Universidad Zaragoza
1542

Trabajo Fin de Grado

Inteligencia Artificial académica moderna
Modern academic Artificial Intelligence

Autora

Dª Virginia Casino Sánchez

Directora

Dra. Dª Piedad Garrido Picazo

Escuela Universitaria Politécnica de Teruel, Universidad de Zaragoza
2022

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Agradecimientos

En esta página de mi TFG, quería dedicar unas palabras a todas esas personas que han estado en mi vida en estos 4 últimos años correspondientes a mis años de universidad en este grado de Ingeniería Informática.

No han sido unos años fáciles, pero sí que lo han sido entretenidos porque salgo de esta carrera muy satisfecha de haberla elegido, y, las personas que me han acompañado en estos años han hecho que fuera más divertido. Gracias a mi familia y seres queridos que me han apoyado, han estado conmigo en todo momento, han confiado en mí, mucho más de lo que a veces hacía yo misma, y no me han soltado nunca de la mano, al igual que mi directora de TFG, Piedad Garrido que de igual manera a aguantado todos mis agobios como una más de mi familia. Además, sin todas las nociones que he aprendido a lo largo de estos años tampoco hubiera sido posible estar hoy aquí, por ello, gracias a todos mis profesores que me han inculcado y compartido todo sus conocimiento y sabiduría.

Como he dicho, acabo esta carrera muy contenta y satisfecha por lo que he aprendido tanto de conocimientos académicos, como de la vida, y lo que ha supuesto para mi, por las personas que he conocido y por las que me demuestran día a día que van a estar siempre para mí ayudando y apoyándome en todo momento.

Resumen

En este documento se presenta un Trabajo Fin de Grado (TFG) con un enfoque centrado en la docencia, donde se lleva a cabo el desarrollo de una práctica, sobre agentes basados en objetivos, para que los alumnos la puedan realizar en la asignatura de Inteligencia Artificial (IA) en el Grado de Ingeniería Informática (GII) impartido en la Escuela Universitaria Politécnica de Teruel (EUPT).

El objetivo que se desea conseguir con la práctica que se ha realizado es preparar al alumnado para determinar cuándo un enfoque es adecuado para la resolución de un problema concreto, identificando la representación apropiada, el mecanismo de razonamiento, así como su implementación y evaluación.

Se lleva a cabo desde cero, empezando por la labor de investigación de realizar algo que todavía no existe, la búsqueda de una interfaz útil para no destinar tiempo a algo innecesario, la implementación sobre el código de la interfaz de los algoritmos de búsqueda exigidos en la asignatura y, parámetros necesarios para la dotación de la inteligencia artificial al agente y alguna prueba o resultado sobre el código implementado. Se emplea el juego del Pac-Man para que sea más amena y entretenida la labor del estudiante al no tener que programar algo "sin sentido", sino que se trata de un videojuego lo cual suele resultar más atractivo.

Palabras claves

Agente basado en objetivos, Gamificación, Inteligencia Artificial (IA), Pac-Man, Heurística

1. Introducción y Objetivos

El presente Trabajo Final de Grado (TFG) tiene como objetivo enfocar el estudio de la Inteligencia Artificial (IA) hacia la preparación del alumnado para determinar cuándo un enfoque es adecuado para la resolución de un problema concreto, identificando la representación apropiada, el mecanismo de razonamiento, así como su implementación y evaluación. El principal problema a abordar se centra en conseguir que el alumnado no vea la IA como una materia aislada sino integrada en el resto de disciplinas de su formación, gracias a la elaboración de una serie de novedosos recursos Tecnologías de Información y Comunicación (TIC) de apoyo, que se prevén obtener como resultado de este trabajo académico.

El núcleo central de este TFG radica en la implementación de una IA en un juego a resolver mediante los distintos algoritmos de búsquedas enseñados en dicha asignatura en la Escuela Universitaria Politécnica de Teruel (EUPT) [29, 30], que se basan en agentes por objetivos que se desplazan sobre un espacio de estados. Concretamente los algoritmos empleados en este proyecto son de búsqueda ciega o no informada (primero en profundidad, coste uniforme y primero en anchura) [29] e informada, que a diferencia de las otras aplican conocimiento sobre cómo llegar al objetivo y hacerla más eficiente (voraz y A^*) [30]. Este conocimiento viene dado por una función que estima la "bondad"de los distintos estados, dando preferencia a los que son mejores, según el criterio de una función heurística.

La elección del uso de un juego para explicar estos algoritmos a los alumnos, consiste en darle un enfoque distinto a la carrera, ya que los videojuegos no es un tema que se aborde mucho en la EUPT y es una posible salida laboral más, y, además de hacer el proceso más entretenido y enriquecedor para ellos, porque al programar un juego el atractivo es mayor. El juego a abordar es el clásico Pac-Man, un personaje formado por un círculo amarillo con boca que ha de comerse toda la comida disponible dentro de un laberinto azul con fondo negro. Tras alcanzar el objetivo avanza de nivel, pero en todo momento hay una serie de fantasmas que son los enemigos del Pac-Man y le quitarán vidas cada vez que le alcancen, así hasta que o bien el Pac-Man se quede sin más vidas o consiga pasarse todos los niveles [53]. También existen unas cápsulas que cuando el Pac-Man se las come, es invencible y por mucho que se le acerque un fantasma éste no

le hace daño, sino que se lo come. Eso sí, una vez se ha comido un fantasma, con el efecto de una de estas, deja dicho fantasma de estar asustado y puede volver a quitarle vidas al Pac-Man.

En este trabajo se ha llevado a cabo el desarrollo y creación de la práctica en cuestión que deberían realizar los alumnos para interiorizar y poder llevar a cabo un proceso de mayor comprensión e interiorización de algunos de los algoritmos de búsquedas explicados en la asignatura de IA. Para ello, primero se ha llevado a cabo una investigación de si es cierto que mediante los videojuegos los alumnos adquieren mejor los conocimientos, además de para conocer todo lo que se ha realizado hasta hoy. Luego se analizan ciertos proyectos ya disponibles en Internet, por otras personas, para dotar a sus Pac-Mans de inteligencia, pero como se verá no siguen los mismos criterios que en este proyecto. Posteriormente, se expone la propuesta de Pac-Man, con su análisis y diseño, así como su implementación que parte de un código ofrecido por la Universidad de Berkeley y los resultados obtenidos. Más adelante se exponen los temas de accesibilidad, usabilidad, licencias software y documental. Y, para finalizar, las distintas conclusiones y el trabajo futuro. En el apartado de anexos, se encuentra un posible borrador de lo que podría ser el enunciado de la práctica a mostrar para los alumnos.

En definitiva, los principales objetivos del TFG son:

1. Enfocar el estudio de la IA hacia la preparación del alumnado para determinar cuándo un enfoque es adecuado para la resolución de un problema concreto, identificando la representación apropiada, el mecanismo de razonamiento, así como su implementación y evaluación.

2. Cumplir con el Objetivo de Desarrollo Sostenible (ODS) de Educación de Calidad, en concreto 4.4. que consiste en aumentar considerablemente el número de jóvenes y adultos que tienen las competencias necesarias, en particular técnicas y profesionales, para acceder al empleo, el trabajo decente y el emprendimiento [50].

2. Contextualización

Este apartado ha sido añadido para explicar un poco el propósito del porqué he elegido este proyecto como TFG. Mi idea desde muy pequeña era ser profesora pero claro nunca se nos ocurre pensar en un profesor de pequeños y pensar en los de la universidad. Por lo que, cuando entré en la carrera, mis disposición fue ya intentar buscar el poder acabar dando clases de lo que aprendía en ésta. Piedad, mi directora de TFG, al conocer mis pensamientos, me planteó el poder llevar un proyecto, desde el punto de vista del docente: desarrollar un dosier con la práctica que tus alumnos han de realizar.

El proyecto que se ha llevado a cabo no ha pretendido ser una implementación de inteligencia artificial donde el Pac-Man fuera capaz de resolver todos los laberintos sin ser comido. Si se ha dotado al Pac-Man de inteligencia, pero desde un nivel muy bajo que emplea uno de los conceptos más básicos y simples de la inteligencia artificial como son los algoritmos de búsqueda, por lo tanto, no porque sea racionalmente inteligente va a hacerlo todo y lo va a hacer perfecto, todo lo contrario. Se pretende que yo y los alumnos podamos ver que prácticamente cualquier juego puede emplear inteligencia artificial con los algoritmos de búsqueda y que no todas las implementaciones van a ser las mejores, ya que no darán los mismos resultados para el mismo juego e incluso para el mismo laberinto, cada uno es un mundo.

La implementación del software parte de conocimientos básicos enseñados en la EUPT, como las estructuras de datos o grafos, esto no es un tipo de programación al uso sino que son pequeños programas, donde unos agentes ejecutan una idea según los algoritmos que le han sido indicados. Estos tratan de buscar el camino más eficiente posible, y aunque hay veces que se creen haberlo encontrado, hay alguno más listo que encuentra uno mejor al suyo.

3. Estado del Arte

3.1. Parte investigación

En este apartado se expone el estado del arte de una selección de artículos de la bibliografía donde se recoge la labor de investigación llevada a cabo sobre el videojuego del Pac-Man, con distintas implementaciones que tienen en común la utilización de la IA. Aclarar que todas las veces que se nombra agente, se está denominando, al protagonista del juego Pac-Man que suele estar bajo el control de los usuarios o los posibles fantasmas en alguno de los casos.

Primero, se van a comentar una serie de artículos que plantean un uso de algoritmos de búsqueda para llevar a cabo la dotación de dicha "inteligencia"del agente.

En el artículo [20], se plantea el uso de algoritmos de búsqueda para recorrer el camino más eficiente con el agente, con la finalidad de alcanzar la mayor cantidad de comida.

El proyecto [51], realiza 4 experimentos en base a distintos algoritmos de búsqueda centrado en el juego del Pac-Man para luego poder realizar una comparación en términos de rendimiento, integridad y optimización.

En el [58], se lleva a cabo la implementación de varios algoritmos de búsqueda de árboles avanzados con el objetivo de poder realizar una comparación entre estos en base al juego del Pac-Man. El modelo aplicado que es más óptimo, para este juego, es el aprendizaje por refuerzo basado en modelos.

En el artículo [39], se expone un ejemplo de un algoritmo de búsqueda en árbol puesto en práctica sobre el juego del Pac-Man, empleando la técnica de captura de pantalla y algoritmos evolutivos aplicados sobre el agente.

El artículo [10], se centra concretamente en el algoritmo de lógica difusa haciendo uso del Q-learning. Gracias a este enfoque se logra poder abordar los aspectos no deterministas del juego del Pac-Man y encontrar un auto-aprendizaje o una estrategia adaptativa, para el agente, en base a unos valores como la distancia a la píldora más cercana o distancia al fantasma más cercano.

Sobre el juego del Pac-Man, se plantean dos formas para controlar el agente en el artículo [26]. Una son las reglas bien definidas diseñadas por humanos, basadas en el algoritmo de Dijkstra y la segunda es el empleo de la computación evolutiva, mediante las redes neuronales. Ambas maneras pueden coexistir e incluso ofrecen un mayor rendimiento que por separado.

En el artículo [3], también es empleado el algoritmo de Dijsktra que sigue unas reglas además de un algoritmo de búsqueda de árboles para la implementación del agente y la ayuda de una cuadrícula gráfica base para representar el estado del juego.

El [14], se centra en el juego del Pac-Man enfocado con un propósito educativo para ayudar a los estudiantes a realizar una comprensión de los algoritmos de búsqueda de la inteligencia artificial. Los resultados que se obtuvieron fueron excelentes y afirman que el empleo de un juego para el aprendizaje de conceptos teóricos hace más sencillo y ameno el proceso, además, de facilitar su comprensión.

El [11], se centra en el ámbito escolar, como el anterior artículo y en el juego del Pac-Man, donde se plantean 4 enfoques distintos que son: búsqueda en el espacio de estado, búsqueda multiagente, inferencia probabilística y aprendizaje por refuerzo. Para ello los alumnos han de hacer uso de heurísticas de búsqueda, funciones de evaluación y características. El mero hecho de realizar el trabajo para el estudiante mediante un videojuego, queda demostrado que resulta de mayor provecho y entusiasmo para ellos.

El artículo [44], también se centra en realizar el proceso de adquisición de conocimientos para el alumnado de IA, mediante un juego, concretamente el Pac-Man. En este caso abarca los algoritmos genéticos de optimización de parámetros. Destacar que hace hincapié en que para que un alumno comprenda mejor los algoritmos y sus diferencias entre ellos, es mejor trabajar siempre sobre el mismo marco de trabajo, ya que hace que se percaten menos de las posibles diferencias a notar si no se realiza de esta manera.

Al igual que se plantea implementar IA en el agente o los fantasmas, en el artículo [41], se propone el uso de ésta para generar distintos laberintos. Se realiza mediante un algoritmo genético en función de aptitud y propone un fin para el juego.

De la misma forma que el artículo anterior, la programación genética es en más casos empleada para la implementación de una IA en el agente, como ocurre en el artículo [1].

Existen otras variantes de búsqueda, en árbol, mediante el algoritmo de Monte-Carlo. En los artículos [17, 48, 23, 28, 2, 16, 42], se plantea tanto para el agente como para los fantasmas en tiempo real en algunos casos. Incluso se abordan

distintos comportamientos para el agente, pudiendo ser prioridad la puntuación más alta o solo alcanzar el siguiente. Para el caso de los fantasmas se trata de predecir los movimientos de los agentes para poder alcanzarlo mediante esta técnica, adaptando la capacidad personal del jugador de manera proporcional al nivel del desafío. A pesar de proporcionar buenos resultados y ser un algoritmo muy eficiente, estos requieren de una IA más curanzada, ya que la asignatura de IA del GII de la EUPT es básica y sólo cubre 6 créditos, por lo que no se tendrán en cuenta para la implementación del juego en la práctica a realizar.

Una alternativa menos vista en los artículos encontrados, es la implementación del agente mediante un modelo de mapa de influencia, como se lleva a cabo en el artículo [55]. Los resultados que se obtienen a pesar de ser buenos, no son los mejores comparados con otros artículos y dio problemas tanto el modelo como el software a la hora de su implementación.

Al igual que en el artículo anterior se exponía una nueva alternativa no tan frecuente como otras, en éste [57], ocurre de manera similar. Se hace uso de una malla de navegación (NavMesh) para optimizar el problema de la búsqueda. Destaca por su gran área de implementación, especialmente en el ámbito de los juegos. El método hace uso del algoritmo A* y un motor de juego Unity 3D. Este algoritmo dota de inteligencia a 3 de los fantasmas, y se notó una diferencia con respecto a los restantes a la hora de las pruebas.

Otra implementación para el agente sería mediante un aprendizaje automático.

En el [27], se realiza una implementación de aprendizaje automático para los fantasmas, modificando el camino de estos, pudiendo hacer un mejor o peor movimiento dependiendo del nivel en el que se encuentre un jugador. Pero, también se aplica dicho aprendizaje automático al agente, con tres niveles de juego (principiante, intermedio y experto), para poder llevar a cabo las suficientes pruebas para determinar un modelo de aprendizaje adecuado a los criterios.

El artículo [4], se centra en el aprendizaje mediante refuerzo a través de algoritmos de extracción, que producen entradas para una red neuronal Q-learning. Los resultados obtenidos son muy buenos a pesar de las limitadas entradas que tiene la red neuronal, e incluso, Pac-Man es capaz de desenvolverse en distintos laberintos.

El artículo [37], se centra también en el aprendizaje automático por refuerzo mediante el uso de redes neuronales profundas, proponiendo un nuevo método empleando Deep Neuronal Network (DNN). En otros casos, como el artículo [13], se emplea una máquina de estados finitos simples y un conjunto de reglas para la

aplicación de un aprendizaje incremental basado en la población (PBIL) para ajustar los parámetros del agente.

Al igual que los anteriores artículos, el artículo [45] se centra en el paradigma del aprendizaje automático por refuerzo, buscando una optimización del comportamiento para reducir la duración y maximizar el rendimiento del aprendizaje, mediante un nuevo algoritmo que realiza una extracción de información útil de demostraciones de expertos y lo emplea para su mejora. Esta aplicación demuestra una evidencia estadística significativa de la mejora en el rendimiento final.

En el artículo [8], se plantea un aprendizaje automático únicamente para un fantasma. De tal manera, que mediante el uso de redes neuronales cuyas entradas son la posición y el estado del Pac-Man y cuya salida es la dirección a donde debe moverse dicho fantasma, se puede realizar al final una comparación entre el fantasma con conocimientos y los restantes, que son controlados por el guión tradicional, ya que el primero aprende correctamente y juega mejor que los otros fantasmas.

Uno de los principales desafíos en la IA es lograr, en la mayor medida posible, simular el comportamiento humano. En el artículo [22], se lleva a cabo el desarrollo de jugadores virtuales mediante el uso de la neuroevolución. Esto es una forma de aprendizaje automático que hace uso de algoritmos evolutivos para entrenar redes neuronales artificiales. Se emplea el juego del Pac-Man para poder comparar dos metodologías: datos sin procesar extraídos del rastro humano y agregar niveles de juegos más elaborados. En base a unos parámetros de medición, como la puntuación final se evalúa la importancia de estas características, para poder imitar de la mejor manera el juego humano.

En otros casos, se hace uso del Webcam Pacman, un navegador de juegos donde se puede jugar al Pac-Man, empleando un modelo de aprendizaje automático. El conocimiento es aprehendido de un conjunto de datos de entrenamiento y se transfiere a otro modelo. Esta manera es mencionada y analizada en los artículos [46] [5].

Actualmente existen competiciones para que, sobre un tema, se aplique la IA y la implementación de esta sea la más óptima y eficaz. El artículo [40], se centra en una competición para el juego del Pac-Man donde los participantes deben desarrollar controladores para el agente o para los fantasmas interactuando directamente con el motor de éste. Se recogen en él alguna de las revisiones de los trabajos previos y puesta en marcha, que se presentan a esta competencia.

3.2. Parte académica

En esta parte del trabajo se pretende mostrar la novedad o la diferencia de lo que ya está realizado y se puede encontrar por la web y, el enfoque que se quiere dar al proyecto para que pueda ser empleado en un ámbito académico. Se ha elegido un tema tan popular para realizar este trabajo, que tras elaborar el estado del arte se llegó a pensar que podría estar muy machacado y estudiado, pero no fue así, tal y como se va a explicar a continuación.

Aunque haya muchas fuentes, códigos y muchas implementaciones para dotar de IA al Pac-Man, no se ha llegado a encontrar ninguna que trabaje desde el bajo nivel desde el que se parte en este caso. Hay mucho repositorio en la plataforma Github que han dado soluciones al código inicial ofrecido por la Universidad de Berkeley sin dicha inteligencia, como por ejemplo el que se puede encontrar en la siguiente url: `https://github.com/karlapalem/UC-Berkeley-AI-Pacman-Project/blob/master/search` o `https://github.com/errikosg/Berkeley-AI-Pacman/blob/master/Project1-search` o `https://github.com/LtVaios/Berkeley-Pacman-Project/blob/main/Pacman_project1/search`. Como se verá más adelante en esta propuesta, se desea implementar unos algoritmos de búsqueda que requieren principalmente la implementación de: un espacio de estados, una función sucesor, el concepto de nodos expandidos y generados. En estos casos expuestos, se emplea sin embargo, mucho código del ya ofrecido por la Universidad de Berkeley siendo una de las principales diferencias que cada algoritmo de búsqueda tiene su método y distintas resoluciones, mientras que en el caso que se plantea en ese trabajo, se pretende que los alumnos entiendan que tanto las búsquedas no informadas como las informadas parten de la misma base, trabajando de manera diferente uno de los elementos más importantes de estas técnicas: la frontera. Por lo que, compartirán código a excepción de dicha frontera. Tal como está hecho en la bibliografía ese concepto no se muestra a los alumnos.

Además, no queda definida claramente la función sucesor, que ésta tenga un objetivo de manera externa al código donde se aloja el algoritmo de búsqueda aunque estén relacionados. La función sucesor contiene un método donde se obtienen todas las posibles acciones desde una posición. Incluso ya en esta se tienen en cuenta los conceptos de coste de camino, el concepto de visitado y expandido, y, en la implementación que se va a trabajar en este proyecto, sólo es empleado dentro de los algoritmos de búsqueda, una metodología de programación y organización del código que no se encuentra en la bibliografía.

De igual manera, otra parte muy importante es, la definición de un estado con parámetros que puedan indicar si se ha alcanzado la meta final o no, es empleada, pero

de una manera que resulta tediosa, no se ve a simple vista y emplea una lógica un tanto compleja para resolver un problema que se puede enfocar de otra manera más sencilla. Remarcando también, que el objetivo de muchos de estos casos, no es el acabar con la comida sino alcanzar las cuatro esquinas del laberinto por lo que no se tiene en consideración acabar el juego sino llegar a estos cuatro puntos, de ahí la complejidad de ese posible estado.

En definitiva, la estructuración del trabajo a plantear debe seguir rigurosamente los conceptos de un estado claro y definido del cuál se pueda partir y alcanzar una meta, que irá ligada a la cantidad de comida que queda en el laberinto. Esto supondrá una parte, la otra podrá ser la definición de la función sucesor que generará distintos estados. Dentro de los algoritmos de búsqueda ya se podrán distinguir los términos de expandido, generado, coste temporal del camino y expansión desde el cuál se llamará a la función sucesor.

4. Propuesta Pac-Man - EUPT

En este apartado se lleva a cabo la explicación del análisis, diseño e implementación de la propuesta del TFG. Este trabajo consiste en la creación de una práctica para la asignatura de IA, concretamente para los algoritmos de búsquedas impartidos en el temario 2 de la asignatura [29, 30].

Para realizar este proyecto, se ha partido de un código que ya consta de la interfaz del Pac-Man. Al ser un juego tan famoso se encontraron muchas interfaces, existen incluso torneos para ver quién dota de la mejor IA al Pac-Man para que pase el mayor número de niveles solo, o al fantasma para que consiga que encuentre al fantasma en el menor tiempo posible. El código que se ha escogido ha sido obtenido de la siguiente url: `http://ai.berkeley.edu/search.html`. Es un código proporcionado por la Universidad de Berkeley, que pretende que cualquier usuario de la red, pueda dotar de inteligencia al Pac-Man y ofrece recursos para ello, pero estos recursos y la lógica que quiere emplear para incorporar dicha inteligencia no sigue los principios básicos que se imparten en la asignatura de la EUPT, por lo tanto, sirve para emplear la interfaz, pero no el resto de la lógica de la aplicación.

4.1. Análisis y Diseño

Comenzando por la fase de análisis del trabajo, como se parte de un código ya diseñado, se ha tenido que llevar a cabo una labor de conocimiento de éste para poder estudiar donde se podía introducir dicha inteligencia. Para ello, se busca poder aislar la interfaz del resto de código [9].

En el código de la Universidad de Berkeley se pueden encontrar los siguientes ficheros:

- La carpeta *layouts*, que contiene una serie de ficheros con distintos diseños de laberintos.
- La carpeta *test_cases* donde están unos ejercicios que plantea la Universidad de Berkeley, para que los alumnos puedan adquirir y poner en práctica sus conocimientos de manera progresiva. Ésta contiene los casos de prueba para cada una de ellos.

- El fichero *search.py* es donde el alumnado debe incluir los algoritmos de búsqueda.
- El fichero *searchAgents.py* contiene los agentes basados en búsquedas.
- El fichero *pacman.py* es el archivo principal que ejecuta los juegos de Pac-Man.
- El fichero *game.py* contiene la lógica detrás de cómo funciona el mundo Pac-Man, como se mueve, las acciones, etc.
- El fichero *util.py* consta de unas estructuras de datos útiles para implementar algoritmos de búsqueda.
- El fichero *graphicsDisplay.py* implementa los gráficos para Pac-Man.
- El fichero *graphicsUtils.py* es donde se encuentra el soporte para los gráficos del Pac-Man.
- El fichero *textDisplay.py* contiene los gráficos ASCII para Pac-Man.
- El fichero *ghostAgents.py* tiene los agentes para controlar fantasmas.
- El fichero *keyboardAgents.py* consta de interfaces de teclado para controlar el Pac-Man mediante teclas.
- El fichero *layout.py* implementa el código para leer los archivos de diseño de la carpeta layout y almacenar su contenido: donde se encuentra la comida, las paredes y la posición inicial de Pac-Man.
- El fichero *autograder.py* que es un autograder de proyectos, es decir, un autocalificador que simula el proceso de probar un programa [38].
- El fichero *testParser.py* analiza los archivos de prueba y solución del *autograder*.
- El fichero *testClasses.py* vinculado a las preguntas que plantea la Universidad de Berkeley, son clases generales de prueba de autocalificación.
- El fichero *searchTestClasses.py* son clases de prueba de autocalificación específicas de los algoritmos de búsquedas, implementadas para el banco de preguntas comentado anteriormente.

Partiendo de esta estructura, se va a observar el código con un mayor lujo de detalle, para discernir cuál es necesario para el trabajo académico que se pretende desarrollar. Pero antes, se va a mostrar el diagrama de relación entre los distintos paquetes que hay

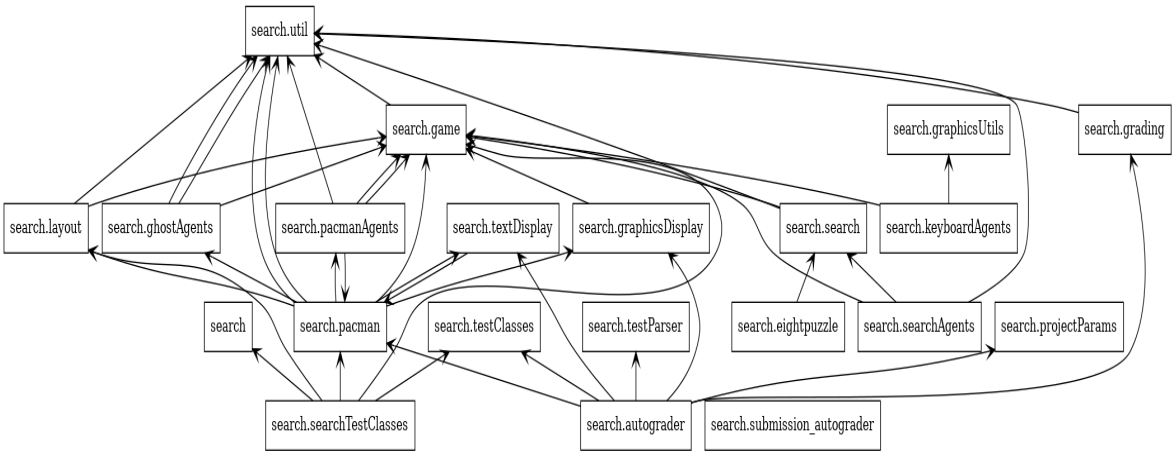

Figura 4.1: Paquetes del código original

en este proyecto, generado con la herramienta de Python, *pyreverse*, que puede verse en la Figura 4.1.

Con respecto al diagrama de clases, debido a su tamaño se deja como para su consulta en el respositorio de GitHub del proyecto.

Ahora se procede a observar el proyecto de manera detenida para conocer su ejecución. Comenzando por el fichero principal, que contiene la función *main* del programa. La función principal es un *entry point* o punto de entrada, que es la ubicación en el código donde se produce una transferencia de control (ejecución) del programa [21]. Por lo tanto, cuando se ejecuta la aplicación, ésta es llamada la primera. En ella se lo primero que se hace es recoger una serie de parámetros pasados por consola y los transforma. Luego, llama al método *runGames* con dichos valores recogidos.

Antes de seguir con el análisis, para el trabajo académico a desarrollar se modificará esta forma de lanzar la aplicación, para evitar pasar los parámetros por consola, ya que suele resultar más tedioso. Se hará de manera que tendrán que modificar el valor de una variable para indicar la modificación de estos.

Los datos pasados por parámetros pueden ser todos los que se ven en la Figura 4.2, destacando el número de veces que se puede jugar, el laberinto que se va a emplear, el agente que se va a considerar tanto para el Pac-Man como para los fantasmas, el número de fantasmas que se desea tener, el zoom para ver la pantalla del juego y la velocidad con la que se desplacen los agentes.

Una vez obtenidos los datos pasados como parámetros por consola son tratados. Se observan aquellos que se han considerado interesantes. Para la interfaz del laberinto, se ha de obtener el diseño, en el caso de los agentes, tienen que ser procesados, llamándose a otra función denominada *loadAgent*. En ésta, lo que hace es comprobar si el agente

```
parser.add_option('-n', '--numGames', dest='numGames', type='int',
                  help=default('the number of GAMES to play'), metavar='GAMES', default=1)
parser.add_option('-l', '--layout', dest='layout',
                  help=default('the LAYOUT_FILE from which to load the map layout'),
                  metavar='LAYOUT_FILE', default='mediumClassic')
parser.add_option('-p', '--pacman', dest='pacman',
                  help=default('the agent TYPE in the pacmanAgents module to use'),
                  metavar='TYPE', default='KeyboardAgent')
parser.add_option('-t', '--textGraphics', action='store_true', dest='textGraphics',
                  help='Display output as text only', default=False)
parser.add_option('-q', '--quietTextGraphics', action='store_true', dest='quietGraphics',
                  help='Generate minimal output and no graphics', default=False)
parser.add_option('-g', '--ghosts', dest='ghost',
                  help=default('the ghost agent TYPE in the ghostAgents module to use'),
                  metavar = 'TYPE', default='RandomGhost')
parser.add_option('-k', '--numghosts', type='int', dest='numGhosts',
                  help=default('The maximum number of ghosts to use'), default=4)
parser.add_option('-z', '--zoom', type='float', dest='zoom',
                  help=default('Zoom the size of the graphics window'), default=1.0)
parser.add_option('-f', '--fixRandomSeed', action='store_true', dest='fixRandomSeed',
                  help='Fixes the random seed to always play the same game', default=False)
parser.add_option('-r', '--recordActions', action='store_true', dest='record',
                  help='Writes game histories to a file (named by the time they were played)', default=False)
parser.add_option('--replay', dest='gameToReplay',
                  help='A recorded game file (pickle) to replay', default=None)
parser.add_option('-a','--agentArgs',dest='agentArgs',
                  help='Comma separated values sent to agent. e.g. "opt1=val1,opt2,opt3=val3"')
parser.add_option('-x', '--numTraining', dest='numTraining', type='int',
                  help=default('How many episodes are training (suppresses output)'), default=0)
parser.add_option('--frameTime', dest='frameTime', type='float',
                  help=default('Time to delay between frames; <0 means keyboard'), default=0.1)
parser.add_option('-c', '--catchExceptions', action='store_true', dest='catchExceptions',
                  help='Turns on exception handling and timeouts during games', default=False)
parser.add_option('--timeout', dest='timeout', type='int',
                  help=default('Maximum length of time an agent can spend computing in a single game'), default=30)
```

Figura 4.2: Posibilidad de parámetros a pasar por consola en el juego del Pac-Man

pasado existe y, si es así, crea un objeto del agente que se ha indicado. Los parámetros restantes son almacenados en distintas variables.

Cuando ya tiene todos los parámetros almacenados, se procede a lanzar el juego, método *runGames*. En éste lo que se hace es llamar a la clase *ClassicGamesRules* para declarar una serie de reglas.

Esta clase contiene el método *init*, cada vez que se crea un objeto de dicha clase es llamado y, se ejecuta el código en su interior que suele ser para inicializar los atributos de este objeto [49]. También contiene el método *newGame* para iniciar un nuevo juego, *process* que procesa si un juego ha terminado o no, *win* para comprobar si el Pac-Man ha ganado y *lose* para ver si ha perdido. El método *getProgress* para ver el progreso del juego en base a cuanta comida le queda por comer, *agentCrash* para cuando un agente por una implementación errónea se salga del laberinto o cruce una pared, y más métodos para obtener valores como el valor de la variable tiempo que lleva a cabo

una cuenta atrás. Algunos de estos parámetros, como el último mencionado, no son de interés para los objetivos buscados.

Posteriormente, una vez se ha creado el objeto *ClassicGamesRules* se mete en un bucle para ejecutar los distintos juegos que se han indicado, o el valor por defecto que es 1. Una vez dentro del bucle, comprueba si se ha introducido un valor para la variable *numTraining* que se corresponde con si se desea llevar a cabo rondas de entrenamiento. Si es así entra en un modo denominado *beQuiet*. Este modo no muestra una interfaz gráfica y las reglas que tiene en consideración son menores. Dicho modo no se va a tener en cuenta para el trabajo ya que es muy importante e interesante que la interfaz sea visible en todo momento para que el alumno pueda ver como se desplaza el Pac-Man por las distintas casillas y como lo hace.

Luego, se crea un nuevo juego, mediante el método de la clase *ClassicGamesRules* comentado con anterioridad. En éste principalmente se declaran e inicializan los valores del estado del juego, como son los agentes de los que se dispone, los cuál los almacena en un vector denominado *agents* pero son distintos objetos. Esta información es relevante ya que se debe tener en consideración como son tratados los fantasmas y el Pac-Man para poder aplicarlo en la implementación futura. Todos son almacenados en el mismo lugar, pero al parecer para diferenciarlos, el Pac-Man recibe el identificador 0 y los fantasmas los números consecutivos a éste. A continuación, crea un objeto del estado del juego, *GameState*. Éste contiene el número de comida que hay en total, cuánta de ésta ha sido consumida, cuantas cápsulas hay y se ha comido, los agentes, el diseño de la interfaz que se va a emplear, cuál es la puntuación y, si ha ganado o perdido. Todos los parámetros han sido explicados anteriormente o se sobreentiende cuál es su valor, pero no se ha indicado cuándo se puntúa y cuál es el valor de dicha puntuación. Cuando el Pac-Man come, la puntuación se incrementa en 10, si consigue finalizar el juego se añade 500 al igual que si pierde se reduce en 500; y, si alcanza a un fantasma asustado ya que se ha comido una cápsula se añade 200. Además, cada vez que pasa un tiempo especificado se va decrementando en una penalización de 1 punto. La puntuación en nuestro caso se modificará para que se incremente cada vez que consuma una comida, para que el alumno pueda ver que su algoritmo ofrece ganancias siempre que alcanza parte de su objetivo, que es comer, y, no tendrá en cuenta posibles penalizaciones por tiempo o afectará si alcanza a un fantasma asustado o no, porque será elección del alumno si desea o no introducir fantasmas en la ejecución del código.

Además del objeto del estado de juego, se crea el juego, que contiene también a los agentes como en el estado del juego, pero además tiene variables para comprobar si un agente se ha salido del laberinto, las distintas reglas, cuál es el agente que comienza

según el identificador especificado, un tiempo y un historial de por donde se han movido los agentes.

Cuando ya se han inicializado todos los parámetros necesarios, se procede a lanzar el juego. Para ello se llama al método *run* que tiene el objeto del juego, denominado *Game*. Cuando se lleva a cabo la ejecución de éste, primero se inicia la interfaz, todos los agentes que deben aparecer en ella, donde habrá siempre un Pac-Man pero no tiene por qué haber fantasmas o no la misma cantidad. Cuando ya todo está listo, entonces se procede a mostrar la pantalla con el juego y comienza éste para el usuario. Para ello, se hace uso de un bucle infinito que va realizando acciones con cada uno de los agentes, comenzando por el Pac-Man, recoge la acción que debe realizar, la lleva a cabo, actualiza la interfaz y procesa si se ha ganado o no según las normas del juego especificadas en una distinta clase. Luego, incrementa el número de identificador para los agentes, y realiza la misma acción para estos, que serán los fantasmas. Una vez se han ejecutado todas las acciones para todos los agentes se vuelve a empezar con el Pac-Man, por su identificador 0.

Las acciones que llevan a cabo los agentes son diferentes para cada uno y dependerá del objeto con el cuál ha sido creado. Este tipo de agente era especificado al pasarlo por parámetro. Por ejemplo, un Pac-Man, puede ser creado según está diseñado el código, como *KeyboardAgent*, *LeftTurnAgent* o *GreedyAgent*. En el primer caso, el agente será manejado por la combinación de las teclas que emplee el alumno. En el segundo tipo de agente, éste siempre que sea posible se moverá a la izquierda y el último, es un caso de implementación de IA, que se pide a los alumnos que es el caso del algoritmo informado denominado voraz. Se puede ver en la Figura 4.3 como primero lleva a cabo la obtención de todas las acciones posibles desde la posición en la que se encuentra llamando a otro método, luego obtiene un número de sucesores devolviendo los distintos estados según las posibles acciones a realizar, comprueba cuál de ellos obtiene mejor resultado y devuelve éste, si hay más de uno con la misma puntuación, entonces entre todos estos retorna uno aleatorio. Como se indica en la parte académica (3.2) del apartado del Estado del Arte, la implementación que se lleva a cabo no es la misma que se pretende que los alumnos implementen porque no se puede trazar la estructura básica de cómo se enseña en las clases teóricas. Aunque es una forma más de implementar inteligencia artificial, no es la que se desea que los alumnos lleven a cabo.

Ya acabada la partida, se procede a almacenar los resultados en un fichero si no era un entrenamiento y supone un texitrecord. Y, si ya se han jugado todas las rondas que se habían indicado con el número de juegos, se muestra al usuario el valor de su puntuación máxima, todas las puntuaciones obtenidas, las veces que ha ganado y la

```python
class GreedyAgent(Agent):
    def __init__(self, evalFn="scoreEvaluation"):
        self.evaluationFunction = util.lookup(evalFn, globals())
        assert self.evaluationFunction != None

    def getAction(self, state):
        # Generate candidate actions
        legal = state.getLegalPacmanActions()
        if Directions.STOP in legal: legal.remove(Directions.STOP)

        successors = [(state.generateSuccessor(0, action), action) for action in legal]
        scored = [(self.evaluationFunction(state), action) for state, action in successors]
        bestScore = max(scored)[0]
        bestActions = [pair[1] for pair in scored if pair[0] == bestScore]
        return random.choice(bestActions)
```

Figura 4.3: Algoritmo voraz desarrollado por la Universidad de Berkeley

media de las mismas.

De manera amplia es el análisis que se ha podido realizar del código desde el inicio de éste, hasta que es ejecutada una partida. Para poder llevar a cabo una implementación del código la labor de investigación ha sido mucho más tediosa, tal y como se podrá ver más adelante y en el código entregado.

Se pretende que los alumnos solo tengan que llevar a cabo la implementación de la IA y que la estructuración del proyecto o del código sea la más cómoda posible, por lo que partiendo del esqueleto básico de la Universidad de Berkeley, los alumnos recibirán un proyecto con la estructuración que se muestra a continuación, donde se desechan los ficheros que no son necesarios y se reestructura la información que hay en estos para que quede de la siguiente manera:

- La carpeta *layouts*, que contiene una serie de ficheros con distintos diseños de laberintos se mantiene igual.
- Se crea otra carpeta denominada *graphics*, que contiene todos los ficheros vinculados con el diseño e implementación de la interfaz (que no será necesario que los alumnos modifiquen para la práctica), ya que sólo deberán consultar en fichero *layout* y se les indicará en el enunciado. Los ficheros que contiene esta carpeta son: *graphicsDisplay.py*, *graphicsUtils.py* y *layout.py*. Este último implementa el código para leer los archivos de diseño de la carpeta layout y almacenar su contenido, como se realizaba ya antes, pero con alguna modificación.
- El fichero *search.py* es donde están todos los algoritmos de búsqueda.
- El fichero *pacman.py* seguirá siendo el archivo principal que ejecuta los juegos de Pac-Man aunque tendrá sus modificaciones.

– Los ficheros *game.py* y *gameState.py* contienen la lógica detrás de cómo funciona el mundo Pac-Man también se mantendrá, pero con los cambios necesarios.

– El fichero *datastructures.py* consta de las estructuras de datos que serán necesarias para implementar los algoritmos de búsqueda.

– El fichero *agents.py* que contiene tanto los agentes como los fantasmas incluido el Pac-Man.

– El fichero *pacmanState.py* que contiene los estados necesarios y parte de la lógica para dotar al Pac-Man de IA.

Se puede ver en la Figura 4.4, el nuevo diagrama que muestra la relación que tendrán los distintos paquetes en el proyecto final.

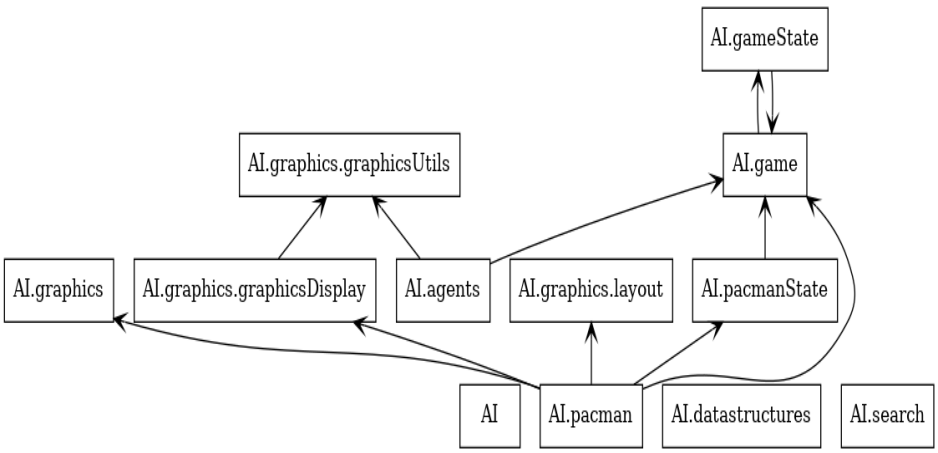

Figura 4.4: Paquetes del código a desarrollar

Para el diagrama de clases, de igual manera que para el del código original, se deja para su consulta en el repositorio de GitHub del proyecto.

Claramente, la disminución de código que se pretende llevar a cabo, que se puede ver con los diagramas, va a ser notable.

4.2. Herramientas tecnológicas empleadas

El lenguaje de programación para llevar a cabo el trabajo es Python [36] [15], debido a que es el que ya se está empleando actualmente en la EUPT para realizar esta práctica de algoritmos de búsqueda.

La versión a emplear de Python se pretende que no sea una específica y cualquiera le pueda servir al alumnado. El código de partida está preparado para la versión 2.7, pero se llevarán a cabo las modificaciones pertinentes para lograr que pueda ser ejecutado con otras versiones más actuales.

La herramienta de software empleada para la implementación del trabajo es PyCharm [32], un entorno de desarrollo integrado de Python. El paquete completo de este entorno consta de las características que se pueden observar en la Tabla 4.1.

Características PyCharm
Editor de Python inteligente
Depurador gráfico y ejecutor de pruebas
Navegación y refactorización
Inspecciones de código
Compatibilidad con Visual Computing System (VCS)
Herramientas científicas
Desarrollo web
Marcos de trabajo web Python
Perfilador Python
Capacidades para desarrollo remoto
Soporte para bases de datos y Structured Query Language (SQL)

Tabla 4.1: Características del entorno PyCharm

Este paquete tiene muchas cosas innecesarias para el desarrollo de la práctica y encima es de pago. PyCharm ofrece un paquete más reducido, denominado PyCharm Community Edition, gratuito y creado en código abierto. Consta de alguna de las característica comentadas anteriormente que son: editor de Python inteligente, depurador gráfico y ejecutor de pruebas, navegación y refactorización, inspecciones de código y compatibilidad con VCS.

Además, tanto la versión completa como la reducida puede ser ejecutada en los siguientes sistemas operativos: Windows, macOS Intel y Apple Silicon, y, Linux. Por lo que no se limita al uso de un sistema operativo concreto para poder llevar a cabo el desarrollo de cualquier proyecto.

4.3. Implementación

Antes de proceder con la implementación del código, se va a proceder a explicar de manera teórica en qué consisten los algoritmos de búsquedas en la IA, para luego poder entender mejor la parte práctica.

El principal objetivo de todo esto es resolver un problema y para ello crear un programa que sea capaz de automatizarlo. La resolución de un problema es considerada una capacidad inteligente.

Para todo esto se necesita un agente, es decir, una persona o cosa que percibe su entorno a través de sensores e interacciona con en él a través de actuadores. Estos agentes parten de un estado inicial y dependiendo de la acción que realicen llegan a

uno u otro, hasta alcanzar aquel que es considerado el estado final que es el objetivo. Existen 4 tipos de agentes: agente reflejo simple, agente reflejo basado en modelo, agente basado en objetivos y agente basado en utilidad. En este trabajo se hace uso de un agente reflejo basado en objetivos o también conocido como agente de resolución de problemas, donde las acciones van dirigidas a conseguir un objetivo. Este tipo de agentes utilizan representaciones atómicas, ya que se identifican los estados sin ninguna estructura compleja. Si empleasen alguna representación más sofisticada serían agentes de planificación.

En este caso la solución a plantear mediante estos algoritmos consiste en hacer búsquedas en un espacio de estados determinado y los pasos generales para poder resolver un problema son:

- El objetivo a alcanzar, en este caso es que el Pac-Man pueda acabarse toda la comida. De primeras pensando en los alumnos no se tiene en consideración que los fantasmas puedan o no alcanzarlo, eso será una consideración que se quiere llevar más adelante.

- Luego, se formula el problema, estados y acciones que se pueden llevar a cabo. Las acciones son ir a la derecha, la izquierda, arriba y abajo siempre cuando las barreras del laberinto lo permitan. La definición de estados viene dada más adelante.

- Posteriormente, se plantea el método de búsqueda que se va a plantear para determinar las posibles secuencias de acciones hasta lograr alcanzar el objetivo.

- Y, por último, se ejecutan todas las acciones de la secuencia.

Se trata de un entorno parcialmente observable ya que es posible detectar todos los aspectos del entorno para la elección de una acción a excepción de los fantasmas según lo que se plantea para el alumno, porque no tiene que tener en cuenta estos para encontrar la solución. En un trabajo futuro se va a llevar a cabo un entorno completamente observable ya que sí se tendrá en cuenta donde se encuentran los fantasmas para determinar la acción a realizar por el Pac-Man. Continuando con el entorno enfocado en la práctica para los alumnos, éste es determinista porque su estado viene determinado por el estado actual y la acción ejecutada por el agente. Es secuencial, porque sus decisiones vienen determinadas conforme a las decisiones tomadas anteriormente y estático porque el entorno no se modifica cuando el agente está determinando la siguiente acción que quiere realizar. Por último, es discreto porque las acciones del agente no dependen de su evolución en el juego y no es multiagente.

El espacio de estados del problema viene definido por el estado inicial y la función sucesor. Este espacio forma un grafo donde los nodos son los estados y las acciones son los arcos entre los nodos. El problema planteado es un grafo dirigido acíclico que para cada escenario será distinto, pero todos comparten que solo haya un estado inicial y uno final.

A continuación, se muestra un ejemplo de un grafo para un escenario concreto donde el estado inicial es el que se muestra en la Figura 4.5 y el estado final el que se muestra en la Figura 4.6. Cada caja representa un estado, en cuyo interior hay un valor que se corresponde con un número, concretamente el de comidas que hay en total en el laberinto escogido, éste es el valor que determinará la evolución del juego, pero también se ha de tener en cuenta la posición en donde se encuentra el Pac-Man en todo momento. La resolución del problema a plantear pretende poder ser empleado en cualquier espacio y no se centra en uno concreto, por lo que la posición no es un aspecto interesante a considerar en los estados para saber si gana o no, ya que si no se hace un estudio anterior de cuál va a ser la mejor posición en la cual acabar, puede suponer que el algoritmo sea menos efectivo por tener en cuenta este valor también y un mayor coste temporal porque requiere de un tiempo alcanzar esa posición óptima. Teniendo en consideración solamente la comida es posible finalizar el juego, y es el objetivo, al fin y al cabo. No es relevante para determinar si se ha finalizado o no, pero es necesario para poder conocer la posición del Pac-Man en todo momento. El estado final será que quede un valor de 0 comidas.

El interior de cada caja contiene el nombre que se le ha dado al estado, la posición que ocupa el Pac-Man entre corchetes, siendo el primer valor correspondiente al eje horizontal y el segundo al vertical; y, entre llaves, el valor realmente importante que es el de la comida.

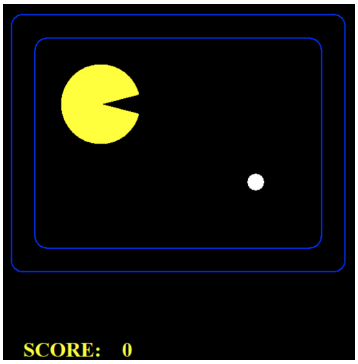

Figura 4.5: Ejemplo PacMan estado inicial

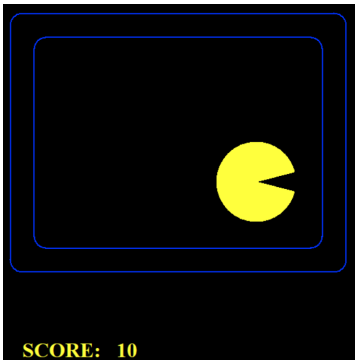

Figura 4.6: Ejemplo PacMan estado final

En la Figura 4.8 se muestra un posible grafo de lo que podría ser una ejecución donde cada arco simboliza una acción, donde se refleja entre corchetes dicha acción que se

está realizando en cada momento y hacia que celda. También se aporta el escenario del Pac-Man dividido en las distintas posiciones que puede ocupar para ese ejemplo (ver Figura 4.7). En este escenario existen hasta 6 cuadrículas por las que el Pac-Man se puede desplazar, partiendo del estado inicial (E0) puede solamente ir a la derecha (E1) o hacia bajo (E2), desde el Estado 1 se si se va a la derecha se puede llegar al Estado 3, volver al Estado 0 si se va a la izquierda, o ir al Estado 4 si se va hacia abajo.

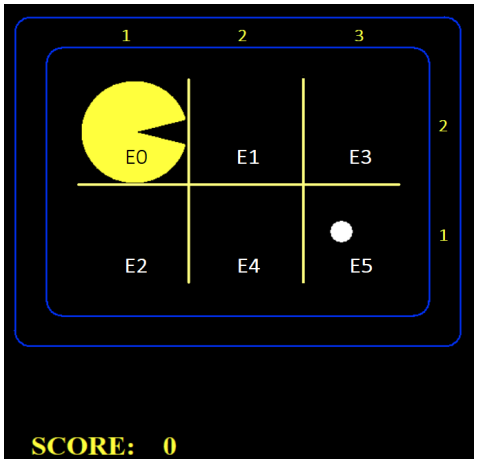

Figura 4.7: Ejemplo PacMan con cuadrícula

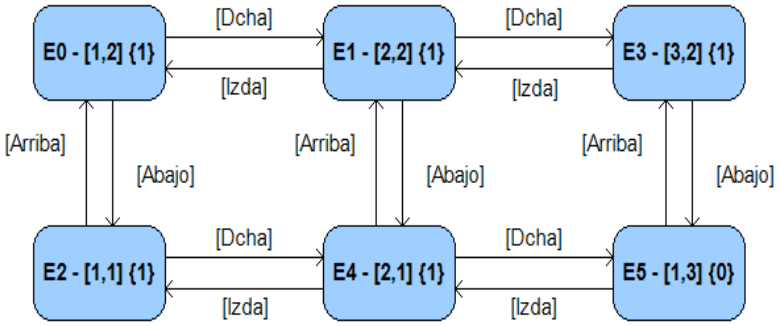

Figura 4.8: Ejemplo PacMan grafo

Una vez se ha planteado el problema a abarcar, se procede a explicar los distintos algoritmos de búsqueda básicas que se han implementado para la práctica.

La resolución de problemas mediante algoritmos de búsquedas emplea espacio de estados, este sigue el modelo de estados que cumple lo siguiente [29]:

- Espacio de estados finito y discreto S
- Estado inicial $s_0 \in S$
- Un conjunto de estados objetivos $G \subseteq S$

– Acciones aplicables A(s) $\subseteq$ A en cada estado s $\in$ S

– Una función transición f(s,a) para s $\in$ S y a $\in$ A(s)

– Y una función de coste c(a,s) > 0

Una solución es una secuencia aplicable de acciones a_i, i = 0, ..., n que lleva desde el estado inicial s_0 al estado objetivo s $\subseteq$ SG; es decir, $s_{n+1} \subseteq$ SG y para i = 0, ..., n

$$s_{i+1} = f(a_i, s_i) \text{ y } a_i \subseteq A(s_i)$$

Y, la solución óptima minimiza el coste es: $\sum_{i=0}^{n} c(a_i, s_i)$.

El estado inicial y la función sucesor definen el espacio de estados del problema (el conjunto de todos los estados alcanzables desde el estado inicial). El espacio de estados forma un grafo, como se ha visto anteriormente, en el cual los nodos son estados, y los arcos entre los nodos son acciones. Además, la función transición (sucesor) devuelve un conjunto, el coste del camino ya que refleja la medida del rendimiento (prestaciones) y la solución óptima tiene el coste más pequeño del camino entre todas las soluciones [29].

En este caso, el estado inicial ya se ha definido más atrás. La función sucesor será una de las partes que los alumnos deberán implementar y hallar para resolver la práctica, ya que ésta deberá devolver todos los estados válidos con base al estado actual. Posteriormente, se explicará cómo ha sido resuelto para este TFG. El coste del camino que refleja las prestaciones de cada posible método de búsqueda es en este caso 1 por cada acción realizada. Y, por último, la solución más óptima para cada caso tendrá un valor, en el caso expuesto con anterioridad (ver Figura 4.8) será 4, porque es el camino más corto entre todas las soluciones, que se puede observar en el grafo ya mostrado y además se explicarán cada uno de los algoritmos a emplear.

Se exigen más de una búsqueda para la solución del problema, y la idea básica, es llevar a cabo una simulación de una exploración del espacio de estados generando los sucesores de los estados ya explorados, realizando expansiones.

Este espacio de estado genera un árbol de búsqueda que consta de un nodo raíz que se corresponde con el estado inicial, nodos que son los distintos estados y las hojas (son nodos sin hijos) generados por la función sucesor. Y el árbol a su vez, suele generar un grafo, donde se permite la posibilidad de regresar a un estado ya visitado y existen diferentes caminos para un mismo estado.

Por lo tanto, partiendo de un nodo que representa un estado, éste se expande, lo que significa que se generan nuevos estados aplicando acciones permitidas sobre uno.

El nodo que se expande se denomina padre y los resultantes los hijos. El conjunto de nodos sin hijos, no expandidos, es conocido como frontera o lista abierta [29].

Teniendo en cuenta estos conceptos sobre los algoritmos de búsquedas, se procede a la explicación de los algoritmos de búsquedas que se han implementado. Las estrategias de búsqueda vienen definidas por la elección del orden de expansión de los nodos y se evalúan de acuerdo a cuatro dimensiones:

- Completitud, ¿se encuentra la solución al problema?
- Optimización, ¿es encontrada siempre la solución de menor coste?
- Complejidad tempral, número de nodos generados y expandidos
- Complejidad espacial, número de nodos almacenados en memoria en la búsqueda

Existen estrategias de búsquedas informadas y no informadas. Para esta práctica se han puesto a prueba tres no informadas y dos informadas.

En el caso de las búsquedas no informadas se hace uso sólo de información disponible en la descripción del problema, no cuenta con ninguna información o conocimiento sobre cómo llegar al objetivo. Se definen por el algoritmo de expansión utilizado, existen seis categorías, aunque las que se piden son: primero en anchura (*breadth first*), coste uniforme (*uniform cost*) y primero en profundidad (*depth first*). Se diferencian por la estructura de datos utilizada para la frontera. En el caso del primero en anchura se emplea una cola FIFO (First In First Out), lo que significa que lo primero que entra es lo primero en salir. Para la búsqueda de coste uniforme se emplea una cola ordenada por coste de caminos, donde cada nodo supone un coste de 1 como se ha comentado con anterioridad. Esta prioridad viene determinada por la función lambda, donde lambda es una función anónima en el lenguaje de programación de Python, que consiste en tomar cualquier número de argumentos, pero sólo puede tener una expresión [52]. Por último, primero en profundidad emplea una cola LIFO (Last In First Out), donde el último en entrar es el primero en salir [29].

En el caso de las búsquedas informadas, se aplica conocimiento al proceso para hacerlo más eficiente. Este viene dado por una función que estima la "bondad"de los estados, y da preferencia a los que son considerados mejores, ordenando la cola de abiertos por la comparación de su bondad estimada. Pretende reducir el árbol de búsqueda para poder ganar eficiencia. Este conocimiento específico que se va a usar sobre el problema está codificado en la función heurística. Se denomina heurística a una función numérica sobre los estados que estima la "distancia al objetivo y siempre

tiene un valor mayor o igual a 0". Será igual a 0 cuando se llegue al estado final y se admite el valor "infinito". Para este tipo de búsquedas se emplean dos algoritmos, voraz o *greedy* y *A**.

En ambos casos se hace uso de una cola con prioridades, pero varían en cuanto a la prioridad de la que hacen uso de ésta. En el primer caso, el algoritmo de búsqueda voraz únicamente se basa en la heurística para el cálculo del coste de la función: $f = g$, donde g representa el coste del camino. En cambio, en el segundo caso de *A**, además de basarse en la heurística, hace uso de la información dada en el problema para hacer el cálculo del coste de la función: $f = g + h$, donde g representa el coste del camino y h el coste estimado por la heurística. En ambos casos se hace uso de la función anónima lambda, ya explicada. La diferencia entre ambas viene determinada por hacer uso del coste de la función, es decir, sería hacer uso de g o no hacer uso de g. En el caso de la voraz solo se hace uso de la heurística para el cálculo del coste para cada acción hasta en nodo meta, y en el caso de la *A** se hace uso de la heurística y el coste de la función. Las funciones heurísticas son definidas más adelante, pero un ejemplo de éstas muy conocido es el de la distancia Manhattan [19].

A continuación, se expone el pseudocódigo para los distintos algoritmos de búsquedas. Ambos métodos de búsqueda hacen uso del algoritmo general de búsqueda de grafos donde la única diferencia entre los informados y los no informados, es el coste de la función, que en cada caso viene determinado por distintas funciones.

El procedimiento de estos algoritmos hace uso de cuatro variables. La primera es el nodo inicial del cuál partimos, dos variables de tipo entero que almacenarán el número de nodos expandidos y generados, y, un vector que contiene los nodos explorados. El algoritmo sigue los siguientes pasos:

- Primero se añade el nodo inicial a la frontera. Es decir, se inicializa la frontera con el estado inicial del problema.

- Luego dentro de un bucle infinito:

  - Se comprueba si la frontera está vacía, si es así se devolverá un objeto vacío
  - Si no está vacía la frontera entonces se elimina el nodo que le corresponde a la frontera en cada caso. Si la frontera es una pila entonces el primer nodo apilado será el nodo eliminado porque pasa a ser un nodo explorado, si es una cola entonces será el último elemento añadido. Si se tratara de una cola con prioridades, entonces sería el elemento que mayor prioridad tuviese.

- Luego se comprueba si el nodo explorado es el estado final, si es el estado final se retorna el nodo explorado, el total de nodos expandidos y el total de nodos generados.
- Si no era el estado final, se añade dicho nodo al vector de nodos explorados.
- Se incrementa el número de nodos expandidos.
- Se expande el nodo y se almacenan todos los nodos obtenidos al expandirse en un vector.
- Se recorre este último vector nombrado, y se comprueba si no están en el vector de nodos explorados o si la frontera no los contiene.
- Si se cumplen ambas premisas entonces, se añade el nodo a la frontera y se incrementa el número de nodos generados.
- Una vez llegado a este punto se regresa a ejecutar el inicio del bucle hasta que se alcanza el nodo final.

A continuación, se procede a explicar cómo funcionan los distintos algoritmos de búsqueda.

Comenzando por los algoritmos de búsqueda no informada, concretamente el de primero en anchura. Partiendo del árbol que se puede observar en la Figura 4.9, se puede ver como el nodo inicial es el 1. Desde que éste se añade a la frontera, se expande. Al hacerlo se generan los 3 nodos que se encuentran en el siguiente nivel, que serían el 2, 3 y 4, por ese orden de izquierda a derecha, son añadidos a una lista. A continuación, se debe expandir el primer nodo que se tiene en la lista, que sería el 2 y se generan el 5 y 6. Estos son añadidos a la lista, que ahora es la siguiente 3, 4, 5 y 6, porque como el 2 ha sido expandido, sale de la lista. Ahora, se debe expandir el nodo que se encuentra primero en la lista, que sería el 3 y genera al 7. Por lo tanto, la lista es 4, 5, 6 y 7. Toca expandir el nodo 4 y genera al 8 y 9, quedando la lista 5, 6, 7, 8 y 9. El nodo 5 es el siguiente y genera el 10 y 11, la lista es la siguiente: 6, 7, 8, 9, 10 y 11. El nodo 6 no tiene nada que generar por lo que sale de la lista y es el turno del 7 que pasa lo mismo que el 6. Le toca el 8 y tampoco genera ningún nodo, pasa al 9 y éste sí genera los nodos 12 y 13. La lista ahora está de la forma que se puede ver a continuación: 10, 11, 12 y 13. Estos nodos no generarán ninguno más y se deben recorrer en el orden que se encuentran en la lista. Este recorrido es bastante sencillo y para entenderlo visualmente se puede ver la Figura 4.10, donde se indica con flechas y número, el orden y lógica de éste.

Por mucho que se haya explicado el recorrido entero del árbol, no siempre va a ser así,éste será recorrido hasta que se genere el noto meta. Es decir, si el nodo meta es

el 4, en el momento que se expanda el nodo 1 se habrá acabado porque al expandirse genera el nodo 4. Por lo tanto, en ese instante la ejecución se para, no tiene sentido seguir ya que se ha alcanzado la meta que se tenía.

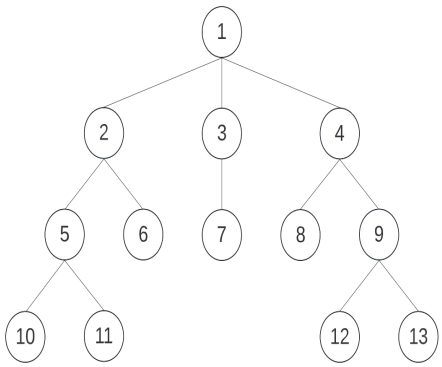

Figura 4.9: Ejemplo algoritmo primero en anchura

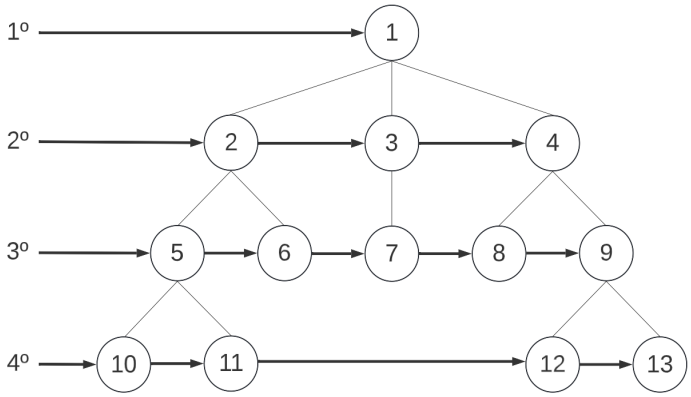

Figura 4.10: Ejemplo algoritmo primero en anchura recorrido

A continuación, se procede a explicar el algoritmo de primero en profundidad (ver Figura 4.11), algoritmo no informado. De igual manera que antes se comienza por el nodo inicial que es el 1, de estos nodos se expanden el nodo 2, 7 y 9. En este caso se debe coger el primer nodo generado de izquierda a derecha, que es el 2, y el que más abajo este dentro de una rama, es decir, en este caso todos los nodos están al mismo nivel por lo que da igual, el nodo 2 genera al 3 y 6. Ahora tenemos 7, 9, 3 y 6, se debe coger aquel que este más a la izquierda y abajo, por lo tanto en el nivel más bajo están el 3 y el 6, y el que más a la izquierda esta es el 3. Este nodo se expande y genera el 4 y 5, la lista ahora es 7, 9, 6, 4 y 5. ¿Cuál de los nodos está en un nivel más bajo? El 4 y 5, y el que más a la izquierda el 4, no tiene nodos a generar, la lista es 7, 9, 6 y 5. El siguiente nodo más abajo es el 5, tampoco genera nodos así que se continúa. En la lista están el 7, 9 y 6, y el que está más abajo es el 6 por lo que este se expande. Ahora están el 7 y el 9, están al mismo nivel por lo que se coge el que más a la izquierda está

que es el 7 y genera el 8. Como el 8 está más abajo se expande antes que el 8 pero no tiene nodos para generar y es el turno del 9, que se expande y genera el 10 y 11. Se coge el 10 porque están al mismo nivel y está más a la izquierda. No tiene nodos a generar. El siguiente porque no hay más nodos en la lista es el 11, éste genera el 12 y 13, ambos al mismo nivel pero primero se expandirá el 12 y luego el 13 porque está más a la derecha este último (ver Figura 4.12).

De igual manera que el anterior, si el nodo está antes del último nodo a recorrer no será necesario realizar el recorrido entero.

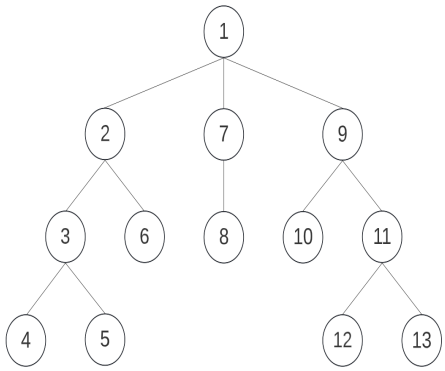

Figura 4.11: Ejemplo algoritmo primero en profundidad

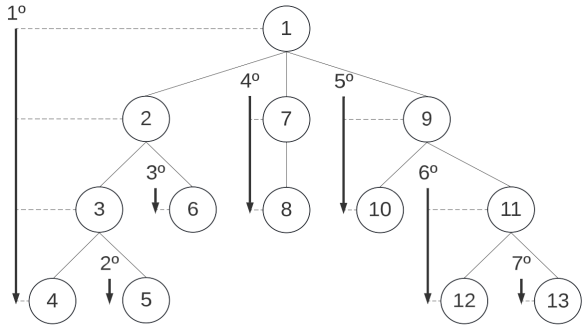

Figura 4.12: Ejemplo algoritmo primero en profundidad recorrido

Para finalizar con los algoritmos no informados, toca el de coste uniforme, para el que se va a emplear como referencia la Figura 4.13. En este caso se especifica que se desea ir desde el nodo A hasta el nodo H. Para ello, desde el nodo A se expande y se generan los nodos B, C y D. A diferencia de los otros árboles, estos tienen un valor en cada línea, que indica el coste del camino, es decir, desde el nodo A al B, cuesta 1 pero desde A al D cuesta 10. En este caso la elección del nodo para expandir viene basada en el menor coste posible, por lo tanto, desde el nodo A el menor coste es por el nodo B que tiene un coste de 1. Este nodo B se expande y tiene el nodo E y F. Desde el nodo A hasta el nodo E se tiene que tener en cuenta la suma del coste por los nodos

que se pasa antes, en este caso sería desde A hasta B y desde B hasta E, suponiendo un total de 4 de A a E y 3 de A a F. Por lo tanto, ahora se debe elegir aquel camino que tenga menor coste de todos los encontrados hasta ahora, que en este caso sería el de A a C que era 2. C genera el nodo G, desde A a G hay un coste de 4 igual que desde A hasta E. En este caso, el coste desde A hasta F es el menor por el momento, por lo que se expande F y hasta H tiene un coste de 8, superior al de 4 que tenían A hasta E y A hasta G. Si se escoge desde A hasta E se va al nodo H ya expandido, desde A hasta H pasando por B y E, el coste es de 11, superior al de pasando por B y F. En cambio, si se escoge desde A hasta H pasando por G el coste total es 6, y sería el camino más óptimo de llegar. El camino de D no es nombrado tan apenas ya que se haya una solución antes por otro camino de un valor menor al del coste simplemente de A a D.

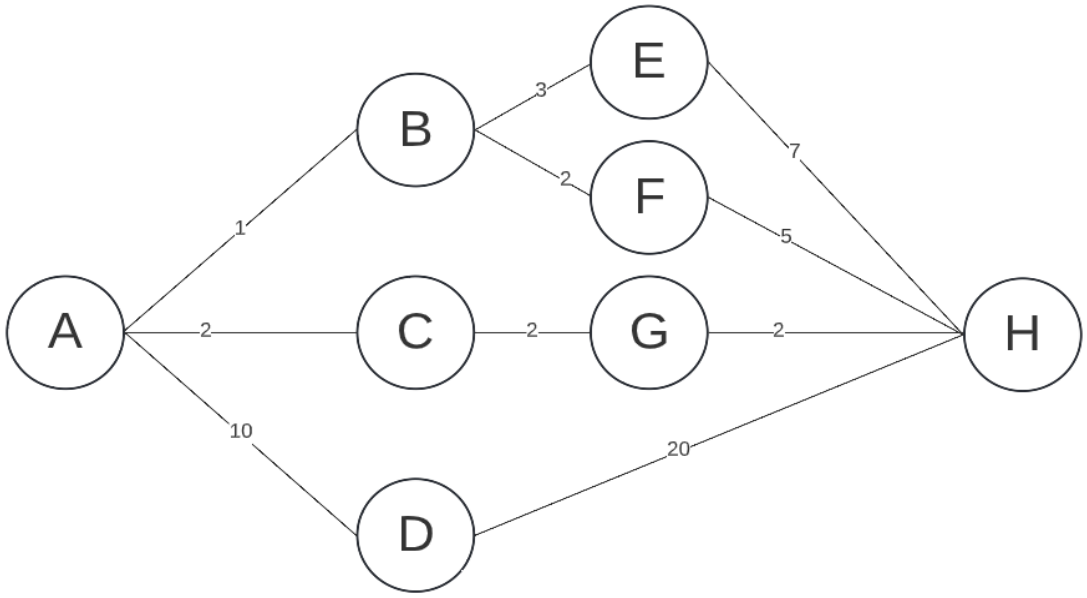

Figura 4.13: Ejemplo algoritmo coste uniforme

Una vez terminados los algoritmos no informados, toca los informados, estos funcionan de manera similar al de coste uniforme pero el valor que se tiene en consideración para el coste, en el algoritmo de voraz es el de las heurísticas y el de $A*$ es una unión del voraz y coste uniforme porque tiene en cuenta el coste del camino y el estimado por las heurísticas. Para la explicación de éstas se va a emplear el ejemplo que se encuentra en la Figura 4.14.

Primero se va a comenzar a explicar el algoritmo voraz. Para ello se parte del nodo A y se desea llegar hasta el nodo H. En este caso, aparece la diferencia con respecto al

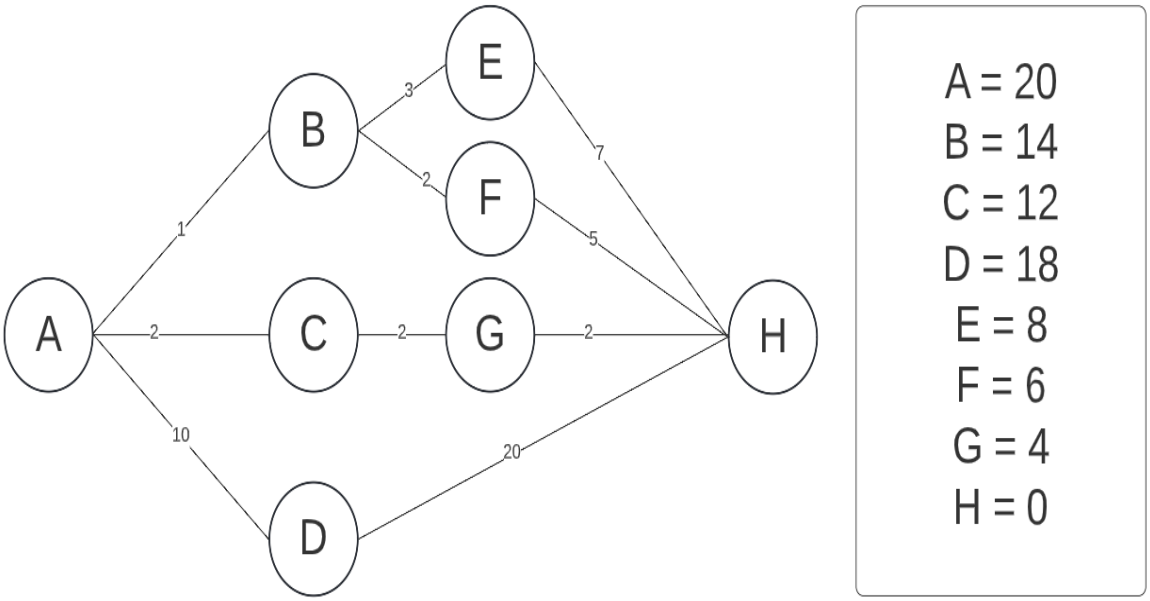

Figura 4.14: Ejemplo para algoritmo de búsquedas informadas

algoritmo de coste uniforme, es una tabla, donde se puede observar que para cada nodo hay un valor, aleatorio, que se obtendría al aplicar una heurística desde el nodo en el que se encuentra hasta el nodo final. Como en este caso el algoritmo voraz trabaja con los valores del coste estimado de la heurística, para calcular el coste del camino de A a H será tomado de las cifras de la tabla. Por lo tanto, en este caso los cálculos para ir desde A hasta H por los 4 distintos caminos posibles serían:

- Pasando por B y E, la suma de A más B más E más H: A + B + E + H = 20 + 14 + 8 + 0 = 42
- Pasando por B y F sería : $A + B + F + H = 20 + 14 + 6 + 0 = 40$
- Pasando por C y G: $A + C + G + H = 20 + 12 + 4 + 0 = 36$
- Y, por, D: A + D + H = $20 + 18 + 0 = 38$

En este caso el camino óptimo es el que va desde A hasta H pasando por C y G. Pero por ejemplo se puede observar que en el algoritmo voraz, pasar por D era el peor camino y en este caso es el segundo mejor. No siempre tienen que coincidir, el posicionamiento de los mejores caminos, cuando se emplea el coste del camino o el coste estimado por la heurística. Además de que no siempre distintas heurísticas darán tampoco el mismo posicionamiento.

Para finalizar, se procede a explicar el algoritmo $A*$. Éste para obtener los resultados del coste tiene en cuenta el coste del camino y el estimado de las heurísticas. Se va a proceder al igual que en el caso anterior a explicar cuál sería el coste para cada posible

camino y observar cuál sería el mejor, considerando que g es el coste del camino y h de la heurística. Los costes para ir desde A hasta H:

- Pasando por B y E, la suma de A más B más E más H: *A(h) + B(h) + E(h) + H(h) + A-B-E-H(g)= 20 + 14 + 8 + 0 + 11 = 58*
- Pasando por B y F sería : *A(h) + B(h) + F(h) + H(h) + A-B-F-H(g) = 20 + 14 + 6 + 0 + 8 = 48*
- Pasando por C y G: *A(h) + C(h) + G(h) + H(h) + A-C-G-H(g) = 20 + 12 + 4 + 0 + 6 = 42*
- Y, por, D: *A(h) + D(h) + H(h) + A-D-H(g) = 20 + 18 + 30 = 68*

Para este caso, como es lógico ya que en ambos algoritmos tanto voraz como coste uniforme, el camino pasando por C y G desde A hasta H era el mejor, sigue siéndolo.

Hasta ahora se ha explicado el concepto teórico que se ha puesto en práctica para el desarrollo del trabajo. A continuación, se procede a exponer la implementación de la inteligencia en el Pac-Man.

Para ello, primero se tiene que dejar claro qué es necesario que contengan los distintos estados del espacio, aunque algunos parámetros luego no se tengan en cuenta para comprobar si se ha llegado al estado final. Como se ha indicado anteriormente, el único parámetro para comprobar será la cantidad de comida que falta por comer, siendo 0 cuando se alcance el estado último.

Se necesita conocer en todo momento dónde se encuentra el Pac-Man, es decir su posición. Y en dicha posición se tendrá que saber si hay o no comida, si se pasa por dicha posición, se tendrá que decrementar el número total de comida que queda y en esta posición indicar que ya no habrá comida porque ya se ha pasado por ésta.

Teniendo claros los parámetros que se han de tener en consideración, se investiga como se pueden obtener del código del que se parte.

Con respecto a la posición, se tiene que conocer de donde parte el agente y aquellas por las que pasa. La posición inicial se puede obtener del objeto *Layout* que contiene los siguientes valores: anchura y altura del laberinto, dónde se ubican las paredes, comida y capsula en el laberinto, las posiciones de los distintos agentes, el número total de fantasmas y el total de comida. Tras observar alguno de los diseños (ver Figura 4.15) se puede ver que:

- El símbolo de porcentaje (%) es empleado para indicar las paredes del laberinto.

– Los puntos (.) para la comida.

– La letra “G”mayúscula, representa a los fantasmas.

– La letra “P”mayúscula al agente Pac-Man.

– La letra “o”minúscula las capsulas.

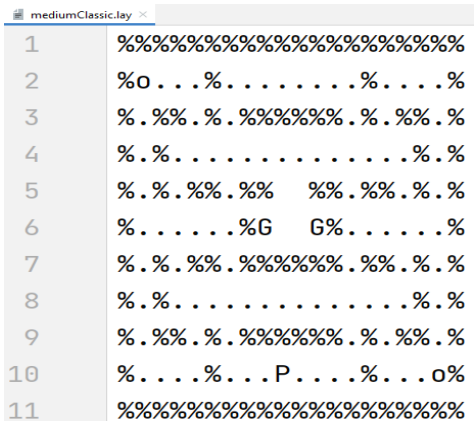

Figura 4.15: Diseño de la interfaz ”mediumClassic.lay”

Y, empleando el *debugger* para conocer el contenido de:

– La variable que contiene la ubicación de la comida. Se puede ver como se utiliza una matriz para ello, con las dimensiones del laberinto, siendo en el caso que se pone de ejemplo de la Figura 4.15 de ancho 20 y altura 11. Se indica en esta con un *True* cuando hay comida en esa posición (ver Figura 4.16).

– La posición inicial del Pac-Man, es recogida en una variable junto a las posiciones de los agentes restantes. Ésta contiene una tupla con un valor booleano y luego la posición. El valor booleano indica si el agente del que se trata es el Pac-Man (valor *True*) o si de es un fantasma (valor *False*). El agente Pac-Man ocupa la primera posición de este vector siempre, por lo que, consultando el segundo valor de la tupla de la primera posición de éste, se obtendrá el valor que se desea.

– Aunque esta variable es más sencilla, también se obtiene mediante este objeto el número total de comidas que hay en este diseño, en la variable *numFood*.

Al constar de la posición inicial del Pac-Man, las restantes pueden ser calculadas incrementando o decrementando el valor de las variables x e y, siendo x la representación del eje horizontal e y el eje vertical, dependiendo de si la acción es para arriba, abajo, izquierda, derecha o quedarse parado. El formato que sigue la posición es el siguiente: (x, y). Pero no siempre se podrá llevar a cabo cualquier acción

Figura 4.16: Matriz de la comida para la interfaz "mediumClassic.lay"

porque existe la limitación de las paredes, hay otra matriz que indica en que posición se pueden encontrar éstas, pero existe una función, disponible en el fichero *game.py*, concretamente en la clase *Actions* que tiene el método *getPossibleActions* (ver Figura 4.17). Éste recibe como parámetros un objeto Configuration que consta de una tupla con la posición y una acción, y, la matriz que contiene la pared. Por lo tanto, partiendo de la posición pasada y las paredes del laberinto se obtienen todas las posibles acciones a llevar a cabo.

```
def getPossibleActions(config, walls):
    possible = []
    x, y = config.pos
    x_int, y_int = int(x + 0.5), int(y + 0.5)

    # In between grid points, all agents must continue straight
    if abs(x - x_int) + abs(y - y_int) > Actions.TOLERANCE:
        return [config.getDirection()]

    for dir, vec in Actions._directionsAsList:
        dx, dy = vec
        next_y = y_int + dy
        next_x = x_int + dx
        if not walls[next_x][next_y]:
            possible.append(dir)

    return possible

getPossibleActions = staticmethod(getPossibleActions)
```

Figura 4.17: Método getPossibleActions en Actions, game.py

Con los parámetros indicados se podrían establecer los estados para este juego que

se va a denominar *PacmanState* (ver Figura 4.18). Éste tendrá una variable que serán las paredes comunes para todo los estados, pero son necesarias para poder obtener las acciones posteriores, el número total de comidas, la posición que será almacenada en el objeto *Configuration*, por lo que también se tendrá la acción, y, la matriz de la ubicación de la comida, ya que para cada estado podrá ser distinta al anterior.

```
class PacmanState(object):
    walls = set()

    def __init__(self, numFood, position, direction, food):
        self.numFood = numFood
        self.config = Configuration(position, direction)
        self.food = food
```

Figura 4.18: Estado del Pac-Man

Remarcar la importancia de que, aunque en el estado se tengan que tener en consideración todos los parámetros que éste contiene, no determinarán si se ha llegado al estado final o no, solo lo hará la cantidad de comida que quede. Pero, es necesario conocer cómo se va actualizando la matriz de comida conforme se generan nuevos estados y, para ello, la posición que va ocupando en cada momento.

Una vez se tiene claro el estado, se debe crear la función sucesor que proporciona los posibles estados consiguientes al estado en el que se encuentra en ese momento. Para ello, primero se crea un método dentro de *PacmanState* denominado *next_states* que llamará al método *succ* donde se encontrará la función sucesor. De esta manera se sigue la lógica requerida en la práctica actual de la asignatura de IA. Esta clase puede ser encontrada en el fichero *pacmanState.py*.

En el método *next_states* (ver Figura 4.19)se parte de un estado actual y primero se obtienen todas las posibles acciones, para desde este estado poder desplazarse, ya que no todo es viable porque existe la limitación de las barreras. La acción de estar parado se elimina ya que no es útil para este caso y una vez se tienen éstas, se llama a la función sucesor para cada una de las posibilidades. Ésta devuelve cómo sería el estado para cada una las posibilidades de acciones generadas para dicho estado. Y, se generará un vector con los distintos estados obtenidos, que será empleado por los algoritmos de búsqueda como se verá un poco más adelante.

En la función sucesor (ver Figura 4.20) se recibe como parámetro el estado actual y la acción que se ha de llevar a cabo. Con esta información se comprueba qué nuevo estado se quedaría. Para ello, se aplica la acción a la posición del estado actual y

```
def next_states(self, state):
    self = state
    new_states = []

    legal = Actions.getPossibleActions(self.config, self.walls)

    if Directions.STOP in legal:
        legal.remove(Directions.STOP)

    for action in legal:
        state = 0
        if action is not None:
            state = self.succ(action)
        if state is not None:
            new_states.append([state, state.config.direction])

    return new_states
```

Figura 4.19: Método next_states

se comprueba si en dicha nueva posición hay comida. Si hubiere comida, el total del número de comida es decrementado en uno y en la matriz de comida dicha posición que estaría a *True* pasa a ser *False*. La modificación de la matriz, en un principio, dio problemas porque cuando se crea una variable y se le asigna el valor de otra variable, siguen siendo distintas. Pero en el caso de la matriz, cuando a una variable se le trataba de asignar el valor de la otra matriz como se hacía con las variables, está en vez de hacer una copia, hacía un enlace a la misma variable. Dicho problema fue solventado empleando la función *deepcopy* [34]. Estos nuevos valores del total de número de comida, posición y la matriz de la comida generá un nuevo estado que es devuelto al método *next_states*.

```
def succ(self, action):
    numFood = self.numFood
    food = copy.deepcopy(self.food)
    config = self.getNewPosition(action)
    x, y = config.pos
    if food[x][y]:
        numFood -= 1
        food[x][y] = False

    return PacmanState(numFood, config.pos, config.direction, food)
```

Figura 4.20: Método succ

Ya definida la función sucesor y el método que devolverá todos los nuevos estados, se procede a explicar los distintos algoritmos de búsqueda.

En el fichero *search.py* se puede ver toda la lógica implementada de estos algoritmos. Ésta hace a su vez uso de otro fichero denominado *datastructures.py* que contiene las distintas estructuras de datos que son empleadas para los distintos algoritmos. Estas

estructuras son: pila o (*stack*), cola o (*queue*) y cola con prioridad o (*priority queue*).

Dentro del fichero *search.py* se encuentran los algoritmos de búsqueda tanto informada como no. La lógica que sigue los algoritmos es la que se ha explicado anteriormente. Ambos algoritmos parten de un nodo, concretamente el nodo inicial, que este contiene el estado inicial y otros valores que son: el nodo del cual proviene el estado y si es el inicial será vacío, la acción por la cual se ha llegado hasta dicho estado, el coste del camino desde el estado inicial hasta el nodo actual, siendo 0 en el caso del inicial y en el caso de las informadas el coste estimado por la heurística, desde el nodo en el que se encuentra hasta el nodo meta que contiene el estado final.

Se lleva una cuenta de los nodos que son expandidos y generados para luego poder realizar una comparativa con los distintos algoritmos y observar cuál de ellos es más eficiente para cada laberinto. También se declara un vector que almacena todos los nodos ya explorados.

Cuando se han terminado de declarar las variables, se inserta el nodo inicial en la frontera como ya se había indicado, y, se realiza el procedimiento que ya ha sido explicado anteriormente. En la Figura 4.22, se puede ver el caso de búsqueda informada, que difiere de la Figura 4.21, el caso de la búsqueda no informada, en la parte de abajo, donde tras la sentencia de *node.g = explored_node.g +1* que se corresponde con el coste de camino, en la búsqueda informada se añade el cálculo del coste estimado por la heurística que es la siguiente: *node.h = heuristic(node.state, goal_state)*, donde se llama a una heurística a la que se le indica cuál es el estado inicial y cuál es el estado meta.

```
def uninformed_search(initial_state, goal_state, frontier):
    initial_node = Node(initial_state, None, None)
    expanded = 0
    generated = 0
    explored = []

    frontier.insert(initial_node)

    while True:
        if frontier.is_empty():
            return None

        explored_node = frontier.remove()

        if explored_node.state.numFood == goal_state.numFood:
            return explored_node, expanded, generated

        explored.append(explored_node.state)
        expanded += 1

        expand = explored_node.expand()
        for node in expand:
            if node.state not in explored and not frontier.contains(node):
                node.g = explored_node.g + 1
                frontier.insert(node)
                generated += 1
```

Figura 4.21: Método de búsqueda no informada

```python
def informed_search(initial_state, goal_state, frontier, heuristic):
    initial_node = Node(initial_state, None, None)
    explored = []
    expanded = 0
    generated = 0

    frontier.insert(initial_node)

    while True:
        if frontier.is_empty():
            return None

        explored_node = frontier.remove()

        if explored_node.state.numFood == goal_state.numFood:
            return explored_node, expanded, generated

        explored.append(explored_node.state)
        expanded += 1

        expand = explored_node.expand()
        for node in expand:
            if node.state not in explored and not frontier.contains(node):
                node.g = explored_node.g + 1
                node.h = heuristic(node.state, goal_state)
                frontier.insert(node)
                generated += 1
```

Figura 4.22: Método de búsqueda informada

Destacar, recordando lo que ya se había comentado, que cuando se realiza la comprobación de si se ha llegado al estado final, por mucho que se tengan en cuenta más de una variable, sólo se comprueba si todavía queda comida, los demás parámetros no se tienen en este momento en consideración. Y, cuando se lleva a cabo una comprobación de si dos estados son iguales sí se tiene en cuenta la posición, el número de comida que queda y la matriz de la ubicación de la comida.

En todo este código para los algoritmos, no aparece la función sucesor ni la de los siguientes estados, para poder saber en cuál estamos o cuales se pueden a partir de uno. Esto se lleva a cabo cuando se ejecuta el método denominado *expand*, donde se crea un vector para almacenar los posibles sucesores y se llama al método *nextstates* (ver Figura 4.23) que retorna los siguientes posibles estados.

```python
def expand(self):
    successors = []
    for (newState, action) in self.state.next_states(self.state):
        newNode = Node(newState, self, action)
        successors.append(newNode)
    return successors
```

Figura 4.23: Método *expand*

En la Figura 4.24, se puede ver que las tres primeras funciones se corresponden

con los algoritmos de búsqueda no informada, donde en cada uno de los casos se les pasa la estructura de datos que emplean a la variable *frontier*, que es la frontera. Y las dos últimas funciones son los algoritmos de búsqueda informada, donde la estructura de datos que se pasa es una cola de prioridad, donde dicha prioridad en el caso del algoritmo *greedy* o voraz, viene solo determinado por el coste estimado por la heurística, y, en el caso de la $A*$ por el coste del camino y el coste estimado por la heurística.

```
# Functions for uninformed search


def breadth_first(initial_state, goal_state):
    frontier = Queue()
    return uninformed_search(initial_state, goal_state, frontier)


def depth_first(initial_state, goal_state):
    frontier = Stack()
    return uninformed_search(initial_state, goal_state, frontier)


def uniform_cost(initial_state, goal_state):
    frontier = PriorityQueue(lambda x: x.g)
    return uninformed_search(initial_state, goal_state, frontier)

# Functions for informed search


def greedy(initial_state, goal_state, heuristic):
    frontier = PriorityQueue(lambda x: x.h)
    return informed_search(initial_state, goal_state, frontier, heuristic)


def a_star(initial_state, goal_state, heuristic):
    frontier = PriorityQueue(lambda x: x.g + x.h)
    return informed_search(initial_state, goal_state, frontier, heuristic)
```

Figura 4.24: Algoritmos de búsqueda informada y no informada

Solo faltan por explicar en este apartado las heurísticas [47].

Las heurísticas que se suelen emplear para aplicar en los algoritmos son: la distancia Manhattan [6] o Euclidiana [7]. Estas heurísticas trabajan con distancias y referencias entre dos puntos distintos que constan de una x y una y, por lo que en este caso que el estado inicial y el estado final van determinados por el valor de una variable que solo indica cantidad, no son válidas.

Al no encontrar ninguna heurística clásica que pudiera ser empleada para este planteamiento del juego, se han creado distintas heurísticas con diferentes operaciones matemáticas básicas.

Las heurísticas que se han planteado son las que se pueden observar en la Figura 4.25. En la primera se hace una división por 2 de la cantidad de comida que queda en

dicho estado. En el segundo caso una multiplicación por 2, en la heurística tercera una raíz cuadrada, en la cuarta dicha cantidad por la potencia de 2 y la potencia de 3 en el quinto caso. Una raíz cuadrática de la cantidad de comida entre 2 y la potencia de 2 del valor entre 2.

```
def h1(current_state, goal_state):
    return current_state.numFood / 2

def h2(current_state, goal_state):
    return 2 * current_state.numFood

def h3(current_state, goal_state):
    return math.sqrt(current_state.numFood)

def h4(current_state, goal_state):
    return pow(current_state.numFood, 2)

def h5(current_state, goal_state):
    return current_state

def h6(current_state, goal_state):
    return current_state * 13

def h7(current_state, goal_state):
    return math.sqrt(current_state.numFood) / 2

def h8(current_state, goal_state):
    return pow(current_state.numFood, 2) / 2
```

Figura 4.25: Heurísticas para la búsqueda informada

Una vez finalizada la implementación de la IA para este juego, falta fusionarla con la interfaz. Para ello, se genera un vector con los distintos pasos a tomar según el camino óptimo. Luego, se introduce en el método *run* del fichero *game.py*. Como se había comentado, en este método se consideran todos los agentes, por lo que se tiene que comprobar en cada iteración si es el Pac-Man, y si lo es consultar una posición del vector para saber la acción que se debe tomar e incrementar el contador del índice del vector en uno. Si el índice del agente es 0 se sabrá que es el Pac-Man y sino será un fantasma. El código del fantasma ha sido modificado para que funcione de manera aleatoria, en cada momento se observará qué posiciones son legales desde su estado y se cogerá una de todas las acciones válidas. En el *run* se pueden ver dos maneras de juego distintas, en el caso del Pac-Man las acciones que se van a realizar se saben antes de iniciar la interfaz, en cambio, el fantasma tiene poder de tomar una decisión en tiempo real, en el momento que es su turno, por lo que se comprueba su posición y se decide. Esta lógica de tiempo real para el Pac-Man va a ser trabajada en un futuro, para que cuando vaya a tomar una decisión, se valore si hay un fantasma cerca es un peor estado.

Otros aspectos que no tenían que ver con la dotación de IA eran que en algunos aspectos el código que se planteaba resultaba complejo de comprender, por lo que, tras ver que sobraban muchos métodos e incluso clases, también se realizó una

reestructuración y reorganización del código y una limpieza, eliminando aquello que no fuera necesario.

Además, añadir que el código de por sí, de la base que se partía, tenía errores para poder ser ejecutado con un compilador de Python de versión 3.7 o superior, pero no para la 2.7. Para que a los alumnos no les genere ningún problema la versión con la cuál trabajar, se ha planteado un código que puede ser compilado tanto con una versión de 2.7 que está en sus últimas etapas, hasta la versión más actual que es la 3.10 [35].

El código que se ofrecerá a los alumnos, se encuentra disponible en un respositorio de GitHub. GitHub es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git [54]. El enlace para visitarlo es el siguiente: `https://github.com/vircas/Pac-Man-EUPT`

4.4. Resultados o pruebas

Una vez se ha llevado a cabo toda la implementación del código, se procede a ver que algoritmo se adapta mejor al juego dependiendo de cada laberinto, ya que no se garantiza que un algoritmo funcione igual de bien para todas las configuraciones. Por ello, se van a plantear dos laberintos, distintos para todos los algoritmos de búsqueda, con el objetivo de analizar su comportamiento y ver cuál puede ser mejor en cada caso.

Los laberintos a analizar varían en su tamaño uno un poco más grande y con más cantidad de comida (ver Figura 4.26), caso 1 y otro de un tamaño más pequeño y menos comida, como se puede observar en la Figura 4.27, caso 2.

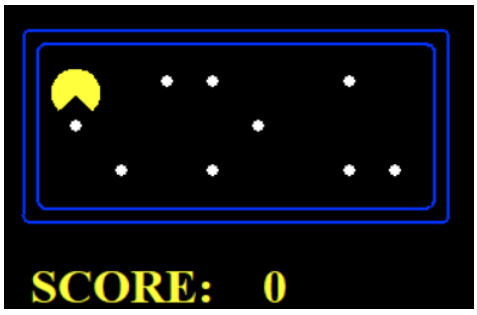

Figura 4.26: Diseño del laberinto para el caso 1

Los resultados obtenidos para los distintos algoritmos de búsqueda en el caso 1, son los que se pueden ver en la Tabla 4.2. Se puede observar que el camino con menor pasos conseguidos es de un total de 18, y se ha obtenido con los algoritmos: *breadth first, uniform cost, A** con la heurística 1, 3, 4, 5, 6 y 7. Siguiendo con los que han conseguido un camino óptimo con un menor número de pasos, cabe destacar que los

Figura 4.27: Diseño del laberinto para el caso 2

que han hallado este camino en un menor número de nodos, tanto expandidos como generados, ha sido el $A*$ con las heurísticas 4 y 5. Éstas empleaban la potencia de 2 y 3 sobre el valor del estado actual. Se procede a realizar esta operación con una potencia distinta sobre el mismo escenario para ver como evolucionan los datos. Se aplican potencia de 4, 5 y 10. Para estos casos el camino óptimo son los mismos pasos, pero los nodos generados son más siendo 30 los expandidos y 58 los generados para los 3, por lo que no se puede afirmar que todas las potencias obtienen un buen resultado sino que solo han sido conseguidos buenos los mejores resultados con la potencia de 2 y 3.

En el caso del camino óptimo encontrado con un mayor número pasos ha sido *depth first*, aunque los nodos expandidos o generados no hayan sido tantos. Si se realiza una reflexión, es mejor el *breadth first* que el *depth first*. Uno ha encontrado un camino óptimo con un menor número de pasos, pero el otro ha hallado un camino en una menor cantidad de nodos. Para poder saber cuál es mejor, se debe plantear qué criterio se tiene en consideración. De igual manera, este criterio no afecta si se compara el *depth first* con el de $A*$ de heurística 4 o 5 porque ambos han conseguido un resultado mejor en todos los aspectos.

Los resultados del algoritmo *greedy* a excepción del primero, se han mantenido en todos los parámetros para las distintas configuraciones.

Los resultados obtenidos para los distintos algoritmos de búsqueda en el caso 2, son los que se pueden ver en la Tabla 4.3.

Este se trata de un caso más sencillo, porque es más pequeño, solo hay 3 comidas y estas están en una línea recta. El camino óptimo, de un total de 6 pasos, es obtenido por todos los algoritmos menos por el *depth_first*, y *greedy* con la heurística 1. Los mejores resultados han expandido 6 nodos y 15 generados, por los algoritmos que emplean heurísticas, concretamente *greedy* con las heurísticas 2, 3, 4, 5 y 6, y $A*$ con las heurísticas 4 y 5. Resulta llamativo que $A*$ con las heurísticas 4 y 5 vuelven a ser

Algoritmo	Total pasos solución	Nodos expandidos	Nodos generados
BREADTH FIRST	18	6372	7269
DEPTH FIRST	30	35	72
UNIFORM COST	18	6266	7176
GREEDY H1	20	46	78
GREEDY H2	21	30	60
GREEDY H3	21	30	60
GREEDY H4	21	30	60
GREEDY H5	21	30	60
GREEDY H6	21	30	60
GREEDY H7	21	33	63
A* H1	18	4687	5654
A* H2	19	424	709
A* H3	18	4138	5162
A* H4	18	30	58
A* H5	18	30	58
A* H6	18	5166	6094
A* H7	18	32	60

Tabla 4.2: Resultados algoritmos de búsquedas caso 1

Algoritmo	Total pasos solución	Nodos expandidos	Nodos generados
BREADTH FIRST	6	29	48
DEPTH FIRST	8	23	30
UNIFORM COST	6	21	39
GREEDY H1	10	18	33
GREEDY H2	6	6	15
GREEDY H3	6	6	15
GREEDY H4	6	6	15
GREEDY H5	6	6	15
GREEDY H6	6	6	15
GREEDY H7	6	22	30
A* H1	6	12	24
A* H2	6	6	15
A* H3	6	11	21
A* H4	6	6	15
A* H5	6	6	15
A* H6	6	18	33
A* H7	6	8	21

Tabla 4.3: Resultados algoritmos de búsquedas caso 2

unos de los que ofrecen el mejor resultado con un menor número de nodos expandidos y generados.

El laberinto que se expone en la Figura 4.28, no ha sido resuelto todavía ya que para el algoritmo *breadth_firts* lleva más de un día en ejecución y todavía no ha finalizado,

en una hora se expandieron alrededor de 50.000 nodos y de un total de 54 comidas en el laberinto, el nodo que más cerca está de la meta tenía todavía 37 comidas.

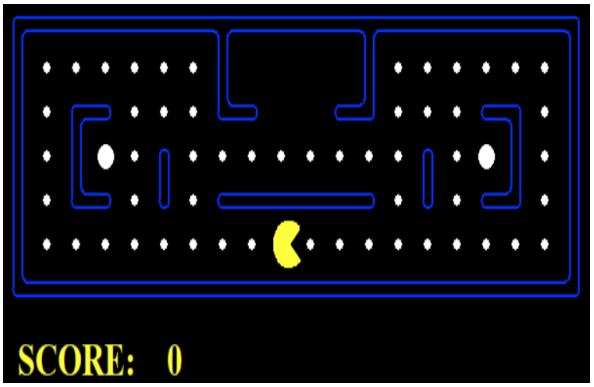

Figura 4.28: Diseño del laberinto *smallClassic*

En el enlace que se muestra a continuación, se puede observar una ejecución del Pac-Man con los distintos algoritmos y heurísticas para el caso 1: `https://drive.google.com/file/d/1ru8GvEmJu0t2o_EwVeZTXDZraXM6v-re/view?usp=sharing`.

5. Accesibilidad y Usabilidad

Este apartado se centra en los distintos parámetros de accesibilidad y usabilidad que han sido cumplidos y considerados para el proyecto llevado a cabo.

5.1. Accesibilidad

Se entiende como accesibilidad la condición que deben cumplir los entornos, productos y servicios para que sean comprensibles, utilizables y practicables por todos los ciudadanos, incluidas las personas con discapacidad [12]. Hasta ahora lo que se ha tenido en consideración hasta el momento han sido aquellos alumnos, que tendrán que llevar a cabo la realización de la práctica. Se les da un código, el cual no han de entender por completo, para la implementación de los algoritmos de búsqueda, ya que sigue una estructura simple y está especificado en cada momento que han de emplear para poder llevar a cabo su práctica. No es necesario que entiendan cada línea para adquirir los conocimientos que se pretende que interioricen y comprendan. Por eso, como se ha comentado en el apartado de la propuesta, una de las tareas que se hicieron fue una limpieza de código para una menor complejidad de éste. Aun así, la accesibilidad para este proyecto es un tema que se encuentra todavía en desarrollo y es una de las tareas a realizar en el trabajo futuro.

5.2. Usabilidad

Con respecto a la usabilidad, hace referencia a la facilidad con que un usuario puede utilizar una herramienta fabricada por otras personas para alcanzar un determinado objetivo [43]. Para medirla en este proyecto, se va a hacer uso de los 10 principios de usabilidad de Jakob Nielsen [24]. A continuación se va a llevar a cabo un análisis de cada uno de los principios, para estudiar con cuántos de ellos cumple la aplicación.

Visibilidad del estado del sistema

Este principio dice que el diseño siempre debe mantener a los usuarios informados sobre lo que está sucediendo, a través de comentarios apropiados dentro de un período de tiempo razonable.

Para el objetivo académico buscado sí se cumple el principio pero pensando en un público más general quizás se podría realizar otra versión explicando más algunos de los

pasos que se van realizando, como:"se está estableciendo la inteligencia del PacMan.º "se inicia el PacMan inteligente".

Coincidencia entre el sistema y el mundo real

El segundo principio indica que el diseño debe hablar el idioma de los usuarios. Use palabras, frases y conceptos familiares para el usuario, en lugar de jerga interna. Siga las convenciones del mundo real, haciendo que la información aparezca en un orden natural y lógico.

Todo lo que se comunica son frases u oraciones cortas que emplean palabras sencillas sin complejidad alguna por lo que se cumple con el principio.

Control y libertad del usuario

Este principio se centra en que los usuarios a menudo necesitan echar marcha atrás, porque no desean continuar, para dejar una acción no deseada sin tener que pasar por un proceso prolongado.

Pensando en el alumnado, estos podrán parar el proceso de ejecución con el botón cuadrado rojo, en el momento que deseen o utilizar el *debugging* para tener el control, por lo que se cumple con el principio.

Consistencia y estándares

El cuarto principio plantea que no existan posibles ambigüedades para los usuarios ya que no deberían tener que preguntarse si diferentes palabras, situaciones o acciones significan lo mismo.

Con respecto a las palabras son claras y concisas, como puede ser: "Ha ganado". Las situaciones, el usuario únicamente ve la pantalla del pacman y comienza el juego solo. Con las acciones podría haber confusión de que teclas son necesarias para poder mover el PacMan aunque emplea las habituales que son las flechas o las letras WASD.

Prevención de errores

En este principio se abarca el tema de prevención de errores. Para prevenir distintos errores se llevan a cabo comprobaciones de si se pasan objetos nulos o vacíos. Ciertos controles estaban ya realizados en el código original por lo que hay algunos que han sido revisados, pero todavía se siguen encontrando algún error y se subsana en el momento. El código de implementación propia, comprueba en todo momento los parámetros y se emiten mensajes de errores si es necesario ya que parte de este tema también es faena de los alumnos como objetivo de la práctica para aprender. Por lo tanto, se cumple con el principio.

Reconocimiento en lugar de recuerdo

El sexto principio tiene como objetivo minimizar la carga de memoria del usuario haciendo visibles los elementos, las acciones y las distintas opciones.

Para este caso una vez implementada la inteligencia, el usuario no requiere de recordar nada a la hora de interactuar con la interfaz. Antes de introducir ese conocimiento, solo tiene que recordar que teclas se emplean para poder mover el PacMan si desean jugar a éste. Se cumple con el principio.

Flexibilidad y eficiencia de uso

Este principio abarca unos aspectos que no se pueden mejorar en el juego del PacMan tal como está implementado ya que no permite una mayor flexibilidad o eficiencia de uso porque no hay ninguna acción como para incrementar una mejoría en este aspecto. No se cumple con el principio.

Diseño estético y minimalista

El principio séptimo indica que las interfaces no deben contener información que sea irrelevante o que rara vez se necesite. Cada unidad adicional de información en una interfaz compite con las unidades de información relevantes y disminuye su visibilidad relativa.

Solo se enseña información relevante mediante la interfaz que es el propio juego, sin ningún mensaje adicional y los colores o figuras empleadas venían con el código empleado y es el original. En la pantalla se ve el laberinto, el PacMan, la comida, las capsulas y los fantasmas, todo necesario y nada en exceso. Se cumple con el principio.

Ayudar a los usuarios a reconocer, diagnosticar y recuperarse de errores

Para cumplir este principio se muestran mensajes de error expresados en un lenguaje sencillo (sin códigos de error), indicando con precisión el problema y para el caso de los alumnos no se sugiere dependiendo del error ninguna solución ya que es un objetivo de la práctica que se peleen con el lenguaje, su código implementado y los algoritmos. Se cumple con el principio.

Ayuda y documentación

Para finalizar, con el décimo principio a los propios alumnos se les aportará una documentación que les permitirá tener una ayuda en la implementación del código, pero para el uso de la interfaz no es necesario saber nada más que lanzar el programa, que tal como es dado para cualquier versión de Python es ejecutable, y las teclas a emplear para mover el PacMan. Se cumple con el principio.

6. Licencia Software y Documental

Llegados a este apartado, se va a proceder a comentar tanto la licencia de software como la licencia documental.

En cuanto a la licencia de software se va a emplear Berkeley Software Distribution (BSD). Se trata de una licencia de software libre permisiva como puede ser OpenSSL o la MIT License. Existen diferentes tipos de licencias, en el caso de este TFG se ha utilizado la licencia "BSD modificada", "BSD revisada", "BSD-3" o "BSD de 3 cláusulas" [18].

Al igual que sucede en el mundo del software, se tienen que buscar formas de garantizar las libertades asociadas al trabajo elaborado y su inviolabilidad futura. Para garantizar que la libertad esté asociada al documento se buscan métodos, uno de ellos es la licencia GNU Free Documentation License GFDL).

El propósito de esta Licencia es hacer que en el caso de este TFG sea "gratuito" en el sentido de libertad: para asegurar a todos la libertad efectiva de copiarlo y redistribuirlo, con o sin modificarlo, ya sea comercial o no comercialmente. En segundo lugar, esta licencia preserva para el autor y el editor una forma de obtener crédito por su trabajo, sin ser considerado responsable de las modificaciones realizadas por otros. Es una especie de "copyleft", lo que significa que las obras derivadas del documento deben ser libres en el mismo sentido. Si por algún motivo se emplea este documento y se modifica, debe realizar una serie de acciones indicadas en el sitio web oficial de GNU [56].

Tampoco hay que olvidar que este documento, por defecto, está al amparo de la licencia 7.3, por su inclusión en el Repositorio Institucional de Documentos de la Universidad de Zaragoza: ZAGUAN.

Figura 6.1: Licencia de ZAGUAN

7. Conclusiones y Trabajo Futuro

Para poder llevar a cabo este trabajo, han sido relevantes todas y cada una de las asignaturas del GII, porque de manera indirecta la formación de cada una de ellas ha hecho posible llegar a este punto con las capacidades y destrezas necesarias para lograr finalizar este proyecto y la titulación, pero no todas han influido de manera directa en éste. Las que sí lo han hecho son:

- Programación 1, Programación II, Tecnología de la Programación para todo el tema de la estructura, organización e implementación del código.
- Estructura de Datos y Algoritmos para el empleo de distintas estructuras de datos para las fronteras de los algoritmos de búsqueda, además de generación de árboles, grafos y recorridos de estos.
- Teoría de la Computación, para la lógica de los estados y grafos del problema planteado
- Inteligencia Artificial para todo el concepto de los agentes de resolución de problemas.
- Sistemas de Ayuda a la Toma de Decisiones para la lógica de los estados del Pac-Man.

Se puede confirmar haber cumplido con los objetivos propuestos para el TFG exitosamente, que eran los siguientes:

1. Enfocar el estudio de la IA hacia la preparación del alumnado para determinar cuándo un enfoque es adecuado para la resolución de un problema concreto, identificando la representación apropiada, el mecanismo de razonamiento, así como su implementación y evaluación.
2. Cumplir con el Objetivo de Desarrollo Sostenible (ODS) de Educación de Calidad, en concreto 4.4. que consiste en aumentar considerablemente el número de jóvenes y adultos que tienen las competencias necesarias, en particular técnicas y profesionales, para acceder al empleo, el trabajo decente y el emprendimiento [50].

Este trabajo ha supuesto para mí un antes y un después de la carrera, marca el final de una etapa que, aunque ha sido dura y ha requerido de un gran esfuerzo, ha merecido la pena y además de adquirir mucho conocimiento técnico y aprender a buscarnos la vida, para solventar posibles errores, como nos pasará en el día de mañana, también me ha hecho crecer como persona y madurar más. El trabajo en una primera instancia no creía que fuera a tener la complejidad que, a día de hoy, una vez ha finalizado ha supuesto.

La labor de investigación conllevó un tiempo, la que se requería encontrar si se había llevado a cabo antes esta idea y un código con una interfaz separable de la lógica con la que se movían los distintos agentes.

El principal muro que ha supuesto este TFG para mí ha sido la lógica para dotar de ïnteligencia al Pac-Man", los algoritmos de búsqueda estaban claros, pero faltaba por definir qué estados determinarían que el juego no había acabado todavía y cuales sí. Esto me supuso más de un quebradero de cabeza, sin exagerar pude estar dándole vueltas durante 2 o más semanas sobre cómo podría plantear este concepto. Cuando tenía cualquier rato libre el Pac-Man estaba dentro de mi cabeza, e incluso como no disponía de un tiempo completo para ello, siempre que se me iban ocurriendo ideas me lo apuntaba hasta que un día di con un posible enfoque de plantear que los estados pudieran ir determinados por la cantidad de comida que quedará en el laberinto hasta alcanzar la nada.

Tras tener todo algo más claro me dispuse a implementarlo, labor que tampoco resultó ser sencilla ya que al partir de un código, sin apenas documentación... (si tanto nos insisten por la buena documentación en el grado... por algo será) resultó ser más tediosa de lo que puede parecer, ya que había mucho método sin sentido aparente que encima la dificultaba. Por fin conseguí los parámetros necesarios para poder seguir desarrollando la implementación y cuando pareció estar todo enfocado se cayó en la cuenta de que, se necesitaba tener en consideración la posición del agente en cada momento. Tras darle vueltas y parecer algo complejo, era tan sencillo como comprobar solo la cantidad de comida para ver si se había alcanzado el estado final y ya está. Pero todavía no funcionaba correctamente porque faltaba conocer cómo manejar la ubicación de la posición en la que estaba la comida, algo que en un principio al no darle mucha vuelta no se tuvo en cuenta. Tras añadir esto, se consiguió dotar de "inteligencia.ªl Pac-Man.

Otro pequeño quebradero de cabeza fueron las heurísticas que se podían plantear para un valor de un estado actual, ya que el estado final con un 0 mucha información no podía aportar.

Al final, se plantean operaciones matemáticas no muy complejas pero que llaman la atención, ya que dan mejores resultados de lo esperado. Se buscará en un futuro intentar darle un enfoque nuevo, planteando heurísticas más elaboradas.

Para finalizar con las conclusiones, comentar que la obtención de los resultados no fue algo que requirió mucho tiempo y no se diría que fue lo más complejo del trabajo sino un análisis de distintos datos. Pero como siempre las he cuidado desde que entré la carrera, considero que las memorias no son complejas sino costosas en tiempo a invertir.

Dicho esto, la programación no resultó ser compleja sino la lógica detrás de todo esto y la utilización de un código no propio con escasa documentación.

En lo que respecta al trabajo futuro. Se podría centrar en:

- La actualización del proyecto para mejorarlo y mantenerlo usable con el paso del tiempo, haciendo así las modificaciones necesarias para que pueda seguir siendo usado con las nuevas versiones de Python publicadas en el repositorio de GitHub.
- Como modificaciones más específicas del código:
  - La búsqueda de una heurística más acertada para este proyecto o algún laberinto en concreto.
  - En la búsqueda actual no hay adversarios porque la presión del fantasma no se ha de tener en cuenta para la resolución del problema, pero se plantea una nueva versión de futuro donde se lleve a cabo una implementación del juego, pero con adversarios que sería el PacMan contra los fantasmas. Para ello, sería empleado el algoritmo MINI-MAX, la Poda Alfa-Beta y/o funciones de evaluación [31].
  - La modificación del código para que si un fantasma está cerca o no en el camino del Pac-Man se tenga en consideración y escoja un estado diferente.
  - Conseguir que funcione para todas las versiones de Python, tratando de solventar el error que produce la función lambda para las versión 3.10 en Python.
- Mejorar la accesibilidad para los alumnos y crear una aplicación que pueda ser usada por cualquiera, pero para ello hay que mejorar en mayor medida la accesibilidad.
- Y, para finalizar, se pretende publicar este trabajo en un revista de educación de impacto.

Muchas gracias por haber llegado hasta aquí. ¡Espero que te haya gustado mucho mi trabajo!

8. Bibliografía

[1] Alhejali, A. M., and Lucas, S. M. Evolving diverse ms. pac-man playing agents using genetic programming. In *2010 UK Workshop on Computational Intelligence (UKCI)* (2010), pp. 1–6.

[2] Alhejali, A. M., and Lucas, S. M. Using genetic programming to evolve heuristics for a monte carlo tree search ms pac-man agent. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (2013), pp. 1–8.

[3] Bell, N., Fang, X., Hughes, R., Kendall, G., O'Reilly, E., and Qiu, S. Ghost direction detection and other innovations for ms. pac-man. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games* (2010), pp. 465–472.

[4] Bom, L., Henken, R., and Wiering, M. Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)* (2013), pp. 156–163.

[5] Carney, M., Webster, B., Alvarado, I., Phillips, K., Howell, N., Griffith, J., Jongejan, J., Pitaru, A., and Chen, A. Teachable machine: Approachable web-based tool for exploring machine learning classification. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, Apr. 2020), ACM.

[6] Cuemath. Distance formula - derivation, examples, types, applications. `https://www.cuemath.com/distance-formula/`.

[7] Cuemath. Euclidean distance formula - derivation, examples. `https://www.cuemath.com/euclidean-distance-formula/`.

[8] Dai, J.-Y., Li, Y., Chen, J.-F., and Zhang, F. Evolutionary neural network for ghost in ms. pac-man. In *2011 International Conference on Machine Learning and Cybernetics* (2011), vol. 2, pp. 732–736.

[9] de Berkeley, U. Materiales de ia de berkeley. http://ai.berkeley.edu/search.html.

[10] DeLooze, L. L., and Viner, W. R. Fuzzy q-learning in a nondeterministic environment: developing an intelligent ms. pac-man agent. In *2009 IEEE Symposium on Computational Intelligence and Games* (2009), pp. 162–169.

[11] DeNero, J., and Klein, D. Teaching introductory artificial intelligence with pac-man. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence* (01 2010).

[12] Española, R. A. Definición de accesibilidad - diccionario panhispánico del español jurídico - rae. https://dpej.rae.es/lema/accesibilidad#:~:text=Adm.,TRLGDPD%20%2C%20art.

[13] Gallagher, M., and Ryan, A. Learning to play pac-man: an evolutionary, rule-based approach. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.* (2003), vol. 4, pp. 2462–2469 Vol.4.

[14] Grivokostopoulou, F., Perikos, I., and Hatzilygeroudis, I. An educational game for teaching search algorithms. In *In Proceedings of the 8th International Conference on Computer Supported Education* (01 2016), vol. 2, pp. 129–136.

[15] Guido Van Rossum, F. L. D. The python language reference manual - guido van rossum, fred l. drake - google libros. https://books.google.es/books/about/The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ&redir_esc=y. (Accessed on 06/24/2022).

[16] Hao, Y., He, S., Wang, J., Liu, X., jiajian Yang, and Huang, W. Dynamic difficulty adjustment of game ai by mcts for the game pac-man. In *2010 Sixth International Conference on Natural Computation* (2010), vol. 8, pp. 3918–3922.

[17] Ikehata, N., and Ito, T. Monte-carlo tree search in ms. pac-man. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)* (2011), pp. 39–46.

[18] Initiative, O. S. The 3-clause bsd license — open source initiative. https://opensource.org/licenses/BSD-3-Clause. (Accessed on 06/17/2022).

[19] IQ, O. Manhattan distance [explained]. https://iq.opengenus.org/manhattan-distance/#:~:text=Manhattan%20distance%20is%20a%20distance,all%20dimensions%20of%20two%20points.

[20] Luo, W. PAC-MAN game based on SAPF algorithm. In *2018 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS)* (Jan. 2018), IEEE.

[21] MathWorks. Configure generated c function interface for model entry-point functions - matlab & simulink. https://www.mathworks.com/help/rtw/ug/configure-c-code-generation-for-model-entry-point-functions.html#:~:text=An%20entry%20point%20is%20a,when%20the%20application%20starts%20executing.

[22] Miranda, M., Sánchez-Ruiz, A. A., and Peinado, F. A neuroevolution approach to imitating human -like play in ms. pac-man video game.

[23] Nguyen, K. Q., and Thawonmas, R. Monte carlo tree search for collaboration control of ghosts in ms. pac-man. *IEEE Transactions on Computational Intelligence and AI in Games 5*, 1 (2013), 57–68.

[24] Nielsen, J. 10 usability heuristics for user interface design. https://www.nngroup.com/articles/ten-usability-heuristics/, 11 2020.

[25] Norvig, P. R., and Intelligence, S. A. A modern approach. *Prentice Hall Upper Saddle River, NJ, USA: Rani, M., Nayak, R., & Vyas, OP (2015). An ontology-based adaptive personalized e-learning system, assisted by software agents on cloud storage. Knowledge-Based Systems 90* (2002), 33–48.

[26] Oh, K., and Cho, S.-B. A hybrid method of dijkstra algorithm and evolutionary neural network for optimal ms. pac-man agent. In *2010 Second World Congress on Nature and Biologically Inspired Computing (NaBIC)* (2010), pp. 239–243.

[27] Ortega, D. B. Machine learning applied to pac-man final report, 6 2015.

[28] Pepels, T., Winands, M. H. M., and Lanctot, M. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games 6*, 3 (2014), 245–257.

[29] Picazo, P. G. Tema 2 (i). resolución de problemas y búsqueda, 2020.

[30] Picazo, P. G. Tema 2 (ii). búsqueda informada, 2020.

[31] Picazo, P. G. Tema 2 (iv). búsqueda entre adversarios - juegos, 2020.

[32] PyCharm. Pycharm: el ide de python para desarrolladores profesionales, por jetbrains. https://www.jetbrains.com/es-es/pycharm/. (Accessed on 06/21/2022).

[33] PyPI. pyreverse · pypi. https://pypi.org/project/pyreverse/.

[34] Python. copy — shallow and deep copy operations — python 3.10.5 documentation. https://docs.python.org/3/library/copy.html. (Accessed on 06/22/2022).

[35] Python. Download python — python.org. https://www.python.org/downloads/. (Accessed on 06/21/2022).

[36] Python. Welcome to python.org. https://www.python.org/. (Accessed on 06/21/2022).

[37] Qu, S., Tan, T., and shuhuiq, Z. Z. Reinforcement learning with deeping learning in pacman, 2014.

[38] Respuestas, T. ¿qué es un autograder? - tus respuestas. https://tusrespuestas.net/que-es-un-autograder/.

[39] Robles, D., and Lucas, S. M. A simple tree search method for playing ms. pac-man. In *2009 IEEE Symposium on Computational Intelligence and Games* (2009), pp. 249–255.

[40] Rohlfshagen, P., and Lucas, S. M. Ms pac-man versus ghost team cec 2011 competition. In *2011 IEEE Congress of Evolutionary Computation (CEC)* (2011), pp. 70–77.

[41] Safak, A. B., Bostanci, E., and Soylucicek, A. E. Automated maze generation for ms. pac-man using genetic algorithms. *International Journal of Machine Learning and Computing 6*, 4 (2016), 226–230.

[42] Samothrakis, S., Robles, D., and Lucas, S. Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games 3*, 2 (2011), 142–154.

[43] SEAS. ¿qués es la usabilidad de una interface? — blog seas. https://www.seas.es/blog/informatica/ques-es-la-usabilidad-de-una-interface/#:

~:text=La%20usabilidad%2C%20seg%C3%BAn%20la%20RAE,para%20alcanzar%20un%20determinado%20objetivo%E2%80%9D., 10 2019.

[44] Silla, C. N. Teaching genetic algorithm-based parameter optimization using pacman. In *2016 IEEE Frontiers in Education Conference (FIE)* (Oct. 2016), IEEE.

[45] Tamassia, M., Zambetta, F., Raffe, W. L., Mueller, F., and Li, X. Learning options from demonstrations: A ¡italic¿pac-man¡/italic¿case study. *IEEE Transactions on Games 10*, 1 (2018), 91–96.

[46] Tang, D. Empowering novices to understand and use machine learning with personalized image classification models, intuitive analysis tools, and mit app inventor, 1 1970.

[47] Techopedia. ¿qué es la heurística? - definición de techopedia. https://www.techopedia.com/definition/5436/heuristic, 3 2022.

[48] Tilkidagi, S. Monte-carlo tree search algorithm in pac-man identification of commonalities in 2d video games for realisation in ai (artificial intelligence), 11 2021.

[49] Udacity. __init__ in python: An overview — udacity. https://www.udacity.com/blog/2021/11/__init__-in-python-an-overview.html#:~:text=The%20__init__%20method%20is%20the%20Python%20equivalent%20of,is%20only%20used%20within%20classes. (Accessed on 06/21/2022).

[50] Unidas, N. Educación - desarrollo sostenible. https://www.un.org/sustainabledevelopment/es/education/.

[51] VERI, B. M. R. Project - 1 cs:246 artificial intelligence, 09 2019.

[52] W3Schools. Python lambda. https://www.w3schools.com/python/python_lambda.asp.

[53] Wiki, P.-M. Pac-man (videojuego) — pac-man wiki — fandom. https://pacman.fandom.com/es/wiki/Pac-Man_(videojuego).

[54] Wikipedia. Github - wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/GitHub.

[55] Wirth, N., and Gallagher, M. An influence map model for playing ms. pac-man. In *2008 IEEE Symposium On Computational Intelligence and Games* (2008), pp. 228–233.

[56] y FSF, G. El sistema operativo gnu y el movimiento del software libre. `https://www.gnu.org/home.es.html`. (Accessed on 06/16/2022).

[57] Zikky, M. Review of a* (a star) navigation mesh pathfinding as the alternative of artificial intelligent for ghosts agent on the pacman game. *EMIT. Int. J. Eng. Technol. 4*, 1 (Aug. 2016).

[58] Zou, Y. General pacman AI: Game agent with tree search, adversarial search and model-based RL algorithms. In *2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)* (Sept. 2021), IEEE.

Anexos

A. Enunciado de la práctica

En este anexo se va a dejar un boceto de lo que podría ser el boletín de prácticas para los alumnos.

Esta práctica se corresponde con el Tema 2 de la asignatura, que a su vez está basado en el Capítulo 3 de la bibliografía básica (*"Artificial Intelligence: A Modern Approach, Chapter 3: Solving Problems By Searching"*).

A.1. Objetivos de la práctica

La primera práctica de la asignatura se centrará en la resolución básica de problemas mediante búsqueda por expansión de estados. Se estudiarán diferentes estrategias tanto informadas como no informadas y se estudiará su eficiencia a la hora de resolver problemas en entornos deterministas, completamente observables y estáticos.

Gran parte de los juegos para un solo jugador (sudoku, puzles, etc.) podrían caer en esta categoría, pero también otros problemas a resolver en los que se puedan definir de forma inequívoca los diferentes estados del sistema y las transiciones entre los mismos mediante una serie finita de acciones.

A.2. Búsqueda no informada e informada

Los problemas de búsqueda siempre comienzan con un estado inicial del sistema, y se centran en encontrar un estado final que satisface ciertas condiciones. Por tanto, debemos definir para cada problema los siguientes parámetros:

- El estado inicial del sistema.
- Las posibles acciones que pueden producirse desde cada uno de los posibles estados del sistema.
- El resultado de dichas acciones, es decir, a qué estado nos conduciría una acción dado un estado dado.
- Una función de coste, que asigna un coste a cada serie de acciones. Normalmente se representa como g(n).

– Un test para determinar si un estado es el final (también llamado meta u objetivo).

El esquema general de un algoritmo de búsqueda aparece en la Figura 3.7 de la bibliografía, definida como *TREE-SEARCH* y *GRAPH-SEARCH*. La única diferencia entre ellos es que *GRAPHSEARCH* incorpora una lista de estados visitados (explorados) para evitar repeticiones, por lo que es más eficiente especialmente en entornos con multitud de estados. Por ello, se usará este algoritmo como base para la práctica:

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
    initialize the frontier using the initial state of *problem*
    initialize the explored set to be empty
    loop do
        if the frontier is empty **then return** failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state **then return** the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set

Figura A.1: Descripción informal del algoritmo general de búsqueda en grafos

Existen diferentes variantes de este algoritmo, todas ellas basadas en pequeños matices a la hora de elegir un nodo a expandir de la frontera de estados visitados. Se puede hacer una primera división entre algoritmos de búsqueda no informada, o informada:

– **Búsqueda no informada (ciega):** las estrategias no disponen de información adicional sobre estados más allá de la proporcionada en la definición del problema, por lo que sólo pueden expandir estados y distinguir entre estados objetivos y no-objetivos. Dentro de esta categoría se distinguen:

  - **Búsqueda primero en anchura (*breadth-first*):** se expanden primero los estados de un nivel antes de expandir los del estado siguiente.
  - **Búsqueda primero en profundidad (*depth-first*):** se expanden primero los estados de mayor nivel (los más profundos).
  - **Búsqueda en coste uniforme (*uniform cost*):** se expanden primero los estados con menor coste acumulado de acciones para llegar a él.

– **Búsqueda informada (heurística):** se utiliza información propia del problema más allá de la proporcionada en la definición. Se utiliza una función heurística, normalmente denominada h(n), que estima el coste del camino más corto desde el estado actual hasta el objetivo. Para que la heurística sea admisible, nunca puede indicar un coste mayor que el real para algún estado. En esta categoría se incluirían:

  - **Búsqueda voraz, avariciosa o primero el mejor (*greedy or best-first*):** se expanden primero los estados con el menor valor de la función heurística, h.
  - **Búsqueda A*:** se expanden primero los estados con el menor valor de la suma del coste para llegar al estado más la estimación de alcanzar el estado objetivo, es decir, f = g + h.

A.3. Problema a resolver

Para aplicar las estrategias de búsqueda, resolveremos el juego del Pac-Man, pero no en el sentido de conseguir que pueda ganar todas las partidas, sino que se pueda finalizar toda la comida del laberinto de la manera más óptima sin la presión generada por parte de los fantasmas.

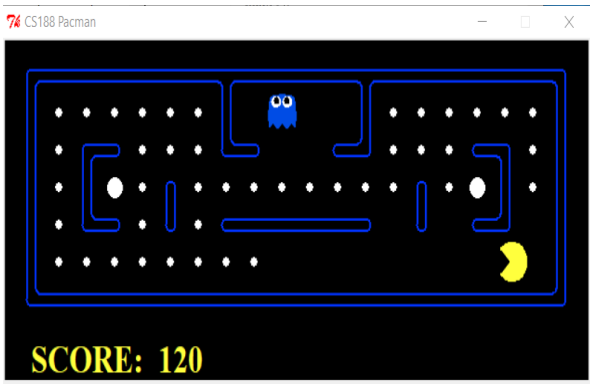

Figura A.2: Ejemplo de laberinto para el juego del Pac-Man

Se parte de un estado en el que el laberinto tiene un número indeterminado de comida, la posición no influye de manera directa aunque como se verá será necesaria para determinar los estados. Las acciones que se podrán llevar a cabo serán ir a la derecha 'East', ir a la izquierda 'West', hacia arriba 'North' y para abajo 'South'.

Por lo tanto, se debe tener en cuenta que aunque el objetivo es finalizar toda la comida, la posición del Pac-Man será relevante para tener en cuenta donde se encuentra y comprobar si en dicha casilla hay comida o no. Ya que si la hay se podrá decrementar

el valor de la comida restante en el laberinto y estar un paso más cerca de alcanzar el objetivo final que sería que no quedase nada de comida en el laberinto.

El coste de una solución (función g) se considerará igual a cada desplazamiento llevado a cabo, es decir, cuando el Pac-Man se mueve a una casilla, cuenta como 1.

A.4. Ficheros de la práctica

La práctica está compuesta por los siguientes ficheros:

- *datastructures.py:* contiene estructuras de datos útiles en Python para la realización de los algoritmos (*Stack, Queue, PriorityQueue*).

- *search.py:* contiene la estructura básica del proceso de búsqueda no informada e informada. Sin embargo, algunas de las funciones no tienen el código necesario para que el algoritmo funcione correctamente. Por tanto, cada alumno debe rellenar el código de las siguientes funciones: *uninformed_search, breadth_first, depth_first, uniform_cost, informed_search, greedy, a_star, h1* u otras funciones heurísticas.

- *pacmanState.py:* contiene la clase que representa cada estado del problema. Cada alumno debe rellenar el código de las siguientes funciones: *succ, next_states*

- *pacman.py:* contiene el programa principal de la práctica, que hace llamadas a los diferentes algoritmos de búsqueda. En este solo se tendrán que modificar los parámetros de entrada como qué tipo de agente de Pac-Man es o el escenario que se quiere emplear, todo se puede cambiar en el main de dicho fichero. Si se desea que no haya fantasmas que interrumpan la ejecución del Pac-Man, habrá que modificar en el método *runGames*, una variable como se ha indicado el código denominada *ghosts*. Con respecto al laberinto que se desea resolver con el Pac-Man se recomienda que no se empiece con uno muy grande ya que puede tardar incluso días la resolución de alguno de estos.

- *graphicsDisplay,py, graphicsUtils.py, layout.py:* proporcionan una interfaz gráfica para poder ver la solución encontrada con alguno de los algoritmos de búsqueda. De estas solo quizás es necesario consultar los parámetros de los que consta *layout.py*.

- *game.py, gameState.py, agents.py:* implementan la lógica necesaria para que el Pac-Man se mueva o consuma la comida

A.5. Entrega de la práctica

Deberán completare las siguientes tareas para considerar la práctica entregada:

1. Completar el código Python necesario para que los algoritmos de búsqueda no informada e informada funcionen correctamente, siguiendo el algoritmo *GRAPH-SEARCH*. Además, deberán proponerse, al menos 2 funciones heurísticas diferentes y admisibles para ser utilizadas en los algoritmos de búsqueda informada. La eficiencia de las funciones heurísticas formará parte de la evaluación de la práctica.

2. Estudiar la optimalidad y eficiencia de los algoritmos en un laberinto concreto que no expanda ni genere muchos nodos, comparando la longitud de las soluciones encontradas y el número de nodos expandidos y generados por cada algoritmo. Discutir los resultados.

3. Estudiar el comportamiento de los algoritmos cuando se escoge un laberinto más grande del ya escogido o con una mayor cantidad de comida.

Todas estas tareas se llevarán a cabo en parejas o de manera individual.

La práctica se entregará a través del Anillo Digital Docente (ADD), para lo cual se deberá realizar una memoria indicando cómo se ha realizado la implementación y los resultados obtenidos por los diferentes algoritmos, y se deberá entregar un fichero comprimido con el código de la práctica y con la memoria realizada en formato PDF. La fecha límite de entrega será el 22/11/2022.

A.6. Recursos adicionales

Explicación del juego del Pac-Man: `https://pacman.fandom.com/wiki/Pac-Man_(game)`