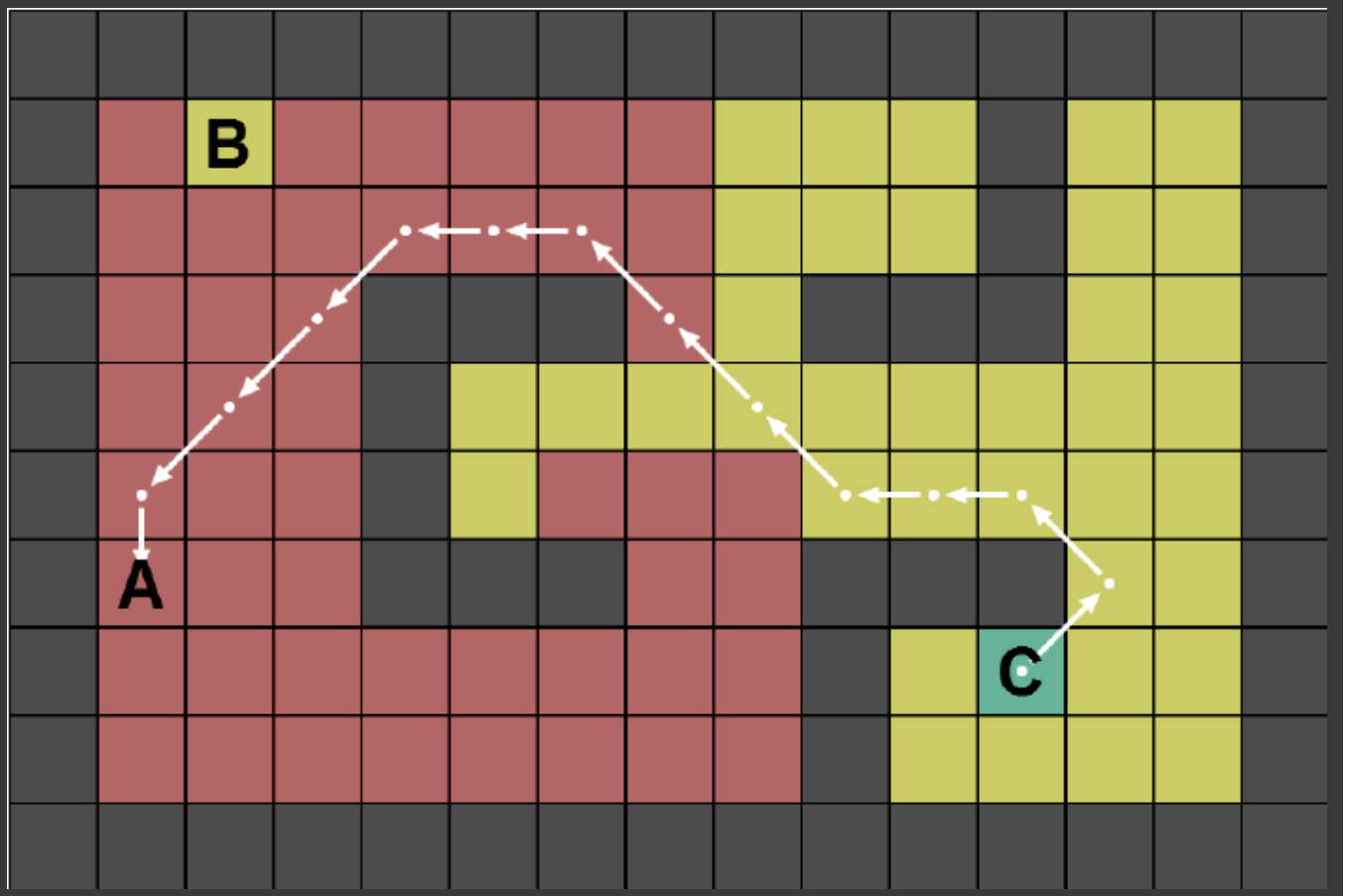
Introducción

El objetivo de la práctica es desarrollar un algoritmo de búsqueda heuristica capaz de encontrar el mejor camino entre dos puntos si lo hubiese. Concretamente tenemos que implmentar el algoritmo A*.



Conceptos básicos

Introducción

El algoritmo A* se clasifica dentro de los algoritmos de búsqueda de grafos. Este algoritmo tiene como objetivo buscar distintas rutas entre un punto inicial y un punto final, sólo se “construyen” los caminos que son candidatos a formar una posible solución, es decir, los caminos que cumplen con ciertas restricciones, en nuestro caso el camino tiene que ser el de menor coste entre ambos nodos. Este algoritmo se basa en el empleo de heurísticas, este concepto se puede definir como una suposición que realizamos partiendo de datos que ya conocemos y de datos sin completar, en otras palabras, es una aproximación cuantitativa de lo que nos falta para llegar a la resolución del problema. La heurística determinará el coste computacional que tendrán los distintos algoritmos que se pueden emplear para la resolución del problema, con esto podemos determinar qué algoritmo será el mejor para la

problema a resolver cuenta con muchos nodos se volvera una tarea complicada al tener que considerar todos los sucesores con los costos respectivos.

Explicación general

Ahora procederemos a explicar el algoritmo A*. Este algoritmo se basa en la siguiente función:

```
f(n) = g(n) + h(n)
```

Donde g(n) representa el coste real que se ha requerido desde el nodo inicial hasta el nodo actual, también tenemos h(n) que se trata de la estimación que haremos desde el nodo actual hasta el nodo meta. Este algoritmo también emplea dos estructuras de datos para almacenar los nodos, por un lado los nodos por los que es factible expandir se almacenarán en la lista Frontera, por su parte aquellos nodos que ya se han expandido y que hemos visitado los almacenaremos en la lista Interior. Gracias a esto el algoritmo puede funcionar de forma “inteligente”, ya que se puede dar el caso de que podamos llegar a la meta por un camino que sea más corto que uno que ya habíamos tomado como mejor, es decir, permite que el algoritmo sea capaz de evaluar nodos que sean más prometedores que los que ya habíamos evaluado.

Propiedades de A*

A* como todo algoritmo de búsqueda en amplitud, es un algoritmo completo, es decir, en caso de que exista una solución siempre dará con ella. La heurística se pueden emplear para controlar el comportamiento de A*, dependiendo del valor de la heurística tendremos distintos escenarios:

- Por un lado tenemos que si para todo nodo n del grafo se cumple que g(n) = 0, nos encontraremos frente a un algoritmo de búsqueda voraz.
- También tenemos que sí para todo nodo n del grafo se cumple que h(n) = 0, A* actuará como un algoritmo de Dijkstra.
- De la misma manera, sí h(n) es coincide con el coste de moverse de n hasta la meta, entonces el algoritmo A* solo seguirá el mejor camino y nunca expandirá nodos demás, lo que hará que sea muy eficiente. Esto no ocurrirá en todos los casos, solo en ciertos casos especiales.

cualquiera de sus nodos hijos (expansiones), el coste coste de llegar al objetivo desde n no será mayor que el de llegar a uno de sus hijos más el coste de llegar desde sus hijos hasta el nodo meta.

Implementación de A*

Para la explicación del pseudocódigo del algoritmo me ayudaré de código para que sea más sencillo.

```
Alg A*
  listaInterior = vacio
  listaFrontera = inicio
  mientras listaFrontera no esté vacía
    n = obtener nodo de listaFrontera con menor f(n) = g(n) + h(n)
    listaFrontera.del(n)
    listaInterior.add(n)
    si n es meta
      devolver
      reconstruir camino desde la meta al inicio siguiendo los punteros
    fsi
    para cada hijo m de n que no esté en listaInterior
      g'(m) = n.g + c(n, m)  //g del nodo a explorar m
      si m no está en listaFrontera
        almacenar la f, g y h del nodo en (m.f, m.g, m.h)
        m.padre = n
        listaFrontera.add(m)
      sino si g'(m) es mejor que m.g  //Verificamos si el nuevo camino es mejor
        m.padre = n
        recalcular f y g del nodo m
      fsi
    fpara
  fmientras
  devolver no hay solución
```

```
        self.g = 0
        self.h = 0
        self.coste = 0
        self.padre = padre
```

También hemos implementado unos getters, que no serán de ayuda en ciertos casos.

```
def getF(self) -> float:
    return self.f

def getG(self) -> float:
    return self.g

def getH(self) -> float:
    return self.h

def getPadre(self):
    return self.padre
```

Y también una especial que nos servirá para comparar los distintos nodos entre si, por ejemplo cuando iteremos sobre una lista, etc. Diremos que dos nodos son iguales si sus filas y columnas coinciden.

```
#Sobrecarga del operador ==
def __eq__(self, other):
    return self.casilla == other.casilla
```

Aquí se llama al operador == de la clase Casilla.

```
def __eq__ (self, other):
```

```
nodoInicial: Nodo = Nodo(origen)
nodoMeta: Nodo = Nodo(destino)

#Se añade a la lista frontera el origen
listaFrontera.append(nodoInicial)

orden = 0
```

```
#Matriz de -1, para los expandidos
def iniciaEstados(mapi):
    estados = []
    for i in range(mapi.alto):
        estados.append([])
        for j in range(mapi.ancho):
            estados[i].append(-1)
    return estados
```

Ahora procedemos a entrar en el bucle, iteraremos mientras haya elementos en la lista o hasta que encontremos solución. Tras entrar en el bucle sacamos el mejor nodo, en nuestro caso se encontrará en la posición 0 de la lista, esto se debe a que al añadir un elemento a la lista ordenamos la lista para que los elementos con menor f estén primero. Y también guardamos el orden en el que se van expandiendo los nodos.

```
while listaFrontera:
    # Cogemos el mejor nodo de la lista Frontera
    n: Nodo = listaFrontera[0]
```

```
    for i in range(mapi.alto):
        for j in range(mapi.ancho):
            print(camino[i][j], end=" ")
        print()
```

Si no es meta, tendremos que borrar el nodo n de la lista frontera y añadirlo a la lista interior. Y procedemos a mirar los vecinos válidos del nodo n/mejor.

```
else:
    """Expandimos nodo"""
    listaInterior.append(n)
    listaFrontera.remove(n)
```

Para sacar los vecinos he diseñado una función, iteramos desde una posición anterior a la fila del nodo n hasta una posterior, y dentro hacemos lo mismo dentro con las columnas. Luego miramos si una posición es correcta/válida, definimos una posición válida como una posición que tiene como carácter '.' o el número 0, ya que así está definido en el mapa, y además hay que comprobar que cada posición sea distinta de la actual, ya que no tiene sentido volver a evaluar la posición actual.

resultado es 1 se tratará de un movimiento en una dirección y tendrá como coste 1, si es 2 se tratará de un movimiento en diagonal y tendrá como coste 1.5.

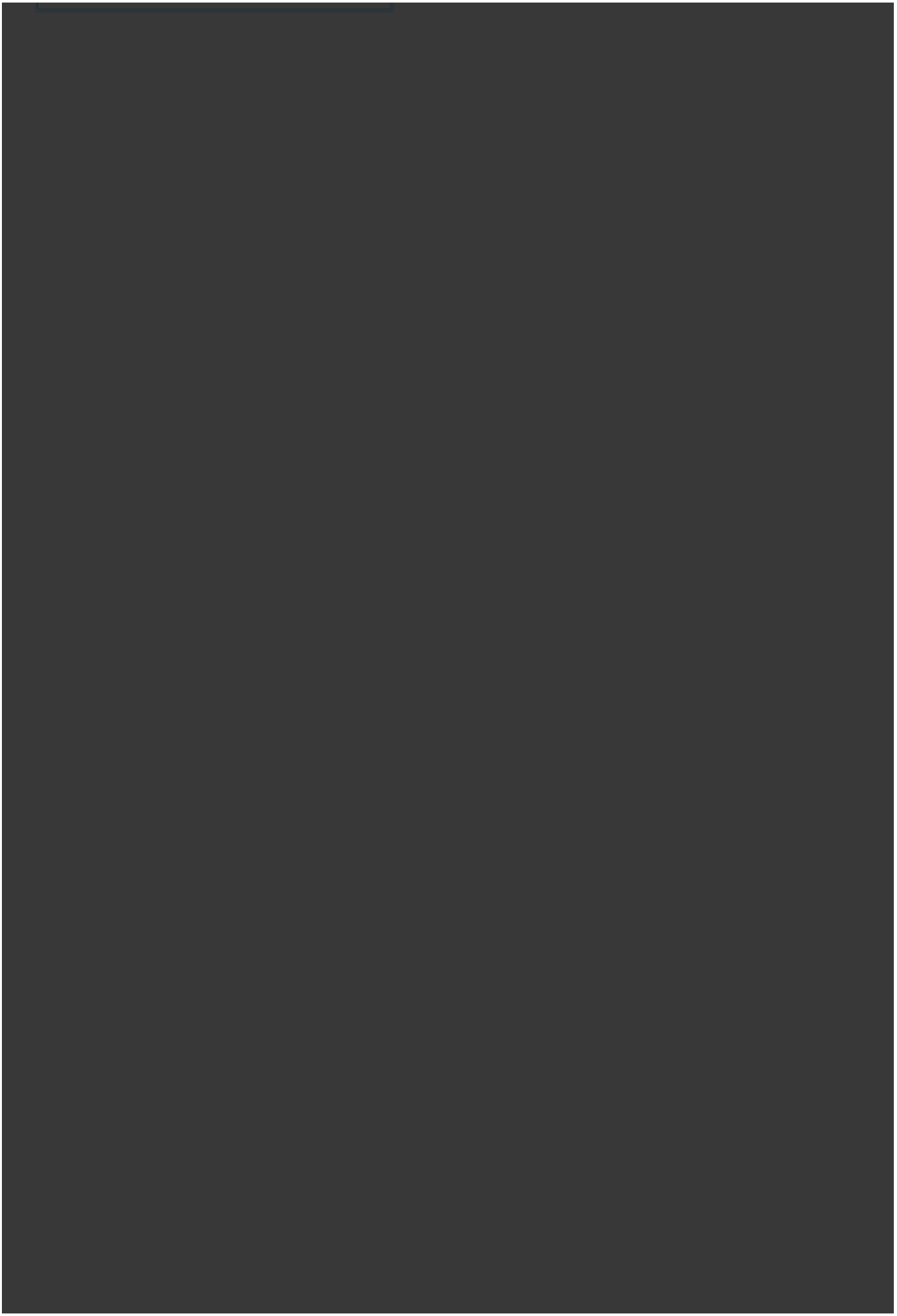
```
def costeCelda(vecino: Nodo, n: Nodo) -> float:
    x = abs(n.casilla.fila - vecino.casilla.fila)
    y = abs(n.casilla.col - vecino.casilla.col)
    if x + y == 1:
        return 1.0
    else:
        return 1.5
```

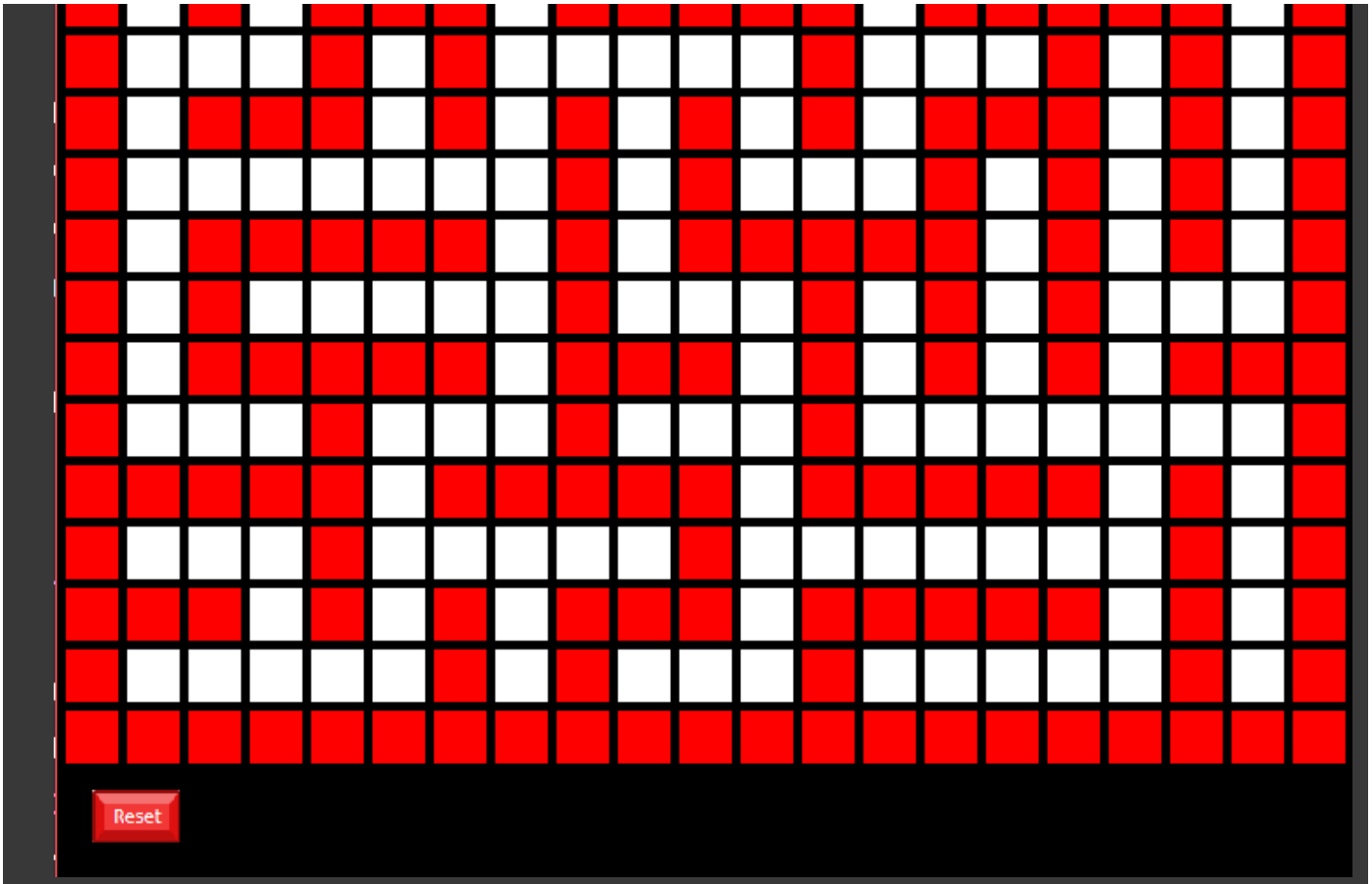
Ahora miramos si el nodo hijo se encuentra en la lista frontera, si no se encuentra el nodo se añade directamente a la lista frontera, antes tendremos que calcular las funciones g, h y f del

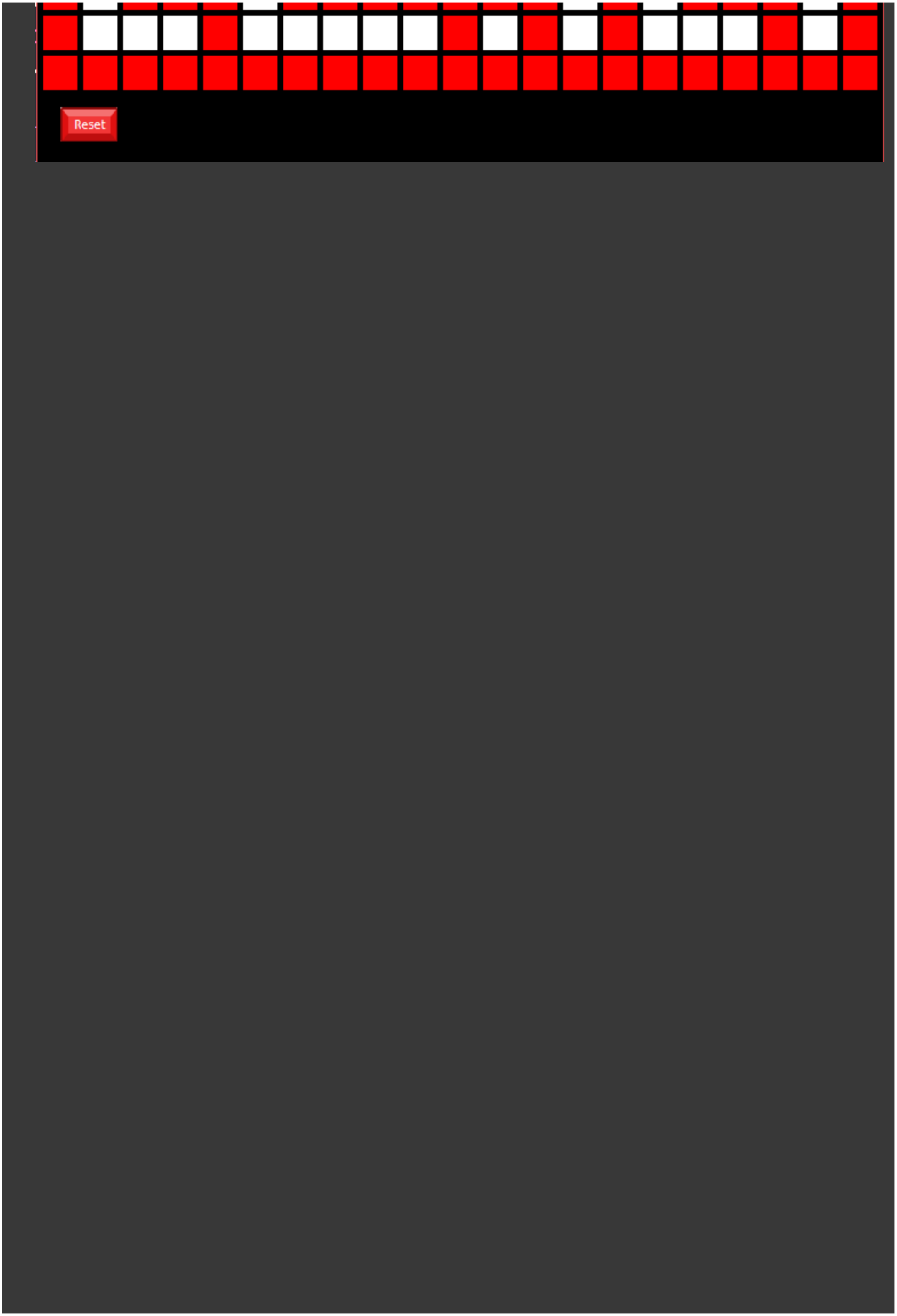
información con la que dotamos de inteligencia a nuestro algoritmo, su objetivo principal es acelerar la búsqueda.

Antes de comentar las distintas heurísticas que se han implementado, hay que comentar lo que es una **heurística admisible**.

Como se ha mencionado en las clases teóricas, una heurística es admisible si nunca sobreestima el coste de alcanzar el objetivo. Es decir, si el valor que da para cada celda es siempre menor o igual que el coste mínimo para alcanzar el objetivo.




Reset


Reset