Aplicación de diferentes técnicas de búsqueda en el espacio de estados al juego 'Wappo Game' y la importancia de A en este contexto.*

David Gil Baeza
Universidad de Sevilla
Sevilla, España
davgilbae@alum.us.es

Roberto Hermoso Núñez
Universidad de Sevilla
Sevilla, España
robhernun@alum.us.es

El objetivo principal del proyecto es poder resolver mediante una inteligencia artificial los niveles, tanto con un monstruo como con dos monstruos, del juego Wappo game, inspirado en el juego de Teseo y el minotauro [1], inventado por Roger Abbott.

El objetivo del juego es guiar el protagonista, localizado en un mapa rectangular o cuadrado a una casilla objetivo, en el que se encuentran uno dos monstruos, los cuales intentarán alcanzar al protagonista. Destacar que en el caso de dos monstruos, si estos se juntan se fusionaran, serán inmunes a las trampas y se moverán más rápido. El objetivo que el personaje llegue a la casilla objetivo sin ser cazado. El protagonista se moverá de uno a uno, sin embargo, el monstruo se moverá de dos en dos excepto cuando se fusionan, cuyo movimiento será de tres en tres.

Palabras Clave—Inteligencia Artificial, algoritmos, búsqueda en el espacio de estados, búsqueda en anchura, búsqueda en profundidad, búsqueda con A*, heurística, coste, nodos, python.

I. Introducción

Este proyecto está orientado para ser resuelto mediante una Inteligencia Artificial, programada en el lenguaje Python. Dentro del ámbito de la Inteligencia se usará la búsqueda en el espacio de estados [2], en la cual se trata de encontrar una solución a un problema modelado y representado, definiendo un *espacio de estados,* un estado inicial, que será un nivel cualquiera de Wappo Game (nodo), el estado final, que es el caso en el que el protagonista llegue al objetivo, y las reglas a partir de las acciones disponibles, que en este caso son los movimientos que puede hacer el personaje (arriba, abajo, derecha o izquierda).

La resolución por espacios de estados consiste en buscar, mediante un árbol de nodos, la solución. Existen varias formas de realizar este árbol de nodos, pero en este caso hemos usado:

1. Achura
2. Profundidad
3. A*

Estos tres algoritmos serán desarrollados y explicados en los siguientes apartados. Primero explicaremos exhaustivamente en el apartado de "Preliminares" en cómo funcionan los algoritmos de búsquedas por espacios de estados, utilizando los tres métodos de resolución .

II. Preliminares

Métodos empleados

Como hemos mencionado en la introducción, hemos usado búsqueda en espacio de estados. En esta técnica se usan árboles de búsqueda. En cada nodo de ese árbol se encuentra un estado del problema concreto, y los hijos que cuelgan de un nodo son consecuencia de aplicar una de las posibles acciones del problema.

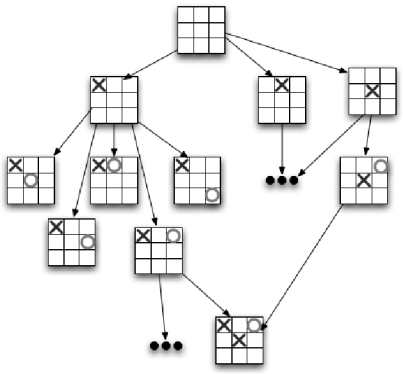


Fig 1. Ejemplo de Problema de búsqueda básico. Extraido de http://www.cs.us.es/~fsancho/?e=33

Existen diferentes métodos de búsquedas de la solución, como mencionamos anteriormente. Nosotros usaremos en concreto tres algoritmos diferentes, los cuales describiremos a continuación:

1) Anchura

El método por resolución por Anchura consiste en explorar los nodos de tal forma que se priorice la anchura del árbol. Como ya veremos en el apartado de "Resultados", este algoritmo de búsqueda en los niveles con los cuales vamos a experimentar es eficiente y eficaz, y es capaz de devolver la solución óptima, cosa que no ocurre siempre con la búsqueda en profundidad que veremos a continuación.

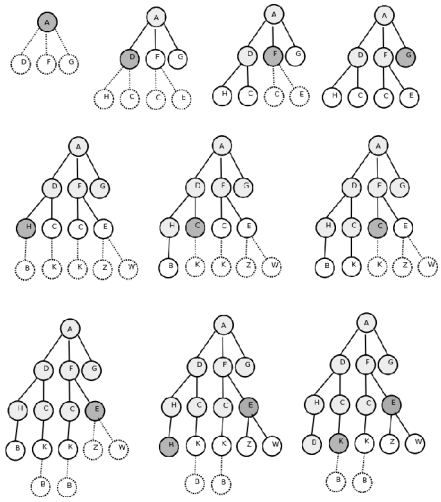


Fig 2. Ejemplo por búsqueda en anchura, extraido de: http://www.cs.us.es/~fsancho/?e=95

2) Profundidad

Consiste en encontrar la solución en el árbol explorando primero cada rama al completo hasta llegar a su fin.

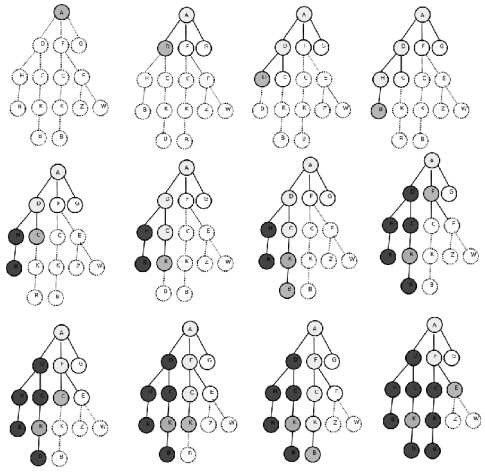


Fig 3. Ejemplo por búsqueda en profundiad, extraido de: http://www.cs.us.es/~fsancho/?e=95

En ocasiones es más eficiente que el algoritmo de búsqueda en anchura, aunque no tiene por qué devolver la solución óptima.

*3) A**

Este algoritmo es el más interesante, ya que puede ser guiado mediante el uso de funciones de coste y heurística, haciendo que el algoritmo pruebe primero aquellos nodos y ramas que puedan estar más cerca de la solución.

El algoritmo A*[3] es un algoritmo que se usa para el cálculo de caminos mínimos de una red de nodos. Se trata de un algoritmo heurístico, esto quiere decir que evalúa los nodos con una función dada, determinando entonces si el camino es óptimo o no.

Esta función está compuesta a su vez de dos funciones. Una de ellas indica la distancia del nodo origen hasta el nodo a etiquetar ($g(n)$), mientras que la otra indica la distancia estimada desde el nodo actual hasta el nodo destino ($h(n)$).

$$f(n) = g(n) + h(n)$$

h(n) es la función heurística, es la que estima cuán lejos estamos de encontrar la solución, en el caso de que esta función nunca sobrestime el valor de la distancia real entre el nodo y el destino, se dice que es admisible y será entonces una solución óptima.

Por otro lado, la función de coste evalúa cada nodo. De esta manera, el algoritmo juega con la heurística y el coste intentando devolver la solución cuyo camino sea el de menor coste.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18		20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Fig 4. Ejemplo de Algoritmo de búsqueda A*. Obtenido de: https://es.wikipedia.org/wiki/Algoritmo_de_búsqueda_A*

III. METODOLOGÍA

A. Modelado del mapa

Para empezar hemos modelado el problema con las clases *Mapa* y *Mapa2*, según si el nivel contiene uno o dos monstruos, cuya diferencia es que la segunda clase contiene dos monstruos en lugar de uno. Estas clases componen el estado de cada nodo y contiene la siguiente información:

1) *Las celdas*: Una matriz n x m que define el mapa del nivel. La celda superior izquierda será la celda (0,0) y la celda inferior derecha será la celda (n,m). Los valores posibles para cada celda de la matriz están comprendidos entre 0 y 3, indicando:
   a) 0: Casilla normal.
   b) 1: Casilla ardiente, donde el monstruo se quedará paralizado durante 3 turnos, y por lo tanto el personaje podrá moverse libremente sin preocuparse del monstruo. También hay

que tener en cuenta si el personaje colisiona con esta casilla perderá el juego.

c) 2: Casilla Objetivo, es la casilla donde tiene que llegar el personaje.

d) 3: Casilla inaccesible, es una casilla que no puede ser atravesada por ningún elemento del juego, se usa como casilla auxiliar por si la casilla objetivo está fuera del mapa, y así introducir una fila o columna ficticia. Se trata de una casilla que podría equivaler a una casilla normal rodeada de paredes.

- En la definición del mapa juega un papel muy importante la función *hay_pared,* la cual es muy usada en otras funciones del algoritmo y simplifica mucho el código de estas. También es importante la definición de la función *__eq__,* la cual permitirá al algoritmo evaluar correctamente cada nodo evitando repetir nodos que ya han sido visitados. Esto es clave para que los niveles sean resueltos eficientemente.

2) *El monstruo***:** Estará siempre definido su posición en el mapa, indicando la coordenada *i* y la coordenada *j*. También tendrá un contador, inicializado a 0, que indicará el tiempo que está paralizado (tiempo de *stun*) en el caso de que caiga en una casilla ardiente. El monstruo se moverá de dos en dos, priorizando primero estar en nuestra misma columna y después de en la misma fila, este orden es importante para la resolución del problema. En todos los niveles el monstruo se mueve de dos en dos, y cuando pisa una trampa o casilla ardiente se paralizará durante 3 turnos, pero en aquellos niveles con dos monstruos, si estos caen en la misma casilla, se fusionarán, quedando inmunes a las trampas y moviéndose de tres en tres.

3) *El personaje:* Indicamos en todo momento en que coordenadas se encuentra, al igual que con el monstruo. Aquí no hay posibilidad de cambiar el número de movimientos por turno del personaje, por lo que siempre se mueve una casilla por turno.

4) *Paredes*: Definimos también una lista de tuplas de casillas, indicando que entre esas dos casillas se encuentra una pared, estas paredes no se pueden atravesar de ninguna manera ni por el monstruo ni por el personaje. Estas paredes son muy importantes para la resolución de la mayoría de niveles, pues buscaremos siempre encajar el monstruo detrás de una de estas paredes.

Contamos además con otros elementos auxiliares:

- *Celda genérica:* No juega un papel en la representación del mapa, pero se trata de un objeto con coordenadas *i* y *j* con la cual poder referenciar a cualquier celda del mapa de manera sencilla donde se necesite.

B. Acciones: Aplicabilidad y aplicación

Hemos definido las únicas cuatro acciones posibles que puede realizar el personaje: Izquierda, derecha, arriba y abajo. Para ello usamos las clase *MoverPersonaje* y *MoverPersonaje2,* de nuevo según el número de monstruos. Ambas heredan de la clase *Accion y guardan* dos atributos: *coste* y *dirección*. El movimiento del monstruo o monstruos quedará determinado por la nueva posición del personaje tal y como detallaremos a continuación.

La función de aplicabilidad (*es_aplicable),* contenida en ambas clases es la función que define si es posible o no realizar una acción, y es que la que se tendrá que ejecutar antes de aplicar un movimiento. Hemos decidido contar en esta función como acciones imposibles las acciones que conlleven: Atravesar una pared, acceder una casilla inaccesible, salirse del mapa, situarse en la misma casilla que el monstruo o monstruos (ya sea por que el personaje se encuentra junto a un monstruo y una acción llevaría a ocupar la misma casilla que el monstruo o bien porque la acción a realizar conlleva que uno de los monstruos pille al personaje, lo que conlleva calcular *a priori* las futuras posiciones de los monstruos, algo costoso sobre todo teniendo dos monstruos) o pisar una casilla ardiente. Estas dos prohibiciones permiten que se pueda llegar a la solución sin que el personaje sea pillado por el monstruo. Esta función es usada en cualquier algoritmo de búsqueda para evitar buscar por nodos imposibles o que simplemente hagan que perdamos el juego.

Por último, la función *aplicar,* creará el nuevo nodo en base a la acción que haya elegido el algoritmo, cambiando la ubicación del personaje y a continuación calculando la nueva posición de cada monstruo dependiendo de dónde esté el personaje, priorizando primero estar en la misma columna y después en la misma fila. Si tenemos un solo monstruo en el nivel, si está paralizado por haber pisado una casilla ardiente en un nodo anterior, este deberá permanecer en la casilla en la que está y solo reducir el número de turnos sin moverse que le quedan al monstruo. Si de lo contrario tras aplicar la acción pisa una casilla ardiente, habrá que bloquear el movimiento del monstruo para futuros turnos (el monstruo podrá pisar una casilla ardiente tras su primer movimiento del turno o tras el segundo. Sea como sea, se bloqueará el movimiento del monstruo en cuanto pise la casilla ardiente aunque solo haya realizado un movimiento en el turno). Hay que contemplar el caso en el cual el monstruo queda liberado, y por lo tanto puede moverse, al aplicar la acción el monstruo permanece en la pero misma casilla (la casilla ardiente en la cual se encontraba). En este caso, no se bloqueará el movimiento del monstruo en los siguientes

turnos. Como cabe esperar, en los niveles con dos monstruos la dificultad se ve incrementada debido a que hay que tener en cuenta lo mismo que ya hemos mencionado y otros muchos factores, como que uno de los monstruos puede estar paralizado, pero no el otro, y principalmente el tratamiento de la fusión cuando estos coinciden en la misma celda. Estos son los puntos clave que han dificultado la implementación de una acción.

Con esta información que hemos definido ya podríamos realizar pruebas con la búsqueda en anchura y en profundidad, pero nos falta introducir diferentes costes y heurísticas para usar con A* y sacar verdadero partido a este algoritmo, principalmente en niveles de grandes dimensiones, ya que en niveles pequeños no se notará demasiado la diferencia.

C. Coste y heurística

Debido a que en la propia función *es_aplicable* definimos, no solo las acciones imposibles, si no las que nos conducen a la derrota (reduciendo así además los nodos explorados por el algoritmo), nosotros hemos decidido desarrollar una buena heurística para guiar al algoritmo de manera correcta y contemplar solo dos posibles costes para cada nodo.

1) *Coste 0:* De esta manera, no existe una solución mejor que otra. Se ha priorizado que el algoritmo encuentre solución sin importar que el número final de pasos del personaje o acciones a realizar para llegar a la celda objetivo. Tal y como se ha indicado en la introducción a este apartado, lo restrictiva que es la función *es_aplicable* y el uso de una buena heurística permite que el algoritmo encuentre una solución óptima.
2) *Coste 1:* El algoritmo considerará que una solución será mejor que otra si se han dado menos pasos para alcanzarla.

En la heurística hemos tenido en cuenta muchos factores, y el algoritmo echará mano de una de las varias heurísticas que disponemos para estimar cuán lejos está un nodo de la solución.

1) Distancia Manhattan (Personaje - Objetivo) (1)

Para cada nodo la heurística es la distancia distancia en casillas que hay desde el personaje al objetivo. En el caso del nivel 0, lo único que debe hacer el personaje es acercarse hacia la casilla objetivo, no tiene que usar ninguna casilla ardiente para bloquear al monstruo ni tampoco tiene que llevar al monstruo a ninguna pared intermedia, dando rodeos por el mapa. El monstruo empieza ya bloqueado entre las paredes para que el protagonista pueda llegar al objetivo sin problema.

Esta heurística nos sirve para ambos tipos de niveles, con uno o dos monstruos.

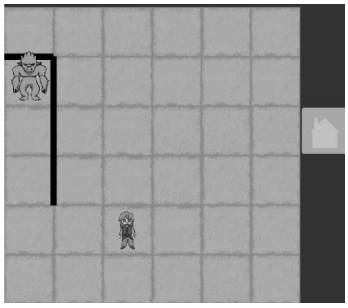


Fig4. Nivel 0 del videojuego Wappo Game

Las siguiente heurísticas son solo aplicables a los **niveles con un solo monstruo.**

2) Heurística contemplando casillas ardientes y diferentes situaciones

No todos los niveles son más sencillos, ya que entran en juego otros aspectos como las casillas ardientes, u otras situaciones en las que será necesario encajar al monstruo en alguna pared, dando un rodeo por el mapa, para llegar al objetivo. Por lo tanto tendremos en cuenta diferentes distancias.

1. *Distancia del personaje al objetivo*
2. *Distancia del monstruo al objetivo*
3. *Distancia del personaje al monstruo*
4. *Distancia del monstruo a una casilla ardiente*
5. *Distancia del personaje a una casilla ardiente*

Estas distancias deben de ser consideradas, ya que en base a ellas y a diferentes situaciones que se planteen la heurística se calculará de una manera u de otra.

Si el monstruo está más cerca del objetivo que del personaje no tiene más remedio que dar una o varias vueltas por el mapa para bloquearle en alguna pared y poder llegar al objetivo. Le podemos bloquear usando las paredes, pero principalmente usando las casillas ardientes, ya que están pensadas para ser utilizadas para paralizar al monstruo y llegar al objetivo, aunque si hay más de una casilla ardiente no significa que tengamos que usarlas todas para llegar al objetivo. Se plantean evalúan las siguientes situaciones o condiciones:

1. *Si existe alguna casilla ardiente, el personaje está cerca de una de esas casillas, el monstruo no está bloqueado en una casilla ardiente y la distancia del monstruo al objetivo es menor que dos veces la distancia del personaje al objetivo, la heurística será:*

   *4 * distanciaMonstruoArdiente*

2. *Si la distancia del monstruo al objetivo es menor o igual a la distancia del personaje al objetivo, el*

monstruo no está bloqueado en una casilla ardiente y la distancia del monstruo al objetivo es menor que dos veces la distancia del personaje al objetivo, la heurística será:

(2 max(filasMapa, columnasMapa)- distanciaPersonajeObjetivo - distanciaPersonajeMonstruo).*

3. *En caso contrario, la heurística será la distancia del personaje al objetivo.*

Finalmente se sumará uno a la heurística calculada en base a lo anterior por cada pared que se interponga entre el personaje y el objetivo.
:

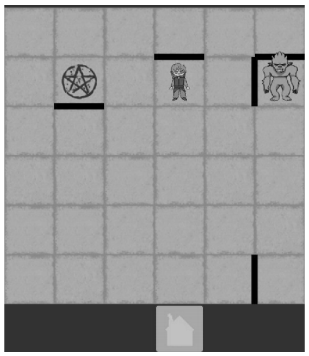


Fig 5. Nivel 1 del videojuego Wappo Game

En los niveles en los que solo dispongamos de una casilla ardiente será suficiente con calcular la distancia del personaje y del monstruo a la casilla ardiente. Por lo tanto, en el caso del nivel 1, como el personaje está más cerca del monstruo que de la casilla objetivo, tratará de hacer que se acorte la distancia del monstruo con respecto a la casilla ardiente, una vez que el monstruo quede paralizado la distancia que tendrá en cuenta la heurística será la de *Distancia Manhattan.*

Sin embargo, en el caso de que se encuentren dos casillas ardientes en el mapa, se tendrá que considerar una de las dos casillas, de tal forma que la casilla ardiente pueda ser elegida:

1. *Aleatoriamente. (4)*
2. *La que esté más alejada al personaje. (3)*
3. *La que esté más cercana al personaje. (2)*

Debido a que dependiendo del mapa será obligatorio para su resolución llevar el monstruo a una casilla u otra, la más universal y polivalente de las tres opciones es la primera, elegida de forma aleatoria, aunque puede provocar que el número de nodos explorados por el algoritmo sea diferente en cada ejecución del mismo al introducir esta aleatoriedad.

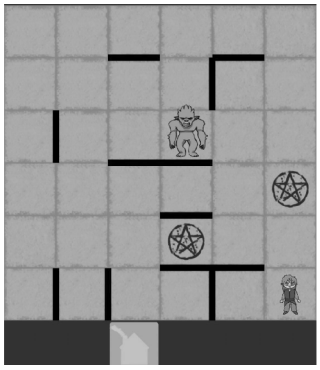


Fig 6. Nivel 3 del videojuego Wappo Game

En el caso del nivel 3, el de la figura 6, será necesario usar la casilla ardiente que se encuentra justo encima del personaje, pero la otra casilla ardiente es totalmente irrelevante para la resolución del nivel.

En este nivel se da una situación aún más compleja que en los anteriores, debido a que para resolverlo hay que encajar al monstruo en las dos paredes que se encuentran más cercanas a la esquina superior derecha del mapa, y una vez que ha sido bloqueado, dirigirse a la casilla objetivo quedando de nuevo bloqueado al monstruo en las dos paredes horizontales del centro del mapa.

3) Heurística favoreciendo situaciones en las que el monstruo sea encajado entre dos paredes contiguas (5)

Se trata de una ampliación de la anterior heurística en la variante en la cual se escoge una casilla ardiente aleatoria. Si no se cumplen las dos primeras condiciones de la heurística planteada anteriormente, en lugar de devolver la distancia entre el personaje y el objetivo se realiza otra comprobación. Calculamos para cada nodo si el monstruo está encajado entre dos paredes contiguas y si además hay una pared o paredes en dirección al personaje, quedando el personaje ‘libre’ de peligro. La heurística quedaría así:

1. *Si existe alguna casilla ardiente, el personaje está cerca de una de esas casillas, el monstruo no está bloqueado en una casilla ardiente y la distancia del monstruo al objetivo es menor que dos veces la distancia del personaje al objetivo, la heurística será:*

   *4 * distanciaMonstruoArdiente.*

2. *Si la distancia del monstruo al objetivo es menor o igual a la distancia del personaje al objetivo, el monstruo no está bloqueado en una casilla ardiente y la distancia del monstruo al objetivo es menor que dos veces la distancia del personaje al objetivo, la heurística será:*

*(2 * max(filasMapa, columnasMapa) - distanciaPersonajeObjetivo - distanciaPersonajeMonstruo).*

3. *Si el monstruo está bloqueado por dos paredes contiguas, el personaje está libre de peligro, el monstruo no está bloqueado en una casilla ardiente y la distancia del monstruo al objetivo es menor que dos veces la distancia del personaje al objetivo, la heurística será:*

*2 * distanciaPersonajeObjetivo - distanciaPersonajeMonstruo*

4. *En caso contrario, la heurística será la distancia del personaje al objetivo.*

D. Definición del problema

La clase que hemos creado para definir el problema, la cual usará el método *buscar* para obtener la solución de un problema concreto, es *ProblemaWappo.* Esta clase contiene dos atributos: el estado inicial del mapa (instancia de la clase *Mapa* o *Mapa2*) y la lista de acciones posibles (cuatro instancias de la clase *MoverPersonaje* o *MoverPersonaje2*, una por cada dirección). En la clase *ProblemaWappo* encontramos la función *es_estado_final,* en la cual indicamos al algoritmo que será estado final aquel estado en el cual el personaje se encuentre en la celda objetivo.

E. Mejoras y cambios en el código base proporcionado

Para la realización de este proyecto hemos realizado pequeños cambios en el código proporcionado por el Dpto. de Ciencias de la Computación e Inteligencia Artificial.

1. Debido a un error en el código el método *buscar* de la clase *BusquedaGeneral* no permitía resolver ningún problema usando A*. Por ello se implementó aparte la clase *BusquedaEstrella* con los cambios necesarios para que funcione con A*.
2. El método *solucion* de la clase *NodoSimple* ha sido modificado para que devuelva una lista con dos elementos:
    a. Una lista con las acciones a realizar para alcanzar la solución.
    b. Una lista con todos los nodos por los que hay que pasar para llegar a la solución. Esto nos permite representarlos gráficamente y poder visualizar el mapa y la posición del monstruo y el personaje tras realizar cada movimiento con la clase *representation* que hemos implementado, de la cual hablaremos más adelante.

Como mejora se ha implementado una interfaz, la cual usaremos a través de la consola de Python (la interfaz ha sido realizada y probada en *Spyder*).

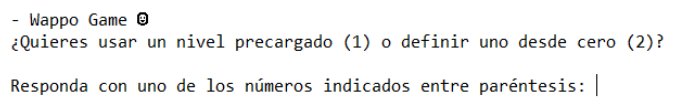


Fig 7. Interfaz desarrollada para la resolución del problema

Mediante el uso de números podremos seleccionar si queremos cargar un mapa predefinido, como un nivel del juego o un nivel de prueba, o si preferimos definir un **nivel desde cero.** De la misma manera podremos seleccionar qué algoritmo de búsqueda, heurística y coste usar, así como si queremos que la ejecución se realice de manera detallada y si queremos que la solución sea mostrada en un fichero 'resolucion_wappo.txt' con la representación del mapa por cada paso del nivel, el cual será generado en la carpeta raíz del proyecto, o bien si queremos que la solución sea mostrada en la misma consola de python. En ambos casos, podremos ver cuántos nodos han sido explorados y el tiempo de ejecución del algoritmo.

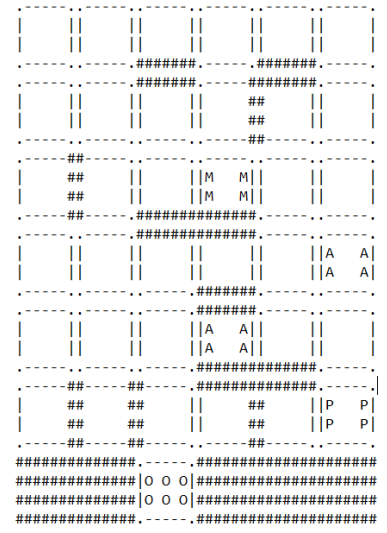


Fig 8. Ejemplo de representación del nivel 3 del videojuego Wappo Game

De igual manera se proporcionan dos ficheros mediante los cuales, si se desea, se podrá cargar un nivel y elegir algoritmo, coste, heurística y demás opciones de forma manual.

F. Medida de la eficacia y eficiencia

Para medir la eficacia veremos cuántos niveles somos capaces de resolver con cada algoritmo, y como medida de eficiencia para aquellos niveles que se hayan resuelto veremos cuántos nodos ha explorado el algoritmo y el tiempo (en segundos) que ha necesitado.

IV. Resultados

Como se ha descrito antes, el algoritmo A* es el más interesante respecto a los demás algoritmos de resolución que hemos probado, ya que nos permite guiar la resolución

mediante el uso de la heurística y reducir los nodos explorados, pero no siempre será el más eficiente. Será más conveniente usar este algoritmo en niveles grandes.

Comprobaremos ahora los diferentes resultados y tiempos que nos han dado los distintos algoritmos. Resolveremos los niveles desde el 0 hasta el 14, **subrayando en las tablas el nivel 12** para diferenciarlo al ser de dos monstruos. Si se desea repetir estas pruebas podrá hacerse uso de la interfaz proporcionada. Destacar que las pruebas se han realizado en un ordenador de sobremesa potente, y que en función del ordenador los resultados pueden variar mínimamente.

Primero comprobaremos los resultados con el algoritmo de **búsqueda en anchura** para todos los niveles. Además también hemos realizado mapas de prueba para comprobar que los métodos de aplicabilidad, aplicar, etc y las restricciones impuestas (como el bloqueo del monstruo en la casilla ardiente) funcionan correctamente, pero de igual manera nos sirven para evaluar también la eficiencia de los diferentes algoritmos. Estos niveles son los siguientes:

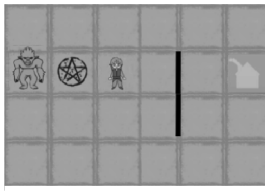


Fig 9. Mapa de prueba 0

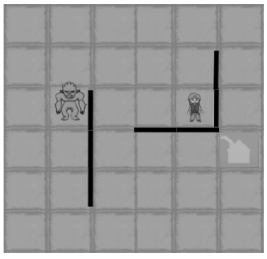


Fig 10. Mapa de prueba 1

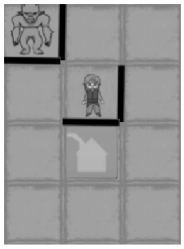


Fig 11. Mapa de prueba 2

Es importante añadir que la experimentación con estos niveles contribuyó en el desarrollo de las diferentes heurísticas expuestas anteriormente.

Table 1. Resultados de la búsqueda en anchura

Mapa	**Búsqueda en anchura**		
	Nodos explorados	***Tiempo (En segundos)***	***Movimientos***
0	39	0.0359	6
1	21	0.0419	21
2	60	0.0429	21
3	60	0.0609	22
4	65	0.0611	21
5	59	0.0589	21
6	81	0.0770	21
7	57	0.0533	21
8	52	0.1029	21
9	72	0.0669	21
10	56	0.0489	21
11	54	0.0539	21
12	56	0.0396	21
13	83	0.0719	21
14	82	0.0729	21
Prueba 0	32	0.0189	5
Prueba 1	16	0.0149	6
Prueba 2	7	0.0056	3

Como podemos ver el algoritmo resuelve todos los niveles planteados en centésimas de segundo, devolviendo para cada nivel la **solución óptima,** es decir, el mínimo número de pasos necesarios para resolver el nivel. El resultado del nivel 12 es bueno, quizás por las carácteristicas del nivel en concreto, aunque se necesita realizar más cálculos para su resolución que en el resto de niveles. A continuación probaremos la **búsqueda en profundidad.**

Table 2. Resultados de la búsqueda en profundidad

Mapa	**Búsqueda en Profundidad**		
	Nodos explorados	***Tiempo (En segundos)***	***Movimientos***
0	8	0.0090	8
1	22	0.0349	21
2	22	0.0309	21
3	23	0.0694	22
4	21	0.0319	21
5	21	0.0539	21
6	27	0.0614	27
7	21	0.0409	21
8	21	0.0759	21
9	27	0.0519	27
10	21	0.0339	21
11	23	0.1189	23
12	21	0.0391	21
13	26	0.0429	23
14	23	0.1159	21
Prueba 0	7	0.0070	7
Prueba 1	6	0.0069	6

Prueba 2	3	0.0019	3

Como podemos observar, este algoritmo de búsqueda es capaz al igual que el anterior de resolver todos los niveles, con una peculiaridad, y es que **no siempre devuelve la solución óptima.** Estos resultados son completamente normales, ya que el algoritmo explora rama a rama, y si en la rama que está explorando el personaje alcanza la casilla objetivo esta será la solución que devolverá el algoritmo, lo que explica que según el nivel necesite explorar menos nodos que en la búsqueda en anchura y por tanto sea más eficiente, al menos en los niveles probados y en otros de dimensiones similares. Destacar como en el caso del nivel 12 justo explora los mismos nodos que componen la solución.

Por último comprobaremos la eficacia y eficiencia de la búsqueda con el algoritmo A*. Realizaremos varias pruebas probando varias de las heurísticas diseñadas.

Primero comprobaremos los resultados obtenidos por la *Distancia de Manhattan*:

Table 3. Resultados de A* - Distancia de Manhattan

Mapa	**Coste**	**A* - Distancia de Manhattan (1)**		
		Nodos explorados	***Tiempo (En segundos)***	***Movimientos***
0	0	7	0.0069	6
1	0	38	0.0326	21
2	0	32	0.0279	21
3	0	51	0.0529	22
4	0	50	0.0491	21
5	0	51	0.0499	21
6	0	42	0.0429	21
7	0	41	0.0409	21
8	0	31	0.0299	21
9	0	51	0.0489	25
10	0	53	0.0529	23
11	0	39	0.0389	21
12	0	55	0.0386	21
13	0	72	0.0669	23
14	0	61	0.0616	21
Prueba 0	0	7	0.0039	5
Prueba 1	0	13	0.0119	6
Prueba 2	0	5	0.0029	3
0	1	16	0.0169	6
1	1	51	0.0399	21
2	1	60	0.0489	21
3	1	52	0.0529	22
4	1	58	0.0559	21
5	1	57	0.0608	21
6	1	56	0.0599	21
7	1	50	0.0469	21
8	1	45	0.0429	21
9	1	60	0.0599	21
10	1	55	0.0559	21
11	1	42	0.0409	21
12	1	56	0.0322	21
13	1	79	0.0739	21
14	1	74	0.0699	21
Prueba 0	1	25	0.0163	5
Prueba 1	1	33	0.0312	6
Prueba 2	1	7	0.0034	3

Como podemos comprobar el algoritmo explora menos nodos que en la búsqueda por anchura, aunque el tiempo no es en todos los casos mejor. Aunque se podría pensar que el tiempo debería ser entonces siempre menor, el tiempo que invierte el algoritmo en calcular la heurística de cada nodo produce que no se aprecie demasiado la mejora en este sentido. Sin embargo, los resultados son peores que en la búsqueda en profundidad, aunque en este caso sí que obtenemos la solución óptima.

Por otro lado, según podemos apreciar en los resultados, es más eficiente usar coste 0 en lugar de coste 1, por ello **en las pruebas siguientes usaremos siempre coste 0, y no probaremos los niveles de prueba 0, 1 y 2 debido a su baja complejidad, ni probaremos el nivel 12 debido al uso de heurísticas para niveles con un solo monstruo.** En la interfaz proporcionada se puede cargar cualquier prueba posible si se desea.

Ahora comprobaremos las *heurísticas contemplando diferentes situaciones,* primero haciendo pruebas con preferencia a la casilla ardiente más cercana al personaje.

Table 4. Resultados de A* - Casilla ardiente más cercana

Mapa	**A* - Casilla ardiente más cercana (2)**		
	Nodos explorados	***Tiempo (En segundos)***	***Movimientos***
0	8	0.0110	6
1	38	0.0339	21
2	28	0.0239	21
3	58	0.0659	22
4	32	0.0329	21

5	46	0.0499	21
6	32	0.0319	21
7	31	0.0329	21
8	37	0.0363	21
9	55	0.0569	21
10	53	0.0579	21
11	33	0.0329	21
13	61	0.0589	23
14	59	0.0599	21

Como podemos ver, en la mayoría de los niveles los nodos explorados y el tiempo han mejorado, pero no en todos los casos. En concreto, los resultados obtenidos han sido mejores en los niveles: 2, 4, 5, 6, 7, 11, 13 y 14. En el caso del nivel 13, el algoritmo no ha devuelto la solución óptima.

Debido a que existen mapas donde la solución es encontrada gracias a la casilla más alejada, realizaremos pruebas con esta la heurística pero teniendo en cuenta la *casilla ardiente más alejada*.

Table 5. Resultados de A* - Casilla ardiente más alejada

Mapa	**A* - Casilla ardiente más alejada (3)**		
	Nodos explorados	***Tiempo (En segundos)***	***Movimientos***
0	8	0.0089	6
1	38	0.0329	21
2	28	0.0250	21
3	59	0.0619	22
4	48	0.0510	21
5	46	0.0496	21
6	34	0.0389	21
7	37	0.0349	23
8	36	0.0379	21
9	55	0.0559	21
10	53	0.0580	21
11	33	0.0379	21
13	61	0.0569	23
14	59	0.0649	21

Tal y como vemos no se aprecian grandes diferencias en los resultados, salvo por que ahora en los niveles 7 y 13 el algoritmo no devuelve la solución óptima. Parece que con esta heurística no se ha logrado mayor eficiencia, pero podría ser interesante en niveles de mayores dimensiones.

Probaremos ahora una heurística que *seleccione una casilla ardiente aleatoria,* viendo que ocurre al introducir este elemento aleatorio.

Table 6. Resultados de A* - Casilla ardiente aleatoria

Mapa	**A* - Casilla ardiente aleatoria (4)**		
	Nodos explorados	***Tiempo (En segundos)***	***Movimientos***
0	8	0.0099	6
1	38	0.0339	21
2	28	0.0250	21
3	61	0.0672	22
4	48	0.0529	21
5	46	0.0499	21
6	29	0.0319	21
7	29	0.0289	21
8	30	0.0292	21
9	55	0.058	21
10	53	0.0569	21
11	33	0.0371	21
13	61	0.0944	23
14	59	0.0569	21

Al introducir cierta aleatoriedad en el algoritmo los resultados pueden ser cambiar de una ejecución a otra. En este caso podemos observar que aunque los resultados son similares, en el nivel 8 por ejemplo el resultado es mejor que en las dos heurísticas anteriores.

A continuación realizaremos las pruebas con la heurística teniendo en cuenta lo anterior y además teniendo en cuenta cuando el monstruo se quede encajado en dos paredes contiguas.

Table 7. Resultados de A* - Monstruo encajado en dos paredes contiguas

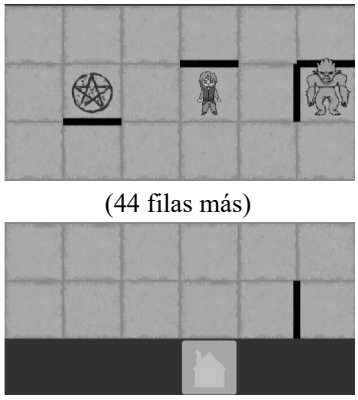
Mapa	**A* - Monstruo encajado en dos paredes contiguas (5)**		
	Nodos explorados	***Tiempo (En segundos)***	***Movimientos***
0	8	0.0100	6
1	38	0.0359	21
2	28	0.0240	21
3	60	0.0669	22
4	48	0.0529	21
5	46	0.0509	21
6	42	0.0449	21
7	31	0.0239	21
8	38	0.0399	21

9	55	0.0579	21
10	53	0.0569	21
11	33	0.0038	21
13	61	0.0619	23
14	59	0.0570	21

En esta última heurística observamos que los resultados son muy similares a los observados con la heurística anterior, principalmente en cuanto al número de nodos explorados.

En resumen, aunque se puede apreciar una mayor eficiencia usando A* con heurística y coste frente a la búsqueda en anchura, la diferencia no es demasiado notable. Pero, **¿qué ocurre si intentamos resolver un mapa mucho más grande?**

Para realizar esta prueba hemos diseñado el nivel de prueba 3, el cual está basado en el nivel 1 pero otorgándole de 50 filas en lugar de 7. De esta forma ahora los algoritmos tienen muchos más nodos que explorar y podremos ver qué diferencia existe si lo resolvemos con los diferentes algoritmos de búsqueda.



(44 filas más)

Fig 12. Mapa de prueba 3

Table 8. Comparativas de los métodos de búsqueda en el mapa de prueba 3

Búsqueda	**Mapa de prueba 3**		
	Nodos explorados	***Tiempo (En segundos)***	***Movimientos***
Anchura	3172	12.936	64
A* - Distancia Manhattan	82	0.0131	64
A* - Casilla ardiente cercana	82	0.1671	64
A* - Casilla ardiente alejada	82	0.1689	64
A*- Casilla ardiente aleatoria	82	0.1728	64
A* - Paredes contiguas	82	0.1729	64

Dado un nivel con unas dimensiones más elevadas, el algoritmo más eficiente es **A*, seguido por anchura y por último profundidad.** Este último no conseguía resolver el algoritmo tras un minuto de ejecución, por lo que se detuvo la ejecución del mismo. Aunque la primera heurística ha sido la más eficiente, en otros niveles mucho más complejos podría ser determinante el uso de cualquiera de las otras.

Para simular niveles complejos de gran dimensión bastaría con eliminar de la clase *Mapa* el método *__eq__*. Los algoritmos pensarán que todos los nodos explorados son diferentes, aumentando la complejidad. Niveles como el 1, 2, 6 o 7 solo podrán ser resueltos en ese caso con A* y con una muy buena eficiencia teniendo en cuenta la alta complejidad que se plantea haciendo ese cambio.

V. Conclusiones

Durante el estudio realizado hemos realizado pruebas con tres algoritmos diferentes: Anchura, profundidad y A*. Tras este proceso de experimentación realizado con estos algoritmos y con los niveles del juego, entre otros diseñados por nosotros, llegamos a la conclusión de que **A* es el algoritmo más interesante** para resolver el juego 'Wappo Game' y otros puzzles similares, sin embargo, **no en todos los casos.**

Si solo nos fijamos en los resultados obtenidos en los niveles del juego, aunque A* es ligeramente más eficiente, no es mucha la diferencia frente a los algoritmos de búsqueda en anchura y profundidad, por lo que quizás no es necesario invertir tiempo en el desarrollo de una buena heurística en estos casos debido a que los resultados de los otros dos algoritmos son suficientes. De hecho, puede que no merezca la pena asumir el tiempo que necesita A* para realizar el cálculo de la heurística de cada nodo.

Sin embargo, tal y como hemos podido comprobar en la última prueba realizada, donde realmente es interesante A* es en **aquellos niveles donde la cantidad de nodos posibles sea mucho más elevada,** ya que **no necesitará explorar todos los nodos para alcanzar la solución.**

En otros juegos o problemas de mayor dimensión y complejidad que 'Wappo Game' puede que ni anchura ni profundidad puedan resolver el problema planteado en tiempos aceptables y que la única opción a plantear sea directamente A*, por ello este algoritmo cobra mucha importancia en este contexto.

Referencias

[1] Teseo y el Minotauro. https://www.microsiervos.com/archivo/juegos-y-diversion/teseo-y-el-minotauro.html

[2] Espacios de estados http://www.cs.us.es/~fsancho/?e=33

[3] Algoritmo A* http://idelab.uva.es/algoritmo

[4] Página web del curso IA 2018/19 de Ingeniería del Software. https://www.cs.us.es/cursos/iais/.