección se explicará de manera introductoria la técnica empleada para resolver el problema y el diseño que se ha llevado a cabo para el mismo. También se detallarán algunos trabajos relacionados de la resolución de otros problemas con esta misma técnica.

A. Métodos empleados

La técnica empleada para la resolución del Hitori ha sido la búsqueda en espacio de estados. Para ello debemos tener en cuenta cuál será la situación inicial desde la que se parte, el objetivo final, las diferentes situaciones por las que puede pasar y qué acciones hay que realizar para pasar de una situación a otra.

Como es necesario partir de una cuadrícula y realizarle diferentes operaciones, he decidido crear un objeto tipo Puzle, de manera que tenga diferentes funciones definidas y cuya representación será la cuadrícula. Dichas operaciones serán detalladas en la siguiente sección.

Se han usado las librerías proporcionadas por el departamento, las cuales son 'búsqueda_espacio_estados.py' y 'problema_espacio_estados.py'. Además, se han usado dos librerías externas llamadas copy y label:

- **Copy**: sirve para realizar copias de objetos de manera que no sea modificable el original. Servirá para realizar comprobaciones sobre estados. [3]
- **Label**: sirve para obtener una tupla sobre el número de elementos conexas en una matriz y dicha matriz con los elementos sustituidos con una enumeración para cada componente conexa. Nos servirá para realizar la comprobación de una de las restricciones. [4]

Para la implementación del problema como búsqueda en espacio de estados he decidido definir una clase llamada 'MarcarCasilla' que sea una instancia de la clase 'Acción', en la cual se comprobará la aplicabilidad de manera que sólo se podrá aplicar una acción siempre que se cumplan una serie de restricciones, las cuales se detallarán en la siguiente sección.

He decidido definir una clase llamada 'Hitori' que sea una instancia de la clase 'ProblemaEspacioEstados' para poder resolver el problema. En esta clase se comprobará cuándo un estado del puzle es final y por tanto solución.

Por último, mediante una función llamada 'resuelveHitori', se resolverá el problema usando uno de los diferentes tipos de

búsqueda contemplados (para la búsqueda A* será necesario definir una heurística, detallada en la siguiente sección).

B. Trabajos Relacionados

Para familiarizarme con la técnica de búsqueda de espacio de estados he tomado como referencia otros problemas en donde se aplica esta técnica:

- Rompecabezas de las Torres de Hanoi [5]
- Búsqueda de caminos en juegos de ordenador [5]
- Decidir jugadas de TETRIS [6]

III. METODOLOGÍA

Esta sección quedará dividida en las partes descritas en la sección de preliminares, es decir, la clase '**Puzle**', la clase '**MarcarCasilla**', la clase '**Hitori**', la **heurística** para la búsqueda A* y la función '**resuelveHitori**'.

1) En primer lugar, se explicarán las diferentes funciones de la clase 'Puzle' que permitirá realizar operaciones sobre la cuadrícula.
Primero, operaciones básicas:

- **numeroColumnas:** esta función devuelve el número de columnas que hay en la cuadrícula.
- **numeroFilas:** esta función devuelve el número de filas que hay en la cuadrícula.
- **casilla:** esta función devuelve el valor de una casilla concreta a partir de la fila y columna pasadas como parámetros.
- **casillaDentro:** esta función devuelve un valor booleano dependiendo de si una casilla concreta está dentro de la cuadrícula a partir de la fila y columna pasadas como parámetros.
- **casillaMarcada:** esta función devuelve un valor booleano dependiendo de si una casilla concreta está marcada en negro a partir de la fila y columna pasadas como parámetros.
- **transponer:** esta función devuelve la cuadricula transpuesta (se cambian las filas por columnas y viceversa).
- **obtenerFila:** esta función devuelve una lista correspondiente a una fila concreta de la cuadrícula a partir de la fila pasada como parámetro.
- **obtenerColumna:** esta función devuelve una lista correspondiente a una columna concreta de la cuadrícula a partir de la columna pasada como parámetro. Para ello se utiliza la función 'transponer'.
- **estaRepetido:** esta función devuelve un valor booleano dependiendo de si un elemento se repite a lo largo de la fila y/o columna, a partir de la fila y columnas pasadas como parámetros. Gracias a esta función se podrá descartar comprobar aquellas casillas que no estén repetidas en la misma fila y/o columna.

A continuación, operaciones necesarias para realizar las acciones del problema:

- **marcarCasilla:** esta función marca en negro una casilla a partir de la fila y columna pasadas como parámetros, además como consecuencia de marcar en negro una casilla, entonces las casillas contiguas verticales y horizontales no podrán ser marcadas nunca, por lo que se usa la función 'marcarCasillasFilasColumnasContigua' que permite marcar en negro aquellas casillas repetidas en filas y columnas a esas casillas que no pueden ser marcadas nunca. Además, la función 'marcarCasilla' devolverá una lista de todas las casillas marcadas en la llamada de la función 'marcarCasillasFilasColumnasContigua', para realizar más adelante una comprobación de que no se ha incumplido alguna restricción. Gracias a esta función se podrá realizar la acción marcar múltiples casillas como consecuencia de marcar una sola casilla.

Función marcarCasilla
Entrada:
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Salidas:
- Una lista de todas las casillas marcadas en la llamada de la función 'marcarCasillasFilasColumnasContigua'

Función:
Marcar en negro la casilla(fila, columna)
casillasMarcadas = marcarCasillasFilasColumnasContigua(fila, columna)
Devolver casillasMarcadas

Pseudocódigo 1. Función marcarCasillas

- **marcarCasillasFilasColumnasContigua:** esta función marca en negro aquellas casillas que se encuentran en la misma fila y columna que la contigua vertical y horizontal de la casilla correspondiente a la fila y columna pasada por parámetros. Dada las posibles situaciones, se realiza las comprobaciones para los 4 lados posibles (izquierda, derecha, arriba y debajo). Para cada lado se verifica que la casilla está dentro de la cuadrícula y si es así, se comprueba si tiene elementos repetidos, en tal caso, se obtienen los índices de los elementos repetidos y se marcan dichas casillas (descartando para que no se marque la casilla contigua a la correspondiente pasada por parámetros). Además, la función 'marcarCasillasFilasColumnasContigua' devolverá una lista de todas las casillas marcadas, para realizar más adelante una comprobación de que no se ha incumplido alguna restricción. [7]

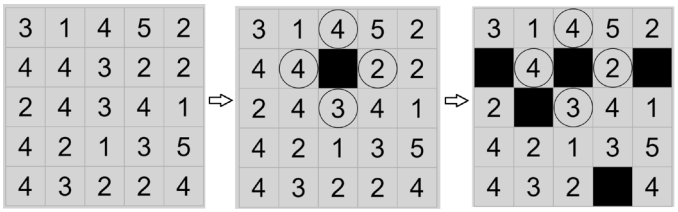

Fig. 1. Ejemplo de usar la función marcarCasilla y como consecuencia marcarCasillasFilasColumnasContigua

Función marcarCasillasFilasColumnasContigua
Entrada:
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Salidas:
- Una lista de todas las casillas marcadas en la llamada de la función 'marcarCasillasFilasColumnasContigua'

Función:
Si la contigua izquierda está dentro de la cuadrícula:
casillaContigua = casilla(fila, columna – 1)
Si se repiten valores iguales a casillaContigua en la misma fila:
Marcar en negro dichas casillas.
Añadir a casillasMarcadas las casillas marcadas.
Si se repiten valores iguales a casillaContigua en la misma columna:
Marcar dichas casillas.
Añadir a casillasMarcadas las casillas marcadas.
Si la contigua derecha está dentro de la cuadrícula:
casillaContigua = casilla(fila, columna + 1)
Si se repiten valores iguales a casillaContigua en la misma fila:
Marcar en negro dichas casillas.
Añadir a casillasMarcadas las casillas marcadas.
Si se repiten valores iguales a casillaContigua en la misma columna:
Marcar dichas casillas.
Añadir a casillasMarcadas las casillas marcadas.
Si la contigua arriba está dentro de la cuadrícula:
casillaContigua = casilla(fila – 1, columna)
Si se repiten valores iguales a casillaContigua en la misma fila:
Marcar en negro dichas casillas.
Añadir a casillasMarcadas las casillas marcadas.
Si se repiten valores iguales a casillaContigua en la misma columna:
Marcar dichas casillas.
Añadir a casillasMarcadas las casillas marcadas.
Si la contigua abajo está dentro de la cuadrícula:
casillaContigua = casilla(fila + 1, columna)
Si se repiten valores iguales a casillaContigua en la misma fila:
Marcar en negro dichas casillas.
Añadir a casillasMarcadas las casillas marcadas.
Si se repiten valores iguales a casillaContigua en la misma columna:
Marcar dichas casillas.
Añadir a casillasMarcadas las casillas marcadas.
Devolver casillasMarcadas

Pseudocódigo 2. Función marcarCasillasFilasColumnasContigua

Finalmente, operaciones más complejas para realizar antes de empezar la búsqueda que nos permitirá descartar posibilidades para comprobar:

- **parRepetidoConsecutivo:** esta función comprueba si una casilla concreta dada a partir de la fila y columna pasada por parámetros, tiene a uno de sus lados un valor, y el par que formaría está de nuevo consecutivo a este (horizontalmente o verticalmente), formando un doble par igual (por ejemplo, de manera horizontal: 5 4 5 4), y por tanto, se sabrá que las dos casillas centrales no podrán marcarse en negro y como consecuencia de ello se marcarán en negros todas las que se repitan a esos dos valores para esa fila y/o columna (excluyendo como es obvio de marcar en negro esas dos casillas centrales). El marcado se realizará llamando a la función 'marcarCasilla' que como consecuencia llamará también a 'marcarCasillasFilasColumnasContigua', descartando así más posibilidades desde un inicio.

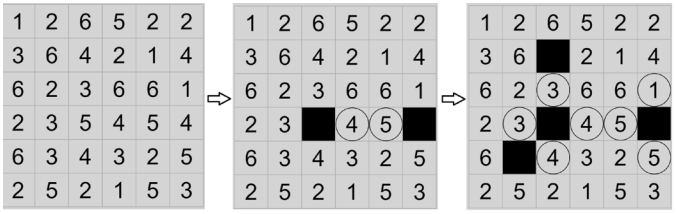

Fig. 2. Ejemplo de usar la función parRepetidoConsecutivo

Función parRepetidoConsecutivo
Entrada:
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Función:
Si las tres casillas contiguas izquierda están dentro de la cuadrícula:
Si casilla(fila, columna - 1) == casilla(fila, columna - 3) y casilla(fila, columna) == casilla(fila, columna - 2):
Usar la función marcarCasilla para cada elemento repetido al pasado por parámetro y al de su izquierda, en la fila
Si las tres casillas contiguas derecha están dentro de la cuadrícula:
Si casilla(fila, columna + 1) == casilla(fila, columna + 3) y casilla(fila, columna) == casilla(fila, columna + 2):
Usar la función marcarCasilla para cada elemento repetido al pasado por parámetro y al de su derecha, en la fila
Si las tres casillas contiguas arriba están dentro de la cuadrícula:
Si casilla(fila - 1, columna) == casilla(fila - 3, columna) y casilla(fila, columna) == casilla(fila - 2, columna):
Usar la función marcarCasilla para cada elemento repetido al pasado por parámetro y al de arriba, en la columna
Si las tres casillas contiguas abajo están dentro de la cuadrícula:
Si casilla(fila + 1, columna) == casilla(fila + 3, columna) y casilla(fila, columna) == casilla(fila + 2, columna):
Usar la función marcarCasilla para cada elemento repetido al pasado por parámetro y al de abajo, en la columna

Pseudocódigo 3. Función parRepetidoConsecutivo

- **estaEntreDosRepetidos:** esta función comprueba si una casilla concreta dada a partir de la fila y columna pasada por parámetros, está entre dos casillas cuyo valor es el mismo, tanto de manera vertical u

horizontal (por ejemplo, de manera horizontal: 5 3 5), y por tanto como una de las casillas cuyo valor es el mismo deberá marcarse, se sabe que la que está entre medio no podrá nunca marcarse. Como consecuencia de ello, se marcarán en negro todas las casillas cuyo valor sea el mismo que el valor central, tanto para su fila como su columna. El marcado se realizará llamando a la función 'marcarCasilla' que como consecuencia llamará también a 'marcarCasillasFilasColumnasContigua', descartando así más posibilidades desde un inicio. Esta función también queda aplicable a una situación en la que tuvieramos el mismo número consecutivo (por ejemplo, en horizontal: 2 2 2), esta función haría que quedaran marcados en negro los elementos de cada lado. Es importante que esta función se use después de usar la función 'parRepetidoConsecutivo', ya que si no descartaría toda posibilidad de encontrar pares repetidos con la función 'parRepetidoConsecutivo'. [7]

Fig. 3. Ejemplo de usar la función estaEntreDosRepetidos

Función estaEntreDosRepetidos
Entrada:
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Función:
Si la casilla contigua a la izquierda y derecha están dentro de la cuadrícula:
    Si los valores de dichas casillas son iguales:
        Usar la función marcarCasilla para cada elemento repetido al pasado por parámetro en la fila y columna
Si la casilla contigua de arriba y abajo están dentro de la cuadrícula:
    Si los valores de dichas casillas son iguales:
        Usar la función marcarCasilla para cada elemento repetido al pasado por parámetro en la fila y columna

Pseudocódigo 4. Función estaEntreDosRepetidos

- **parIgual:** esta función comprueba si una casilla concreta dada a partir de la fila y columna pasada por parámetros, tiene a uno de sus lados (horizontalmente o verticalmente) dos casillas cuyo par tienen el mismo valor, y por tanto, se sabrá que al menos una de esas dos tendrá que ser marcada en negro durante la búsqueda y como consecuencia de ello, entonces se marcarán en negro todas las casillas que tengan el mismo valor para esa fila (si el par está horizontalmente) o esa columna (si el par está verticalmente), excluyendo a marcar las casillas que forman el par. El marcado se realizará llamando a la función 'marcarCasilla' que como consecuencia llamará también a 'marcarCasillasFilasColumnasContigua', descartando así más posibilidades desde un inicio. [7]

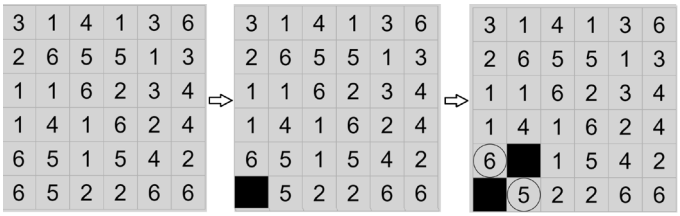

Fig. 4. Ejemplo de usar la función parIgual

Función parIgual
Entrada:
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Función:
Si las dos casillas contiguas de la izquierda están dentro de la cuadrícula:
    Si los valores de dichas casillas son iguales:
        Si se repiten valores para esa fila correspondiente a los valores de dicho par:
            Usar la función marcarCasilla para cada elemento repetido a dicho par
Si las dos casillas contiguas de la derecha están dentro de la cuadrícula:
    Si los valores de dichas casillas son iguales:
        Si se repiten valores para esa fila correspondiente a los valores de dicho par:
            Usar la función marcarCasilla para cada elemento repetido a dicho par
Si las dos casillas contiguas de arriba están dentro de la cuadrícula:
    Si los valores de dichas casillas son iguales:
        Si se repiten valores para esa columna correspondiente a los valores de dicho par:
            Usar la función marcarCasilla para cada elemento repetido a dicho par
Si las dos casillas contiguas de abajo están dentro de la cuadrícula:
    Si los valores de dichas casillas son iguales:
        Si se repiten valores para esa columna correspondiente a los valores de dicho par:
            Usar la función marcarCasilla para cada elemento repetido a dicho par

Pseudocódigo 5. Función parIgual

- **repetidoContiguoEsquina:** esta función comprueba si las casillas de las esquinas de la cuadrícula (de manera independiente) tienen el mismo valor que sus casillas contiguas tanto horizontalmente como verticalmente una casilla (por ejemplo, para la esquina superior izquierda: 44 y debajo del primer 4, otro 4), y, por tanto, se sabrá que la casilla de la esquina deberá ser marcada en negro. El marcado se realizará llamando a la función 'marcarCasilla' que como consecuencia llamará también a

'marcarCasillasFilasColumnasContigua', descartando así más posibilidades desde un inicio. [7]

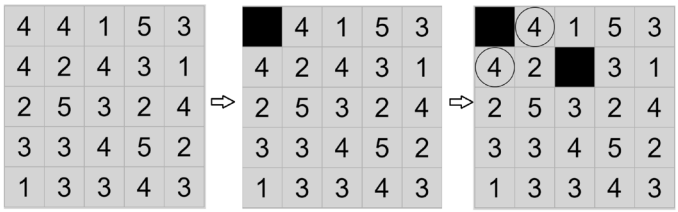

Fig. 5. Ejemplo de usar la función repetidoContiguoEsquina

Función repetidoContiguoEsquina
Entrada:
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Función:
Si el valor de la casilla de la esquina superior izquierda es igual al valor de su casilla contigua derecha y abajo:
    Usar la función marcarCasilla para la casilla de la esquina
Si el valor de la casilla de la esquina superior derecha es igual al valor de su casilla contigua izquierda y abajo:
    Usar la función marcarCasilla para la casilla de la esquina
Si el valor de la casilla de la esquina inferior izquierda es igual al valor de su casilla contigua derecha y arriba:
    Usar la función marcarCasilla para la casilla de la esquina
Si el valor de la casilla de la esquina inferior derecha es igual al valor de su casilla contigua izquierda y arriba:
    Usar la función marcarCasilla para la casilla de la esquina

Pseudocódigo 6. Función repetidoContiguoEsquina

- **parRepetidoEsquina:** esta función comprueba si las casillas de las esquinas de la cuadrícula (de manera independiente) tienen dos pares, en donde cada par está compuesto del mismo número, (por ejemplo, para la esquina superior izquierda de manera horizontal: 44 y debajo 11), y, por tanto, se sabrá que la casilla de la esquina deberá ser marcada en negro y la que hace esquina con esta también será marcada en negro. El marcado se realizará llamando a la función 'marcarCasilla' que como consecuencia llamará también a 'marcarCasillasFilasColumnasContigua', descartando así más posibilidades desde un inicio. [7]

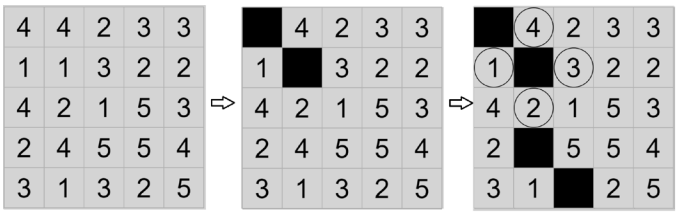

Fig. 6. Ejemplo de usar la función parRepetidoEsquina

Función parRepetidoEsquina
Entrada:
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Función:
Si en la esquina superior izquierda se repiten los valores en dos pares de valores (de manera independiente) de manera vertical u horizontal:
    Usar la función marcarCasilla para la casilla de la esquina y la casilla que hace esquina con esta.
Si en la esquina superior derecha se repiten los valores en dos pares de valores (de manera independiente) de manera vertical u horizontal:
    Usar la función marcarCasilla para la casilla de la esquina y la casilla que hace esquina con esta.
Si en la esquina inferior izquierda se repiten los valores en dos pares de valores (de manera independiente) de manera vertical u horizontal:
    Usar la función marcarCasilla para la casilla de la esquina y la casilla que hace esquina con esta.
Si en la esquina inferior derecha se repiten los valores en dos pares de valores (de manera independiente) de manera vertical u horizontal:
    Usar la función marcarCasilla para la casilla de la esquina y la casilla que hace esquina con esta.

Pseudocódigo 7. Función parRepetidoEsquina

2) A continuación, se explicará el planteamiento de la clase 'MarcarCasilla'.

Se deduce que obviamente el **estado inicial** del problema de búsqueda en espacios de estado será el objeto Puzle creado a partir de la cuadrícula que le pasemos por parámetro. El **estado final** no lo podemos saber a priori, pero sí podemos saber que para que sea estado final se deben cumplir las siguientes restricciones:

- No puede haber dos números iguales en una fila o columna.
- No puede haber dos casillas marcadas en negro juntas horizontal o verticalmente
- Las casillas que no están marcadas en negro deben estar todas conectadas entre sí, esto es que dada una casilla no marcada en negro podemos llegar a cualquier otra moviéndonos horizontal o verticalmente sin pasar por ninguna casilla marcada en negro.

Finalmente, las **acciones** a realizar para este problema será únicamente una: marcar una casilla en negro (que a su vez se pueden llegar a marcar otras como consecuencia del marcado de esta). Esta acción será definida como una clase y tendrá una **aplicabilidad** de manera que sólo podrá aplicarse siempre que se cumpla que:

1. La casilla a marcar está dentro de la cuadrícula.
2. La casilla no está ya marcada
3. La casilla tiene un valor que realmente está repetido.
4. La casilla no tiene ya una casilla contigua horizontalmente o verticalmente ya marcada en negro
5. La cuadrícula no se hace desconexa tras marcar la casilla.

6. Adicionalmente, debido a la consecuencia de marcado en negro de otras casillas al marcar en negro otra, se comprueba que al realizar esta acción no se ha incumplido alguna de las anteriores restricciones.

Para ello será necesario redefinir la función **es_aplicable**, de la clase 'Acción', de la librería importada problema_espacio_estados.py.

Para la comprobación de la restricción sobre si se vuelve o no desconexa una matriz tras el marcado en negro de una casillas, hacemos uso de la librería 'label', de scipy.ndimage. Y a su vez, para poder realizar esta comprobación debemos primero hacer una copia del objeto mediante la librería 'copy'. De esta forma no quedará editado el objeto original y en caso de incumplirse la restricción no sería necesario una "acción" de desmarcado.

Lo mismo ocurrirá para la función **aplicar**, se tendrá que realizar una copia para verificar que si se incumple no se haya editado el Puzle original y se pueda entonces realizar BackTracking sobre esta.

Para la implementación de este problema, he decidido que no habrá coste, es decir, el coste será siempre 1 (o equivalente a la profundidad en la búsqueda).

Las funciones definidas dentro de la clase 'MarcarCasilla' son:

Función casillaContiguaMarcada
Entrada:
- El estado actual del Puzle
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Salidas:
- Un valor booleano que representa si está marcada una casilla contigua horizontalmente o verticalmente a la pasada por parámetro

Función:
```
contiguaMarcada = False
Si estado.casillaDentro(fila, columna - 1):
    contiguaMarcada = estado.casillaMarcada(fila,
    columna - 1)
Si estado.casillaDentro(fila, columna + 1) y no
contiguaMarcada:
    contiguaMarcada = estado.casillaMarcada(fila,
    columna + 1)
Si estado.casillaDentro(fila - 1, columna) y no
contiguaMarcada:
    contiguaMarcada = estado.casillaMarcada(fila - 1,
    columna)
Si estado.casillaDentro(fila + 1, columna) y no
contiguaMarcada:
    contiguaMarcada = estado.casillaMarcada(fila + 1,
    columna)
Devolver contiguaMarcada
```

Pseudocódigo 8. Función casillaContiguaMarcada

Función grupoAislado
Entrada:
- El estado actual del Puzle
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Salidas:
- Un valor booleano que representa si la cuadrícula está conexa o desconexa

Función:
```
estaAislada = False
estadoNuevo = copia(estado)
estadoNuevo.marcarCasilla(fila, columna)
Si número de componentes conexas de estadoNuevo > 1:
    estaAislada = True
Devolver estaAislada
```

Pseudocódigo 9. Función grupoAislado

Función malMarcadasConsecuenciaPorContiguasMarcada
Entrada:
- El estado actual del Puzle
- Un entero representando el número de la fila
- Un entero representando el número de la columna

Salidas:
- Un valor booleano que representa si se ha producido un mal marcado

Función:
```
malMarcado = False
nuevoEstado = copia(estado)
casillasMarcadas = nuevoEstado.marcarCasilla(fila,
columna)
malMarcado = grupoAislado(nuevoEstado, fila, columna)
Si no malMarcado:
    Para cada casilla de casillasMarcadas:
        Si casillaContiguaMarcada(nuevoEstado,
        casilla[0], casilla[1]):
            malMarcado = True
            break
Devolver malMarcado
```

Pseudocódigo 10. Función malMarcadasConsecuenciaPorContiguasMarcada

Función es_aplicable
Entrada:
- El estado actual del Puzle

Salidas:
- Un valor booleano que representa si es aplicable la acción

Función:
```
esAplicable = estado.casillaDentro(self.f, self.c) y no
estado.casillaMarcada(self.f, self.c) y
estado.estaRepetido(self.f, self.c) y no
self.casillaContiguaMarcada(estado, self.f, self.c) y no
self.grupoAislado(estado, self.f, self.c) y no
self.malMarcadasConsecuenciaPorContiguasMarcada(estad
o, self.f, self.c)
Devolver esAplicable
```

Pseudocódigo 11. Función es_aplicable

Función aplicar
Entrada:
- El estado actual del Puzle

Salidas:
- El nuevo estado del Puzle

Función:
```
nuevoEstado = copia(estado)
nuevoEstado.marcarCasilla(self.f, self.c)
Devolver nuevoEstado
```

Pseudocódigo 12. Función aplicar

3) Seguidamente, se explicará el planteamiento de la clase 'Hitori', en la cual se realizará la construcción del problema como búsqueda en espacios de estado.

La cantidad de **acciones posibles** dependerá del estado en el que se encuentre en el momento el Puzle. En un principio, para

el estado inicial, la cantidad de acciones posibles será el número de casillas que hay en total, es decir, será contemplado como N x M, donde N es el número de filas del Puzle y M es el número de columnas del Puzle.

Antes de empezar la búsqueda, con el estado inicial del Puzle, podemos saber que una serie de casillas serán marcadas en negro, reduciendo así la búsqueda dependiendo de la cantidad de casillas que se hayan marcado en negro según la dimensión del Puzle. Para marcar en negro estas casillas que sabemos que serán marcadas obligatoriamente, llamamos a todas las funciones de las operaciones complejas descritas en la clase Puzle.

Finalmente, se debe redefinir la función **es_estado_final** de la clase ProblemaEspacioEstados, de la librería importada problema_espacio_estados.py, donde comprobamos para cada estado obtenido si se trata del estado final o debe continuar la búsqueda. Para que sea estado final, se deben cumplir las 3 restricciones comentadas anteriormente.

Las funciones definidas dentro de la clase 'Hitori' son:

Función para inicializar Hitori como problema de estados
Entrada:
- Una cuadrícula

Función:
```
acciones = []
estado_inicial = Puzle(cuadricula)
n_filas = estado_inicial.numeroFilas()
m_columnas = estado_inicial.numeroColumnas()
estado_inicial.repetidoContiguoEsquina()
estado_inicial.parRepetidoEsquina()
Para cada fila entre 0 y n_filas:
    Para cada columna entre 0 y n_columnas:
        estado_inicial.parRepetidoConsecutivo(f, c)
        estado_inicial.estaEntreDosRepetidos(f, c)
        estado_inicial.parIgual(f, c)
        Se añade MarcarCasilla(f, c) a acciones
Inicialización del problema con (acciones, estado_inicial)
```

Pseudocódigo 13. Función para inicializar Hitori

Función es_estado_final
Entrada:
- Un estado

Salida:
- Un valor booleando que representa si el estado es final o no

Función:
```
esFinal = True
Para cada fila se comprueba que no se repitan valores en las casillas. Si se repite, esFinal = False y devolver esFinal
Para cada columna se comprueba que no se repitan valores en las casillas. Si se repite, esFinal = False y devolver esFinal
Para cada casilla marcada del Puzle se comprueba que no tiene una contigua vertical u horizontal marcada. Si tiene una contigua marcada, esFinal = False y devolver esFinal
Para el estado actual del Puzle, se comprueba que el número de componentes conexas. Si es mayor a 1, esFinal = False y devolver esFinal.
```

Pseudocódigo 14. Función es_estado_final

4) Para la búsqueda A* será necesario definir una heurística. He tomado la decisión de que una buena heurística sería comprobar si se repiten elementos en una fila y/o columna, de manera que el valor de la heurística varia tal que así:

- Si se repiten elementos en alguna fila y columna, el valor de la heurística será 2.
- Si se repiten elementos en alguna fila o columna, el valor de la heurística será 1
- Si no se repiten elementos en ninguna fila y columna (es decir, posiblemente sea estado final), el valor de la heurística será 0.

5) Por último, se hará uso de una función que, a partir de una matriz de números naturales de tamaño MxN (no necesariamente cuadrada), construya el puzle hitori correspondiente a dicha matriz y lo resuelva mediante un algoritmo de búsqueda que también se pasa como argumento, para finalmente devolver la cuadrícula del puzle hitori resuelta.

IV. Resultados

En esta sección se detallará tanto los experimentos realizados como los resultados conseguidos. Los tipos de búsqueda que he decidido usar para los experimentos son la búsqueda en anchura, búsqueda en profundidad, búsqueda óptima y búsqueda A* con la heurística anteriormente definida:

En todos los experimentos realizados se quiere como objetivo resolver el puzle Hitori en el menor tiempo posible. En el documento vamos a centrarnos únicamente en los tiempos de cada búsqueda, para ver el resultado del puzle Hitori, se deberá ejecutar en el fichero de código del trabajo. Para los ejemplos del fichero ejemplos_prueba.txt, se obtienen los siguientes resultados (se indica el número de la línea del fichero, que corresponderá a la matriz que será):

Resultados para ejemplos_prueba.txt					
Número de línea	***Dimensión de la matriz***	***Búsqueda en anchura***	***Búsqueda en profundidad***	***Búsqueda óptima***	***Búsqueda A****
1	3x3	2.73 ms	2.68 ms	2.77 ms	2.61 ms
2	3x3	9.02 ms	7.21 ms	8.59 ms	9.88 ms
3	3x3	3.6 ms	4.3 ms	3.78 ms	3.72 ms
4	3x3	11.2 ms	6.84 ms	7.93 ms	7.59 ms
5	3x3	10.8 ms	8.74 ms	7.03 ms	6.98 ms
6	4x4	6.68 ms	5.29 ms	5.83 ms	5.9 ms
7	4x4	3.01 ms	2.79 ms	2.92 ms	2.32 ms
8	4x4	5.13 ms	6.46 ms	5.96 ms	6.39 ms
9	4x4	9.72 ms	7.89 ms	6.94 ms	5.96 ms
10	4x4	3.75 ms	3.46 ms	3.42 ms	4.24 ms
11	5x5	9.13 ms	8.8 ms	7.62 ms	7.34 ms
12	5x5	41.3 ms	26.3 ms	12.4 ms	13 ms
13	5x5	121 ms	50.4 ms	17.3 ms	16.5 ms
14	5x5	339 ms	94.7 ms	46.2 ms	47.7 ms
15	5x5	5.17 ms	5.98 ms	5.07 ms	5.16 ms

Resultados para ejemplos_prueba.txt					
Número de línea	***Dimensión de la matriz***	***Búsqueda en anchura***	***Búsqueda en profundidad***	***Búsqueda óptima***	***Búsqueda A****
16	6x6	116 ms	21.8 ms	16.9 ms	17 ms
17	6x6	508 ms	56.5 ms	21.1 ms	20.2 ms
18	6x6	10.3 ms	10.3 ms	9.47 ms	8.41 ms
19	6x6	39.7 ms	14.9 ms	12.5 ms	13.7 ms
20	6x6	4.74 s	89.7 ms	267 ms	228 ms

Si realizamos una comparativa con todos los tiempos de búsqueda detallados en la tabla anterior, podemos observar que cuanto menor es la dimensión de la matriz, más rápido se resuelve, como es obvio, y que apenas difieren los tiempos al cambiar el tipo de búsqueda. A mayor va siendo la dimensión de la matriz, mayor va siendo el tiempo de búsqueda y se va notando más la afectividad del tipo de búsqueda utilizado.

Sin embargo, en algunos casos puntuales, a pesar de tener la misma dimensión la matriz, se resuelve más rápidamente o más lentamente que otras. Esto es debido a que puede darse que hay más elementos repetidos o que los elementos están distribuidos de tal manera que desde el estado inicial se puede saber una cantidad mayor de casillas a marcar en negro.

En todo caso, podemos observar que el peor tipo de búsqueda para este problema es el de búsqueda en anchura, teniendo una tasa de tiempos mucho mayor (incluso en la última de dimensión 6x6 tarda segundos).

Entre la búsqueda en profundidad y búsqueda óptima no hay apenas diferencia, ya que, al no haber un coste definido para este problema, la búsqueda óptima coincidirá casi con la búsqueda en profundidad (ya que toma de coste la profundidad); aun así, para los resultados comprobados, sigue siendo mejor la búsqueda óptima en comparación con la de profundidad según los tiempos obtenidos, posiblemente debido a que la solución no tiene por qué estar en un nodo de profundidad máxima.

No cabe duda de que el mejor algoritmo de búsqueda para usar es el A*, debido a que, gracias a la heurística definida, se encuentra más rápidamente la solución, ya que se le da una "pista" de por dónde debe buscar. Esto ayudará a resolver matrices de mayores dimensiones, como son las de 9x9 especificadas en el fichero ejemplos_reto.txt.

Para la resolución de estos puzles de retos 9x9 Hitori he tomado la decisión de que, si el algoritmo de búsqueda A* no es capaz de resolverlo en menos de 1 min y 30 seg, se considerará como fracaso, aunque realmente fuese capaz de encontrar solución en más tiempo (quedará reflejado en la tabla con un guion "-").

Para estos ejemplos se obtienen los siguientes resultados:

Resultados para ejemplos_reto.txt	
Número de línea	***Búsqueda A****
1	111 ms
2	48.6 ms
3	52.2 ms
4	88.2 ms
5	64.7 ms
6	1.58 s
7	36.1 ms
8	47.7 ms
9	-
10	103 ms
11	57.4 ms
12	62.7 ms
13	1min 34s
14	5.45 s
15	1.7 ms
16	-
17	59.7 ms
18	30.6 ms
19	-
20	38 ms
21	55 ms
22	69.4 ms
23	225 ms
24	67 ms
25	1min 9s
26	624 ms
27	929 ms
28	66.7 ms
29	-
30	66.7 ms
31	1.27 s
32	154 ms
33	198 ms
34	628 ms
35	271 ms
36	130 ms
37	145 ms
38	128 ms
39	124 ms
40	6.09 s
41	401 ms
42	1.1 s
43	4.57 s
44	5.79 s
45	1.8 s
46	122 ms

Resultados para ejemplos_reto.txt	
Número de línea	*Búsqueda A**
47	59 ms
48	65.8 ms
49	84.2 ms
50	-
51	86.2 ms
52	-
53	70.4 ms
54	39.7 ms
55	46.4 ms
56	-
57	-
58	90.9 ms
59	52.9 ms
60	941 ms
61	269 ms
62	-
63	-
64	-
65	106 ms
66	-
67	99.7 ms
68	269 ms
69	-
70	-
71	33.3 s
72	2.85 s
73	-
74	-
75	-
76	-
77	-
78	57.1 s
79	-
80	-
81	153 ms
82	147 ms
83	-
84	14.7 s
85	-
86	138 ms
87	-
88	-
89	-
90	-
91	-
92	176 ms
93	-
94	433 ms
95	-
96	-
97	363 ms
98	153 ms
99	-
100	-

Se ha conseguido resolver en un tiempo menor a 1 min y 30 seg, 67 del total de 100 matrices del archivo ejemplos_reto.txt.

V. Mejoras

Se ha añadido como mejora del proyecto una visualización más amigable del puzle. La representación original del puzle es una matriz bidimensional en donde las casillas con valor a 0 significa el marcado en negro. Esto no es muy visual a la hora de ver los resultados.

La representación final es mucho más visual, creando paredes para el tablero y cada casilla, y sustituyendo los valores a 0 por cuadrados negros.

Por ejemplo, para la matriz:

[[1,8,2,4,3,9,4,1,5],
[6,5,5,1,1,4,9,3,3],
[8,4,6,9,5,3,2,6,7],
[4,8,3,8,7,8,8,2,9],
[1,6,4,2,3,6,5,9,1],
[9,5,8,3,5,2,7,6,1],
[2,5,3,1,6,4,4,2,8],
[5,4,4,2,9,3,7,7,6],
[7,9,1,5,1,6,3,8,8]]

La representación final sería la siguiente, junto a su solución:

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

⇨

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

VI. Conclusiones

Como hemos podido ver a lo largo del documento, para haber sido capaz de resolver el puzle Hitori, se han debido de estudiar diferentes estrategias de marcado en negro, jugando con las restricciones del pasatiempo. Combinando estas estrategias se consiguen resolver fácilmente todos los puzles de pequeñas dimensiones.

Sobre la implementación, el hecho de haber podido usar una librería para comprobar si la matriz es conexa o no, ha facilitado muchísimo la implementación, ya que, si no, seguramente habría que haber creado un método que use recursividad para comprobar las casillas que hacen esquinas con la casilla pasada por parámetros y así sucesivamente para cada casilla que hace esquina, para comprobar si finalmente es conexa o no.

Facilita mucho la resolución el hecho de que se cumplan diferentes condiciones en la disposición de las casillas del puzle inicial para poder realizar un marcado en negro de estas antes

de comenzar la búsqueda. Esto ha sido muy clave a la hora de conseguir resolver los puzles.

Respecto a los tipos de búsquedas, está claro que el mejor para usar es la búsqueda A*, ya que el uso de una correcta y buena heurística, facilita mucho la búsqueda.

Finalmente, este proyecto se podría mejorar implementando más estrategias u optimizar el código en un futuro, de manera que fuese capaz de resolver las demás matrices del archivo ejemplos_reto.txt que no fui capaz e incluso matrices de mayores dimensiones.

REFERENCIAS

[1] Página web de Wikipedia sobre Hitori. https://es.wikipedia.org/wiki/Hitori.

[2] Página web para autogenerar puzles Hitori de diferentes dimensiones y dificultades. https://www.psicoactiva.com/juegos-inteligencia/hitori/.

[3] Librería 'copy'. https://docs.python.org/3.5/library/copy.html

[4] Librería 'scipy.ndimage.label'. https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.label.html.

[5] Rompecabezas de las Torres de Hanoi y Búsqueda de caminos en juegos de ordenador. https://www.cs.us.es/cursos/iais-2018/practicas/Pr%C3%A1ctica_03.zip.

[6] Decidir jugadas de TETRIS. https://github.com/camreyaro/tetrIA.

[7] Diferentes técnicas para resolver un Hitori. https://www.conceptispuzzles.com/index.aspx?uri=puzzle/hitori/techniques.