Práctica individual
Inteligencia Artificial

Jorge Rodríguez Fraile, 100405951

ÍNDICE

Introducción

En este documento se recogen las pruebas y resultados obtenidos durante la prueba de los distintos algoritmos de búsqueda vistos en clase aplicados al famoso juego Pacman, más conocido como Comecocos.

El objetivo es, desde una posición inicial y recorriendo el escenario planteado, que es un laberinto, alcanzar la posición final o meta donde se encuentra el premio, llamado comida.

Otro objetivo a alcanzar, haciendo pruebas en otros escenarios diferentes, es distribuir numerosas comidas por el laberinto y que Pacman consiga comérselas todas. Todo ello con el menor número de pasos, para obtener el menor coste posible.

Algoritmos de búsqueda implementados en search.py

Búsqueda en profundidad

Este algoritmo consiste en una pila de nodos que se van generando al recorrer el árbol, lo que quiere decir que mientras vas generando nodos, uno se va poniendo sobre otro y cuando tenemos que extraer uno para avanzar debemos coger el último que hemos introducido. Es decir, el último en entrar es el primero en salir.

No pueden entrar en la pila nodos que ya estén dentro o lo hayan estado, de esta manera no se genera un bucle. En caso de elegir un primer sucesor se elige en orden: arriba, abajo, derecha e izquierda. El algoritmo termina cuando se encuentra el nodo meta, el camino que le ha tenido que recorre es la solución del problema.

Comienza en la posición inicial y va avanzando por una rama hasta llegar a la solución o, si no puede avanzar más, retrocede y empieza a explorar otra rama y así sucesivamente hasta llegar a una solución. Este algoritmo puede encontrar la solución en menos tiempo, pero seguramente más costosa que otros algoritmos.

Búsqueda en amplitud

La búsqueda en amplitud es muy similar a la búsqueda en profundidad, con la diferencia de que no usa una pila de datos, sino una cola. En una cola el primer nodo en entrar es el primero en salir, lo que significa que va recorriendo el árbol de nodos por niveles. Esto hace al algoritmo más complejo espacialmente. Al igual que en profundidad un nodo no entra dos veces en la cola, de esta manera se evitan bucles.

Comienza por la posición inicial y va recorriendo sucesivamente las posiciones adyacentes, después continua por orden con las adyacentes a las anteriores y así sucesivamente, respetando las posiciones ilegales, barreras. Este algoritmo lo que asegura es que se encuentra la solución con el menor número de pasos, aunque implique expandir más nodos.

Dijkstra

Consiste en una lista de los nodos sin explorar ordenados de menor a mayor coste. Aunque el coste unitario de cada paso es el mismo, el orden viene dado por los pasos necesarios para llegar a cada uno de ellos desde el origen. Al igual que amplitud va eligiendo el primer nodo, que en este caso es el de menor coste, lo que nos asegura

que el camino que estamos siguiendo es el más barato. A diferencia de los dos algoritmos anteriores en este caso, en principio, no se desechan los nodos repetidos, ya que finalmente podrían usarse por tener un menor coste que el elegido inicialmente y ser el mejor camino final.

Comienza en la posición inicial y se calcula el coste de todas las posiciones adyacentes, se escoge el de menor coste o si son del mismo coste el que ocupe la primera posición. En esa nueva posición se calculan otra vez los costes de las adyacentes, y se meten en la lista, y se vuelve a elegir el de menor coste. Así sucesivamente hasta que se alcanza la meta. El camino que nos lleva hasta la meta, al ser todos los pasos con coste uniforme, será el que menos pasos requiere.

A* con distancia euclídea como heurística

A* es un algoritmo similar a Dijkstra, pero en este caso además del coste uniforme se le añade el de una heurística que nos proporciona información para localizar la meta en forma de coste, cuanto menor sea la heurística más cerca de la meta nos encontraremos.

En este caso la heurística es la distancia euclídea que mide la cantidad de posiciones entre el nodo actual y el final. Esta distancia es la línea recta entre las dos posiciones. Formula del coste: $h = ((x1 - x2)^2 + (y1 - y2)^2)^{0,5}$. Al igual que Dijkstra pueden aparecer pasos iguales, pero solo se quedará con el de menor coste. El coste de cada paso es la suma de los costes uniformes desde el inicio hasta ese nodo concreto más el coste de la heurística de ese mismo nodo, pero no se acumula la heurística de los nodos anteriores.

Comienza por la posición inicial y para cada posible paso calcula el coste (de la manera explicada anteriormente) y se meten todos en una lista ordenada de menor a mayor. Se escoge el de menor coste y así sucesivamente se van calculan los costes de las posiciones adyacentes hasta que se extrae de la lista el nodo final.

Análisis de PositionSearchProblem

Espacio de estado:

Está compuesto por todas las posibles posiciones que puede ocupar el Pacman en el tablero, sin barreras y en forma de coordenadas x e y. De tal manera que la fila más baja es la y=0 y la columna más a la izquierda es la x=0.

Ejemplo: Si Pacman puede ocupar la casilla de la esquina inferior izquierda, un estado será (1, 1).

No obstante, el algoritmo cuando trabaja sobre el tablero también almacena desde donde se ha llegado a esa posición {North, South, West, East} y cuál es el coste acumulado medido en número de operaciones.

Ejemplo: Si la posición (1,1) es la inicial y se puede desplazar hacia el norte, el estado al que puede llegar se indica con ((1, 2), North, 1); ya que llega al inmediato de arriba, con dirección Norte y como el anterior era inicial el coste del primer será 1.

Estado inicial y test de meta:

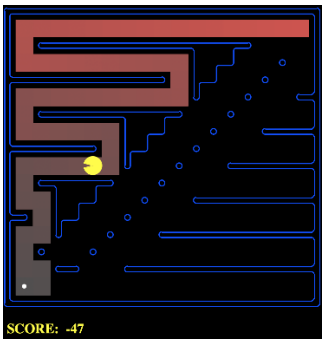
Ambos datos se extraen de la definición del escenario. El estado inicial es donde esté el Pacman, en el escenario se marca con una “P” y el estado meta es donde esté la comida, que se indica con un “.” en el escenario y normalmente está colocado en el (1, 1). El (1,1) es lo más bajo y a la izquierda posible, ya que tiene que haber bordes.

Operadores, con sus condiciones de aplicabilidad y su resultado:

Lo primero, para que la prueba se pueda llevar a cabo, la condición inicial es que esté definida en el escenario la posición que debe ocupar Pacman y que haya solamente una comida.

Para definir los sucesores de un estado lo que hace es probar para cada dirección si hay una barrera. Si no la hay suma o resta 1 dependiendo de la dirección a la que se desplace. Indica la dirección de desplazamiento y suma 1 al coste acumulado del anterior, pero si hay una barrera no meterá en la lista abierta el estado.

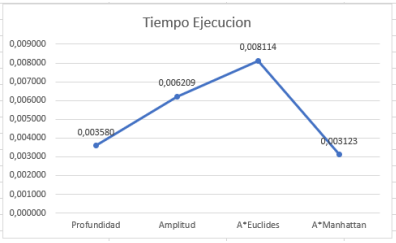
Escenario 1

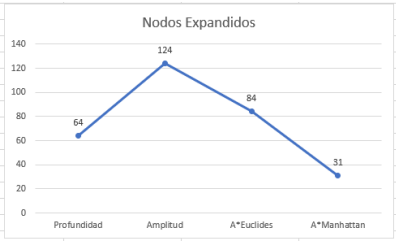


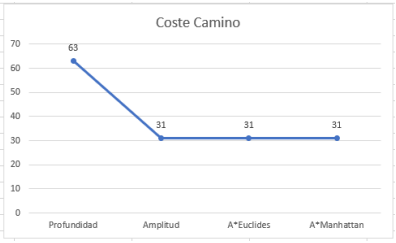
El primer escenario es un tablero de 20x18, en el que hay una bajada diseñada para que A* con heurística euclídea se quede justamente a un paso, ya que no puede avanzar al haber una barrera. Por la parte de arriba a la izquierda del tablero hay un zigzag para que el algoritmo de profundidad sea más costoso y no llegue con el coste óptimo. En amplitud va avanzando por todos a la vez, por lo que será costoso. La parte inferior derecha es una bajada para que A* con distancia Manhattan sea más eficiente. Esa misma zona está abierta para que profundidad avance sin éxito. Estado inicial (19, 17) y final (1, 1)

Los algoritmos de amplitud y los dos de A* encuentran la solución más optima. Sin embargo, amplitud expande muchos más nodos y lo hace menos eficiente. En la gráfica del tiempo de ejecución se puede ver que, aunque A* con Euclides expande menos nodos, el algoritmo hace que tarde más tiempo que el de amplitud.

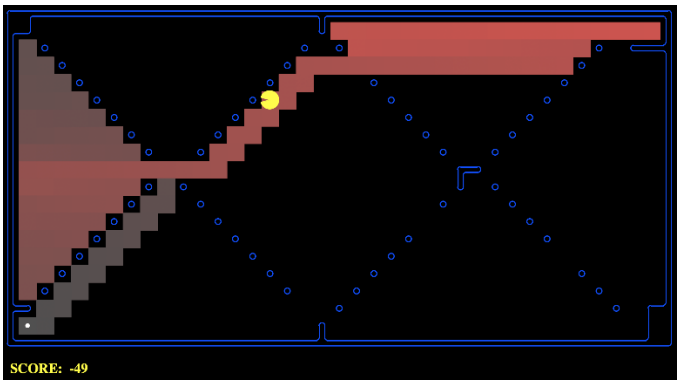
Algoritmo	Tiempo Ejecución (s)	Nodos Expandidos	Coste Camino
Profundidad	0,003580	64	63
Amplitud	0,006209	124	31
A*Euclides	0,008114	84	31
A*Manhattan	0,003123	31	31







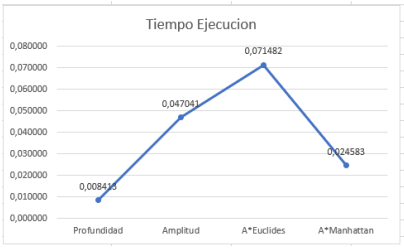
Escenario 2

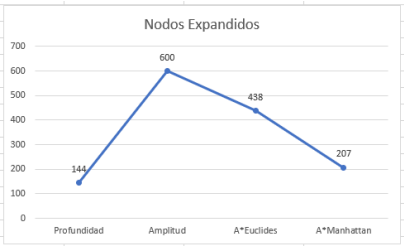


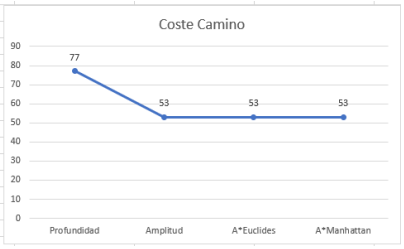
Se trata de un tablero de 40x20 con grandes áreas abiertas. En este caso, al haber áreas tan abiertas el algoritmo de amplitud expande muchos nodos a la vez, lo que hace que sea muy costoso. La forma de rombos (sobre todo el izquierdo y el derecho) hace que las dos heurísticas avancen en línea recta y luego tengan que retroceder y el algoritmo de profundidad, que es el más eficiente, se beneficia del orden de expansión de los nodos y le permite llegar a una solución más rápido, pero que no es la óptima. Estado inicial (39, 19) y final (1, 1)

Como en el escenario anterior, los algoritmos de amplitud y A* llegan a la solución óptima, pero han expandido más nodos que el de profundidad. La complejidad espacial del algoritmo de profundidad es lineal y la de los otros algoritmos es exponencial. También se observa que amplitud, al ser una búsqueda no informada, recorre muchos más nodos, ya que a diferencia del resto de algoritmos no hay unos costes que le guíen hacia la solución.

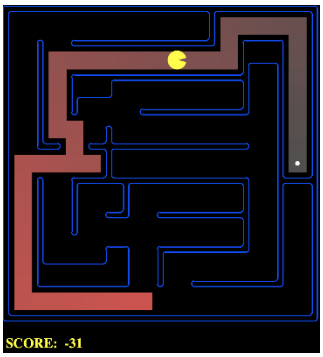
Algoritmo	Tiempo Ejecución (s)	Nodos Expandidos	Coste Camino
Profundidad	0,008413	144	77
Amplitud	0,047041	600	53
A*Euclides	0,071482	438	53
A*Manhattan	0,024583	207	53







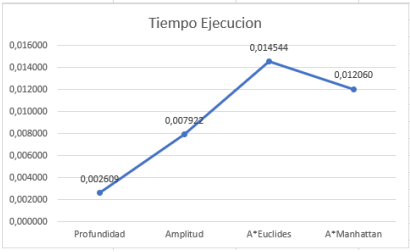
Escenario 3

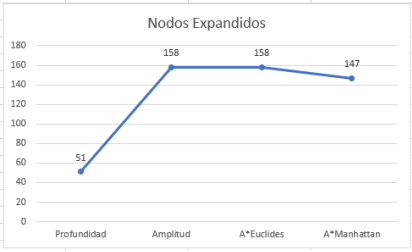


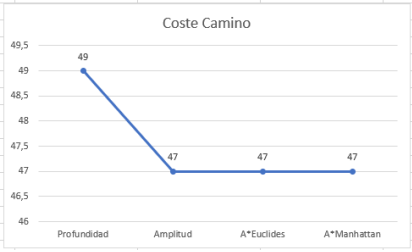
Este escenario, de 20x20, se ha diseñado con muchos caminos posibles que parecen acercarse a la meta, no consiguiendo llegar a la misma. El camino que llega a la solución es precisamente el que, a primera vista, más se aleja de la meta. Este diseño hace que la heurística no sirva de guía inicialmente, ya que solo se mete en ramas sin salida. El algoritmo que menos nodos expande es el de profundidad dado que prioriza ir primero por la rama izquierda, que en este caso es la que lleva a la meta. Estado inicial (10, 1) y final (19, 9)

A pesar de que amplitud y A* son los que más nodos expanden son los algoritmos que hallan el camino con el coste optimo, pero sus tiempos y nodos en hallarlo son considerablemente superiores a los que supone la búsqueda en profundidad, el cual es ligeramente más costoso. En este caso, teniendo en cuenta la poca diferencia existente entre el coste de profundidad y el de los demás algoritmos, así como que el resto expande muchos más nodos, podemos concluir que el mejor algoritmo para este caso específico es el de profundidad.

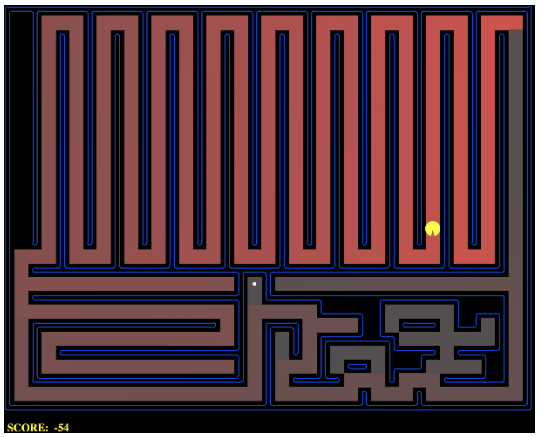
Algoritmo	Tiempo Ejecución (s)	Nodos Expandidos	Coste Camino
Profundidad	0,002609	51	49
Amplitud	0,007922	158	47
A*Euclides	0,014544	158	47
A*Manhattan	0,012060	147	47







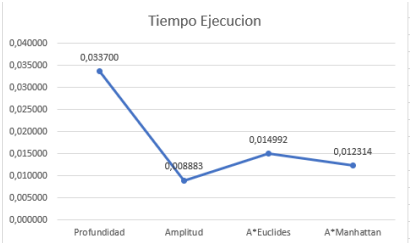
Escenario 4

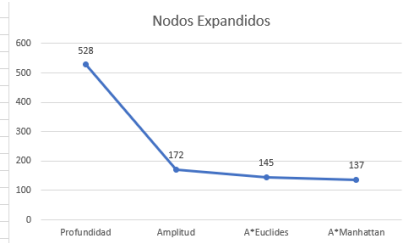


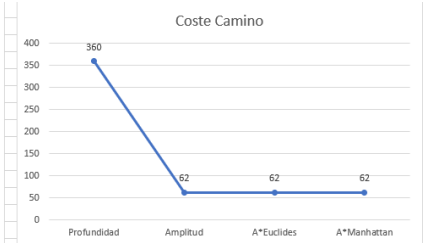
Este escenario de 40x30 trata de mostrar que, a pesar de que en los apartados anteriores parece que el algoritmo de profundidad es el que primero encuentra la meta, la mayoría de las veces esto no es así. Este diseño se aprovecha de que seguirá una rama, para meterlo en un largo recorrido que le lleva a la solución directamente, pero con un elevadísimo coste. Para el resto de algoritmos hay un pequeño laberinto que deben resolver. Estado inicial (39, 29) y final (20, 9)

En este caso la solución de profundidad es mucho más costosa que la del resto. Todos los demás han coincidido en una solución común, que sí que es la óptima. Amplitud encuentra la solución óptima a base de recorrer en cada iteración los nodos expandidos en cada paso anterior ordenadamente, lo que hace que llegue, pero ciegamente. Tanto A* con heurística euclídea como la de distancia de Manhattan están guiadas por el coste de llegar si no hubiese barreras, de esta manera no hace falta que expandan tantos nodos como la amplitud, para hallar el camino optimo, solo guiándose por el coste de la heurística.

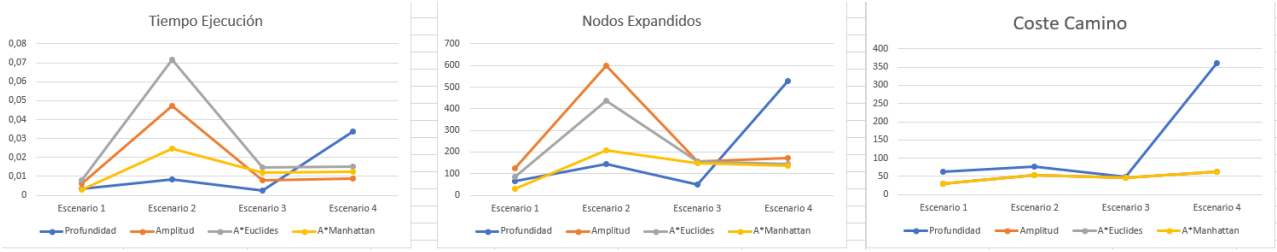
Algoritmo	Tiempo Ejecución (s)	Nodos Expandidos	Coste Camino
Profundidad	0,033700	528	360
Amplitud	0,008883	172	62
A*Euclides	0,014992	145	62
A*Manhattan	0,012314	137	62



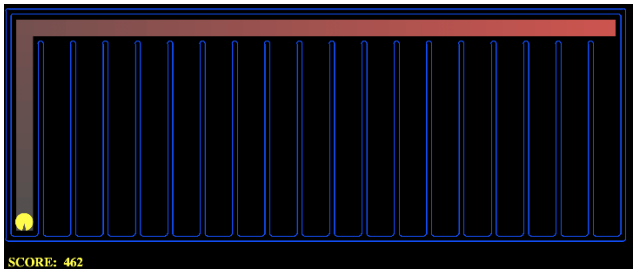




Grafica de los 4 escenarios en conjunto



Escenario adicional 1

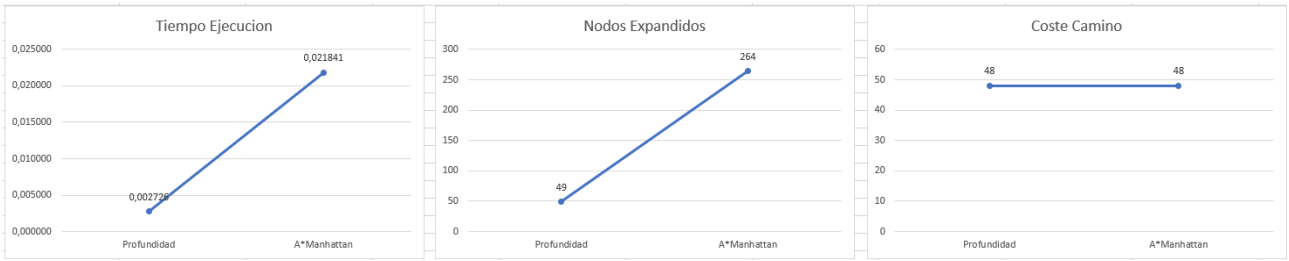


Este escenario de 40x15 está especialmente diseñado para la tarea pedida, que el algoritmo en profundidad halle el camino optimo con menor coste que A* con heurística de distancia de Manhattan. Estado inicial (39, 14) y final (1, 1)

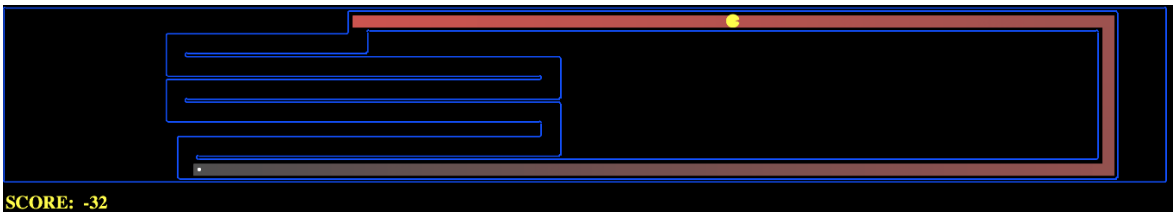
Este diseño se basa en el hecho de que el algoritmo de profundidad sigue un determinado de expansión de nodos. De esta manera, el camino optimo es una L hacia la izquierda y abajo, que corresponde con el orden que seguiría el algoritmo de profundidad.

Además, para que A* con distancia de Manhattan expanda más nodos, aprovechamos que su algoritmo ante igual coste de sucesores elegirá el de abajo para crear columnas que bajan y solo conectan por arriba, pero no le acercan horizontalmente a la meta desde la parte más baja.

Algoritmo	Tiempo Ejecución (s)	Nodos Expandidos	Coste Camino
Profundidad	0,002726	49	48
A*Manhattan	0,021841	264	48
Amplitud	0,014280	264	48



Escenario adicional 2

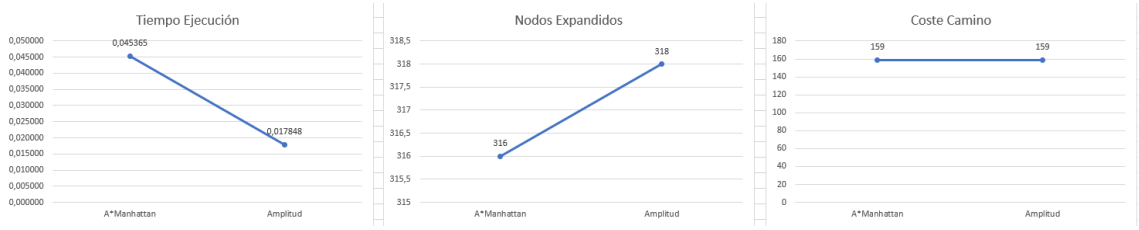


El escenario diseñado intenta que el algoritmo de amplitud expanda menos nodos que A* con distancia Manhattan. Lo intenta creando un zigzag que aparentemente le llevara a la meta de forma más directa que el otro camino. Sin embargo, el camino que se aleja más de la meta y aparentemente es más largo, realmente es 1 paso más corto que el del zigzag.

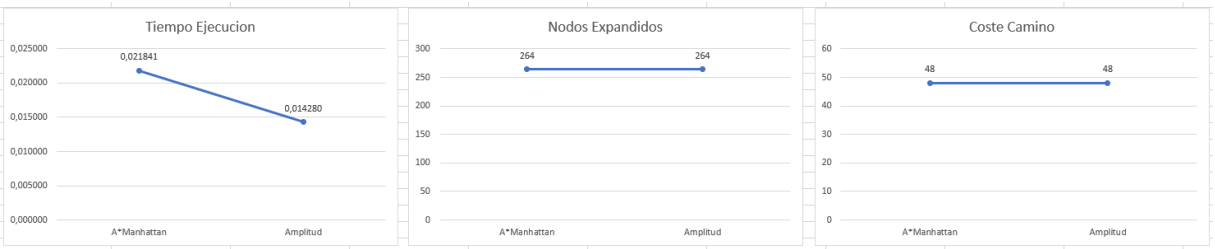
Debido a que la distancia Manhattan informa al algoritmo A*, este no se deja engañar y aunque llega a recorrer parte del zigzag, no lo termina y se va por el camino correcto. El algoritmo de amplitud como recorre todos los posibles sucesores, avanza tanto por el zigzag como por el camino de la derecha y eso hace que expanda más nodos, resultando más costoso.

No es posible para un problema de tablero con coordenadas y barreras que el algoritmo de amplitud visite menos nodos que A*. Se debe a que amplitud recorre todos los nodos que puede hasta encontrar la meta, y la distancia Manhattan solo recorre el sucesor de menor coste, por lo que no se le puede engañar con caminos que sean más largos que la propia solución. Solo será posible si la heurística no fuese admisible.

Algoritmo	Tiempo Ejecución (s)	Nodos Expandidos	Coste Camino
A*Manhattan	0,045365	316	159
Amplitud	0,017848	318	159



Lo máximo que podemos lograr es que ambos expandan los mismos nodos, la gráfica representada más abajo corresponde a hacer profundidad y A* con distancia Manhattan en el escenario adicional 1, en el que se aprecia que ambos llegan a la misma solución con el mismo coste. Además, amplitud alcanza la solución en un menor tiempo, por lo que temporalmente hablando es el mejor de los dos.



Análisis de FoodSearchProblem

Espacio de estados: Los estados son todas las posibles posiciones del escenario que son susceptible de ser ocupadas por Pacman, es decir no almacena aquellas posiciones que son barreras. Junto con una matriz de booleanos de la forma del tablero que indica donde están o no colocadas las comidas, de tal manera que si ocupa una de las posiciones en true pasa a false y será como si se la hubiese comido.

Las posiciones se definen con coordenadas x e y, estando el origen (0, 0) en la esquina inferior izquierda y la matriz con true o false, por ejemplo {true, false, false, true}.

Ejemplo: Tablero 2x2 donde hay comida en todas las posiciones excepto el origen (0, 0) y Pacman está en el inicio: ((0, 0), {true, true, false, true})

Estados iniciales y el test de meta: El estado inicial viene definido en el escenario, que se indica con una "P" donde estará el Pacman al comienzo de juego.

La matriz de booleanos se inicia con el tamaño del escenario y con true en aquellas casillas en las que haya un punto en la definición del escenario.

La meta será cuando el Pacman haya alcanzado todas las comidas, lo que quiere decir que la lista de booleanos interna donde se almacenan las comidas tenga todas las posiciones en false. Que indicará que no queda ninguna comida y habrá terminado.

Agent 1: astar ClosestFoodManhattan

Este algoritmo consiste en hallar para la posición que ocupa Pacman la distancia Manhattan desde él mismo hasta todas las posibles posiciones de comidas marcadas en la matriz, nos quedamos con la de mayor coste (h) y le sumamos el coste de los pasos que lleva desde el origen a la posición en la que se encuentra (g).

Una vez tenemos el coste calculado (f = g + f) para cada casilla colindante de Pacman, escogemos la de menor coste (lista ordenada de menor a mayor) y avanzamos a esa posición, sin desechar el resto de estados con su coste ya que puede retroceder e incluso encontrar menores costes para nodos que ya estén en la lista.

Repetimos el paso anterior hasta que la heurística nos dé un coste 0, que indica que no queda ninguna comida.

El coste de la heurística para la posición x viene dado por la mayor distancia Manhattan desde la casilla x hasta una posición de comida. Para calcular este coste ello en cada posible casilla halla la distancia Manhattan a todas las posibles comidas y se va quedando con la mayor. Se basa en que cuanto más lejos este de los nodos comida más limpio va dejando el mapa de objetivos de búsqueda, hasta que solo queda uno que es el de mayor coste, va a por él y termina.

Agent 2: astar ClosestFoodMazeDistance

Primero, se halla el coste desde la posición de Pacman hasta todas las posibles casillas que contienen comida mediante búsqueda en amplitud y escogemos la mayor distancia de todas las halladas.

A continuación, calculamos lo indicado en el primer paso para cada posición a la que se puede desplazar Pacman (h) y además le sumamos a cada uno de esos estados nuevos el coste desde el inicio hasta la casilla actual (f), ahora se introducen esos costes

en una lista ordenada de menor a mayor y se extrae el de menor coste, sin desechar los anteriores ya que puede ser recorrido o sustituidos por el mismo pero con menor coste. Avanzamos a esa posición, marcando si había comida la casilla en la matriz con false.

Repetimos el paso anterior hasta que no quede ninguna casilla con comida, que quiere decir que el coste de la heurística ha llegado a 0.

La heurística consiste en alejarse por el camino más óptimo de la comida más lejana, de esta manera se consumirán primero las casillas más cercanas al origen y progresivamente llega a la más lejana, donde terminara con un coste 0. El coste de la casilla x viene dado por el mayor coste en búsqueda por amplitud desde la posición x hasta una posición de comida, para ello calcula el coste desde x hasta todas las posibles casillas true de la matriz de comidas.

Agent 3: ClosestDotSearch

Este algoritmo resuelve el problema hallando en cada posición que va ocupando Pacman cuál es la comida más próxima (en número de pasos), y cuando la encuentra se desplaza hasta la posición y repite el proceso de hallar la más cercana y se desplaza hasta que no queda ninguna comida en el tablero.

El cálculo de la casilla con comida más cercana, respetando las barreras, lo hace con búsqueda en amplitud (FIFO) en la que la meta es encontrar una casilla marcada en la matriz de comidas (al ser amplitud la primera que alcance es la más cercana).

Aunque encuentra el camino optimo entre Pacman y la casilla a consumir, al no tener heurística no tiene en cuenta que es mejor limpiar primero una zona del escenario antes de ir a otra, ya que tendrá que volver al final y eso aumentará el coste, numero de pasos necesarios. A pesar de no dar la solución más optima en cuanto al coste en movimientos, este algoritmo es mucho más rápido que los dos anteriores en los que tenían que calcular muchas más posibilidades para cada casilla.

Comparativa en trickySearch

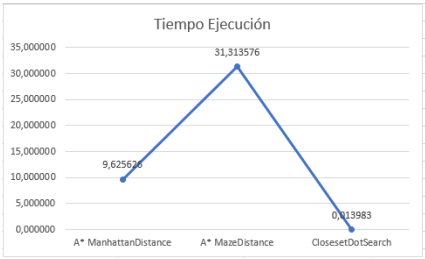
En esta prueba se puede apreciar el coste que supone cada uno de los algoritmos planteados, se puede ver que los que usan heurísticas encuentran la solución más optima. Pero un punto a favor de la que busca los más cercanos, es que, aunque es un poco más costosa, es con diferencia la que menor tiempo y nodos expande que el resto.

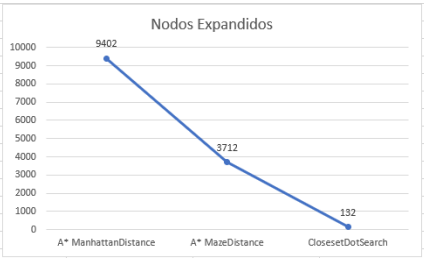
Entre las que usan heurísticas la distancia Manhattan es más barata de calcular, ya que es una operación matemática, sin embargo, Maze Distance debe calcular en cada paso muchas búsquedas en amplitud, lo que supone un coste muy elevado dado que la búsqueda en amplitud es de complejidad temporal y espacial exponencial.

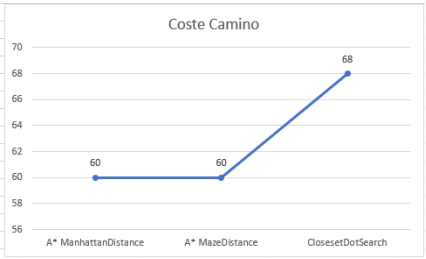
La distancia Manhattan es la que ha expandido más nodos, eso se debe a que ha probado muchas ramas que no han llevado a un camino optimo, ya que el cálculo de esa operación no tiene en cuenta muros, proporcionando menos información que la búsqueda en amplitud que tiene en cuenta cual es la ruta optima entre los dos puntos.

Por lo tanto, podemos sacar en claro que para este caso el mejor algoritmo es Closest Dot Distance, ya que es considerablemente más rápido y la solución proporcionada no difiere mucho de la óptima. El resto no son admisibles, no tardan un tiempo razonable.

Algoritmo	Tiempo Ejecución (s)	Nodos Expandidos	Coste Camino
A* ManhattanDistance	9,625626	9402	60
A* MazeDistance	31,313576	3712	60
ClosesetDotSearch	0,013983	132	68







Comparativa en bigSearch

Esta prueba ha reafirmado lo comentado en el apartado anterior, las dos búsquedas informadas son muy costosas, una temporalmente y otra espacialmente, tanto que no ha sido posible para el tablero bigSearch calcular cual sería el recorrido que debe seguir Pacman.

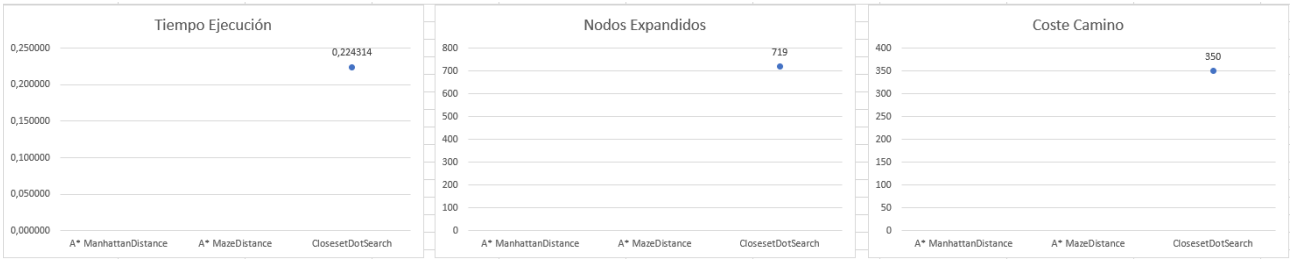
El único que ha logrado superar el mapa ha sido el de Closest Dot Distance que lo ha hecho bastante rápido, incluso más rápido de lo que lo hacían los otros algoritmos en el otro mapa, aun siendo más pequeño y con menor número de comidas.

El problema de que no haya sido posible calcular la ruta para el algoritmo de distancia de Manhattan, se debe a que este expande demasiados nodos y el equipo utilizado para las pruebas no le podía suministrar los recursos que necesitaba.

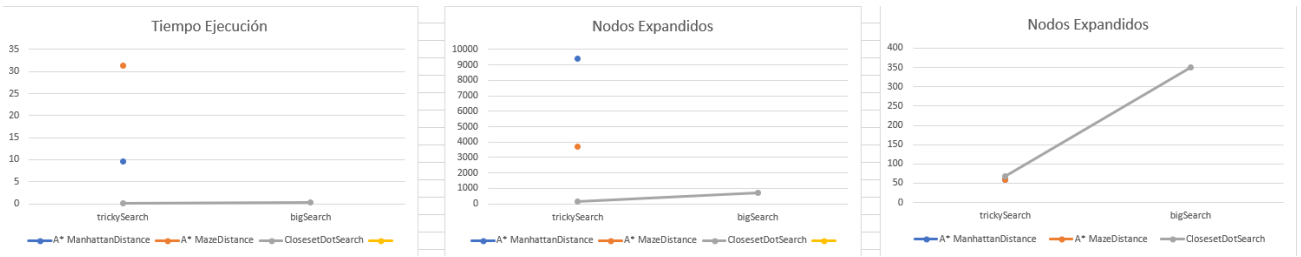
Con Maze Distance ocurría algo similar al caso anterior. Los cálculos internos ha realizar, búsquedas en amplitud desde los sucesores del nodo a todas las comidas del escenario, se incrementaban exponencialmente que el dispositivo se queda sin recursos y no podía terminar la tarea de búsqueda.

De este caso de prueba podemos deducir, que hay que tener en cuenta la complejidad de los algoritmos que utilizamos y a veces se debe emplear un algoritmo más sencillo, menos costoso, aunque esto genere caminos más largos. Es preferible tener una solución, aunque no sea la óptima, a no tener ninguna por usar un algoritmo demasiado complejo o costoso.

Algoritmo	Tiempo Ejecución (s)	Nodos Expandidos	Coste Camino
A* ManhattanDistance			
A* MazeDistance			
ClosesetDotSearch	0,224314	719	350



Grafica conjunta



Conclusiones

En lo referido a la parte de búsqueda de un solo nodo meta, se ha podido comprobar lo eficientes que resultan los distintos algoritmos que hemos aprendido en clase. La mayoría completos y admisibles, han hallado una solución y es la óptima, a excepción de profundidad que no es admisible, ya que no encuentras siempre la óptima.

En cuanto a la búsqueda en amplitud, hemos observado que siempre encuentra la solución más óptima, pero no en el menor tiempo posible y eso es un problema ya que nos podríamos quedar sin recursos. Dado que sus complejidades son exponenciales.

La búsqueda en profundidad, pocas veces proporciona el camino optimo, pero es menos costosa en cuanto a complejidad espacial, lo que hace que en alguna ocasión sea la mejor opción al requerir menos recursos, a pesar de que normalmente requiere alguna operación extra. Su mayor ventaja es poder encontrar un camino rápido, aunque no con el menor número de pasos, debido a su complejidad temporal lineal.

La búsqueda heurística A*, es la que mejores resultados ha proporcionado en las pruebas, gracias a que las heurísticas con sus pesos guiaban al algoritmo hacia el camino más optimo, cosa con la que los dos anteriores no contaban. Ambas heurísticas son admisibles.

En la parte de alcanzar todas las casillas en las que hubiese un punto, comida del juego, nos ha permitido ver algunos algoritmos capaces de resolver este determinado problema. Estos algoritmos eran mezclas de los que ya conocíamos, dos de ellos han resultado en unos más complejos, pero que encontraban la solución óptima, y otro era más simple, pero que no hallaba la mejor solución.

Por todo lo anterior, se llega a la conclusión de que no siempre la opción que está informada o realiza cálculos más complejos es la mejor opción para todos los problemas y a veces hay que optar por otros más simples, que den un resultado aceptable, ya que no siempre se dispone de todo el tiempo o capacidad de procesado que requieren.

Comentarios personales

Este trabajo, a pesar de no entender completamente el lenguaje en el que está programado, ya que en este curso no hemos trabajado en Python, ha sido muy esclarecedor.

Nos ha ayudado a entender cómo funcionan realmente los algoritmos de búsqueda de una forma más interactiva, ya que hemos tenido que diseñar nuestros propios tableros y aplicar sobre ellos los distintos algoritmos comprobando el funcionamiento en cada caso. Hemos llegando así a entender mejor cómo funcionan nuestros propios casos de prueba, al haber podido ver en cada caso el número de nodos expandidos, tiempos y costes, que son datos imprescindibles para entender cada uno de los casos y determinar cuál es el mejor. Incluso hemos podido experimentar que no se pueda realizar un cálculo debido a que se volvía demasiado complejo.