**Practica individual**

**Inteligencia Artificial**

Jorge Rodríguez Fraile, 100405951

INDICE

[Introducción 3](#_Toc38238441)

[Algoritmos de búsqueda implementados en search.py 3](#_Toc38238442)

[Búsqueda en profundidad 3](#_Toc38238443)

[Búsqueda en amplitud 3](#_Toc38238444)

[Dijkstra 4](#_Toc38238445)

[A\* con distancia euclídea como heurística 4](#_Toc38238446)

[Análisis de PositionSearchProblem 4](#_Toc38238447)

[Escenario 1 5](#_Toc38238448)

[Escenario 2 6](#_Toc38238449)

[Escenario 3 6](#_Toc38238450)

[Escenario 4 7](#_Toc38238451)

[Grafica de los 4 escenarios en conjunto 8](#_Toc38238452)

[Escenario adicional 1 8](#_Toc38238453)

[Escenario adicional 2 9](#_Toc38238454)

[Análisis de FoodSearchProblem 10](#_Toc38238455)

[Agent 1 10](#_Toc38238456)

[Agent 2 10](#_Toc38238457)

[Agent 3 10](#_Toc38238458)

[Comparativa en trickySearch 10](#_Toc38238459)

[Comparativa en bigSearch 11](#_Toc38238460)

[Conclusiones 11](#_Toc38238461)

[Comentarios personales 11](#_Toc38238462)

# 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 los nodos que se van generando al recorrer el árbol, lo que quiere decir que mientras vas generando nodos uno se va poniendo tras otro y cuando tenemos que coger uno para explorar cogemos el último introducido. El último en entrar es el primero en salir. No pueden entrar en la pila nodos que ya estén o hayan estado, de esta manera no se hace un bucle. Y el algoritmo termina cuando encuentra el nodo final, el camino que le ha llevado a él es la solución. En caso de elegir un primer sucesor se elige en orden: Izquierda, Derecha, Abajo y Arriba.

Comienza por la posición inicial, y va avanzando por una rama hasta que llega a la solución o no pueda avanzar más, de esa manera se retrocede y se explorara otra rama. 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, si no una cola. La cola lo que hace es que el primer nodo en entrar es el primero en salir, lo que significa que va recorriendo el árbol de nodos por niveles, esto hace el 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 las posiciones adyacentes en orden, y después las adyacentes a esas, así sucesivamente. Este algoritmo lo que asegura es que se encuentra la solución con el menor número de pasos.

## Dijkstra

Consiste en una lista de los nodos sin explorar todos con el mismo coste, ordenados de menor a mayor según el coste para llegar a él desde el origen. Al igual que amplitud va sacando el primer nodo, que este caso es el de menor coste no recorrido, lo que nos asegura que el camino que estamos siguiendo es el de menor coste. A diferencia de los dos anteriores en este caso no se desechan en un principio los nodos repetidos, ya que en este caso el generado puede tener menor coste que el elegido anteriormente y ser un mejor camino.

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 que ocupe la primera posición, en esa nueva posición se calculan los costes de las adyacentes y se meten en la pila, y se vuelve a elegir el de menor coste y 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 no 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 hipotenusa que generan las dos posiciones. Formula del coste: c=((x1-x2)\*2+(y1-y2)\*2)\*0,5. Al igual que Dijkstra puede 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 más el coste de la heurística de ese nodo, pero la heurística no se acumula.

Comienza por la posición inicial y para cada posible paso calcula el coste (suma del coste de la heurística y de llegar a ese nodo con los costes uniformes) y se meten todos en una lista ordenada de menor a mayor coste, se escoge el de menor coste y se calculan los costes de las posiciones adyacentes

# 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).

Aunque 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.

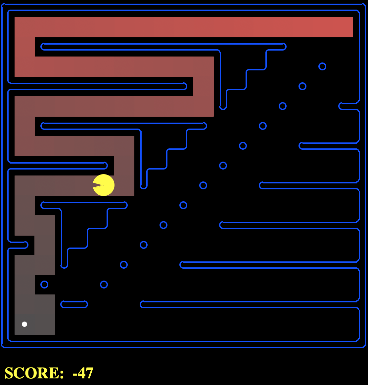
Ejemplo: Si la posición (1,1) es la inicial y se puede desplazar hacia el norte, el estado al que puede llegar se indica con ((1, 2), North, 1); ya que llega al inmediato de arriba, con dirección Norte y como el anterior era inicial el coste del primer será 1.

**Estado inicial y test de meta:** Ambos datos se extraen de la definición del escenario. El estado inicial es donde este el Pacman, en el escenario se marca con un 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 abajo 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 definido en el escenario la posición que debe ocupar Pacman y debe haber solamente una comida.

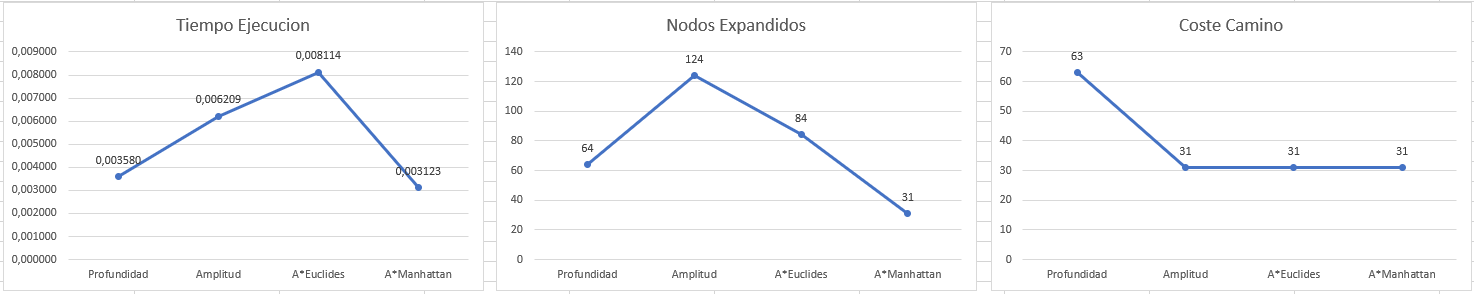
Para definir los sucesores de un estado lo que hace es probar para cada dirección si hay una barrera; y si no la hay suma/resta 1 dependiendo la dirección a la que se desplace, indica la dirección de desplazamiento y suma 1 al coste acumulado del anterior, pero si hay una barrera no meterá en la lista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

## Escenario 1

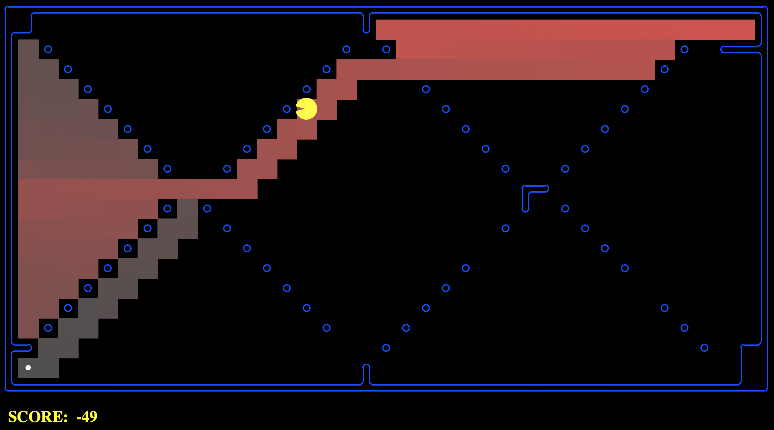


El primer escenario es un tablero de 20x18, en el que hay una bajada diseñada para que A\* con heurística euclídea se quede justamente a un paso ya que no puede avanzar al haber una barrera, por la parte de arriba a la izquierda hay una zigzag para que el algoritmo de profundidad sea más costoso y no llegue con la óptima, en amplitud va avanzando por todos a la vez por lo que será costoso y la parte de abajo a la derecha es una bajada para que A\* con Manhattan sea mas eficiente, en esa misma parte esta 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 grafica del tiempo de ejecución se puede ver que, aunque A\* con Euclides expande menos nodos el algoritmo hace que tarde mas tiempo que el de amplitud.



## Escenario 2



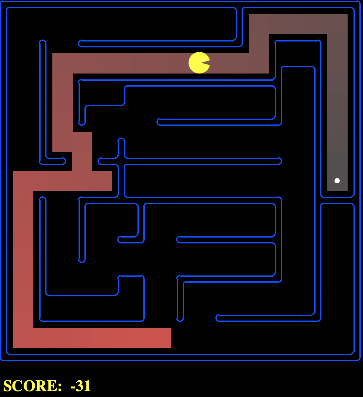
Se trata de un tablero de 40x20 con grandes áreas abiertas, en este caso al haber áreas tan abiertas el algoritmo de Amplitud expande muchos nodos a la vez y lo hace muy costoso, la forma de rombos (sobre todo el izquierdo y primero) hace que las dos heurísticas avancen en línea recta y deban retroceder, y el algoritmo de profundidad, que es el mas eficiente, se benéfica del orden de expansión de los nodos y le permite llegar a una solución más rápido, pero no 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 mas nodos que el de profundidad. La complejidad espacial del algoritmo de profundidad es lineal y el de los otros tres es exponencial. También se observa que amplitud al ser una búsqueda sin información recorre muchos mas nodos, ya que a diferencia no hay unos costes según su posición que le guíen hacia la solución.

Una captura de pantalla de una red social

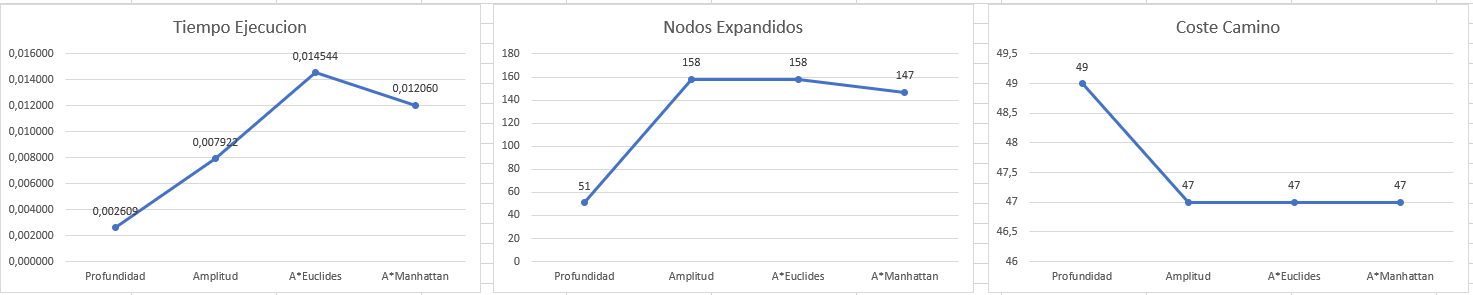
Descripción generada automáticamente

## Escenario 3

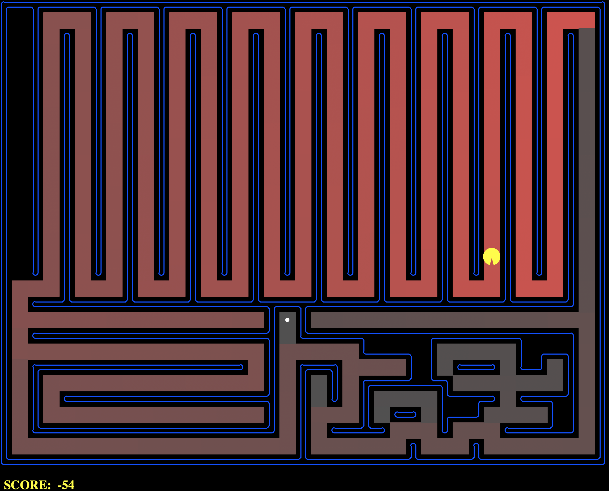


Este escenario de 20x20 se ha diseñado con muchos posibles caminos que parecen acercar a la meta, quedándose muy cerca, pero el camino que llega a la solución es el que de primeras mas se aleja de la meta. Este diseño hace la heurística no sirva de guía inicialmente, ya que solo se mete en ramas sin salida, y 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 mas nodos expanden son los algoritmos que hallan el camino con el coste optimo, pero sus tiempos y nodos en hallarlo son considerablemente superiores a lo que supone la búsqueda en profundidad, el cual solo es un poco mas costoso. En este caso, con la poca diferencia entre el coste de profundidad y el resto y teniendo él cuenta que el resto son bastante mas costoso, el mejor algoritmo para este caso especifico es el de profundidad.

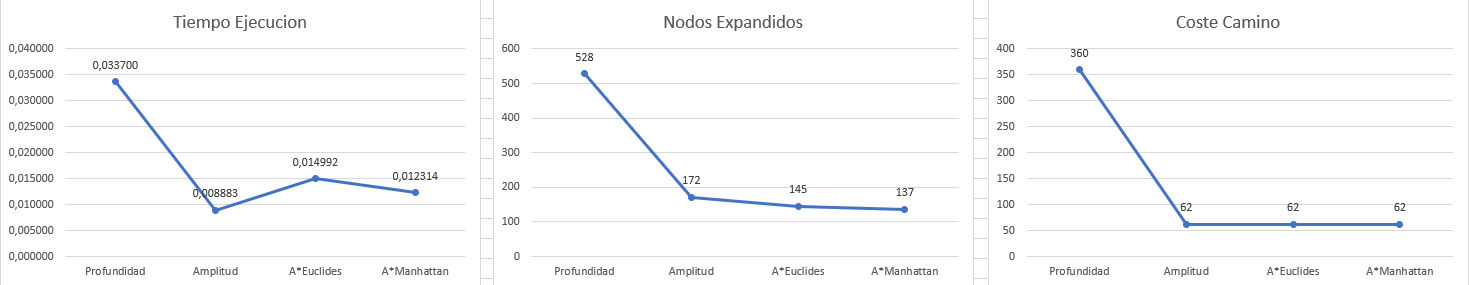


## Escenario 4

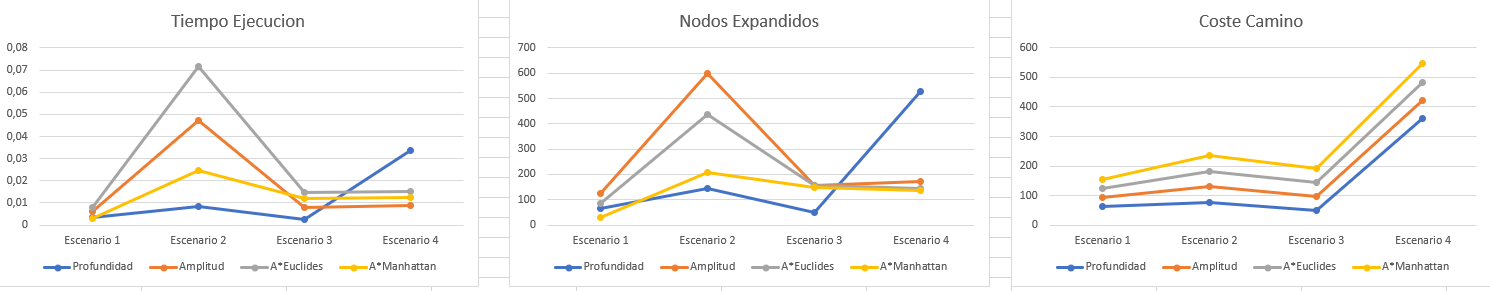


Este escenario de 40x30 trata de mostrar que a pesar de que los apartados anteriores parece que el algoritmo de profundidad es el que primero encuentra la meta, la mayoría de las veces esto no es así. Este diseño se aprovecha que seguirá una rama para meterlo en un largo recorrido que le lleva a la solución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

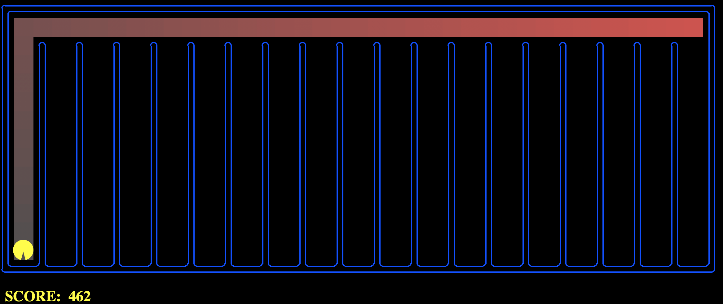
En este caso la solución de profundidad es un mucho mas costosa que la del resto, y el resto han encontrado la misma solución, que es la óptima. Amplitud encuentra la solución óptima a base de recorrer en cada iteración los nodos expandidos en la iteración anterior en orden, lo que hace que llegue, pero 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.



## Grafica de los 4 escenarios en conjunto

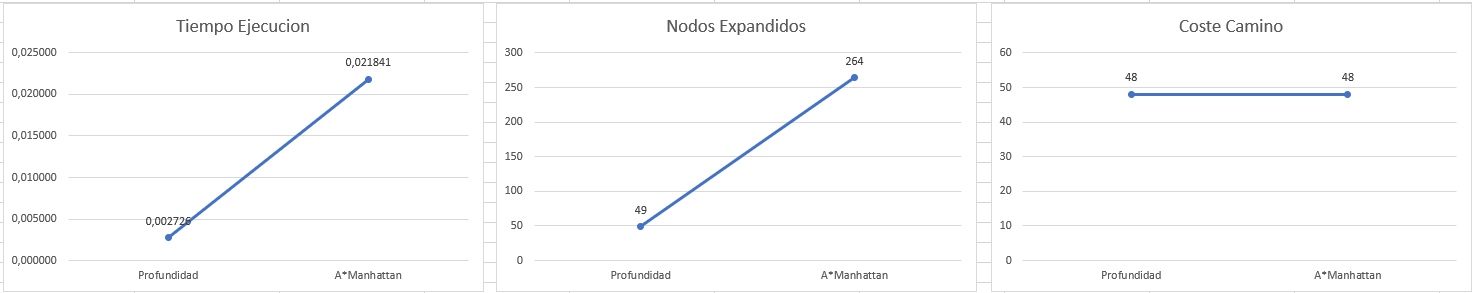


## Escenario adicional 1

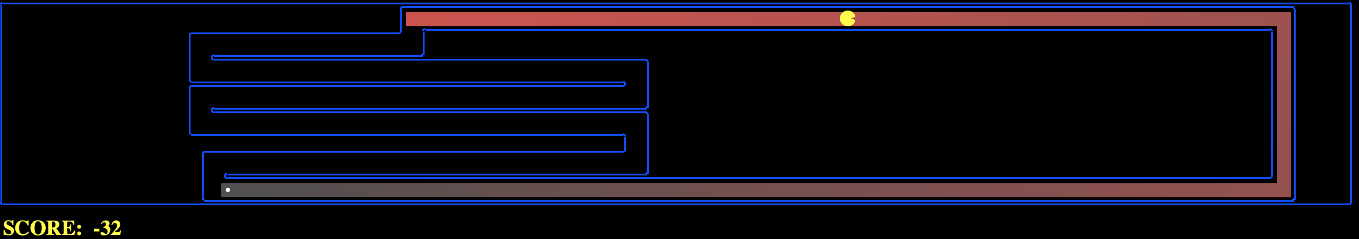


Este escenario de 40x15 está especialmente diseñado para la tarea pedida, que el algoritmo en profundidad halle el camino optimo con menor coste que A\* con heurística de distancia de Manhattan. Estado inicial (39, 14) y final (1, 1)

Este diseño se basa en el hecho de que el algoritmo de profundidad sigue el orden: izquierda, derecha, abajo y arriba. De esta manera, el camino optimo es una L hacia la izquierda y debajo, 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 debajo para crear columnas que bajan y solo conectan por arriba, pero no le acercan horizontalmente a la meta desde la parte más baja.



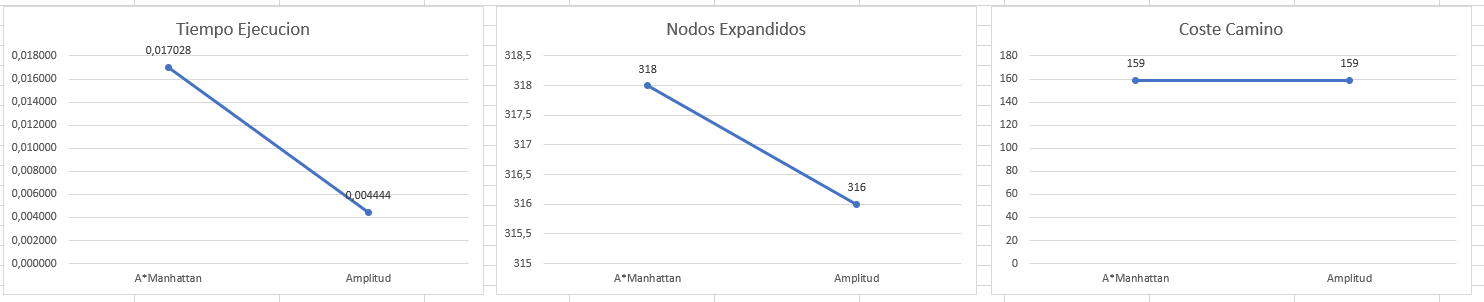
## Escenario adicional 2



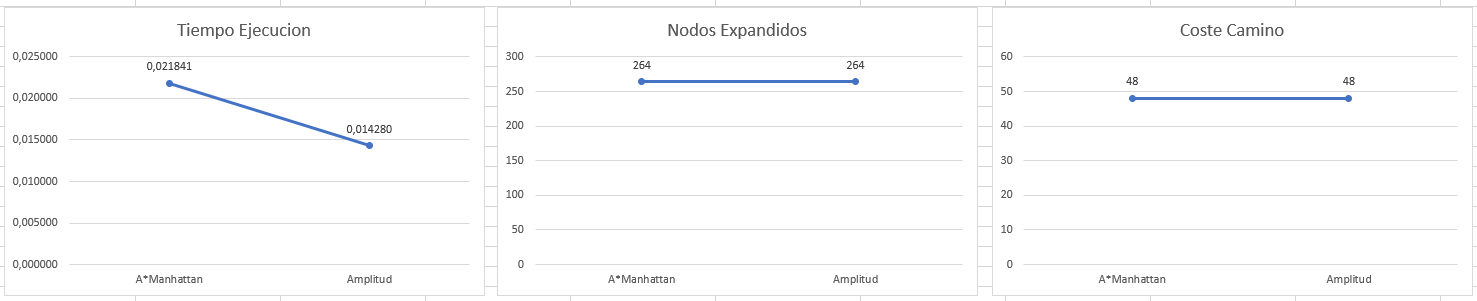
El escenario diseñado intenta que el algoritmo de profundidad expanda menos nodos que A\* con distancia Manhattan, lo intenta creando un zigzag que aparentemente le llevara a la meta más directo que el otro camino, sin embargo, el otro camino se aleja mucho de la meta, pero ese camino es 1 paso más corto que recorrer el zigzag.

Pero debido a que la distancia Manhattan informa al algoritmo A\* no se deja engañar y aunque recorre parte del zigzag no lo hace entero y 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.

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 mas largos que la propia solución.



Lo máximo que podemos lograr es que ambos expandan los mismos nodos, esta grafica 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 con en un menor tiempo, por lo que temporalmente es el mejor de los dos.



# Análisis de FoodSearchProblem

**Espacio de estados:** Los estados son todas las posibles posiciones del escenario que puede ocupar Pacman, eso quiere decir que 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 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 esta 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 un P, donde estará el Pacman al comienzo de juego.

La matriz de booleanos se inicia con el tamaño del escenario y con true en aquella casilla en las que halla un punto en la definición del escenario.

La meta será cuando el Pacman se halla 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 hallado (f=g+f) para cada casilla colindante de Pacman, escogemos la de menor coste (lista ordenada de menor a mayor coste) 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 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 mas 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, escogemos la mayor distancia de todas las halladas.

A continuación, calculamos lo del 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 todos las posibles casilla true de la matriz de comidas.

## Agent 3: ClosestDotSearch

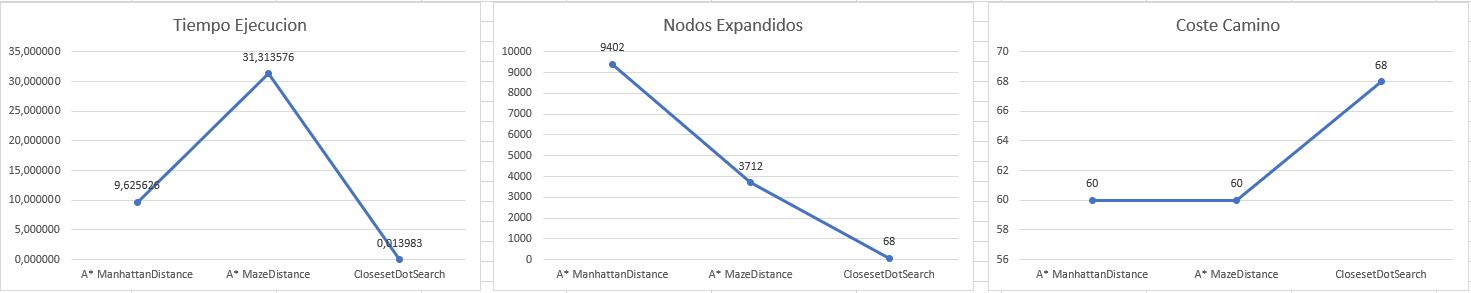
Buscar el mas cercano y va a por él, entonces recalcula el mas cercano de nuevo y va a por él, así sucesivamente hasta que no queda ninguna comida en el tablero.

Explicar el algoritmo y heurística para resolver el problema.

## Comparativa en trickySearch

Grafica de la ejecución de los 3 agentes. Graficas con tiempo de ejecución, numero de nodos expandidos y coste del camino encontrado.

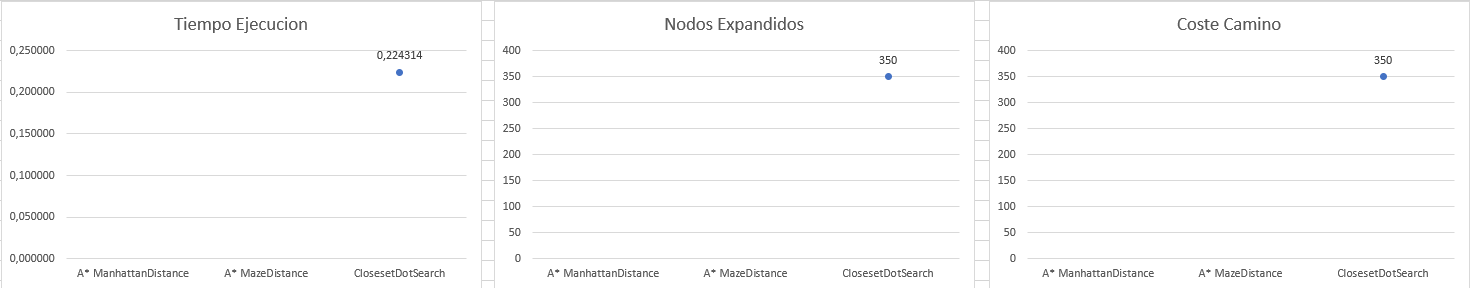
Explicar a qué se debe cada una de las posiciones teniendo en cuenta el algoritmo



## Comparativa en bigSearch

Grafica de la ejecución de los 3 agentes. Graficas con tiempo de ejecución, numero de nodos expandidos y coste del camino encontrado.

Explicar a qué se debe cada una de las posiciones teniendo en cuenta el algoritmo



# 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.

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 temporal, 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 numero 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 mas optimo, cosa con la que los dos anteriores no contaban.

-----------------------------------------------

# 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 sigo 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, llegando así a entender mejor cómo funcionan nuestros propios casos de prueba, al haber podido ver en cada caso el numero de nodos expandidos, tiempo y coste, que son datos imprescindible para entender cada uno de los casos y determinar cuál es el mejor.