Práctica de Sistemas Inteligentes

Grupo: G4.07

ADRIAN MORA PERELA

Oscar Perez galan

Gregorio baldomero patino esteo

2016

Contenido

[Documentación técnica. 2](#_Toc470171241)

[Clases 2](#_Toc470171242)

[App.py 2](#_Toc470171243)

[DatosUsuario.py 2](#_Toc470171244)

[Estado.py 2](#_Toc470171245)

[Pieza.py 2](#_Toc470171246)

[Espacio\_estados.py 3](#_Toc470171247)

[Problema.py 3](#_Toc470171248)

[Nodo.py 4](#_Toc470171249)

[Frontera.py 4](#_Toc470171250)

[Traza de la ejecución del programa. 4](#_Toc470171251)

[Librerías utilizadas 6](#_Toc470171252)

[Manual de usuario. 7](#_Toc470171253)

[Requisitos 7](#_Toc470171254)

[Ejecución 7](#_Toc470171255)

# Documentación técnica.

## Clases

### App.py

Clase principal de la práctica, contiene la lógica necesaria para interactuar con el resto de clases y el control del flujo principal de ejecución.

### DatosUsuario.py

Clase que contiene la lógica necesaria para interactuar inicialmente con el usuario, mostrando un menú de imágenes y el tipo de búsqueda que puede seleccionar, capturando las opciones que seleccione y mandándolas a app.py

### Estado.py

Se ha buscado que la clase estado almacene la menor cantidad de información posible, ya que como se generaran gran cantidad de estados se busca que sean lo más ligeros posible. Esta clase contiene:

* Variables.
  + Número de filas del puzzle.
  + Número de columnas del puzzle.
  + Tablero: Lista que almacena piezas del puzzle, cada pieza contiene la id de una imagen y las coordenadas (x,y) de dicha imagen en la representación en forma de matriz del puzzle.
  + Pivote: Almacena un índice que indica qué pieza del tablero es la pieza negra.
  + PosAnteriorPivote: Almacena la posición anterior del pivote, esta variable se emplea posteriormente a la hora de pintar en pantalla la solución.
* Métodos.
  + CrearEstado(): método que inicializa el estado almacenando el tablero y la posición del pivote.
  + Movimientos(): determina qué movimientos son posibles en un estado y devuelve la secuencia.
  + Gets: Todos los métodos get para obtener alguna variable.
  + Sets: todos los métodos set para establecer el contenido de una variable.

### Pieza.py

Clase que sirve como abstracción de una imagen del total de imágenes en las que se divide la imagen original. Almacena:

* id, que representa el número de imagen (del total de imágenes en las que se divide la imagen original).
* Coordenadas, que representan la posición en una matriz de dicha imagen, para comprobar los movimientos posteriormente y pintarla.
* Pivote, que almacena True o False, solo será True en la pieza negra.

### Espacio\_estados.py

Clase que contiene las operaciones necesarias en el espacio de estados. Inicialmente se pedía implementar:

* EsValido(estado):si o no
* EsObjetivo(estado): si o no
* Sucesores(estado): (acc1,estado1,costAcc1), … , (accN,estadoNostAccN)

Sin embargo, para evitar redundancia se ha definido EsObjetivo(estado) solo en la clase problema. Y si es un estado es válido o no se comprueba inicialmente al cargar las imágenes, ya que al crear el tablero inicial (estado inicial) comprueba que este estado sea válido y, puesto que solo se permiten movimientos que se saben válidos, no se dará el caso de generar estados no válidos, ya que el único susceptible de ser no valido será el primero.

Por tanto, esta clase contiene las siguientes funciones:

* Sucesores(estado, posicionPivote): obtiene los movimientos posibles del objeto estado (estado.movimientos()) y en base a esto genera los estados sucesores y los devuelve.
* Nuevo\_sucesor(movimiento, estado, coste, posPivote): método utilizado por la función anterior que crea un nuevo sucesor con los parámetros necesarios.
* Cambiar\_piezas(id1, id2, estado): método empleado por la función sucesores para cambiar de posición la pieza pivote y la pieza objetivo en base al movimiento para generar el nuevo estado, que posteriormente se almacena en un nuevo sucesor con los parámetros del método anterior.

### Problema.py

Esta clase está definida usando el estado inicial, el espacio de estados y el estado objetivo, por lo cual al crear el objeto problema hay que proporcionarle estos 3 objetos para que los almacene.

Contiene las siguientes funciones:

* esObjetivo(estado): que determina si un estado proporcionado es el estado objetivo, para ello comprueba el orden de las ids de las piezas del tablero.
* heurística(estado): determina la heurística de un estado, en este caso la heurística es el número de piezas desordenadas.

### Nodo.py

Esta clase representa el nodo de un árbol, por lo cual cada vez que se inserte en la frontera algo, deberá estar encapsulado en un nodo, que contiene:

* Padre: referencia al nodo padre del nodo actual.
* Estado: estado de dicho nodo (que será un sucesor de un nodo anterior).
* Costo: Coste de realizar la acción para generar este nuevo estado.
* Acción: Acción realizada para pasar del estado anterior al nuevo.
* Valor: valor por el cual se va a ordenar en la frontera.
* PosPiv: posición del pivote en el estado de este nodo, para acceder más rápidamente en caso de necesitarla.
* Profundidad: profundidad a la que se encuentra el nodo en el árbol.

### Frontera.py

La clase frontera contiene una lista ordenada de nodos del árbol de búsqueda. Esta lista está ordenada de menor a mayor por el campo ‘Valor’

Para esto se ha utilizado la siguiente estructura:



Esta estructura es una lista ordenada (parecida a la sorted list pero que se comporta mejor con un gran volumen, proporcionando mayor velocidad en la ordenación cuando el número de nodos se dispara) que ordena de menor a mayor por un determinado campo, en este caso se especifica que ordene por nodo.valor. La colocación del nodo de manera ordenada se produce de manera automática cada vez que se introduce un dato en la lista.

Contiene las funciones:

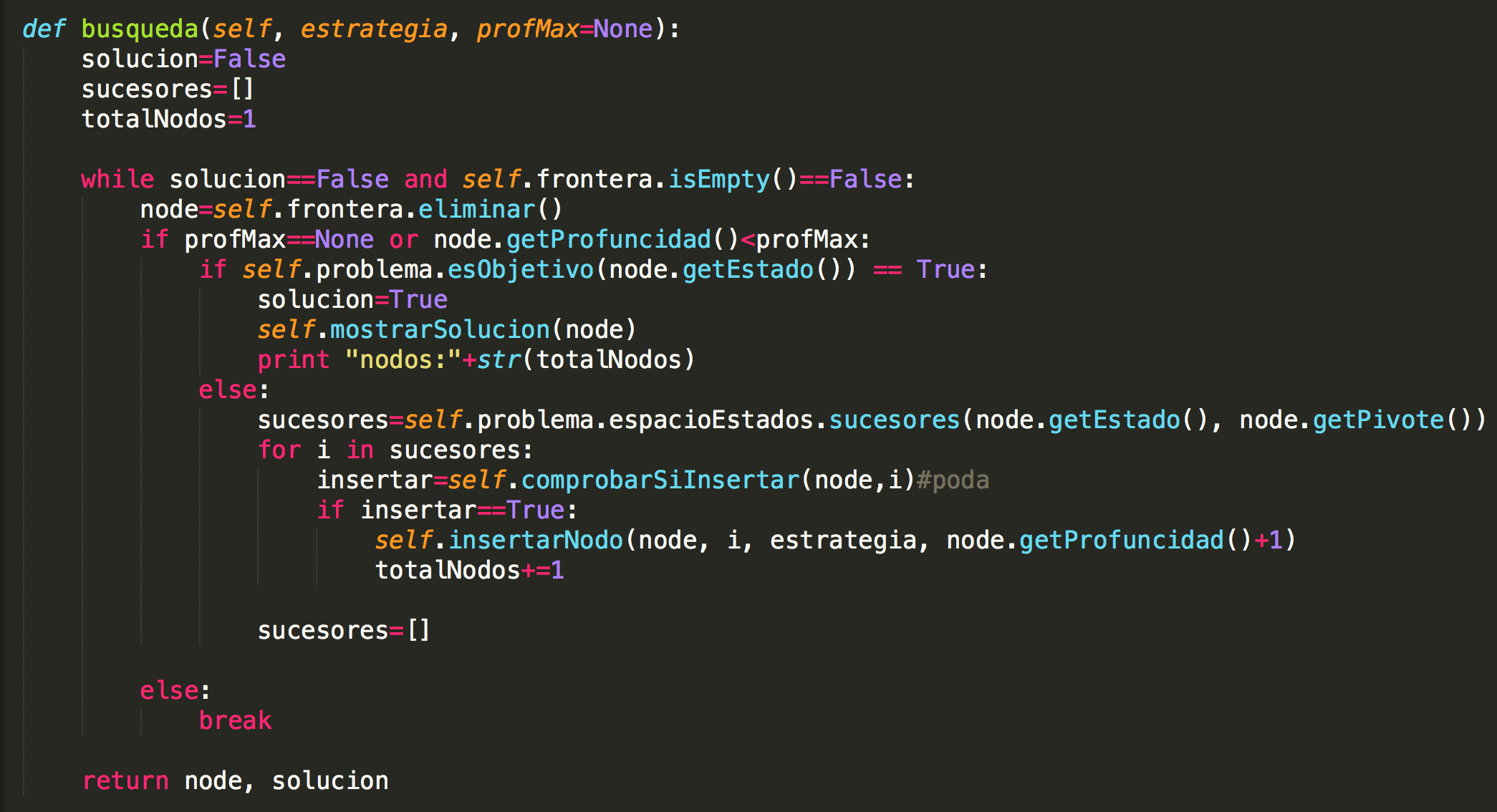
* Contructor de la frontera: inicializa la frontera vacía utilizado sortedcontainers y establece el critero de ordenación.
* isEmpty(): devuelve True si la frontera está vacía y False si no lo está.
* Insertar(nodo): inserta un nodo en la frontera.
* Eliminar(): extrae el primer nodo de la frontera (elimina y retorna el nodo).

## Traza de la ejecución del programa.

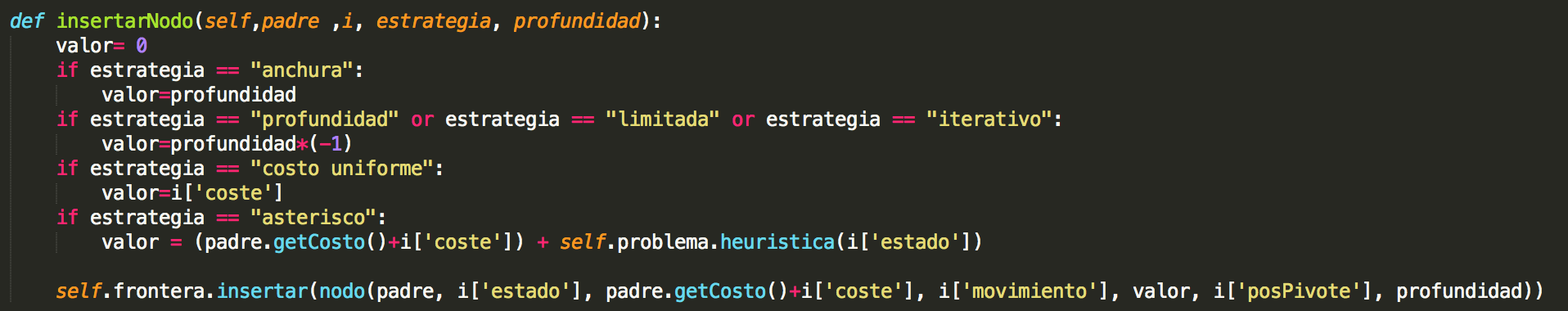
En esta sección se desglosa una traza de la ejecución del sistema para ver cómo funciona e interactúan las clases entre sí.

Cabe destacar que cuando se cargan las imágenes, se almacenan en una lista de trozos de imagen y cada **índice** de esa lista es lo que se almacena en los estados, de forma que cada estado almacene solo una secuencia de números, no trozos de imagen, ya que si fuera así el consumo de recursos del problema sería intratable.

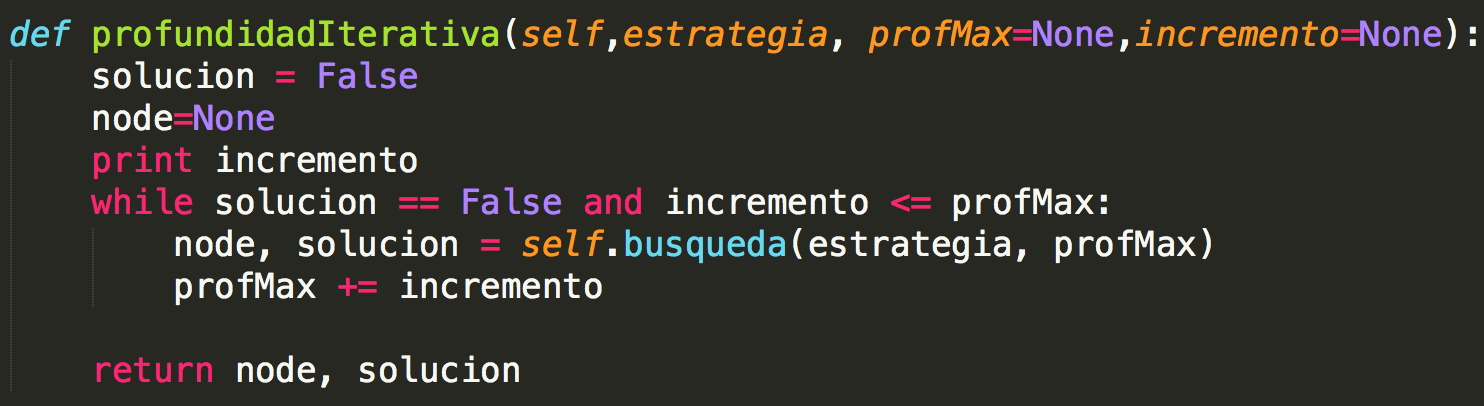
* Al ejecutar app.py se crea un objeto de la clase datosUsuario
* Al crearlo aparecerá un menú para seleccionar las fotos para el puzzle y el método de búsqueda. Una vez seleccionados mediante datosUsuario.obtenerDatosUsuario() se retorna esos datos al flujo principal del programa.
* Con estos datos desde la clase app.py se crea un objeto app al que se le pasan las dos imágenes y las filas y columnas y se lanza la aplicación.
* En primer lugar, obtener\_datos() abre las dos imágenes y establece el ancho y el alto del tablero y de cada pieza en base al tamaño de las imágenes y las filas y columnas.
* Puesto que puede darse el caso de que la imagen no tenga el cuadrado negro se le añade siempre en el método pintar\_pivote(), esté o no esté. En caso de estar simplemente se pinta encima del anterior.
* Una vez están las dos imágenes abiertas el método crear\_tablero\_original() trocea la imagen original y la imagen del estado inicial y las almacena en listas.
* Tras esto el método comprobar\_imagenes() comprueba que ambas listas de imágenes contengan los mismos trozos de imagen y asigna a las imágenes del estado inicial ids que corresponden con las imágenes de la imagen original. En caso de que no coincidan este método devolverá False y el programa avisará de que las imágenes no son iguales y no continuará la ejecución.
* Con todos estos datos se crean el estado inicial y el estado objetivo.
* Tras esto se inicializa el espacio de estados y crea la frontera.
* Se inicializa el problema, que recibe el estado inicial, el estado objetivo y el espacio de estados.
* Pide al usuario que seleccione la estrategia.
* Se llama al método seleccionarBusqueda(), el cual crea el nodo inicial (que contiene el estado inicial) y lo inserta en la frontera. Una vez hecho esto dependiendo de la estrategia llama a un método búsqueda general o al método para la búsqueda iterativa.
* busqueda(): este método se utiliza para todas las búsquedas menos la iterativa.



* El método busqueda() utiliza comprobarSiInsertar() para hacer poda: si en el movimiento n la acción es subir la pieza negra, no se puede dar el caso de que en n+1 sea bajar la pieza, ya que se generaría el mismo estado que hace dos movimientos. Esto se aplica para todas las combinaciones de movimientos.
* También hace uso de insertarNodo(), que asigna a “valor” un dato en función de la estrategia seleccionada.



* Para la búsqueda iterativa usa el método profundidadIterativa:



* Para la búsqueda siempre se toma el tiempo antes y después para mostrar el tiempo total de búsqueda. También el número de nodos y la profundidad a la que encuentra la solución.
* Una vez encuentra solución se almacena en un fichero de texto, se muestra en el terminal y se muestra gráficamente moviendo las imágenes (para esto se ha creado un método mostrar que lanza la pantalla con la imagen y un método mover que mueve las piezas por dicha pantalla).
* Se ha creado un método para limitar la memoria, ya que en caso de no encontrar solución empezará a consumir memoria al insertar miles de nodos. En caso de llegar al límite se aborta el programa.
* También se ha añadido un método que dado un estado genera una imagen.png de dicho estado.

## Librerías utilizadas

* import copy
* from time import time
* import resource
* import signal, os
* import Tkinter as tk
* from PIL import Image, ImageTk
* from os import listdir
* import sortedcontainers

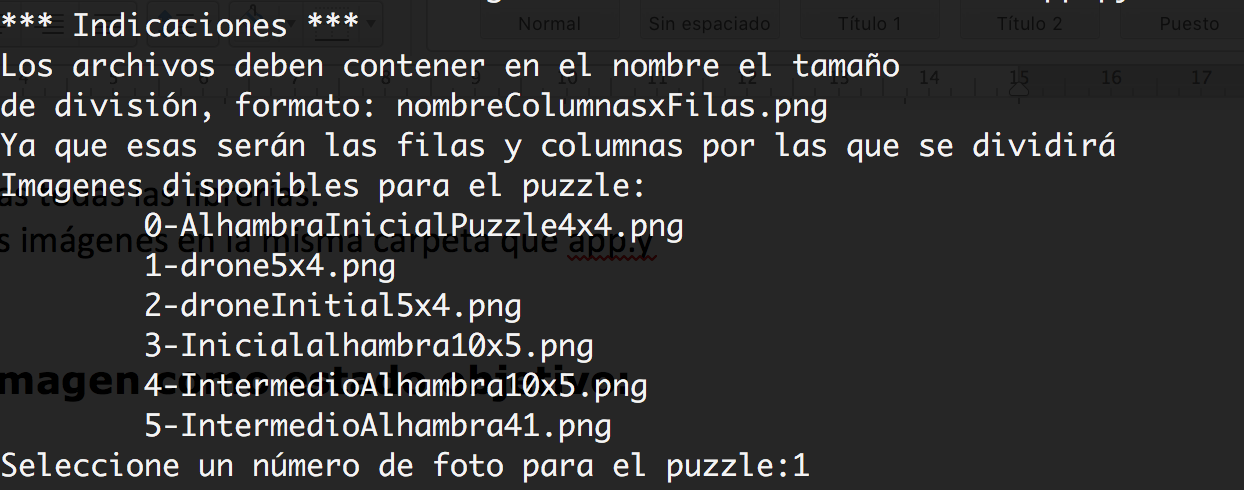
# Manual de usuario.

## Requisitos

* Tener instaladas todas las librerías (dependiendo del SO y la versión algunas vendrán ya instalas).
* Tener todas las imágenes en la misma carpeta que app.y

## Ejecución

**1.Selección de imagen como estado objetivo**

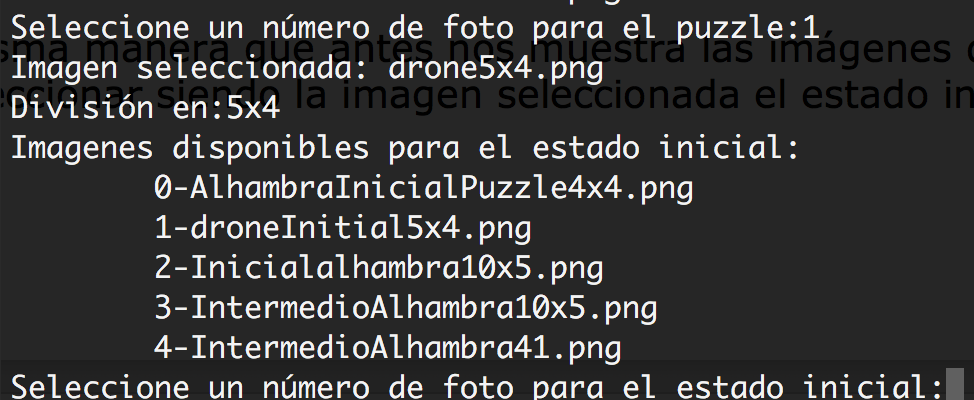


En esta imagen vemos como ejemplo unas imágenes.png que podemos seleccionar a través de sus números identificadores, la foto seleccionada será la foto en su estado objetivo.

Como ejemplo seleccionaremos la imagen drone5x4.png 🡪 1

Como se observa en las indicaciones, la división en columnas X filas debe encontrarse en el nombre de la imagen, ya que el programa las coge automáticamente de ahí.

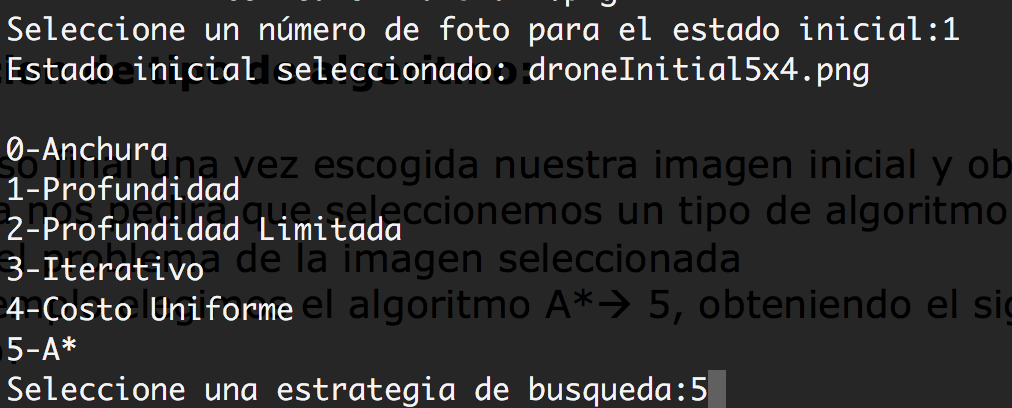
**2.Selección de imagen como estado inicial**

****

De la misma manera que antes nos muestra las imágenes disponibles para seleccionar siendo la imagen seleccionada el estado inicial de la imagen, en caso de seleccionar una imagen diferente de la cogida como objetivo (paso 1), el programa nos notificara que las imágenes no son iguales.

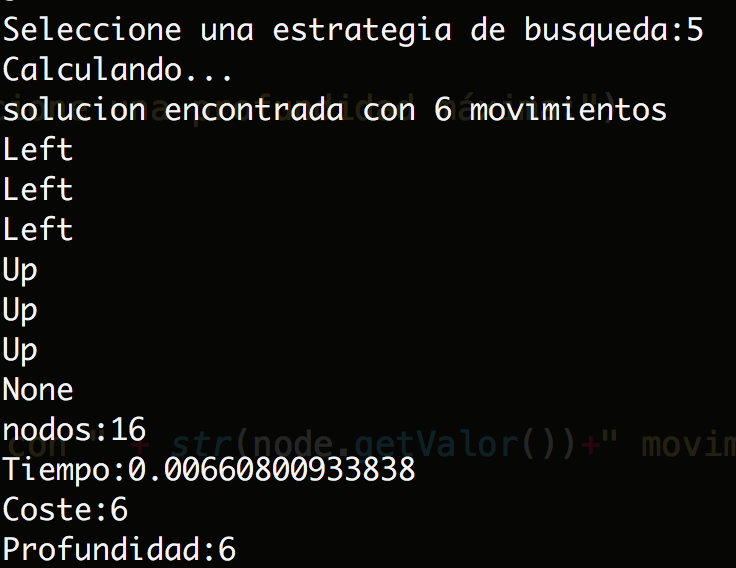
Aquí seleccionamos droneInitial5x4.png 🡪 1

**3.Seleccion de tipo de algoritmo**



Como paso final una vez escogida nuestra imagen inicial y objetivo, el programa nos pedirá que seleccionemos un tipo de algoritmo para resolver el problema de la imagen seleccionada

Como ejemplo elegimos el algoritmo A\*🡪 5, obteniendo el siguiente resultado:



La resolución de cualquier algoritmo siempre que encuentre solución nos informara sobre el número de nodos generados (16), tiempo de ejecución (0.00660800933838 ms), junto con el coste y profundidad que en este caso coinciden porque el coste es 1.

A su vez almacena el resultado en un .txt y muestra gráficamente la secuencia de movimientos:

