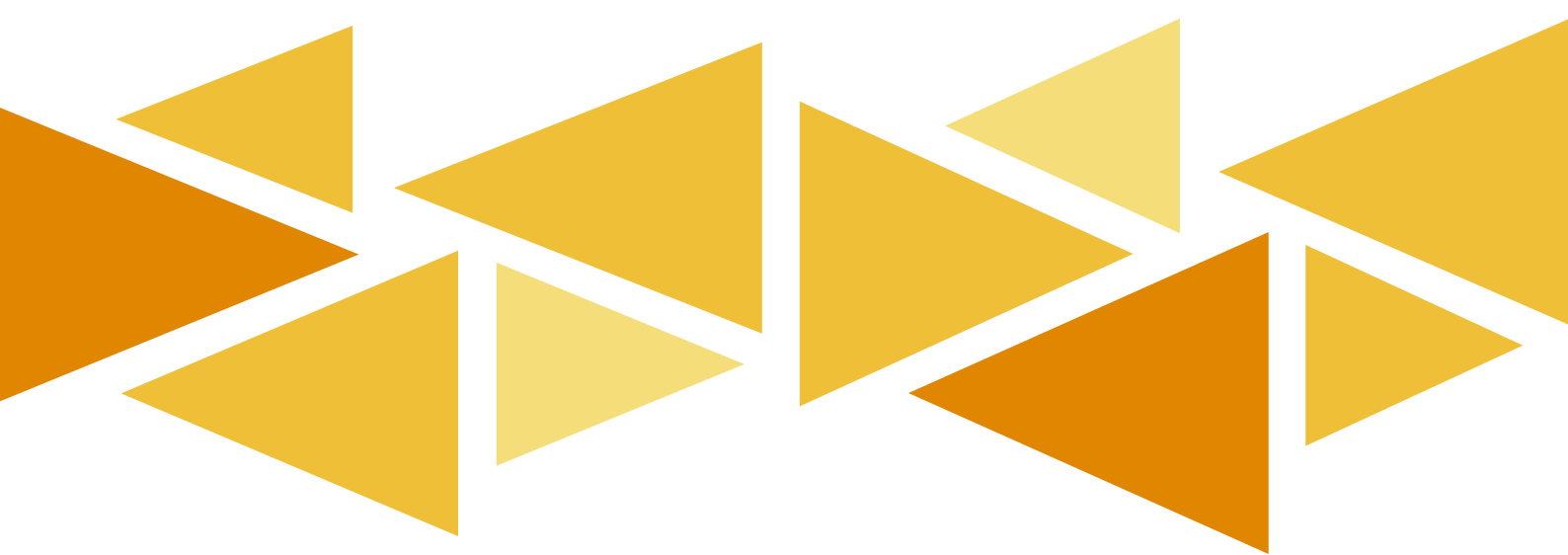


Realizado por:

- Beatriz Espinar Aragón
- Steven Mallqui Aguilar
- Rogger Huayllasco De la Cruz

ALGORITMO A*

PRÁCTICA 1



CONTENIDO

1. Introducción.....	1
2. Entorno.....	2
3. Implementación.....	3
3.1. Clase Main.....	3
3.2. Clase Button	4
3.3. Clase Node.....	4
3.4. Clase Graph	5
3.5. Clase AStar	5
3.6. Clase IndexPQ.....	8
3.7. Clase Utilities.....	9
4. Ampliaciones	11
4.1. Waypoints.....	11
4.2. Casillas peligrosas.....	13
5. Ejemplos	14

1. Introducción

La finalidad de este documento es presentar un informe detallado sobre la práctica, consistente en implementar una versión reducida del algoritmo A*. Se explica el proceso y las decisiones tomadas para lograr una aplicación que permita visualizar el funcionamiento del algoritmo.

A* es un algoritmo de búsqueda inteligente que busca el camino más corto desde un nodo inicial hasta un nodo meta a través de un espacio determinado, y utilizando una heurística óptima. El espacio de navegación se plantea como un tablero de dimensión $rows \times cols$, donde las casillas son los nodos. Ahora bien, no todas las casillas son accesibles, pues algunas serán marcadas por el usuario como inalcanzables. Además, nótese que esta versión del algoritmo permite desplazarse en horizontal, vertical y diagonal.

Por otro lado, es relevante destacar la función de evaluación que se emplea en esta práctica, pues es la que utiliza el algoritmo para establecer el orden en que se irán considerando los nodos. Ya se ha visto que los nodos representan las casillas o celdas del tablero, y decimos que dos nodos son vecinos si están en casillas adyacentes del tablero (diagonal, horizontal o verticalmente). De este modo, podemos representar el conjunto de nodo y relaciones de adyacencia (aristas) mediante un grafo.

Así pues, sea G el grafo que representa el tablero, a el nodo inicial, b el nodo meta, y n un nodo cualquiera, $n \in G$. Entonces la función de evaluación para ese nodo es $f(n) = g(n) + h(n)$, donde

- $g(n)$ = coste (distancia) real de llegar desde a hasta n por el camino más corto que calcula el algoritmo.
- $h(n)$ = coste (distancia) estimado de llegar desde n hasta b . Para estimar esta distancia se utilizará la distancia euclídea, calculada así para dos puntos p_1, p_2 con coordenadas $(x_1, y_1), (x_2, y_2)$ cualesquiera:

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

El coste estimado podría no ser totalmente correcto, dado que pueden existir obstáculos que impidan llegar en línea recta a un nodo. Sin embargo, está demostrado que el algoritmo nunca sobreestima la distancia actual y, por tanto, se mantiene la propiedad de admisibilidad. De esta manera, una vez calculados estos parámetros, se irán escogiendo los nodos en orden creciente según su función de evaluación. Esto es, se considerarán más prioritarias aquellas casillas por las que podamos llegar antes al nodo meta.

Ahora que ya se ha planteado el objetivo del algoritmo, se presentan las secciones que componen este documento para facilitar su lectura:

- [Entorno](#). Se expone brevemente el entorno utilizado para la implementación del algoritmo, así como el lenguaje de programación elegido y otras herramientas de interés.
- [Implementación](#). Se explica con mayor detalle las decisiones relativas al diseño e implementación, tales como las estructuras de datos empleadas o el funcionamiento específico del algoritmo.
- [Ampliaciones](#). Se presentan las ampliaciones realizadas sobre el algoritmo básico, incluyendo la lógica implementada y los cambios aplicados sobre el apartado anterior.
- [Ejemplos](#). Se proponen al usuario una serie de ejemplos ilustrativos para mostrar el funcionamiento del algoritmo, incluyendo ejemplos particulares de las ampliaciones.

2. Entorno

Se explica en este primer apartado el entorno de programación empleado para el desarrollo del algoritmo, con las herramientas y librerías utilizadas.

En primer lugar, el lenguaje de programación escogido para implementar el algoritmo ha sido Python. En particular, la versión más reciente de este lenguaje: 3.11.2. Esto se debe, por un lado, al amplio rango de librerías que incluye Python y que facilitan considerablemente el desarrollo de la interfaz. Además, se trata de un lenguaje que era desconocido para los miembros del equipo, y que tenían especial interés por aprender. Así, la implementación de este algoritmo ha supuesto una oportunidad de profundizar en los conceptos de la asignatura así como de familiarizarse con un nuevo lenguaje de programación muy comúnmente utilizado por grandes empresas.

Para el desarrollo del código se ha optado por utilizar PyCharm, un entorno de desarrollo integrado especializado en Python, con el que resulta muy sencillo encontrar errores en el código y corregirlos rápidamente. Asimismo, el entorno muestra advertencias acerca de la sintaxis y el formato del código, que ayuda al usuario a mantener buenas prácticas de programación.

Por otro lado, en lo relativo a las librerías utilizadas, se listan a continuación las más relevantes para este proyecto:

- `Pygame` → Se trata de una librería orientada al desarrollo de videojuegos en 2D con Python. Funciona con cualquier sistema (MacOS, Windows o Linux), y facilita la creación de la estructura básica de un videojuego con programación orientada a objetos. En esta práctica se ha utilizado principalmente para el desarrollo de la interfaz, permitiendo no sólo mostrar los botones y capturar los eventos, sino dibujar todo el tablero y colorear los nodos conforme se va ejecutando el algoritmo.
- `Math` → Se trata de un módulo que proporciona acceso a las funciones matemáticas definidas en el estándar de C. En esta práctica se ha utilizado en la clase del algoritmo (véase [Clase AStar](#)) para implementar las funciones $g(n)$ y $h(n)$, puesto que era necesario usar potencias y raíces cuadradas.
- `PriorityQueue` → Pertenece al módulo `queue`, y permite la implementación de colas de prioridad. En esta práctica se ha utilizado en la clase del algoritmo (véase [Clase AStar](#)) porque se utiliza en una de las ampliaciones (véase [Waypoints](#)) para tratar los waypoints que quedan por visitar.
- Otras: `sys` (para cerrar la aplicación) y `tkinter` (para mensajes pop-up).

Por último, el equipo ha hecho uso de otras tecnologías que han permitido desarrollar la práctica, entre las cuales cabe destacar: GitHub y GitHub Desktop, para el control de versiones y facilitar el trabajo en equipo mediante la creación de un repositorio común; Discord o Whatsapp, como plataformas de comunicación para trabajar simultáneamente sobre el código; Word (Office 365) con OneDrive, para la elaboración de la memoria; y Canva y Paint, para el diseño de los botones y la interfaz gráfica.

3. Implementación

En esta sección se profundiza en la implementación del algoritmo A*, explicando los módulos en los que se ha dividido el proyecto y las estructuras de datos utilizadas. Además, se explican los pasos que sigue el algoritmo y cómo ha sido plasmado en la interfaz para que el usuario visualice todo el proceso.

El proyecto se ha dividido en 6 módulos principales, que se explicarán a lo largo de este apartado. En «Main» se explicará el funcionamiento general de la aplicación, que se divide en dos pantallas principales. Para la interfaz se utiliza la siguiente clase, «Button», y posteriormente se presentarán las clases relativas a la lógica del algoritmo. El problema queda representado con la estructura de datos «Graph», que a su vez está formado por «Node»'s. Además, para la implementación del algoritmo se ha creado la clase «AStar» (en esta sección es donde se explica el flujo principal del algoritmo), que hace uso de una estructura de datos implementada en la clase «IndexPQ», como se verá más adelante. Por último, en «Utilities» se incluyen las constantes empleadas y se describirá la representación gráfica del algoritmo.

Es relevante destacar que, además de las explicaciones más detalladas de este informe, el código de cada clase está documentado, de forma que se incluye al principio de cada clase un breve comentario descriptivo del objetivo de la clase y los atributos que necesita. Por supuesto, el resto del código está igualmente comentado.

NOTA: Todos los conceptos relativos a las ampliaciones se omiten en este apartado, pues se explican ya en los apartados correspondientes (véase [Ampliaciones](#)).

3.1. Clase Main

En primer lugar, contamos con la clase `Main`, esto es, la clase que controla el flujo principal y desde donde se invoca al algoritmo. La aplicación se presenta como una pantalla inicial donde el usuario introduce los parámetros de entrada deseados, que son:

- `rows` → Número de filas del tablero
- `cols` → Número de columnas del tablero

Esta pantalla inicial es controlada por el método `start()` de la clase, donde se dibuja la interfaz y se parsean los parámetros introducidos. Una vez obtenidas las dimensiones de la cuadrícula, la aplicación redirige a la pantalla principal, donde se construye el tablero. Para inicializarlo, se crea el grafo (véase [Clase Graph](#)) y se resetean los siguientes atributos:

- `start_node` → Nodo origen. Necesario para el constructor de la Clase `AStar`, y utilizado para saber si ya se ha establecido un inicio.
- `end_node` → Nodo destino. Necesario para el constructor de la Clase `AStar`, y utilizado para saber si ya se ha establecido una meta.
- `path` → Booleano que indica si se ha ejecutado el algoritmo, para que el usuario no pueda modificar nada en caso de ser así.

El siguiente paso es capturar los eventos generados por acciones del usuario. El criterio utilizado para que el usuario “de forma” al enunciado del problema, es decir, personalice el grafo antes de ejecutar el algoritmo, es el que sigue:

- El click izquierdo del botón sirve para establecer el nodo inicio, el nodo meta y los nodos barrera, por este orden. Así, si no existía ya inicio, el primer click pulsado sobre una casilla del tablero la convertirá en el nodo inicial. Del mismo modo, si no existe final y ya se ha establecido un inicio, el siguiente click convertirá la casilla en el nodo meta. Una vez establecidos ambos, el resto de clicks crearán nodos inalcanzables o nodos barrera, que el algoritmo no podrá visitar.
- Con el click derecho el usuario puede resetear una casilla del tablero, convirtiéndola en un nodo “normal”, fuera cual fuera su estado anterior.

Nótese que los tipos y el estado de los nodos se diferencia mediante un código de colores, explicado en el apartado de la [Clase Utilities](#). Además, un método imprescindible para el correcto funcionamiento de todo esto es el método `get_clicked_pos()` que, dada la posición absoluta en la pantalla, calcula la fila y columna correspondiente en el tablero. Para ello es necesario tener en cuenta los márgenes ocupados por el encabezado y la zona de los botones, así como los posibles márgenes superior-inferior/laterales en caso de que el tablero no sea cuadrado.

De este modo, la clase tiene un método para cada tipo de evento, que actualizará los valores de los atributos y realizará las acciones correspondientes. Así, cuando el usuario pulse el botón de `START`, se comprobará que existe un nodo inicio y uno meta, pues en caso contrario no se ejecutará el algoritmo. El botón `RESET` reiniciará el estado de todos los nodos y creará de nuevo el grafo, una funcionalidad útil para reejecutar el algoritmo tantas veces como se desee sin tener que cerrar el programa y volver a abrir. Por último, con el botón `EXIT` se regresa a la pantalla inicial, por si el usuario quisiera cambiar los parámetros iniciales.

Cuando el usuario pulse `START`, si todo es correcto creará la clase del algoritmo y lo ejecutará, mostrando un mensaje de error en caso de que no existiera camino posible.

3.2. Clase Button

Esta clase se utiliza para representar los botones que tiene la aplicación, por lo que es utilizada por la [Clase Main](#). Los dibuja en la pantalla y detecta si han sido pulsados o no, para que Main invoque su comportamiento.

3.3. Clase Node

La clase «Node» es una de las estructuras de datos básica para representar el problema a resolver, puesto que un nodo es el equivalente a una casilla o celda del tablero, el cual se representa a su vez mediante un grafo.

La información que es necesario mantener para cada nodo es:

- `row, col` → Es la posición del tablero en la que se encuentra, lo diferencia del resto de nodos. Se podría decir que es el identificador del vértice correspondiente del grafo. Estos atributos inducen la creación de un getter, imprescindible para el funcionamiento del algoritmo.
- `color` → Representa el estado del nodo en un momento determinado. Los colores se explicarán más adelante, pero es fundamental almacenarlo no sólo para pintarlo en pantalla sino para implementar el algoritmo. Con este atributo se generan los correspondientes getters y setters, para cumplir los requisitos de la POO y asegurar la mantenibilidad del código.

- `neighbors` → Se trata de una lista que guarda los nodos vecinos al nodo, es decir, aquellos que se encuentren en casillas adyacentes (horizontal, vertical o diagonal) y que sean nodos alcanzables. Se utiliza para expandir un nodo, paso fundamental del algoritmo.
- `x, y` → Es la posición absoluta del nodo en la pantalla, útil para poder dibujarlo.

Además de los métodos ya mencionados, tiene un método que dibuja el nodo.

3.4. Clase Graph

Una vez vistos los nodos que representan las casillas, veamos la estructura de datos que los contiene: el grafo. El grafo representará al tablero y guardará los siguientes atributos:

- `gap` → Es el tamaño de cada casilla, útil para poder dibujarlas. Nótese que todas las casillas son cuadradas e iguales. Este parámetro se calcula a partir de las filas/columnas indicadas por el usuario, de la siguiente manera:
 - $gap = DIM \% \max(rows, cols)$, donde *DIM* es una constante definida que indica el tamaño máximo que puede tomar el tablero.
- `total_rows, total_cols` → Son los parámetros `rows` y `cols` introducidos por el usuario, necesarios para dibujar el tablero, así como construirlo a nivel lógico.
- `pad_rows, pad_cols` → Es el espacio entre el borde horizontal (/vertical) del tablero y el de la pantalla, útil también para dibujar la cuadrícula.
- `nodes` → Es el atributo más importante, puesto que representa el contenido del grafo. Se trata de una lista de los nodos que componen el grafo.

En esta clase se implementan las funciones que dibujan el tablero, además del método que construye los nodos y, con ellos, el grafo. Además, incluye el método `update_neighbors()`, que genera los nodos vecinos de un nodo si no habían sido ya generados, recorriendo las 8 posibles casillas adyacentes (o menos, si salen de los límites del tablero) añadiendo sólo aquellos que no sean inalcanzables.

3.5. Clase AStar

Esta es la clase principal y fundamental de esta práctica, pues contiene la implementación del algoritmo A*. En este apartado se estudiarán los atributos y métodos de la clase, las estructuras de datos necesarias, y el funcionamiento del algoritmo.

En primer lugar, la clase tiene los siguientes atributos: el grafo que representa el problema, que es el espacio que se irá recorriendo expandiendo los nodos según se vayan considerando; el nodo origen y destino, para saber dónde empieza y cuándo acaba el algoritmo; y una variable denominada `nodes_visited`, que es una lista de los nodos que se han ido visitando en orden inverso. Esta lista fue añadida tras implementar la ampliación de los [Waypoints](#), tal y como se explicará en dicha sección, y sirve para poder reconstruir el camino desde el nodo meta hasta el inicio pasando por todos los waypoints en el orden en que se visitaron.

Por otro lado, en cuanto a los métodos de la clase, AStar cuenta con un método `reconstruct_path()` que reconstruye el camino entre dos nodos una vez terminado el algoritmo, el cual se explicará más adelante. Además, tiene las dos funciones $g(n)$ y $h(n)$, mencionadas anteriormente en el apartado de [Introducción](#). Nótese que el método `g()` calcula la función $g(n)$ para un nodo n_2 dado su nodo padre n_1 , del siguiente modo:

$$g(n_2) = g(n_1) + d, \text{ donde } d = \begin{cases} 1 & \text{si } n_2 \text{ está en una casilla horizontal/vertical de } n_1 \\ \sqrt{2} & \text{si } n_2 \text{ está en una casilla diagonal de } n_1 \end{cases}$$

También se utiliza un método `is_special()` para saber si un nodo no debe ser pintado de ningún otro color para poder ser distinguidos; el método que implementa el algoritmo A* propiamente dicho entre dos nodos dados (no tienen por qué ser el nodo inicio y final, uno de ellos podría ser un waypoint); y el método principal de la clase, `algorithm()`, que invoca al anterior y reconstruye el camino. Cabe destacar que este método tuvo que añadirse al refactorizar el código para la ampliación de los [Waypoints](#), pues antes no era necesario.

Ahora que se ha dado una visión general de la clase, veamos las estructuras de datos que se han utilizado para implementar el algoritmo de la manera más eficiente posible:

- `g_score` → Cada vez que se ejecuta el algoritmo es necesario guardar la función $g(n)$ para todos los nodos que se van visitando, porque necesitaremos comprobar si un nuevo camino es mejor que el encontrado hasta el momento, y para calcular la g de un nodo necesitamos la de su nodo padre. Para almacenar esos valores se ha optado por utilizar un diccionario, donde la clave es el nodo n y el valor es $g(n)$. De esta forma, acceder al valor de la función para un nodo tiene un coste perfectamente asequible, y es fácilmente modificable.
- `f_score` → Análogo al anterior, pero para la función $f(n)$. Es imprescindible almacenar este valor porque es el utilizado para priorizar los nodos en la lista ABIERTA, determinando el orden en que se consideran los nodos. Se utiliza de igual manera un diccionario, donde para cada nodo n (clave) guardamos su $f(n)$ (valor). Recordemos que este se calcula así: $f(n) = g(n) + h(n)$.

NOTA: Obsérvese que no es necesario guardar en una variable el valor de $h(n)$, sino que se puede calcular directamente al visitar un nodo y así obtener $f(n)$, que sí se guarda.

- `count` → Esta variable entera representa el momento en que un nodo n entra en la lista `opened`. Sirve para identificar el elemento dentro de la cola (véase [Clase IndexPQ](#)), y para decidir entre iguales (a igualdad de f , sale antes de la lista el que entrase primero).
- `count_dict` → Para poder actualizar la prioridad de un elemento de la cola de prioridad, es necesario conocer el `count` de cada nodo. Por ello, se utiliza otro diccionario que guarda para cada nodo (clave) el valor `count` asociado (valor). Además, este diccionario es útil para saber si un nodo está en ABIERTA de manera inmediata.
- `opened` → Esta variable representa la lista ABIERTA, donde se mantienen los nodos que se han obtenido al expandir sus nodos padre, pero que aún no han sido expandidos-visitados. Los nodos llevan directamente asociados su función de evaluación $f(n)$, y deben salir de la lista por orden creciente según esta, ya que el nodo más prometedor es aquel que puede tener un camino más corto al nodo meta. Por ello, y dado que los nodos se sacarán e insertarán en la lista constantemente, y cambiarán su función de evaluación (\sim prioridad), conviene representar esta lista mediante una cola de prioridad con prioridades variables. Los elementos de esta cola son tuplas que contienen la $f(n)$ del nodo, que será el factor que

determine la prioridad, la variable `count` y el nodo n . Así pues, un elemento de la cola que represente el nodo n_1 será más prioritario que otro que represente n_2 si y sólo si

$$\rightarrow f(n_1) < f(n_2) \vee (f(n_1) = f(n_2) \wedge count_{n_1} < count_{n_2})$$

Utilizando una cola de prioridad en lugar de una lista (array), por ejemplo, las operaciones de inserción, eliminación o cambio de prioridad serán notablemente más eficientes.

- `closed_set` → El algoritmo A* hace uso también de la denominada lista CERRADA, donde se van guardando los nodos que ya han sido expandidos (visitados) y, por tanto, no deben considerarse más. Como sólo es necesario saber si un nodo dado está o no en CERRADA, podemos utilizar un único conjunto sobre el que realizar operaciones de inserción y de comprobación (`contains()`).
- `came_from` → Por último, es necesario almacenar de alguna manera el camino que va encontrando el algoritmo, para luego poder reconstruirlo. Con ese objetivo, se ha optado por crear otro diccionario que guarde para cada nodo n_2 (clave) qué nodo n_1 (valor) es su predecesor en el camino solución.

Ahora que se han analizado las estructuras de datos empleadas, sólo queda estudiar el comportamiento del algoritmo y cómo funciona.

1. El primer paso es **inicializar las variables**. Tanto $f(n)$ como $g(n)$ se inicializan para todos los nodos con el valor ∞ . Por un lado, $g(n)$ se inicializa con ∞ para que cada vez que el algoritmo encuentre un nodo no visitado anteriormente siempre mejore su $g(n)$ y, por tanto, lo consideremos para meterlo en ABIERTA. El único que no se inicializa a ∞ es $g(i)$, siendo i el nodo inicio, porque el coste de llegar desde el nodo inicio hasta sí mismo es 0.

Por otro lado, el valor de $f(n)$ realmente se define cuando se genera un nodo a partir del que se está expandiendo, así que con este valor indicamos que aún no se ha llegado a ese nodo. Ahora bien, $f(i)$ empieza con otro valor: $g(i) + h(i) = 0 + h(i) = h(i)$, es decir, la distancia euclídea entre el nodo inicio y el nodo meta.

La variable `count` mencionada para decidir entre iguales se inicializa a 0, la lista CERRADA empieza vacía, y la lista ABIERTA empieza teniendo el nodo inicio, para dar comienzo al algoritmo. Por tanto, el diccionario `count_dict` empieza teniendo el nodo inicio con el valor asociado 0. Por último, el diccionario `came_from` guarda por ahora que el nodo anterior al inicio es sí mismo.

2. Ahora empieza a ejecutarse el algoritmo, que parará cuando se cumpla una de estas 3 condiciones:
 - a. El usuario ha pulsado el botón X para salir
 - b. La lista ABIERTA está vacía → No hay solución
 - c. Hemos encontrado el nodo meta
3. Lo primero que hace el algoritmo es sacar el **nodo más prioritario** de ABIERTA – nos referiremos a él como n – y meterlo en CERRADA para indicar que ya se ha visitado. Si n era el nodo meta, entonces se devuelven dos booleanos con valor `True` para indicar que ha terminado con éxito, y pasamos al paso 5.
4. A continuación, toca **expandir** n , generando los nodos adyacentes a él. Para cada nodo vecino v se ejecutan los siguientes pasos, sólo si v no está en CERRADA:

- a. Se calcula $g(v)$ accediendo desde n , mediante la fórmula explicada al comienzo de este apartado, y con ese valor y el que obtengamos con $h(n)$ se calcula $f(v)$.
 - b. Si $f(v)$ es mejor (es decir, **menor**) que la $f(v)$ que teníamos almacenada hasta el momento, entonces seguimos con los siguientes pasos. Si no, se pasa al siguiente nodo vecino y se vuelve a ejecutar el paso anterior. Nótese que la primera vez que el algoritmo llega a un nodo la condición anterior siempre se cumple, pues la f se inicializó a ∞ .
 - c. El camino solución ahora pasa por v , así que se **actualizan** los valores correspondientes: $came_from[v] = n$ (el padre de v es n), y el valor de g y f pasa a ser el que se acaba de calcular.
 - d. Si v no estaba ya en ABIERTA, lo **añadimos** con la f que se acaba de calcular y $count+1$, y se actualiza el $count$ del nodo v en el diccionario. Si estaba, entonces actualizamos su prioridad con la nueva f calculada.
5. Una vez encontrado el nodo meta, el último paso es **reconstruir** el camino solución, para lo cual se utiliza el diccionario $came_from$. Basta empezar por el nodo meta e ir obteniendo el nodo padre hasta que se llegue al nodo inicio.

De esta forma, el algoritmo A* encuentra el camino más corto entre dos puntos, generando una solución aplicable a un inmenso número de problemas reales en distintos ámbitos. Por ejemplo, para simular viajes de un vehículo circulando por un terreno con diferentes alturas, algunas inaccesibles, o un vuelo de avión donde las zonas de tormenta deben ser evitadas.

Si bien es cierto que se contaba con la opción de desarrollar la aplicación de manera más “realista”, por ejemplo, con alguna de las situaciones anteriores, se ha considerado que esta forma de representar gráficamente el algoritmo permite a un usuario con conocimientos sobre el algoritmo visualizar realmente el comportamiento de este. Gracias a la representación de los nodos que están en ABIERTA y en CERRADA, la aplicación permite ver fácilmente cómo A* va considerando los nodos el algoritmo y cuándo cambia de dirección porque encuentra nodos más prioritarios, por ejemplo, si ha encontrado una “pared” de nodos inalcanzables. De este modo, el usuario no sólo obtiene el camino solución, sino mucha más información sobre el algoritmo implementado.

3.6. Clase IndexPQ

Para implementar el algoritmo de la clase anterior, ha surgido la necesidad de crear esta clase para implementar una nueva estructura de datos: la cola de prioridad con prioridades variables. A pesar de que Python proporciona el módulo «queue» con el que es posible hacer uso de la `PriorityQueue`, entre otros, esta estructura no es suficiente para los requisitos del algoritmo.

La `PriorityQueue` permite operaciones de inserción y eliminación del elemento más prioritario, además de otras operaciones de consulta. No obstante, los nodos de la lista ABIERTA necesitan cambiar su prioridad (es decir, su función $f(n)$) conforme avanza el algoritmo, una operación no soportada por esta clase.

Vista la necesidad de que los elementos de la cola puedan actualizar su prioridad, se ha implementado una cola de prioridad con prioridades variables (IndexPQ ~ Index Priority Queue) utilizando montículos binarios. Esta estructura de datos fue estudiada en profundidad por los miembros del equipo en la asignatura de Técnicas Algorítmicas de Ingeniería del Software, analizando los costes de las operaciones y la implementación a bajo nivel de los montículos.

Así pues, la clase cuenta con tres atributos principales: el número de elementos que puede contener la cola, un vector (lista) `array` que contiene los elementos de la cola en cada momento, y otro vector `positions` que se explicará a continuación.

Recordemos que un elemento de la cola está formado por tres parámetros: la prioridad ($f(n)$), el nodo del grafo, y el `count`, que representaba el momento en que el nodo entró en la cola y que identifica el elemento dentro de la cola de prioridad. De este modo, el vector de posiciones contiene para un elemento con `count=c`, en qué posición está de la lista `array`, o 0 si no está. Por tanto, se cumple que `array[positions[c]] = elemento c`.

Las operaciones que invoca la [Clase AStar](#) sobre esta clase son las que siguen:





- `put()` → permite insertar un elemento en la cola de prioridad
- `update()` → dado un elemento de la cola, permite modificar su prioridad
- `empty()` → comprueba si la cola de prioridad está vacía
- `top()` → devuelve el elemento más prioritario de la cola
- `pop()` → elimina de la cola el elemento más prioritario





Además de estas, la clase cuenta con una serie de métodos privados: `size()`, para conocer el tamaño real de la cola en un momento determinado; `prior()`, que establece el criterio de orden de la cola (la prioridad); y los métodos `float()` y `sink()`. Estos métodos se utilizan para mantener la consistencia del montículo binario, y operan sobre árboles equilibrados de búsqueda (AVL). Sin embargo, no se entrará en detalles de implementación, pues no se consideran relevantes para esta práctica. Lo importante es conocer que la raíz del árbol contiene el elemento más prioritario, y que cuando este se elimina, o cuando se inserta un nodo hoja, o cuando se modifica la prioridad de un nodo, el árbol se actualiza para mantener la propiedad de la estructura de datos.

3.7. Clase Utilities

Por último, se presenta la clase «Utilities», donde se definen las constantes utilizadas por el resto de las clases. Contiene parámetros relativos a la interfaz (dimensiones de la pantalla, márgenes...), el conjunto de direcciones en las que el algoritmo puede avanzar (horizontales, verticales y diagonales), y los colores que representan los estados de los nodos.

A continuación, se muestra la leyenda de colores utilizada:

COLOR	SIGNIFICADO
 AMARILLO	Nodo inicio
 VERDE	Nodo meta
 GRANATE	Nodos inalcanzables (barrera)
 MARRÓN	Nodos que han quedado en ABIERTA

COLOR	SIGNIFICADO
 BEIGE	Nodos que han quedado en CERRADA
 AZUL	Nodos que pertenecen al camino solución
 MORADO	Waypoints
 ROJO	Casillas peligrosas

Además de estos colores se definen otros para los colores básicos de la aplicación, las líneas del tablero y el texto, entre otros. Nótese que hay nodos que pertenecen al camino solución pero no se pintan de azul por su naturaleza: el nodo inicio, meta, y los waypoints y casillas peligrosas se mantienen de su color original para poder ser distinguidos.

4. Ampliaciones

En este apartado se presentan las ampliaciones realizadas sobre la práctica básica, que incluye únicamente el cálculo del camino más corto entre un nodo inicio y uno meta teniendo nodos inalcanzables que el algoritmo debe evitar, y todo en un tablero con unas dimensiones definidas por el usuario.

En este caso, se ha optado por implementar las dos ampliaciones que resultaban de mayor interés para los miembros del equipo especialmente por su posible aplicación en escenarios reales: la posibilidad de añadir waypoints y la opción de tener casillas peligrosas.

Estas ampliaciones sobre el enunciado básico se explican en los siguientes apartados, junto a una explicación de la refactorización del código que han provocado y cómo han sido implementadas.

4.1. Waypoints

La primera ampliación consiste en permitir al usuario añadir casillas predeterminadas, denominadas “waypoints”, por las que el algoritmo debe pasar antes de llegar al nodo meta. Esto podría entenderse de dos maneras, según lo ha visto el equipo: el usuario indica una serie de casillas por las que es necesario pasar en el orden en que las ha marcado, o son celdas por las que hay que pasar, pero debe encontrarse el mejor orden para recorrerlas.

Dado que el enunciado deja este aspecto abierto a posibles interpretaciones, el equipo ha optado por intentar implementar la segunda alternativa, pues parecía de mayor interés. Recorrer unas casillas en el orden establecido por el usuario se traduce sencillamente en ejecutar tantas veces el algoritmo como waypoints haya, recorriendo una lista ya generada desde el principio para ir modificando el nodo inicio-fin. Esto, realmente, no supone una gran ampliación sobre el algoritmo que ya se tenía.

Sin embargo, la segunda opción supone un reto mayor, pues es necesario averiguar el orden para recorrer los waypoints que componga el camino total más corto, sabiendo que el último en ser visitado debe ser el nodo meta. Una vez hecho este apunte, veamos cómo se ha implementado esto.

Para decidir el orden en que se visitan los waypoints, se ha optado por utilizar una cola de prioridad (PriorityQueue) donde los elementos son los waypoints que aún no han sido visitados en un momento dado, y la prioridad es la distancia euclídea (función $h(n)$) al nodo desde el cual se está ejecutando en ese momento el algoritmo. Esta cola se genera en el método `order_waypoints()` cada vez que se invoca al algoritmo con un nodo inicio diferente, para así intentar ir primero al nodo más cercano al actual.

Por supuesto, esta estrategia no asegura que el camino total sea el más corto, pero aproxima más a esa solución ideal. Además, para mejorar aún más el algoritmo, se ha refactorizado añadiendo la siguiente condición después de comprobar si hemos alcanzado el nodo meta:

- Si el nodo más prioritario que se ha extraído de ABIERTA es un waypoint (y no es el waypoint al que se pretendía llegar y se consideraba como “nodo sub-meta”) entonces el algoritmo termina, porque hemos averiguado que es más rápido visitar primero ese waypoint. De este modo, será necesario recalcular la cola de prioridad de los waypoints, introduciendo el que se pretendía visitar originalmente, y sacando el que se ha encontrado de forma imprevista.

De nuevo, esta estrategia supone una mejora con respecto a la anterior, aunque sigue sin asegurar al 100% que sea la mejor. Se consideró realizar esta operación (recalcular la cola de prioridad) cada vez que el algoritmo visita el siguiente nodo, para actualizar la decisión del waypoint destino en cada paso del algoritmo y mejorar aún más el algoritmo. Ahora bien, esto supone un coste excesivamente elevado, por lo que al final resulta ser una poda más costosa que el propio beneficio de la poda, y se ha decidido no implementarla.

En lo relativo al código, la implementación de esta ampliación ha supuesto una serie de cambios significativos:

- En primer lugar, ha surgido la necesidad de añadir los siguientes atributos: `waypoints`, un conjunto que contiene todos los waypoints establecidos por el usuario (útil para saber si el nodo que se está visitando es un waypoint); `waypoints_visit`, la cola de prioridad mencionada anteriormente que mantiene los waypoints que aún no han sido visitados por orden creciente de distancia al nodo inicio en una determinada ejecución del algoritmo; `waypoint_visit_set`, un conjunto con el mismo contenido que la cola, necesario para conocer los waypoints que quedan por visitar (recordemos que la cola de prioridad sólo permite ver el elemento más prioritario); y la lista `nodes_visited`, ya mencionada en apartados anteriores. En esta lista se guardan los nodos meta que se han ido visitando en orden inverso (se inserta por el principio), para poder reconstruir el camino desde el nodo meta real hasta el nodo inicio real pasando por los waypoints que se han ido visitando en ese orden.
- Se ha refactorizado el método `reconstruct_path()`, que antes utilizaba los nodos inicio y meta, pero ha sido generalizado para reconstruir el camino entre dos nodos cualesquiera mediante su correspondiente diccionario `came_from`, pues es necesario llamar a este método una vez por cada waypoint.
- En el método del algoritmo `astar()` antes se invocaba a `reconstruct_path()` tras encontrar el nodo meta, pero ahora este comportamiento se ha trasladado a una función general que invoca las veces necesarias tanto al algoritmo como a ese otro método. Además, se ha añadido el caso ya explicado en el que el algoritmo se encuentra con un waypoint inesperado. Obsérvese que el resto del comportamiento del algoritmo es idéntico, pues los waypoints se tratan como nodos inicio/meta de forma igual que los originales.
- Se ha creado el método `order_waypoints()`, ya mencionado anteriormente, que dado un nodo genera la cola de prioridad de los waypoints que quedan por visitar por orden de distancia euclídea a dicho nodo.
- Se ha añadido un nuevo posible estado a los nodos ([Clase Node](#)), con los correspondientes getters y setters.
- En cuanto a la interfaz, fue inmediato añadir el nuevo comportamiento para la tecla 'W', que añade un waypoint, y, conforme el usuario los añadía, generar el conjunto que posteriormente se pasaría como parámetro al constructor de la clase AStar. Esta parte de la ampliación no es demasiado relevante para la práctica, por lo que no se entrará en mayor detalle.
- Por último, se ha creado el método general `algorithm()` que controla y ejecuta el algoritmo completo. Este método invoca primero a `order_waypoints()` para obtener el primer

waypoints a visitar, e invoca al algoritmo mientras queden waypoints por visitar, y mientras no haya encontrado un waypoint inalcanzable, en cuyo caso aborta porque el problema no tiene solución. Nótese que ahora `astar()` devuelve un nuevo parámetro, que es el nodo donde debe empezar la siguiente ejecución del algoritmo. Este nodo, en principio, debería ser el nodo meta de la ejecución anterior, pero puede cambiar si el algoritmo encontrase un waypoint inesperado. Una vez visitados todos los waypoints, ejecuta una última vez el algoritmo para buscar el nodo meta, y reconstruye el camino por partes recorriendo la lista `nodes_visited`.

4.2. Casillas peligrosas

Esta segunda ampliación es considerablemente más sencilla de implementar que la anterior. El objetivo consiste en permitir al usuario añadir casillas que sean peligrosas, pero no inalcanzables. Esto es, nodos que suponen un riesgo hasta cierto punto asumible, lo cual se traduce en que la función de evaluación se verá penalizada por utilizar uno de estos nodos.

De nuevo, el enunciado no concreta los valores de las penalizaciones, por lo que el equipo ha optado por dejarlo a elección del usuario, añadiendo un nuevo parámetro de entrada (además de las dimensiones del tablero). De este modo, el usuario tiene la oportunidad de probar diferentes valores de riesgo/penalización, y ver cómo el algoritmo cambia su comportamiento en función de este, lo cual parece tener mayor interés que mantener la penalización como una constante previamente definida que no permite entender realmente cómo se ve afectado el algoritmo.

La modificación más importante introducida por esta ampliación ha sido, por supuesto, añadir la penalización cuando se calcula $f(n)$ si n es un nodo peligroso. Esta penalización aumenta el valor de la función, lo cual provoca que baje la prioridad del nodo dentro de ABIERTA y sea considerado más tarde de lo previsto, poniendo por delante otros nodos, a pesar de estar aparentemente más cerca del nodo meta.

Además de la codificación de esta funcionalidad, ha sido necesario refactorizar otros aspectos del proyecto:

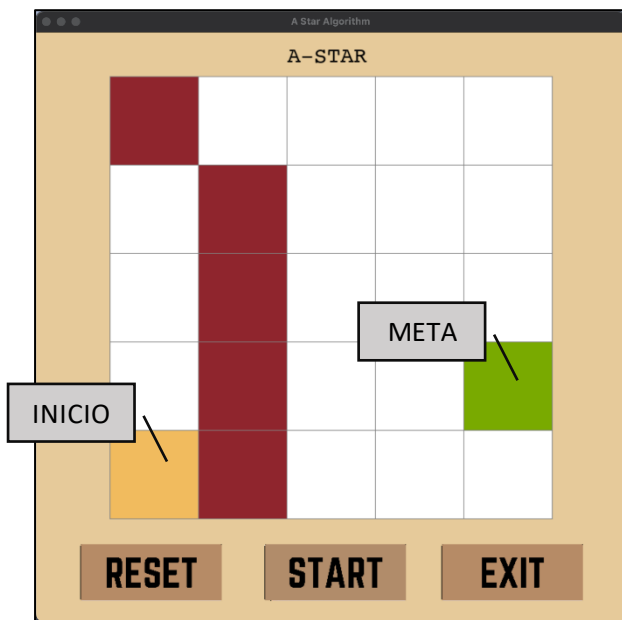
- Por un lado, de igual manera que la ampliación anterior, se ha extendido la captura de eventos en la [Clase Main](#) para añadir el nuevo comportamiento de la tecla 'R', que añade una casilla peligrosa. Estas casillas se van incorporando a un conjunto que posteriormente será pasado como parámetro al constructor de la [Clase AStar](#).
- Ya en la clase del algoritmo ha sido necesario añadir un parámetro que guarde la penalización que supone pasar por un nodo peligroso, así como el conjunto mencionado con los nodos peligrosos seleccionados por el usuario.
- Tal y como se ha explicado, se ha añadido en la pantalla inicial un nuevo campo input para que el usuario introduzca el riesgo que desea que tengan las casillas peligrosas (sólo son validos valores positivos o iguales a 0).
- Se ha añadido un nuevo posible estado a los nodos ([Clase Node](#)), con los correspondientes getters y setters.

5. Ejemplos

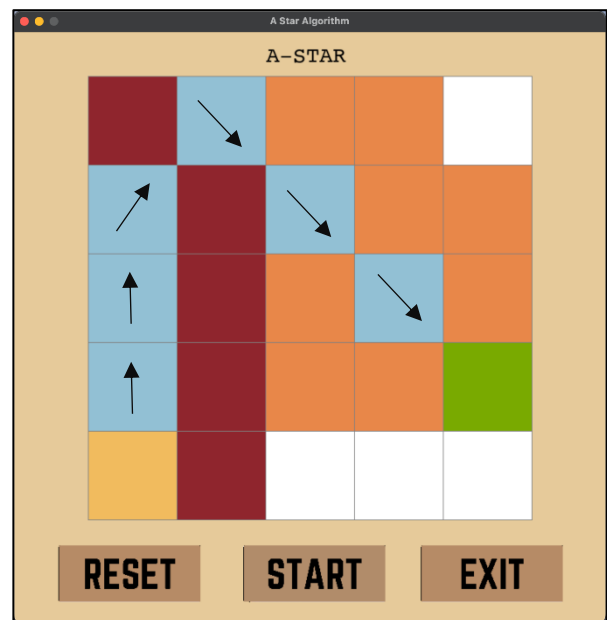
En esta última sección se proponen una serie de ejemplos de ejecución del algoritmo, para poder visualizar todo lo explicado anteriormente en el informe en un escenario real. Se plantea la situación que constituye el ejemplo, y se incluyen capturas para ver el estado antes y después de ejecutar el algoritmo.

- EJEMPLO 1

Este primer ejemplo es el equivalente al que propone el enunciado de la práctica, para mostrar un ejemplo básico que no incluya ninguna de las ampliaciones:



Ejemplo 1. Problema

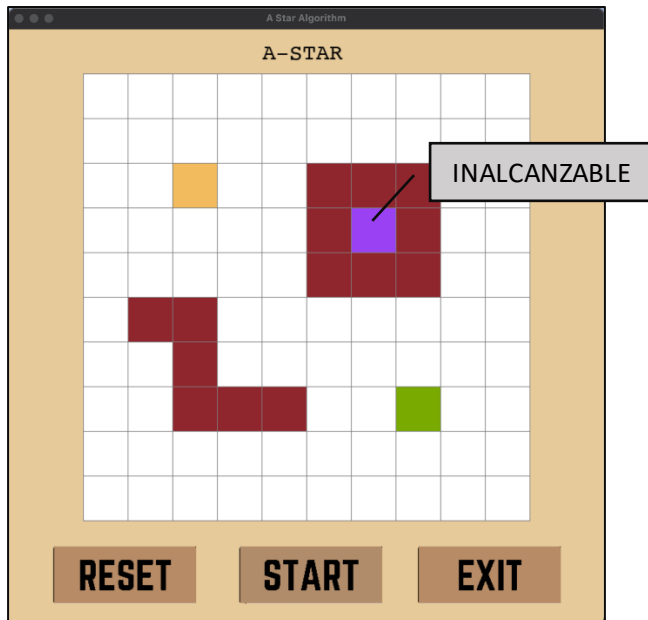


Ejemplo 1. Solución

NOTA: Los nodos señalados en marrón-naranja son los que han quedado en ABIERTA tras finalizar el algoritmo.

- EJEMPLO 2

En este ejemplo se muestra un problema en el que no existe posible solución. En este caso, el nodo no alcanzable es el waypoint por el que debe pasar el algoritmo antes de llegar al nodo meta.



Ejemplo 2. Problema

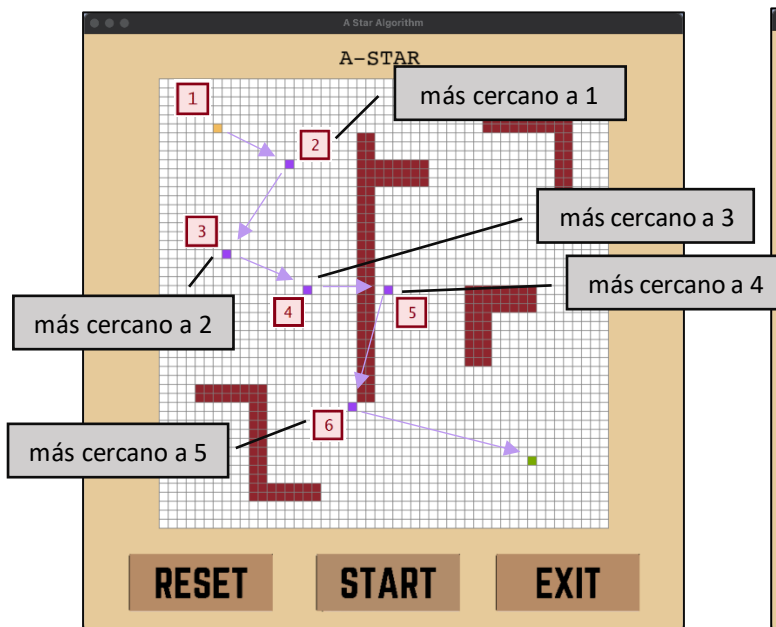


Ejemplo 2. Solución

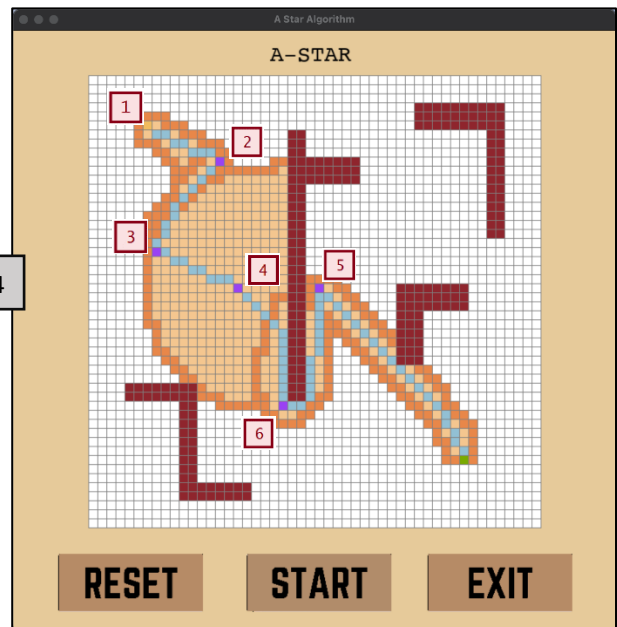
Efectivamente, todos los nodos del tablero están en beige, que indica que están en CERRADA. Es decir, el algoritmo ha parado porque la lista ABIERTA se ha quedado vacía.

- EJEMPLO 3

Con este ejemplo se ilustrará el comportamiento de la primera ampliación de la práctica, esto es, cuando el usuario marca varios waypoints.



Ejemplo 3. Problema

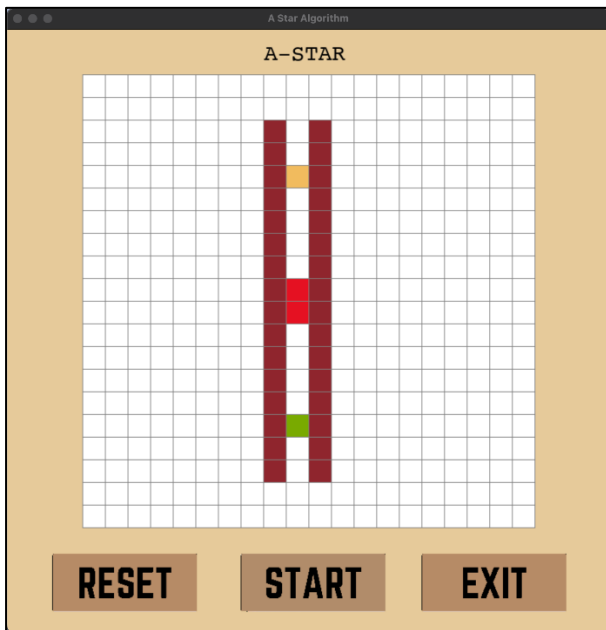


Ejemplo 3. Solución

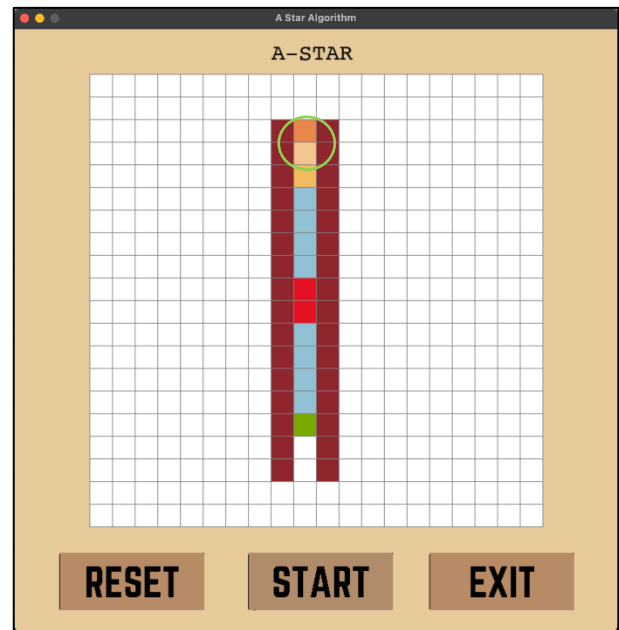
En la primera imagen, se señala el orden en que podría pensarse que se recorrerían los waypoints, pues cada vez que se alcanza un waypoint se calcula el siguiente más cercano a él. Sin embargo, en la solución vemos que los nodos 5 y 6 se recorren al revés. Esto se debe a que cuando la aplicación pasa a ejecutar el algoritmo desde el waypoint 4, con el objetivo de llegar al 5, se encuentra antes con 6. De esta forma, el algoritmo rectifica y finaliza en 6, y reejecuta el algoritmo desde 6 buscando el 5.

- EJEMPLO 4

Este ejemplo ilustrará el funcionamiento de la segunda ampliación de la práctica, por la cual el usuario puede marcar ciertas casillas como casillas peligrosas. Para mostrar mejor el comportamiento del algoritmo en función del parámetro *risk*, se ha ejecutado el mismo escenario para dos valores distintos: *risk* = 3 y *risk* = 100.



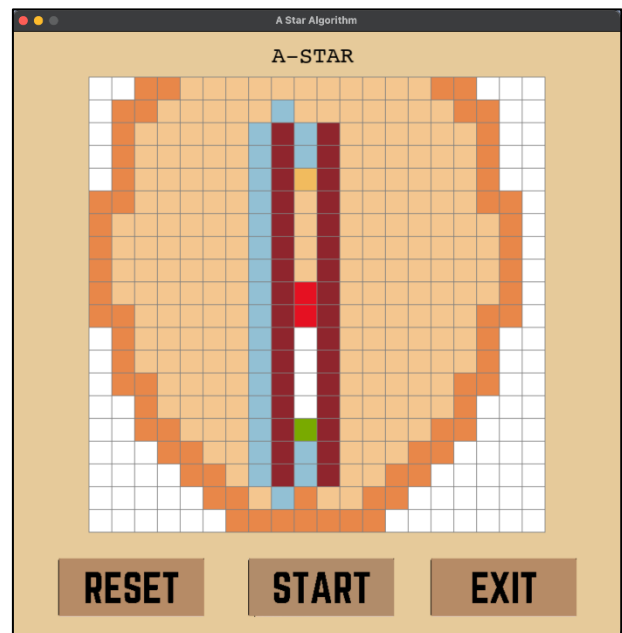
Ejemplo 4. Problema



Ejemplo 4. Solución (risk = 3)

En el primer caso, cuando la penalización por pasar por las casillas peligrosas (rojas) no es demasiado elevado, el algoritmo opta por atravesarlas. De hecho, se puede ver en los nodos señalados con un círculo en la imagen que ya sólo avanzando dos casillas hacia arriba desde el inicio deja de compensar seguir por ese camino.

En cambio, si analizamos el segundo caso, donde la penalización es notablemente más alta, el algoritmo opta por rodear la barrera para alcanzar el nodo meta. Al llegar a la primera casilla peligrosa esta entra en ABIERTA con un valor $f(n)$ tan elevado que todos los demás nodos son considerados antes que este, encontrando por tanto el camino alternativo.



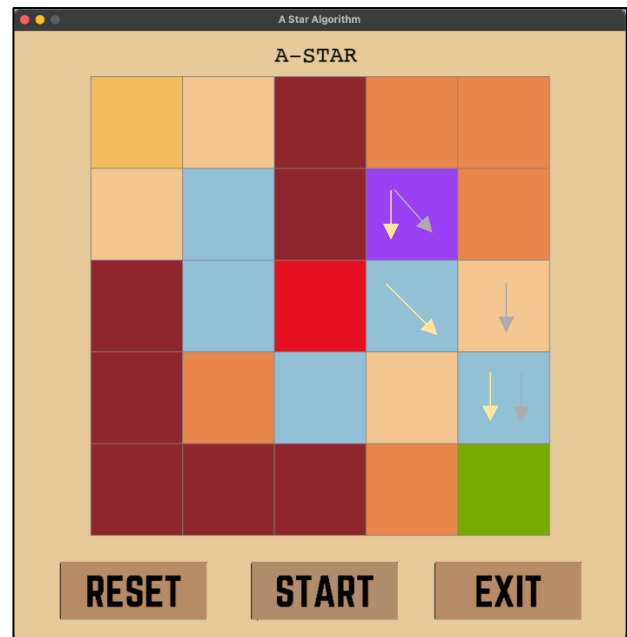
Ejemplo 4. Solución (risk = 100)

• EJEMPLO 5

Por último, se ha decidido incorporar un ejemplo adicional para analizar el funcionamiento del algoritmo con mayor grado de profundidad, comprobando que lo que se muestra en pantalla sigue, de hecho, los pasos del algoritmo. Para poder desarrollarlo, se ha propuesto un escenario bastante sencillo con unas dimensiones asequibles.



Ejemplo 5. Problema



Ejemplo 5. Solución

Así pues, veamos los pasos que sigue el algoritmo:

- El primer waypoint (y único) que se debe visitar es el más cercano al nodo inicio, es decir, el nodo (1,3). Por tanto, se ejecuta una primera vez el algoritmo buscando ese nodo destino.
- En ABIERTA se encuentra el nodo inicio, es decir, el (0,0) en este caso. El nodo inicio sale de ABIERTA y entra en CERRADA, y se expanden los nodos adyacentes, que son (0,1), (1,0) y (1,1).
- De ellos 3 el camino seguirá por (1,1), pues moverse en diagonal tiene un coste menor que en vertical u horizontal. Ahora bien, al expandir los nodos desde el (1,1), nos encontramos con barreras (que el algoritmo ignorará) y una casilla peligrosa. Esta casilla está en la diagonal y es la más cercana al waypoint, por lo que podría parecer que el camino debería atravesarla. Sin embargo, la penalización hace que el nodo entre en ABIERTA con un valor $f(n)$ más elevado, dando prioridad al nodo de debajo.
- Veamos que, de hecho, la función del nodo (2,1) es menor que la de la casilla peligrosa (2,2):
 - $f((2,1)) = g((2,1)) + h((2,1)) = (\sqrt{2} + 1) + \sqrt{5} \approx 4,65$
 - $f((2,2)) = g((2,2)) + h((2,2)) + penalizacion = (\sqrt{2} + \sqrt{2}) + \sqrt{2} + 3 \approx 7,24$
- Así pues, el algoritmo “esquiva” la casilla peligrosa y sigue hacia el waypoint. Una vez alcanzado, se ejecuta de nuevo el algoritmo, siendo el nodo inicio el waypoint (que entra en ABIERTA), y el nodo meta el original.
- Nótese que hay dos posibles caminos igual de cortos entre el waypoint y el nodo meta, señalados en la imagen con amarillo y gris. En este caso, el algoritmo ha escogido el amarillo

porque el nodo (2,3) habrá sido generado y, por tanto, insertado en ABIERTA, antes que el (2,4), por lo que tendrá un `count` menor y será considerado más prioritario.

- Una vez encontrado el nodo meta, basta reconstruir el camino desde dicho nodo hasta el waypoint, y luego desde el waypoint hasta el nodo inicio, obteniendo el camino que se ve en la imagen solución.