JITORI

Álvaro de la Flor Bonilla
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
alvdebon@alum.us.es

Antonio Manuel Salvat Pérez
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
antsalper4@alum.us.es

Hitori es un pasatiempo lógico de origen japonés cuyo objetivo principal consiste en, dada una cuadrícula con cifras, determinar cuáles hay que quitar para conseguir que no haya elementos repetidos ni en las filas ni en las columnas.

En el presente documento proponemos una resolución al problema mediante el uso de técnicas de búsquedas de espacios de estado.

Búsqueda de estado, heurística, coste, python, pasatiempos.

I. INTRODUCCIÓN

La búsqueda en el espacio de estados es una técnica informática que pretende conseguir la solución a un problema propuesto a partir de la combinatoria de distintos estados definidos, hasta concluir en un estado final que satisfaga los requisitos del problema.

En particular, durante nuestros estudios en la escuela, prácticamente no hemos tenido contacto con algoritmos de búsqueda avanzada de resolución de problemas teniendo, sin embargo, un leve contacto en la asignatura Análisis y Diseño de Datos y Algoritmos. En ella realizamos una breve introducción a:

1. Algoritmos genéticos
2. Backtracking
3. Coloreado de grafos
4. Algoritmo A*
5. Recursividad
6. Búsqueda en anchura
7. Búsqueda en profundidad

Como contra, utilizábamos un código prácticamente completo y desarrollado (en nuestro caso por el profesor Miguel Toro) en el que no éramos conscientes de su funcionamiento, simplemente aplicábamos una función de coste ya que los ejemplos propuestos tenían un nivel de simpleza tal que no era necesario el desarrollo de funciones de heurística.

Por otro lado, en este caso como objetivo principal pretendemos llegar a la resolución del problema a partir del desarrollo de un código propio del que somos plenamente conscientes de su funcionamiento, de tal forma que conozcamos que modificar para maximizar su rendimiento con los mejores resultados posibles.

En esta ocasión se ha planteado la resolución de un tablero Hitori.

El juego consiste en dado un tablero con MxN casillas (siendo M y N un número natural positivo con el mismo valor o distinto) donde todas las casillas se encuentran rellenas, elegir las casillas a tachar hasta conseguir que no existen número repetidos tanto en filas como en columnas. Además, se establecen como añadidos una serie de reglas para que la solución sea válida:

- No puede haber dos casillas contiguas tachadas tanto vertical, como horizontalmente.
- El tablero final debe configurar una única componente conexa.

  Una casilla rodeada por celdas tachadas, por ejemplo, formaría una nueva componente conexa.

En esta ocasión, para la resolución del tablero hemos decido priorizar el desarrollo de la búsqueda informada, que no es más que aplicar al proceso de búsqueda un conjunto de datos que priorice los mejores estados, comparando su efectividad esperada y pudiendo realizar así un criterio de orden y podado del conjunto de estados dado.

Durante todo el desarrollo del proyecto nos hemos sostenido en los dos pilares básicos de los criterios de evaluación de la búsqueda de estados:

- Complejidad en tiempo

Tiempo de cálculo total empleado para la resolución del tablero propuesto

- Complejidad en espacio

Número de estados que el sistema almacena en memoria hasta encontrar la solución.

Por último, una vez diseñado todo el código se ha pretendido desarrollar una interfaz lo más intuitiva para el usuario, la cual permite el uso de elección de distintas configuraciones de algoritmos a utilizar, aunque será configurado el A* como uso preferente.

Hay que destacar que para la comprobación de las soluciones de una forma ágil ha sido utilizado un repositorio de GitHub el cual devuelve las soluciones de los tableros introducidos en un fichero de texto de una ruta de nuestro equipo. Este algoritmo es capaz de resolver tableros de un tamaño de 12x12 en menos de un segundo. [1]

II. Estructura de trabajo

A. *Tecnología y entorno de trabajo*

Como lenguaje de programación ha sido utilizado Python en su versión 3.7 y como IDE de desarrollo se ha optado por el uso de PyCharm.

Se ha hecho uso además de un conjunto de librería externas.

- Matplotlib [2]
  Para la representación del estado final

- Numpy [3]
  Para facilitar el uso del manejo de matrices

- Scipy [4]
  Para la comprobación del número de componentes conexas

B. *Control de versiones y estrategia de desarrolo*

Para la gestión de código ha sido utilizado un repositorio en GitHub [5].

Existe parte del código que es completamente modular. Muestra de ello puede ser la comprobación de cada una de las restricciones válidas para que una acción sea aplicable.

Cada una de las divisiones nombradas anteriormente fueron analizadas y asignadas a uno de los integrantes de manera equitativa en función de la complejidad con anterioridad al desarrollo del código.

Sin embargo, existen partes del código que resultaba imposible su división. Es por ello por lo que en estos casos ha sido utilizada la técnica de *pair programming* [7].

Una vez la función era función se utilizaba la metodología GitFlow que consiste en una estrategia de creación de ramas [8].

C. *Organización del código*

Todo nuestro código queda distribuido en 5 archivos principales.

- problema_espacio_estados.py
- búsqueda_espacio_estados.py

Ambas clases fueron proporcionadas en la propuesta del proyecto y se encargan de generar la acción a aplicar y el problema a resolver en el primer caso y generar el algoritmo de búsqueda en el segundo caso. Aunque se han realizado ciertas modificaciones para que se ajusten a nuestro problema en específico

- investiga.py

  Será la encargada de, utilizando una acción ya generada, analizar si esta acción es aplicable y en el caso de que lo sea calcular su coste.

- hitori.py

Inicializa todas las funciones, requiere al usuario el tablero a resolver el tipo de búsqueda a realizar y finalmente aplica el algoritmo de búsqueda.

- auxiliar.py

Contiene los métodos auxiliares que requiere hitori.py para la inicialización del problema.

III. Decisiones de diseño

Para la construcción del problema a resolver se han tomado las siguientes decisiones.

La representación del estado queda definida como el tablero al completo, donde cada celda queda representada por un valor numérico. Si el valor de la celda es 0 significa que esa celda se encuentra tachada.

Las acciones quedan definidas por el tachado de cada una de las celdas del tablero, por tanto, se crearán MxN acciones donde M son el número de filas que tiene el tablero y N el número de columnas.

Por último, se ha decidido que antes de la inicialización del algoritmo de búsqueda se realice una primera optimización de código como posteriormente será explicado. Como breve resumen en esta optimización se realiza una primera eliminación de las casillas que se conoce a ciencia cierta (sin necesidad de aplicación de algoritmos de búsqueda) que deben ser tachadas. En el siguiente apartado será explicado este procedimiento al completo.

IV. Metodología

Para la resolución de nuestro proyecto hemos utilizado cuatro tipos de búsquedas:

- Búsqueda en Anchura
- Búsqueda en Profundidad
- Búsqueda Óptima
- Búsqueda A*

Todas las búsquedas citadas anteriormente son completamente funcionales, sin embargo, a mayor tamaño de tablero alguno de estos métodos se ven muy castigados en su ejecución.

Además, todas ellas usan un mismo código para la aplicabilidad, cálculo de coste y comprobación de estado final. Sin embargo, a partir de una matriz con tamaño 6x6 se hace

inviable el uso de cualquier otro método que no sea la búsqueda A*, ya que es el único que permite el desarrollo de una búsqueda no ciega (búsqueda informada).

El código común entre las distintas búsquedas consiste en los siguientes bloques:

En primer lugar, se analiza la matriz inicial en busca de tríos conjuntos. La existencia de un trío adyacente conlleva la obligatoriedad de eliminación de las casillas extremas, salvaguardando la celda central del trio ya que en caso contrarío se generarían dos casillas contiguas tachadas.

Un ejemplo de trío contiguo puede ser el siguiente:

5	5	5
1	3	2
8	7	6

Las celdas marcadas en rojo deben ser las eliminadas. Cualquier otra variante generaría la existencia de dos celdas contiguas tachadas.

La siguiente tabla muestra el pseudocódigo del método anterior:

Adjacent Triplet
Entrada:
- Un estado status

Salidas:
- Un estado pruned_status con el estado entrada podado

Algoritmo:
1. Mientras j<longitud(columna de status)
   a. Mientras i<longitud(fila de status)
      b. Si longitud(fila de status) – i >= 3
         i. Si status[i][j] == status[i][j+1] == status[i][j+2]
            1. status[i][j] = 0
            2. status[i+2][j] = 0
2. Mientras i<longitud(fila de status)
   a. Mientras j<longitud(columa de status)
      b. Si longitud(columna) – j >= 3
         i. Si status[i][j] == status[i][j+1] == status[i][j+2]
            1. status[i][j] = 0
            2. status[i][j+2] = 0
3. Devolver OV

Pseudocódigo 1 Algoritmo de detección de tripletes

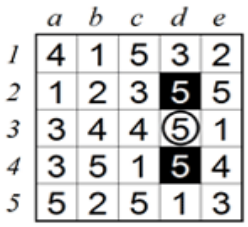

Fig. 1. Ejemplo de tablero resuelto

Eliminados los tripletes pasamos a la eliminación de “pareja + celda”. Este método localiza las parejas de celdas existentes en el tablero. La existencia de una pareja de celdas con un mismo valor significa que, además de que se debe tachar una de las dos celdas, si aparece alguna celda más en la fila (o columna) de esta pareja deberá ser tachada, ya que se debe cumplir el criterio de no repetición de elementos.

La siguiente estrategia para realizar es la eliminación de parejas que consiste en que si en una fila (o columna) existe una pareja de celdas con el mismo valor, todos los elementos restantes con el mismo valor en esa fila o columna deben ser eliminados.

Acto seguido se realiza el método de eliminación de celdas basado en el conjunto de ceros fijados anteriormente.

Analisis de ceros fijados (PARTE 1)
Entrada:
- Un estado status

Salidas:
- Valor booleano

Algoritmo:
newStatus = status
res = 0
Mientras i<longitud(fila de status)
Mientras j<longitud(columna de status
Si status[i][j] == 0
Celda = status[i][j]
-Si i > 0
valorArriba = status[i-1][j]
Mientras jj < lenColumnas
Si valorArriba tiene el mismo valor pero su j es distina
Tacha celda y sumo a res
Mientras i2 < lenFilas
Si valorArriba tiene el mismo valor pero su i es distina
Tacha celda y sumo a res

-Si i < número de filas – 1
valorAbajo = status[i+1][j]
Mientras jj < lenColumnas
Si valorAbajo tiene el mismo valor pero su j es distina
Tacha celda y sumo a res
Mientras i2 < lenFilas
Si valorAbajo tiene el mismo valor pero su i es distina
Tachar celda y sumo a res

-Si j < número de columnas – 1
ValorRight = status[i][j+1]
Mientras jj < lenColumnas
Si ValorRight tiene el mismo valor pero su j es distina
Tacha celda y sumo a res
Mientras i2 < lenFilas
Si ValorRight tiene el mismo valor pero su i es distina
Tacha celda y sumo a res

Adjacent Triplet (PARTE 2)

```
-Si j > 0
valorLeft = status[i][j-1]
Mientras jj < lenColumnas
        Si valorLeft tiene el mismo valor
        pero su j es distina
                Tacha celda y sumo a res
Mientras i2 < lenFilas
        Si valorLeft tiene el mismo valor
        pero su i es distina
                Tacha celda y sumo a res
Si res > 0 devuelve true
```

Pseudocódigo 2 Algoritmo de detección de ceros y eliminación de celdas

Este método se encarga de capturar todos los ceros introducidos en la anterior optimización. Estos ceros, que se encuentran fijados, no pueden tener otro cero adyacente, ya que incumpliría una norma básica, con lo cual, todas las casillas adyacentes a cada cero deben ser fijadas. Fijar estas casillas conlleva a que su valor debe ser único, tanto en filas como en columnas por lo que todas las celdas repetidas deben ser eliminadas, y por cada cero generado debe volver a repetirse toda esta metodología.

Después de este paso el tablero está optimizado.

El siguiente paso consiste en generar las posibles acciones a aplicar, generar el problema a resolver y comenzar la búsqueda elegida por el usuario.

La creación de un problema requiere la definición tanto de la representación del estado del conjunto de acciones a realizar.

En nuestro caso el conjunto de acciones viene definido por la creación de una acción por cada una de las celdas que tiene nuestro tablero. Aplicar una acción consiste en tachar la celda que define la acción.

Una vez definido el estado inicial (que ha sido optimizado) y creado el conjunto de acciones, donde cada acción viene definida por el estado completo, el número de la fila y el número de la columna. Puede iniciarse el proceso de búsqueda.

El primer paso que realiza el algoritmo es analizar las acciones aplicables (aunque existen variantes de heurísticas como explicaremos posteriormente).

Para que una acción sea aplicable debe cumplir que no se encuentre tachada.

En el caso de que no encuentre tachada pasaríamos al método aplica donde diferenciamos dos tipos de acciones: tachado de la celda o fijación de la celda. Para que una celda pueda ser tachada debe cumplir las siguientes restricciones:

- La celda no se debe encontrar tachada.
- No se puede tachar esa celda si su valor es único en fila y columna
- Al aplicarse la acción, la celda tachada no puede crear un aislamiento a ninguna celda

  Un aislamiento viene definido por 5 celdas, una central y sus cuatro celdas adyacentes. Si el valor de la casilla central es distinto a 0 y el valor de todas sus celdas adyacentes es 0 se ha producido un aislamiento.

  Por tanto, una casilla tachada es capaz de generar 4 aislamientos a analizar.

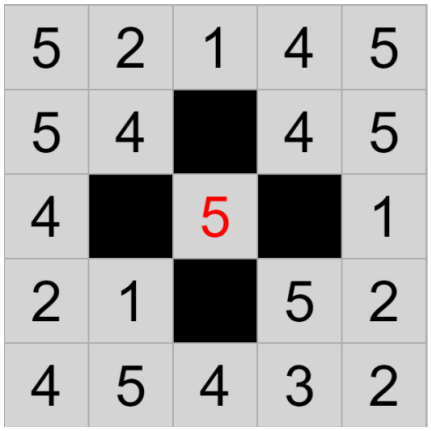

Fig. 2. Ejemplo de tablero resuelto

- Un tablero hitori resuelto solo puede tener una componente conexa.
- Si la celda a analizar tiene a sus lados dos celdas con el mismo valor no puede ser tachada, ya que en algún momento alguna de sus dos casillas adyacentes debe ser eliminada (para no repetir un mismo valor en una fila o columna).

En caso de que estas restricciones no se cumplan se pasaría a la fijación de celda, lo que conlleva a que en ninguna fila ni columna pueda existir el mismo valor.

En ambas acciones (fijación y tachado) se generan un conjunto de ceros en el que se puede volver a aplicar el método de optimización por localizado de ceros.

La siguiente comprobación es calcular el número de componentes conexas existentes, si el número resultante es distinto a 1 la acción no es aplicable y se vuelve el nodo a su estado inicial anterior al realizado del aplica. También se comprueba que no exista ninguna pareja de ceros. En el caso de que exista el nodo vuelve, también en este caso, a su estado inicial.

El siguiente paso es calcular el coste de aplicar. En nuestro caso, el coste de la aplicabilidad viene definido por:

Coste Aplicar
Entrada:
- Un estado status y una celda cell

Salidas:
- Coste coste

Algoritmo:
1. coste = 0
2. numberRow = cuentaRepetidosFila(estado, celda)
3. numberColumn = cuentaRepetidosColumna(estado, celda)
4. Si numberRow > 1
    a. coste = coste + 10000 * numberRow
5. Si numberColum > 1
    a. coste = coste + 10000 * numberColum
6. Devolver coste

Pseudocódigo 3 Coste de aplicar

Una vez llegado a este punto el algoritmo conoce todos los parámetros necesarios para iniciar una búsqueda de nodos, la cual variará en función del criterio de búsqueda elegido.

Nos centraremos en el criterio de búsqueda A*.

Para la utilización de este método, es necesaria la definición de un criterio de heurística. En nuestro caso, queda definido por:

Cálculo de heurística
Entrada:
- Un nodo nodo

Salidas:
- Valor de la heurística del nodo h

Algoritmo:
1. estado = nodo.estado
2. row = 1000000000000000
3. repetidos = calcularRepetidos(estado)
4. row = row + 1000000 * repetidos
5. Devolver row

Pseudocódigo 4 Coste de heurística

A diferencia de otros métodos de búsqueda, el algoritmo A* ordena las acciones a aplicar en función de la combinación entre la diferencia entre el coste de aplicar esa acción y la heurística que posee ese nodo.

Una vez ordenados los nodos a aplicar comienza su combinatoria de búsqueda repitiendo todo el procedimiento anteriormente explicado, pero añadiendo información en cada paso que avanza en función de los nodos descartados y/o desarrollados.

El algoritmo terminará cuando el estado en que se encuentre sea un estado final o haya recorrido todos los nodos disponibles.

V. Experimentos

Para la comprobación de nuestro algoritmo nos hemos apoyado en el código realizado por Arpan Ghosh [1]. Gracias a su programación podíamos comprobar si nuestra matriz solución coincidía con el verdadero resultado.

Podemos diferenciar dos fases diferenciadas por las que ha pasado el desarrollo de nuestro código.

En primer lugar, nos centramos en la resolución de tableros sencillos 3x3 con el uso de la búsqueda en Anchura. En cada fallo de tablero solución se hizo un debugueado de código hasta encontrar la raíz del fallo. Una vez detectado el error era solventado he intentado mejorar en eficiencia.

Este proceso era repetido por cada introducción de dificultad (4x4, 5x5…) hasta llegar a tableros de dificultad 6x6. Para llegar a estos niveles de resolución fue necesario aplicar técnicas de estrategias de resolución de tableros hitori, siguiendo una web especializada en su resolución [11].

Una vez conseguido la resolución de todos los tableros comenzamos a centrar el desarrollo en la optimización de tiempo de ejecución.

El primer paso que realizamos fue realizar una pequeña heurística para permitir la ejecución del algoritmo A* (ya que es algoritmo que permite el mayor rendimiento).

Con cada ejecución de las matrices fuimos analizando que parámetros eran los que castigaban con mayor nivel la expansión de un nodo y se fue variando las funciones de cálculo de heurística y coste de aplicación hasta llegar al punto de que nuestro algoritmo fue capaz de resolver matrices 9x9 con facilidad.

Llegados a este punto mejoramos las funciones de preprocesado para mejorar aún más el rendimiento de tiempo de ejecución. Tras realizar este procedimiento pasamos a realizar la ejecución de todas las matrices 9x9 en una solo tanda de ejecución para el estudio de resultados como explicaremos en el siguiente punto.

El rendimiento ha sido optimizado hasta el punto de que somos capaces de hallar la solución de matrices 20x20 en apenas segundos.

VI. Análisis de Resultados

Para este apartado comenzaremos con un análisis de los ejemplos de prueba ofrecidos para este proyecto.

Dicho documento consta de matrices de tamaño 3x3 hasta 6x6, usando el algoritmo A* podemos ver que conseguimos un tiempo de resolución prácticamente inmediato en cada uno de ellos con lo que conseguimos que en 0,26 segundos aproximadamente se consiga una resolución completa del fichero “ejemplos_prueba.txt”.

Se le adjuntará un archivo con el nombre “ResultadosEjemplosPrueba.txt” en el que se detalla el tiempo de ejecución de cada puzle, su solución obtenida y el tiempo de ejecución total del archivo.

Este sería el formato del archivo:

```
20 [[3,4,5,5,1,3],[5,6,2,3,2,1],[5,3,1,4,5,4],[1,4,3,4,2,2],[3,1,6,1,4,5],[1,2,1,5,3,4]]

Realizando la búsqueda, sea paciente por favor.

Matriz solución:
[0, 4, 5, 0, 1, 3]
[5, 6, 2, 3, 0, 1]
[0, 3, 1, 0, 5, 4]
[1, 0, 3, 4, 2, 0]
[3, 1, 6, 0, 4, 5]
[0, 2, 0, 5, 3, 0]
Número de componentes conexas: 1

La ejecución ha tardado: 0.06902074813842773

La ejecución total del archivo ha tardado: 0.2695159912109375 segundos

Process finished with exit code 0
```

Fig. 3. Ejemplo de tablero resuelto

A posteriori realizamos un análisis de los puzles del archivo "ejemplos_reto.txt" que data de 100 puzles de tamaño 9x9, a la hora de resolver dichos puzles nos encontramos desde puzles que son solucionados instantáneamente como algunos que son una gran minoría que tardan entre 20 y 60 segundos.

A la hora de ejecutar el algoritmo sobre dichos puzles no hemos sido capaces de lograr obtener una solución del puzle número 63(según la línea en la que se encuentre en el archivo) en un tiempo comprensible por lo que se paró la ejecución de dicho puzle.

Cuando se realizó la ejecución de todo el archivo se logró un tiempo de resolución de todos los puzles (excepto el 63 que se obvió) de 267 segundos (4 minutos y 45 segundos aproximadamente).

Como con el anterior archivo se ha adjuntado un documento con nombre "ResultadosEjemplosReto.txt" en el que se detalla el tiempo de ejecución de cada puzle, su solución y el tiempo total de ejecución.

A la hora tomar tanto los resultados de prueba como los de retos se ha obviado el uso de la interfaz gráfica usada como mejora, solo ha tenido en cuenta el uso del mismo entorno de programación ya que queremos enfocarnos en el tiempo de resolución.

En el caso de que se quiera realizar una ejecución de un archivo entero puede usar "executeTXT.py" con la ruta correspondiente del archivo, dicha clase leerá por cada línea si existe un puzle y lo introducirá en el algoritmo A* mostrando su solución y tiempo de ejecución.

```
archivo =
open('C:/Users/anton/Documentos/IA/hitori/
ejemplos_prueba.txt')
```

Ya que obtuvimos un buen rendimiento se realizaron pruebas con puzles de mayor tamaño a los ofrecidos como reto recogidos de varias fuentes tanto de tamaño 12x12 como 20x20 con los que se obtuvieron unos tiempos de resolución menores a un segundo, esto se explicará más detalladamente en mejoras.

Tras todo el desarrollo del algoritmo hemos llegado a la conclusión de que el hecho de implementar las técnicas explicadas anteriormente ofrece un gran rendimiento ya que somos capaces de introducir un puzle con casillas ya tachadas al algoritmo, cosa que nos reduce sensiblemente tanto el número de nodos como la complejidad del puzle.

Con especial hincapié en el hecho de que al tachar una casilla se revisen las adyacentes con el fin de "fijarlas a su valor" y tachar las que tengan su mismo valor tanto en su fila como en su columna.

1	1	3	5	5
4	2	5	3	1
4	3	3	5	4
5	1	2	3	3
3	3	4	1	1

Fig. 4. Ejemplo de tablero resuelto

Al tachar el 5 se ha "fijado" la casilla con valor 3 de abajo y se procedería a tachar la otra casilla con el mismo valor en la misma columna que se encuentra redondeada en rojo.

VII. Mejoras

Como mejora ha sido desarrollada una interfaz de uso amena donde se requiere la interacción del usuario.

En primer lugar, se le solicita al usuario que introduzca el tablero hitori a resolver. Este tablero viene definido por un array de filas, donde cada fila es un nuevo array.

Un ejemplo de introducción de tablero podría ser [[1,2,1],[2,2,1],[3,1,2]] el cual representa el siguiente tablero:

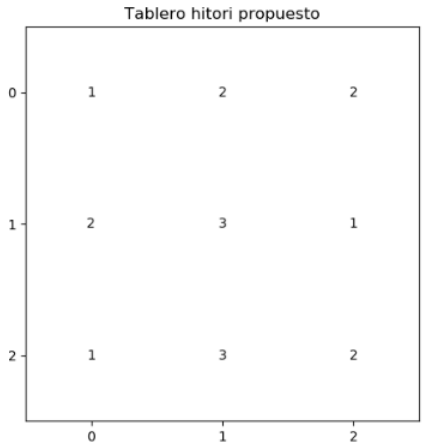

Fig. 5. Ejemplo de tablero propuesto

Una vez ha sido introducido un tablero válido se le solicitará al usuario que introduzca el tipo de búsqueda a aplicar pudiendo ser búsqueda en anchura, búsqueda en profundidad, búsqueda óptima o búsqueda A*.

Para todos los tipos de búsqueda (a excepción de la búsqueda A*) se le solicita al usuario que indique si desea mostrar los detalles de la búsqueda o no.

Después de establecer los parámetros personalizables comienza la búsqueda. Una vez definida esta, si una solución ha sido encontrada se devuelve el tablero anterior con la celda oscurecida en el caso de que la celda haya sido tachada:

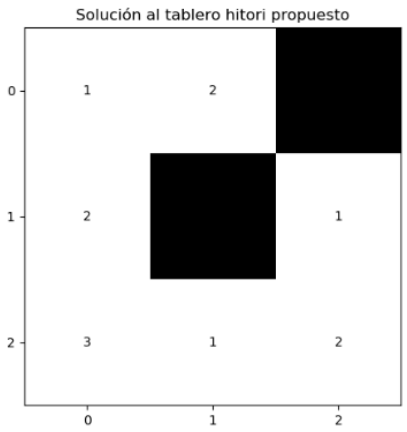

Fig. 6. Ejemplo de tablero resuelto

Una vez terminada la búsqueda se le da la opción al usuario de realizar una nueva o finalizar la ejecución.

Por otro lado, se ha encapsulado todo el código en un único archivo .exe por lo que el usuario solo deberá ejecutar este archivo y comenzar la búsqueda. En su uso tenga en cuenta que tras abrir el programa tarda entre 20 y 40 segundos en cargar.

Puede ver una captura de ejemplo de funcionamiento de nuestro código en YouTube [9]

En la siguiente captura puede verse un ejemplo de la ejecución del programa.

Fig. 7. Ejecución gráfica del código resolviendo un tablero

Una vez cierre la pestaña de solución se le indica al usuario si quiere continuar, solicitando un nuevo tablero o terminar con la ejecución del programa.

Queremos destacar que nuestro algoritmo es capaz de resolver matrices de hasta 20x20 en menos de un segundo, como muestra de ello, hemos usado la matriz que se encuentra en el archivo reto 20x20, la cual ha sido generado en la web indicada en la propuesta de problema [10]

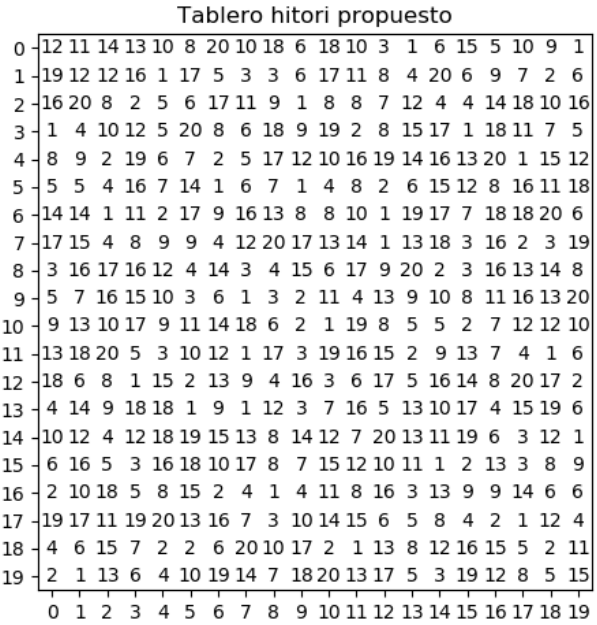

Fig. 8. Tablero 20x20 a resolver

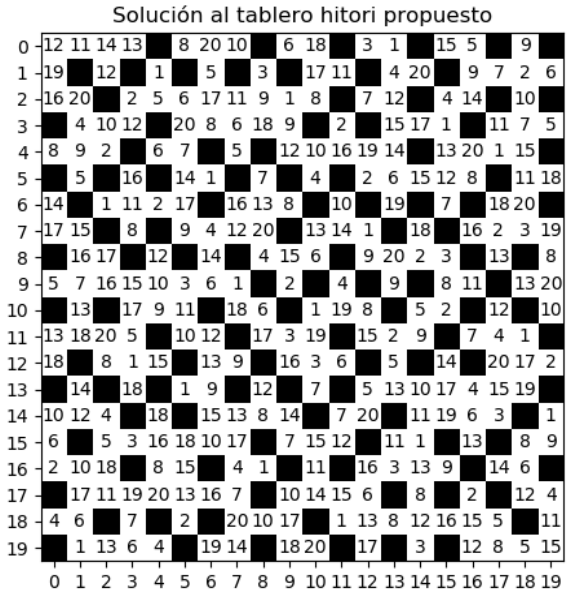

Fig. 9. Tablero 20x20 resuelto

Como mejora se ha realizado una clase que dado un archivo con un puzle por cada línea es capaz de resolver todas leyendolas una por una y mostrando por consola su solución, su tiempo de ejecución y el tiempo de ejecución del archivo entero, para su uso es necesario generar el archivo con los puzles con su sintaxis adecuada y habría que pasarle la ruta del archivo para que la ejecute

VIII. Conclusiones

Concluimos que hemos desarrollado un algoritmo bastante eficiente ya que es capaz de realizar matrices 20x20 cuando la propuesta de reto esta conformada por matrices 9x9.

Sin embargo, el punto fuerte de nuestro algoritmo se centra en el preprocesado de las matrices, que es capaz de eliminar una gran cantidad de celdas tachadas, con lo que ahorramos combinatoria a la hora de expandir nodos.

Somos conscientes de que no contamos con una buena definición heurística ya que en el mejor de los casos solo nos ahorra 1 ó 2 segundos en la resolución de grandes matrices.

En lo referente a nuestra interfaz gráfica creemos que hemos logrado diseñar un sistema bastante intuitivo y personalizable, en el que el usuario no necesita conocer apenas función alguna del sistema para comenzar a usarlo.

IX. Referencias

[1] Repositorio GitHub de apoyo para la resolución de tableros https://github.com/arpanghosh8453/programs/tree/master/myprojects-Python_3/Python_hitori_solver

[2] Librería Matplotlib https://matplotlib.org/

[3] Librería Numpy https://www.numpy.org/

[4] Librería Scipy https://www.scipy.org/

[5] Repositorio de desarrollo https://github.com/alvar017/ia

[6] Librería Scipy https://www.scipy.org/

[7] Pair programming https://en.wikipedia.org/wiki/Pair_programming

[8] ¿Qué es GitFlow? http://aprendegit.com/que-es-git-flow/

[9] Muestra de la interfaz gráfica en funcionamiento https://youtu.be/QASOYTZuO_U

[10] Web referencia para la generación de tableros hitori http://www.menneske.no

[11] Web de estrategia de resolución hitori https://www.conceptispuzzles.com/index.aspx?uri=puzzle/hitori/techniques