PRÁCTICA 1
INTELIGENCIA ARTIFICIAL

21 NOVIEMBRE

AUTOR:
Mario Ventura Burgos 43223476-J
Luis Miguel Vargas Durán 43214929-E

Grado en Ingeniería Informática (GIN 3)
CURSO 2022-2023

ÍNDICE

INTRODUCCIÓN

EXPLICACIÓN DEL PROBLEMA

El enunciado de la práctica plantea una situación que se deberá "resolver" mediante la implementación de unos algoritmos. El escenario que se plantea es el de una rana que se encuentra en un entorno (en este caso, un tablero bidimensional de tamaño 8x8) donde también se encuentra una porción de pizza. La rana deberá poder, mediante su sentido del olfato, detectar dónde se encuentra la pizza para ir hacia ella con el objetivo de comérsela.

Dado que existen varios caminos que seguir desde la posición de la rana hasta la posición de la pizza, se deberán implementar unos algoritmos que permitan a la rana escoger cuál de estos caminos seguir.

Los algoritmos que se deberán implementar son:

1. Búsqueda en **amplitud.**
2. Algoritmo de búsqueda en **A***.
3. Algoritmo **Genético**.
4. Algoritmo de búsqueda adversial: **Minimax**.

Algunos de estos algoritmos pretenden que la rana pueda llegar cuanto antes (mejor solución) a la pizza, mientras que otros simplemente buscan una solución válida sin contemplar que esta sea la óptima (amplitud vs A*).

Antes de ver los algoritmos que se han implementado, cabe mencionar ciertas cosas que algunos de estos tienen en cuenta tales como el coste de las acciones que puede llevar a cabo la rana.

El agente tiene la posibilidad de hacer 3 acciones:

1. **Movimientos simple:** Se desplaza una casilla en alguna de las 4 direcciones que se permiten (Norte, Sur, Este, Oeste) con coste 1
2. **Salto:** Se desplaza 2 casillas en alguna de las 4 direcciones que se permiten (Norte, Sur, Este, Oeste) con coste 6
3. **Esperar:** La rana no se mueve, espera durante un turno sin cambiar su posición. Coste 0'5

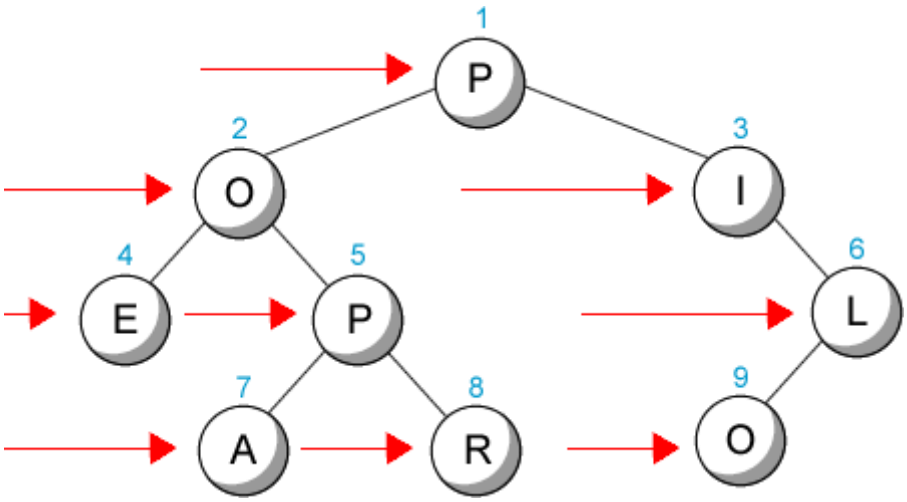
Ahora sí, conociendo la situación y el objetivo de la práctica, veamos que planteamiento se ha seguido para implementar cada uno de estos algoritmos.

SOLUCIÓN PROPUESTA

Los algoritmos implementados son:

ALGORITMO 1: BÚSQUEDA EN AMPLITUD

Este algoritmo pretende encontrar una solución al problema, aunque esta no sea la mejor de ellas. Esto implica que no se tiene en cuenta cuál es el camino con menos coste hacia la pizza. No tiene en cuenta cual es la solución con el número menor de pasos o movimientos, cuál es la que cruza el menor número de casillas, o cualquier otra medida que pudiese usarse para cuantificar cómo de buena es una solución respecto a otra.



Ejemplo de recorrido en amplitud en un árbol

La búsqueda en amplitud consistirá en ir desplegando todos los posibles caminos sin tener en cuenta los costes de las acciones (siendo estos una secuencia de movimientos) hasta que uno de esos lleve a la solución. Una vez esto suceda, se dejará de explorar más caminos y se seguirá este camino descubierto hacia la solución, sin tener en cuenta la posibilidad de que exista un camino mejor sin explorar todavía. Veamos cómo se ha implementado el código que lleva esto a cabo.

En el archivo ***agent.py*** tendremos una clase estado que nos permita definir el estado en el que se encuentra el agente dentro del entorno, y una clase Rana (es decir, el agente) donde se realizará la búsqueda de posibles caminos y se escogerá uno de ellos cuando se encuentre la solución. El funcionamiento del algoritmo que hace uso de todo esto es el siguiente:

En la clase Rana, la función *búsqueda()* se encargará de generar las listas de estados abiertos y cerrados (inicialmente vacías). Posteriormente, una iteración irá generando todos los posibles estados a los que se pueden llegar dado el estado actual y las posibles acciones de la rana en el entorno. Estos serán los estados hijos, y se generarán mediante llamadas a *generar_hijos()*.

También se comprobará si estos estados son válidos (su posición no coincide con la de una pared ni se encuentra fuera del tablero), si estos estados son solución o meta (la posición del agente coincide con la posición de la pizza) y se registrará las acciones que se han necesitado para llegar a estos estados hijo. Una vez se encuentre que uno de los estados hijos es meta, se usará la lista de acciones y estados guardados para saber qué camino debería seguir la rana para llegar a ese estado meta.

La función *actua()* de la clase Rana es la que llamará a búsqueda(). También esta función es en la que se retornará la acción que debe hacer la rana, acompañada de un print que muestre por consola qué acción ha hecho la rana (para que se pueda ver la secuencia de acciones a seguir sin imprimir la lista entera).

De esta forma, se despliega el árbol del juego y se busca una solución en amplitud.

ALGORITMO 2: BÚSQUEDA EN A*

El algoritmo de búsqueda de A* consiste en escoger el mejor de los caminos posibles hacia la solución, haciendo que cada movimiento del agente acerque a este cada vez más a esta solución, que en este caso es llegar a la posición de la pizza sobre el tablero. Así pues, es evidente que presenta una complejidad un tanto más elevada que la búsqueda/recorrido en amplitud, ya que para que podamos saber si con cada movimiento que hacemos estamos más cerca de la solución, deberemos obtener alguna magnitud cuantificable que nos permita medirlo.

En el caso de A*, la toma de decisiones se hace en función de las magnitudes siguientes:

1. A* sí que tiene en cuenta **los costes de las acciones** para ir de un estado a otro
2. A* tiene en cuenta costes de **heurística**

Así pues, surge una cuestión que deberemos resolver. Los costes de las acciones son los costes del desplazamiento de la rana (coste 6 para el salto, coste 1 para movimiento simple, coste 0’5 para esperar), pero ¿cuál es la heurística? ¿cómo sabemos el coste de esta heurística si no sabemos cuál es la heurística?

La respuesta es que tendremos que ser nosotros mismos los que decidamos qué heurística emplear. En este caso, la heurística que se usará es la distancia entre la rana y la pizza (la solución).

Teniendo en cuenta que el tablero sigue una estructura matricial donde hay dos dimensiones, para calcular la heurística se deberá tener en cuenta la fila y columna de rana y pizza.

El cálculo de la heurística se hará de la siguiente manera:

(filaAgente – filaPizza) + (columnaAgente - columnaPizza)

El algoritmo funciona de forma que, al igual que en un recorrido en amplitud, una función *actua()* llama a otra función *búsqueda()*. La diferencia reside ahora en que para tener en cuenta los costes de movimiento y los costes de heurística, se ha añadido un atributo de tipo entero llamado peso a la clase estado, que representa el coste de los movimientos; y además se ha creado una función que calcula la heurística de cada uno de estos estados.

Ahora cuando se llame a *generar_hijos()* y se cree un nuevo objeto estado, se le pasará como parámetro peso su peso actual sumado al coste de la acción (*self.__peso + coste*).
Además, en la función de búsqueda, se tendrá en cuenta la heurística mediante las llamadas a la función *calcular_heuristica().*

Por último, para asegurarnos de que se escoge el camino óptimo se hace que la lista de estados abiertos sea una cola de prioridad **Priority Queue** (en lugar de un simple array, que es como se hizo en el recorrido en amplitud) cuya prioridad sea la menor f(n).
Si definimos f(n) como la suma de los costes + la heurística para cualquier estado n, el resultado que obtenemos es que en la cola de abiertos se encontrarán ordenados los estados y esto nos permitirá obtener el camino mínimo hacia la solución.

Priority queue sort function = *f(n)*

f(n) = *g(n)*+*h(n)* is the estimate of total cost to goal

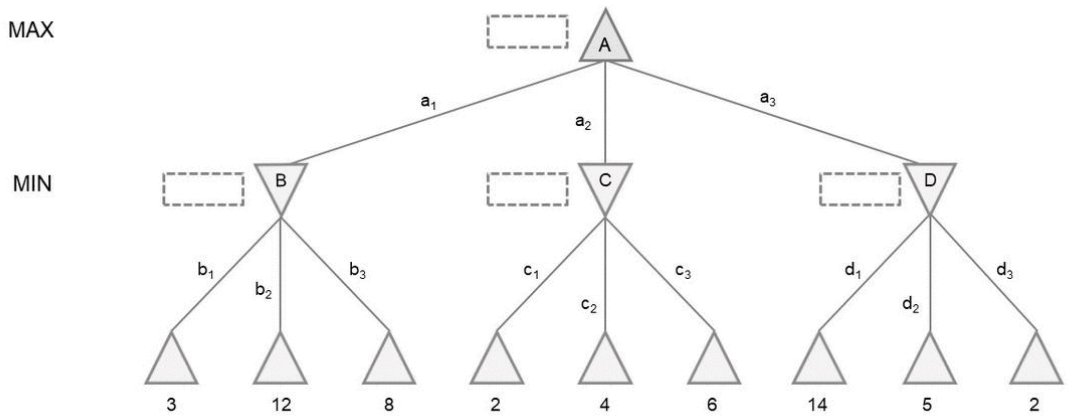
- *g(n)* is the known path cost so far to node *n*
- *h(n)* is the estimate of (optimal) cost to goal from node *n*
- *Priority = minimum f(n)*

ALGORITMO 3: BÚSQUEDA ADVERSIAL MINIMAX

El algoritmo minimax es un algoritmo que tiene el objetivo de minimizar la pérdida máxima esperada en juegos con adversario y con información perfecta (entornos totalmente observables como en nuestro caso). Se trata de un algoritmo que despliega todo el árbol de posibilidades del juego dada una configuración inicial (padre del árbol), y lo recorre en busca de escoger cuál es la mejor jugada posible en el momento dado. Es, por tanto, una técnica que se usa cuando se tiene que un agente tiene un adversario: MAX vs. MIN

De forma más detallada, la idea consiste en comenzar en la posición actual del juego y usar el generador de movimientos legales (desplegar árbol) para generar las posibles posiciones sucesivas hasta un cierto límite de niveles. En el caso de nuestra práctica, por una cuestión de reducir el coste computacional y evitar tiempos de ejecución muy elevados, se establece un límite de profundidad de 4 en el árbol de posibles estados del juego.

A continuación, se aplica una función de evaluación a los últimos estados obtenidos. Esta función es capaz de decidir cuál de estos estados es el mejor y por tanto, qué camino seguir.



Entrado más en términos de código, con este algoritmo ahora tendremos dos agentes en vez de uno, que "jugarán" entre ellos de la forma que se ha explicado anteriormente. El hecho de que haya dos agentes complica la implementación de este código porque los actos de uno interfieren en la percepción del entorno del otro. Cada agente deberá tener en cuenta los movimientos del otro agente para hacer los suyos y elegir el mejor movimiento para él, y que este movimiento al mismo tiempo sea el peor para su contrincante

También tenemos que ser capaces de distinguir a los dos agentes.

Algunos de los cambios y los conceptos que hay que tener en cuenta para el código son:

- A parte del main que se usa de forma habitual para ejecutar el resto de algoritmos, ahora habrá un nuevo main con ciertos cambios que permitirán la aparición de dos agentes en el tablero. Esto se hará modificando el primer parámetro de la instrucción:

  lab = joc.Laberint([ranaM, ranaL], paredes=True)

  Esto se debe a que este primer parámetro (mirar subrayado) es una lista de los agentes que van a intervenir en el juego. Ahora deberemos crear otro objeto rana al que llamamos ranaL, y lo añadimos a la lista mencionada. De esta forma, "metemos" dos agentes en el juego

- Por otro lado, en la clase Estado se deberá añadir un atributo nombre. Esto sirve para distinguir una rana de otra ya que, debido a que ahora el entorno es multi-agente, el estado en el que se encuentra el tablero depende de la posición de todos los agentes que se encuentran en él. Deberemos por tanto ser capaces de distinguir un agente de otro. Ahí es donde los nombres entran en juego, actuando como claves a las cuales el juego debe poder acceder con el objetivo de distinguir a todos los agentes.

- Por último, cabe mencionar, antes de pasar a una visión y explicación más general de cómo este algoritmo funciona, que la función minimax() de este algoritmo, es la única que se ha implementado con recursividad.

El funcionamiento del algoritmo es el siguiente:

Desde la función actua(), se crean objetos estado mediante ClauPercepcio. Posteriormente, se llama a la función minimax() pasándole como parámetro este estado actual detectado para un agente dado, un límite de profundidad, y un booleano que indique si es el turno de Max o de Min. Esta función será la que se encarga de buscar el mejor movimiento para el agente que la llame en el estado que la llame, siguiendo el planteamiento explicado previamente.

Después de que se realice un movimiento, se comprueba para todos los agentes interviniendo en el juego si este ha provocado que alguno de estos agentes acabe en un estado meta. De ser así este agente será el ganador. Se pondrá a True un booleano HAY_SOLUCIÓN que sirve precisamente para indicar que el juego ha llegado a su fin.

Cada agente hará su respectiva mejor acción en cada momento; es decir, en cada estado en que se encuentre. Todo esto se consigue mediante llamadas a la función minimax() desde actua().

En minimax(), se ha aprovechado el pseudocódigo facilitado por el profesor para la implementación.

Minimax() presenta una recursividad cuyo caso trivial es que el estado que se visite tenga una score, cosa que indicaría que hemos llegado al nodo hoja del árbol y por tanto podemos darle este valor al padre para seguir con el algoritmo. Sin embargo, es importante pensar en el hecho de que el gran abanico de posibilidades de desarrollo del juego haría que este árbol fuese infinitamente grande (metafóricamente hablando).

Por ello, se establece un límite de profundidad que se pueda alcanzar en el recorrido en profundidad del árbol. En nuestro caso, este límite será 4, y la comprobación del nivel actual del árbol pasará a ser también un caso trivial de la recursividad ya que, si este caso se da, se evaluará el estado de igual manera y se devolverá su score como si fuese un nodo hoja.

Mediante llamadas a las funciones max() y su homóloga min(), se escogerá el máximo o el mínimo score de los hijos de cada nodo respectivamente, y de esta forma se seguirá la lógica de un algoritmo minimax.

Todas estas funciones actúan como engranajes de una mayor máquina cuyo propósito general es encontrar una solución para cada agente, y que ambos tengan una disputa por llegar a ella (2 ranas peleando por 1 único trozo de comida: situación adversial).

ALGORITMO 4: ALGORITMO GENÉTICO

El funcionamiento de este algoritmo se basa en un cruce genético real entre especies. La implementación es un tanto compleja, por tanto, para simplificar la explicación y no extenderla en exceso, (todas las funciones implementadas en el código ya van acompañadas de comentarios explicativos) se explicará únicamente el funcionamiento general y las funciones principales que hacen que este algoritmo funcione y que papel desempeñan en la totalidad del programa.

En primer lugar, es importante destacar que existen 3 clases:

1. **Clase Individuo** → Individuo de una población. Tiene su propio fitness y una lista de información genética
2. **Clase Genética** → Clase que contiene toda la información y operaciones que se pueden hacer sobre la información genética de una población o de un individuo. Estas son obtener fitness, evaluar si una posición es válida, realizar cruces genéticos, mutaciones, entre otras tantas
3. **Clase Rana** → Clase rana con sus funciones típicas. Sobre todo destacan *actua()* y *búsqueda()*

Para empezar, empezaremos por entender que un **Individuo** tiene una lista de información genética y un fitness individual. El fitness se declara inicialmente como -1 por norma, pero en realidad este valor nunca se llegará a usar. Cuando posteriormente se escojan los individuos más válidos de una población, este valor de fitness será el que determine la validez del individuo. Cuanto menor sea el valor del fitness, mayor probabilidad tendrá de sobrevivir y ser escogido para una nueva generación en la población nueva (se establece 1000 como máximo y peor fitness, y 0 como el mínimo y mejor fitness).

Respecto a la **clase Genética**, esta contiene funciones *setters* y *getters* del fitness de la población, funciones de evaluación de la posición del agente sobre el tablero (comprobar si es válido y si es meta). Sin embargo, estas son funciones bastante simples que podrían ser encontradas en cualquier otro algoritmo.
Dado que en esta clase hay un total de 11 funciones sin contar el ***__init__,*** se explicarán solamente solo algunas de ellas que sea necesario entender para comprender el funcionamiento del algoritmo

Funciones destacables de la clase Genética:

1. **Función fitness()** → Esta función recibe como parámetros un objeto individuo y tiene un valor de retorno void (es decir, no retorna nada en específico). Esta función se dedica a establecer el fitness del individuo recibido por parámetro.
   Inicialmente este individuo ya tiene un fitness, pero este va a cambiar en función de la acción que desempeñe el agente, la dirección en que lo haga, y el coste de esta acción. La función fitness será capaz de modificar el fitness del individuo recibido por parámetro en relación con la acción que este haga, siendo este el desplazamiento de fila + desplazamiento de columna + coste total acumulado de las acciones.

2. **Función calcular_fitness_poblacion()** → Esta función realiza el cálculo del fitness medio de la población. Esto es de vital importancia para posteriormente escoger qué individuos son candidatos sobrevivir a la selección de una nueva población, aunque esto ya se explicará en su momento.
   El cálculo se hace recorriendo toda la lista de individuos de la población, obteniendo el fitness de cada uno de ellos mediante un getter, y sumándolo a un contador. Hecho esto, se hará una media aritmética (fitness total acumulado / número de individuos de la población) y este será el valor que se use para establecer el fitness de la población.
   El retorno de esta función es void

3. **Función seleccionar_individuos()** → Esta función se encarga de escoger qué individuos serán seleccionados para la nueva población. Para ello es necesario saber cuál es el fitness de la población, cosa de la cual se encarga la función explicada previamente.
La selección se hace recorriendo toda la lista de individuos de la población actual, cogiendo el fitness de este mediante un getter, y comparándolo con el fitness de la población.

   Como dijimos anteriormente, cuanto menor sea el fitness, más probabilidades tendrá este individuo de ser escogido. El fitness de todos los individuos será comparado con el de la población, donde se estudiarán dos posibilidades:

   - **Fitness del individuo <= fitness población:** Se escoge el individuo ya que el fitness es bueno, por tanto, se añadirá a una lista llamada candidatos, que contendrá los candidatos a pertenecer a la nueva población
   - **Fitness del individuo > fitness población:** El fitness no es bueno, por tanto, este individuo no será escogido y se añadirá a una lista de individuos descartados.

   Si durante este recorrido se encuentra algún individuo con fitness 0, se establece a este como true y se usa un booleano para indicar que se ha encontrado una solución. Si esto sucediera no haría falta seguir con el recorrido ya que ya se ha encontrado una solución y no es necesario seguir generando poblaciones y cruzando individuos.

   Si no se ha encontrado ninguna solución deberemos continuar con el transcurso de la función. Ahora tenemos una selección guardada en candidatos[], pero debemos tener en cuenta que no es del mismo tamaño que la población anterior, ya que de esta hemos escogido solo a los mejores individuos.
   Lo que haremos a continuación será hacer cruces genéticos entre individuos ya seleccionados como candidatos, y posteriormente, mutarlos para que sean todavía mas diferentes. Esto generará nuevos individuos que se añadirán a la nueva población hasta que esta tenga el mismo tamaño que la anterior.
   En un futuro, sobre esta nueva población se aplicará el mismo proceso hasta encontrar un individuo con fitness con valor 0, que se escogerá como solución.

   Los cruces genéticos se hacen escogiendo dos individuos aleatorios (padre y madre), obteniendo sus arrays de información genética, partiéndolos en 2 partes y reensamblándolo estos arrays con las mitades generadas de uno y otro. De esta forma esta información se modifica y mezcla, pero sigue siendo parecida a la de los individuos padre.

Por último, las mutaciones se harán mediante ciertas comparaciones que harán llamadas a la función **mutar_individuo()**

4. **Función mutar_individuo()** → Esta función usa una lista con todas las posibles acciones del agente y todas las posibles direcciones en las que estas acciones se pueden hacer para, de forma aleatoria, cambiar los genes de la lista de genes del individuo.

Estas funciones se llaman desde otras funciones, operando conjuntamente desde la clase Rana u otras para conseguir el objetivo de alcanzar la pizza.

En la **clase Rana** (el agente) existen dos funciones cuya implementación deberemos entender. Estas usan las otras funcionalidades de las otras clases ya explicadas para conseguir llegar a una solución.

1. **Función búsqueda()** → Busca una solución dada la con figuración inicial del tablero. Cuando se llama a esta función se genera una población con su respectivo fitness y se selecciona qué individuos sobrevivirán. Se hacen cruces genéticos y mutaciones hasta tener una nueva población del mismo tamaño que la anterior (hay un atributo general en todo el programa que establece el tamaño de la población con la que se tratará). Hecho esto, la función genera una lista de acciones que el agente deberá seguir para llegar a la solución encontrada en la llamada a seleccionar_individuos() y se establece que el atributo acciones de la clase Rana sea esta lista obtenida.

2. **Función actua()** → La función actua() es la que le dice al agente/rana que actúe. La función comienza detectando la posición de la pizza, la posición de las paredes y la del agente mediante una ClauPercepcio. Hecho esto, genera un estado llamado estado_actual, usando estas percepciones.
   Si la lista de acciones está vacía (es decir, no se ha buscado una solución todavía), se llama a la función búsqueda(), que devuelve una solución y la lista de acciones que debe hacer la rana.
   Una vez se haya hecho esto, desde otra parte del programa, mediante varias llamadas a actua(), se irán sacando acciones de esta lista y ejecutándolas hasta que la lista se vacíe y, como consecuencia, se habrá llegado a la solución.

De esta forma, mediante varias llamadas a actua(), que llama a búsqueda(), que a su vez usa las operaciones genéticas definidas en la clase Genética, se implementa un algoritmo genético

que es capaz de encontrar una solución mediante la creación de nuevos individuos fruto de cruces genéticos, mutaciones y evaluaciones de fitness de individuos y poblaciones.

El hecho de usar cruces genéticos y mutaciones aleatorias hace que la búsqueda hacia la solución no sea óptima, y que el tiempo de ejecución sea totalmente aleatorio ya que las mutaciones y cruces también lo son. Esto hace que la generación de poblaciones de individuos sea impredecible y por tanto el camino hacia la solución también lo sea.

Todo esto hace que se simule un cruce genético real entre especies.

COMPARATIVA DE RENDIMIENTOS

Para comparar como de buenas son las soluciones que cada uno de estos algoritmos ofrece, se pueden medir ciertas magnitudes y compararlas entre ellas. De esta forma veríamos como un algoritmo que busca un camino óptimo como A* dista en cuanto a resultados respecto a otro algoritmo como amplitud, donde el coste de llegar a la solución no se tiene en cuenta.

Sin embargo, debido a que la posición inicial de la rana y la posición en la que se encuentra la pizza son totalmente aleatorias, es absurdo medir esto con únicamente una ejecución.

Para poder tener un análisis realista que sirva para hacer una comparación, se estudiarán ciertas magnitudes mediante 10 ejecuciones por algoritmo. Cada uno de los algoritmos se ejecutará 10 veces y los datos se recopilarán para hacer una comparativa, de forma que la aleatoriedad de las posiciones de agente y solución no será un factor que influya en la medida.

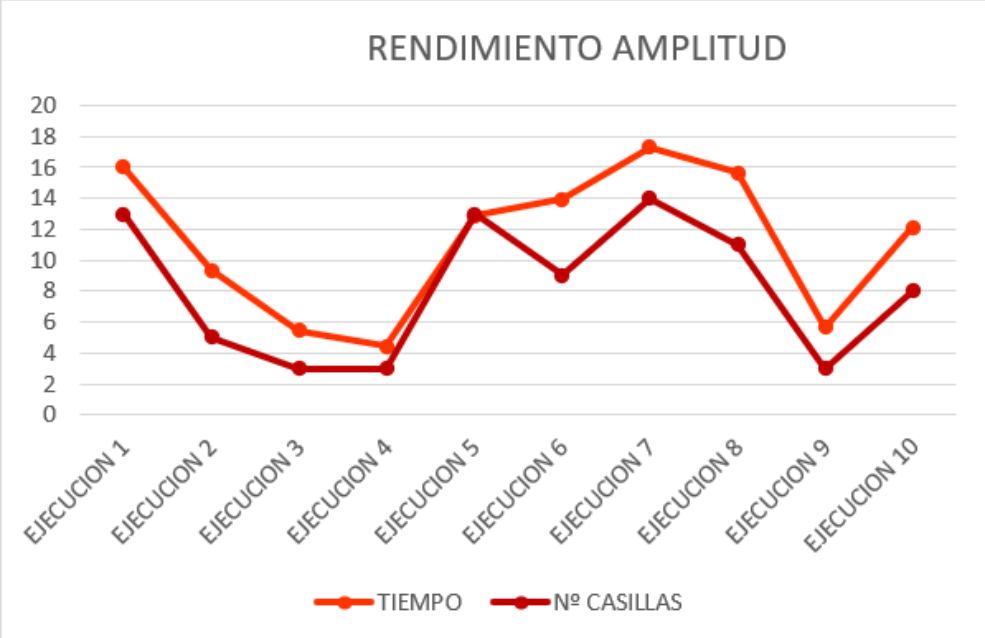
Se podrían usar varias magnitudes para medir y comparar los algoritmos (tiempo de ejecución, casillas recorridas, número de movimientos, etc…).

En nuestro caso, usaremos las dos primeras magnitudes mencionadas.

RENDIMIENTO DE UNA BÚSQUEDA EN AMPLITUD

Las ejecuciones para recorrido en amplitud nos han dado los siguientes resultados respecto al tiempo de ejecución y casillas recorridas:

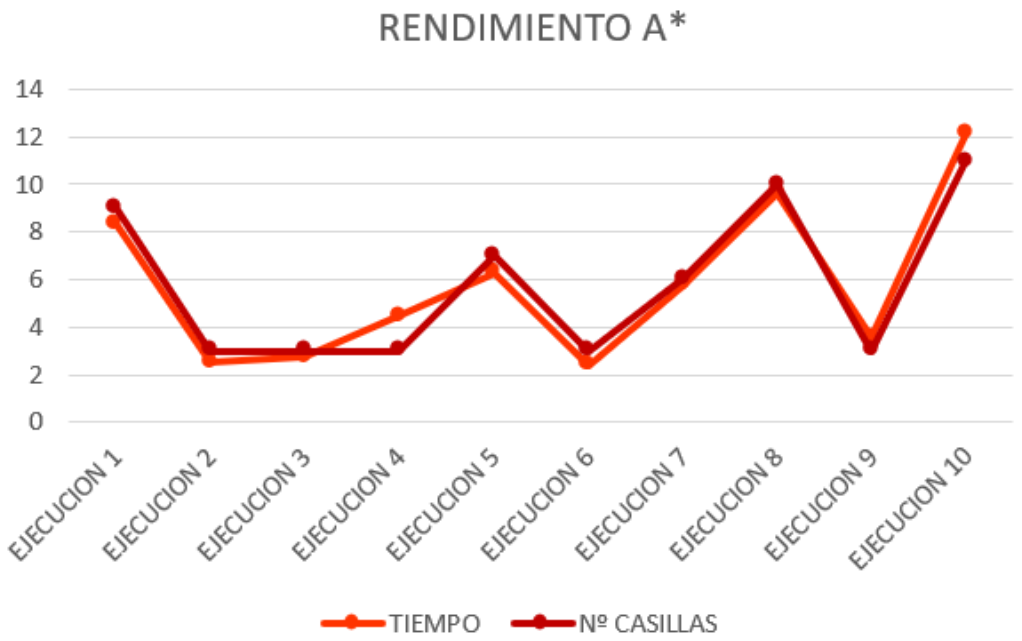
AMPLITUD	EJECUCIONES	TIEMPO(s)	Nº CASILLAS
	EJECUCION 1	16,01	13
	EJECUCION 2	9,32	5
	EJECUCION 3	5,44	3
	EJECUCION 4	4,45	3
	EJECUCION 5	12,85	13
	EJECUCION 6	13,92	9
	EJECUCION 7	17,31	14
	EJECUCION 8	15,63	11
	EJECUCION 9	5,65	3
	EJECUCION 10	12,12	8
	MEDIA	**11,27**	**8,2**



RENDIMIENTO DE A*

En este caso, de A*, tras 10 ejecuciones hemos obtenido lo siguiente:

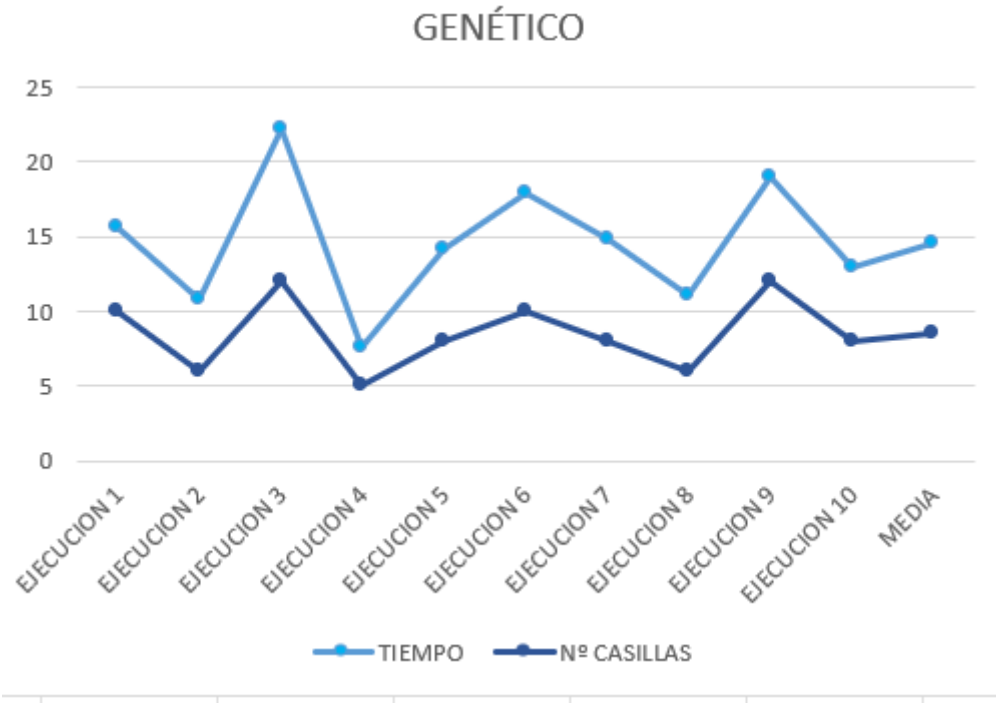
A*	EJECUCIONES	TIEMPO (s)	Nº CASILLAS
	EJECUCION 1	8,35	9
	EJECUCION 2	2,49	3
	EJECUCION 3	2,76	3
	EJECUCION 4	4,45	3
	EJECUCION 5	6,26	7
	EJECUCION 6	2,4	3
	EJECUCION 7	5,71	6
	EJECUCION 8	9,59	10
	EJECUCION 9	3,55	3
	EJECUCION 10	12,12	11
	MEDIA	**5,768**	**5,8**



RENDIMIENTO DEL ALGORITMO GENÉTICO

Los resultados obtenidos en cuanto a tiempo de ejecución y casillas recorridas, en el caso de minimax es el siguiente:

GENÉTICO	EJECUCIONES	TIEMPO (s)	Nº CASILLAS
	EJECUCION 1	15,62	10
	EJECUCION 2	10,74	6
	EJECUCION 3	22,22	12
	EJECUCION 4	7,65	5
	EJECUCION 5	14,15	8
	EJECUCION 6	17,9	10
	EJECUCION 7	14,82	8
	EJECUCION 8	11,02	6
	EJECUCION 9	18,98	12
	EJECUCION 10	12,96	8
	MEDIA	**14,606**	**8,5**

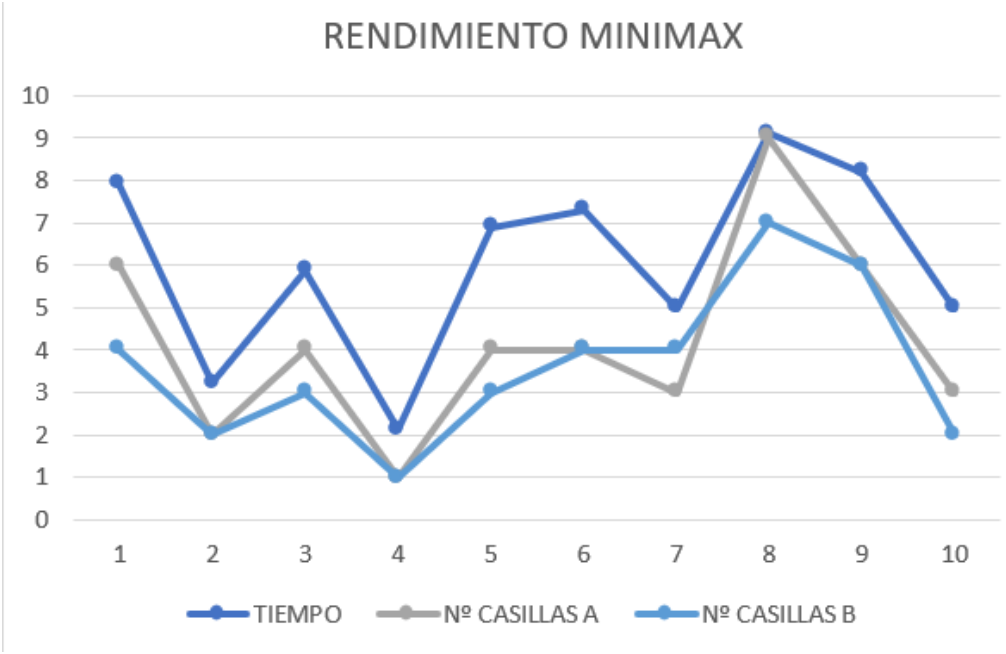


RENDIMIENTO DE MINIMAX

Los resultados obtenidos tras las ejecuciones (el tiempo es el mismo para ambos pero el recorrido de casillas cambia para cada uno de los agentes), en el caso de minimax es el siguiente:

MINIMAX	EJECUCIONES	TIEMPO	Nº CASILLAS A	Nº CASILLAS B
	EJECUCION 1	7,92	6	4
	EJECUCION 2	3,21	2	2
	EJECUCION 3	5,88	4	3
	EJECUCION 4	2,11	1	1
	EJECUCION 5	6,9	4	3
	EJECUCION 6	7,31	4	4
	EJECUCION 7	4,99	3	4
	EJECUCION 8	9,11	9	7
	EJECUCION 9	8,2	6	6
	EJECUCION 10	4,96	3	2
	MEDIA	**6,059**	**4,2**	**3,6**

(el color verde indica el ganador)



CONCLUSIONES

Respecto al rendimiento de los algoritmos, se pueden extraer varias conclusiones. Estas son algunas de ellas:

1. El hecho de buscar una implementación más compleja que permita obtener el camino óptimo hacia una solución hace que posteriormente, esta búsqueda tenga una reducción coste en cuanto a tiempo bastante significativa. Esto se puede ver comparando los resultados obtenidos con amplitud y A*. Basta con comparar las medias aritméticas obtenidas.

2. En algoritmos donde se da prioridad a una búsqueda de la solución óptima, teniendo en cuenta los costes de las acciones (A*), se puede ver en el gráfico como la relación entre casillas recorridas y tiempo de ejecución en segundos, prácticamente no se separa en ningún momento. Se podría decir que van directamente relacionados y que el aumento de uno es consecuencia o causa del aumento de otro, cosa que podría ayudarnos de cara a hipotéticas pruebas reales con datos esperados.

3. Dado que en el algoritmo genético se presenta cierto factor de aleatoriedad, las ejecuciones de este muestran una total imprevisibilidad de los datos que se esperarían obtener. La aleatoriedad proviene de la generación de poblaciones y del cruce y mutación genética, y esto provoca que el tiempo que se tarde en encontrar una solución dependa (dentro de unos baremos o límites) sea realmente impredecible. Si a esto se suma el hecho de que la posición de la rana y de la pizza ya son aleatorias por concepto, esto provoca la gran variación de tiempos y casillas recorridas que se pueden ver tanto en el gráfico como en la tabla.

4. Respecto a minimax, usar recursividad y establecer límites de profundidad ayuda a encontrar una solución ciertamente rápida sin reparar en costes demasiado elevados. De no ser así, si no se pusieran límites de profundidad o no se usase un planteamiento recursivo, el coste del algoritmo sería muchísimo mayor. Se puede ver cierta coherencia entre los datos de la tabla.