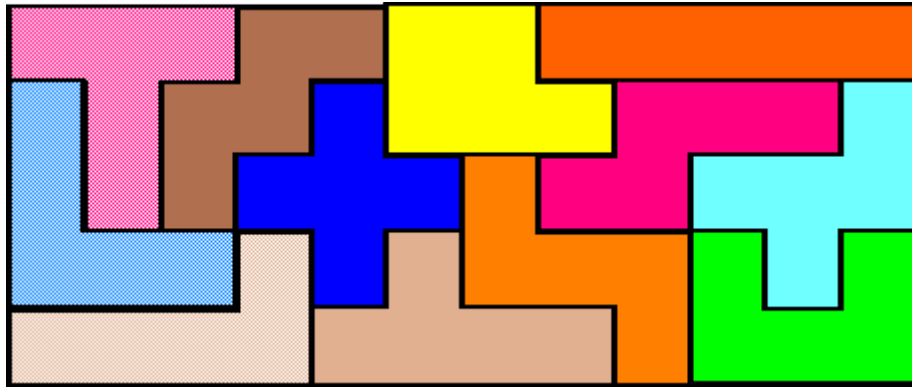


# Inteligencia Artificial



## Pentominós

<b>Integrantes del grupo</b>
Juan Alfonso Fernández Serrano
Mario Sánchez Rodríguez

Profesora: Antonia M. Chávez González

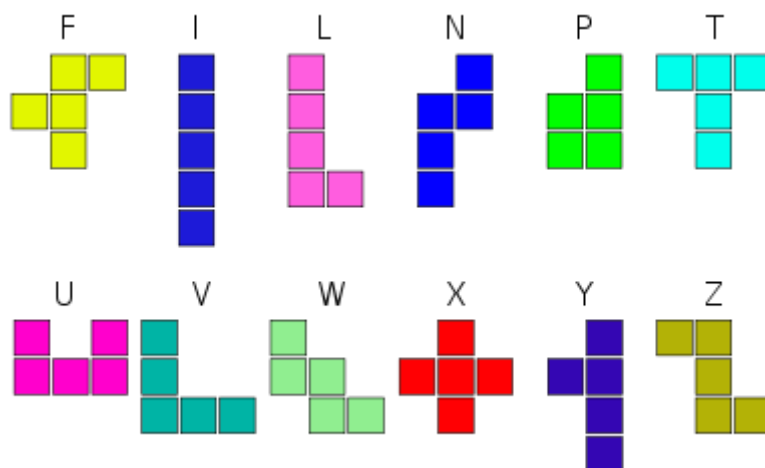
# ÍNDICE

## Contenido

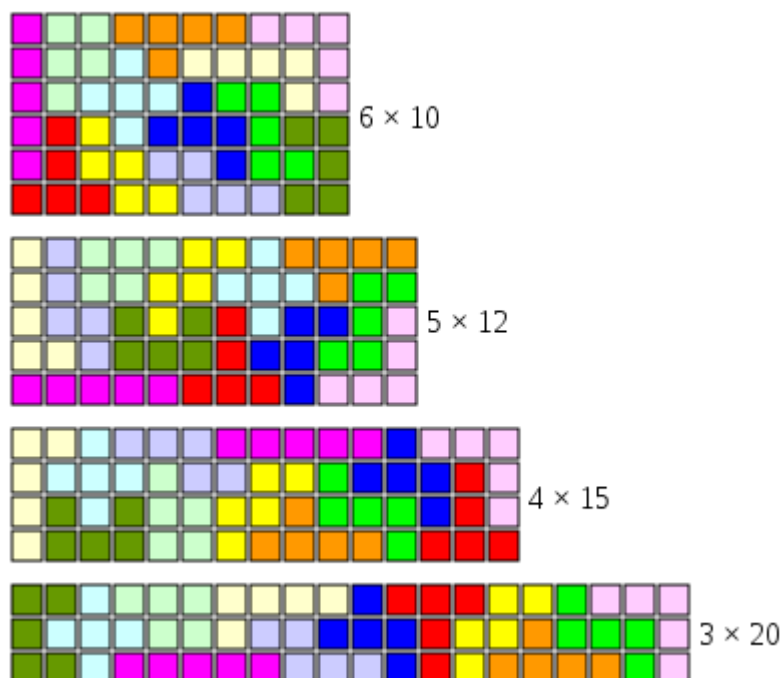
Descripción del problema .....	3
Objetivos del trabajo.....	4
Diseño de los elementos del problema .....	5
Desarrollo de la implementación.....	7
Resultados obtenidos.....	8
Problemas encontrados durante el desarrollo del trabajo.....	10
Fortalezas y debilidades.....	11
Posibles mejoras .....	12
Conclusiones .....	12
Referencias bibliográficas .....	12

## Descripción del problema

Un pentominó es una figura geométrica compuesta por cinco cuadrados unidos por sus lados. Existen doce pentominós diferentes que se nombran con las diferentes letras del abecedario a las que se asemejan.



Se pueden realizar diversas actividades con estas figuras. Por ejemplo, al ser 12 pentominós de 5 cuadrados cada uno, podemos construir rectángulos de 60 cuadrados usando una sola vez cada una de las figuras (sin huecos ni solapamientos).



En este trabajo planteamos una variante de este puzzle. Consistirá en construir un rectángulo de cualquier tamaño pudiendo repetir el uso de pentominós y permitiendo hasta 4 huecos vacíos.

## Objetivos del trabajo

- Representar mediante búsqueda en espacio de estados el problema de completar sin solapamientos y con un máximo de 4 huecos, un rectángulo de cualquier tamaño usando pentominós, no necesariamente los 12 (con sus giros y simetrías) ni necesariamente una sola vez cada uno. Implementar en Python los tipos de datos y métodos auxiliares necesarios, así como heurísticas. Pueden utilizarse las librerías de la práctica 2.
- Comparar, desde el punto de vista experimental, los diferentes algoritmos de búsqueda vistos en el tema para resolver algunas instancias no triviales de este problema, incluyendo preferiblemente diversas heurísticas. Considerar la opción de penalizar el uso de la figura I aumentando su coste respecto al de las demás figuras. Elaborar una tabla con los datos de la comparativa.
- Añadir un menú sencillo para la introducción de las condiciones de partida.
- Junto con el código (comentado) en Python, elaborar una memoria en la que se detallen las decisiones de diseño para la representación de los elementos del problema, el desarrollo de la implementación, las referencias bibliográficas y los comentarios y conclusiones sobre los resultados obtenidos en la fase de experimentación. Incluir la tabla con los datos de la comparativa.

## Diseño de los elementos del problema

Hemos diseñado los estados de tal manera que reciba tres parámetros, el primero será una matriz que hará de función de tablero de Pentominós, el segundo será el número de huecos vacíos que hay en el tablero y el tercer y último parámetro será el número de veces que se puede colocar la figura I (tanto horizontal como vertical).

Un estado será final si hay 4 huecos o menos en el tablero.

Un estado inicial será aquel estado que está compuesto por un tablero vacío y por lo tanto, número de huecos igual a filas\*columnas

Las acciones implementadas posicionarán una figura en el tablero, de tal forma que habrá una acción por cada figura.

Paso previo a colocar la figura será el de comprobar si dicha figura se puede posicionar en la fila y columna indicada (parámetros de entrada de las acciones), para ello debemos comprobar que al colocar la figura se va a situar dentro de los márgenes del tablero y comprobar que la figura a poner no se solapará con ninguna otra figura puesta previamente en el tablero.

```
class poner_Figura_I(probée.Acción):  
  
    def __init__(self, f, c):  
        nombre = 'Pongo la figura I en la posición: ({} , {} )'.format(f,c)  
        super().__init__(nombre)#este será el init de la clase heredada  
        self.fila=f  
        self.columna=c  
  
    def esta_vacio_puntos(self, estado, f, c):  
        matriz = estado[0]  
        res = [True]  
        for i in range(f, f+5):  
            cont = matriz[i][c]  
            if (cont != ' '):  
                res.append(False)  
                break  
        return all(res)  
  
    def limites(self, estado, f, c):  
        matriz = estado[0]  
        if (f + 4) <= matriz.shape[0]-1 and c <= matriz.shape[1]-1:  
            return True  
        else:  
            return False  
  
    def es_aplicable(self, estado):  
        return self.limites(estado, self.fila, self.columna) and self.esta_vacio_puntos(estado, self.fila, self.columna) and estado[2]>=0  
  
    def poner_figura(self, estado, fil, col):  
        nuevaMatriz = estado[0]  
        for i in range(fil, fil+5):  
            nuevaMatriz[i][col] = 'I'  
        estado[1] -= 5  
        estado[2] -= 1  
        return estado  
  
    def aplicar(self, estado):  
        nuevo_estado = copy.deepcopy(estado)  
        self.poner_figura(nuevo_estado, self.fila, self.columna)  
        return nuevo_estado
```

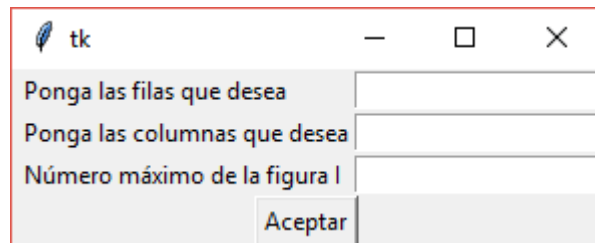
---

La creación del espacio de estados se realizará en la clase Pentominos cuando el usuario inserte los datos correspondientes en la interfaz de usuario (fila, columna, número máximo de figuras I que se puede utilizar).

```
root = Tk()
filas = Label(root, text=" Ponga las filas que desea").grid(row=0, sticky=(W))
columnas = Label(root, text=" Ponga las columnas que desea").grid(row=1, sticky=(W))
nMaxI = Label(root, text=" Número máximo de la figura I").grid(row=2, sticky=(W))
etiqueta = Label(root, text="").grid(row=2, sticky=(W))
valor = StringVar
valoresEntrada1 = Entry(root, textvariable = valor)
valoresEntrada1.grid(row=0, column=1)
valoresEntrada2 = Entry(root, textvariable = valor)
valoresEntrada2.grid(row=1, column=1)
valoresEntrada3 = Entry(root, textvariable = valor)
valoresEntrada3.grid(row=2, column=1)

boton = Button(root, text="Aceptar", command = asignarValores).grid(columnspan=2)

root.mainloop()
```



```
class Pentominos(probee.ProblemaEspacioEstados):
    def __init__(self, filas, cols, nMaxI):
        matriz = np.zeros((filas,cols), dtype=np.str_)

        acciones=[]
        cad='poner_Figura_'
        ac=[]
        letras = ['I','IH','L','Z','N','P','T','U','V','W','X','Y','F']
        for a in letras:
            res = cad+a
            ac.append(res)

        for n in ac:
            for j in range(0,cols):
                for i in range(0,filas):
                    cad2 = 'fichaas.'+n+'(i,j)'
                    acciones = acciones + [eval(cad2)]

        estado_inicial = [matriz, filas*cols, nMaxI]
        super().__init__(acciones, estado_inicial)

    def es_estado_final(self, estado):
        return estado[1] <= 4 #Tiene que haber 4 espacios o menos
```

## Desarrollo de la implementación

Comenzamos implementando las acciones a aplicar. Primero probamos únicamente con la figura I y también la clase Pentominós para poder hacer la prueba.

Al hacer la prueba vimos que no funcionaba y procedimos a solucionar el error.

Después de esto, comprobamos que había otro error en la clase *búsqueda\_espacio\_estados*. Procedimos a solucionar este error para poder seguir con la resolución del trabajo.

A continuación comenzamos a implementar todas las acciones en una factoría externa para poder hacer la prueba con varias acciones, pero nos encontramos con un problema, que era que no sabíamos cómo hacer que el algoritmo ejecutase más de una acción. Cuando conseguimos solucionar este problema, ya pudimos probar el ejercicio y comprobamos que encontraba soluciones pero en algunos casos algunas figuras se solapaban.

Tras revisar la factoría con todas las acciones, localizamos el error que se producía y lo corregimos. Entonces pudimos probar las acciones con los diferentes algoritmos y comprobamos que funcionaba correctamente.

En último lugar, procedimos a realizar un menú sencillo de inicio para establecer las condiciones iniciales del problema.

Para finalizar, organizamos, optimizamos y limpiamos todo el código para que fuese mucho más sencillo de leer e interpretar. Tras esto, comenzamos a realizar la memoria del trabajo para poder entregarlo.

## Resultados obtenidos

Algoritmo	Filas	Columnas	Nº máx. de l	Solución
Anchura ['Z','T','L','P']	6	4	0	<pre> [' ', 'Z', 'Z', ' '] ['Z', 'Z', 'Z', ' '] ['L', 'Z', 'Z', 'Z'] ['L', 'Z', 'Z', 'P'] ['L', ' ', 'P', 'P'] ['L', 'L', 'P', 'P']           </pre>
Profundidad [TODAS]	6	5	2	<pre> ['T', 'T', 'T', 'F', 'F'] ['I', 'T', 'F', 'F', 'I'] ['I', 'T', 'X', 'F', 'I'] ['I', 'X', 'X', 'X', 'I'] ['I', 'U', 'X', 'U', 'I'] ['I', 'U', 'U', 'U', 'I']           </pre>
Profundidad acotada [TODAS]	6	5	2	<pre> ['T', 'T', 'T', 'F', 'F'] ['I', 'T', 'F', 'F', 'I'] ['I', 'T', 'X', 'F', 'I'] ['I', 'X', 'X', 'X', 'I'] ['I', 'U', 'X', 'U', 'I'] ['I', 'U', 'U', 'U', 'I']           </pre>
Profundidad iterativa ['Z','T','L','P']	6	4	1	<pre> [' ', 'P', ' ', 'P'] ['P', 'P', 'P', 'P'] ['P', 'P', 'P', 'P'] [' ', 'P', ' ', 'P'] ['P', 'P', 'P', 'P'] ['P', 'P', 'P', 'P']           </pre>
A* ['Z','T','L','P']	6	4	1	<pre> [' ', 'Z', 'Z', ' '] ['Z', 'Z', 'Z', ' '] ['L', 'Z', 'Z', 'Z'] ['L', 'Z', 'Z', 'P'] ['L', ' ', 'P', 'P'] ['L', 'L', 'P', 'P']           </pre>
Óptima ['Z','T','L','P']	6	4	1	<pre> [' ', 'Z', 'Z', ' '] ['Z', 'Z', 'Z', ' '] ['L', 'Z', 'Z', 'Z'] ['L', 'Z', 'Z', 'P'] ['L', ' ', 'P', 'P'] ['L', 'L', 'P', 'P']           </pre>

En la tabla podemos observar que para los algoritmos de *búsqueda en profundidad* y *búsqueda en profundidad acotada* encuentra la misma solución para los dos casos.



Para todos los demás algoritmos, hemos limitado a únicamente 4 posibles figuras entre las que se puede elegir, porque si ponemos todas las figuras, el tiempo para encontrar una solución es excesivamente elevado.

Con esta condición, el algoritmo de *búsqueda en profundidad iterativa* encuentra una solución diferente a los anteriores casos. Introduce iterativamente la figura P hasta en 4 ocasiones para encontrar una solución válida.

Para los algoritmos de *búsqueda en anchura*, *óptima* y *A\** encuentra la misma solución en los 3 casos. Esto es debido a que no hemos podido encontrar una heurística apropiada para la resolución del ejercicio.

Algoritmo	Filas	Columnas	Nº máx. de l	Solución
Profundidad	7	4	2	<pre>[ ' ', 'F', 'F', 'P' ], [ 'F', 'F', 'P', 'P' ], [ 'I', 'F', 'P', 'P' ], [ 'I', ' ', 'F', 'F' ], [ 'I', 'F', 'F', ' ' ], [ 'I', 'U', 'F', 'U' ], [ 'I', 'U', 'U', 'U' ]</pre>
Profundidad	4	7	2	<pre>[ 'Z', 'Z', 'I', 'I', 'I', 'I', 'I' ], [ 'V', 'Z', 'T', 'T', 'T', 'F', 'F' ], [ 'V', 'Z', 'Z', 'T', 'F', 'F', ' ' ], [ 'V', 'V', 'V', 'T', ' ', 'F', ' ' ]</pre>

Como se puede observar en la tabla, hemos comprobado los resultados que se obtienen, con el algoritmo de *búsqueda en profundidad*, de dos tableros al intercambiar las filas y las columnas.

En el primer caso, seleccionamos un tablero de 7x4, y encuentra una solución dejando 3 huecos vacíos.

En el segundo caso, seleccionamos un tablero de 4x7, y encuentra otra solución distinta dejando también 3 huecos vacíos.


La razón por la que no elige la misma solución es porque no tenemos implementadas las rotaciones y simetrías de todas las figuras. De ser así, podría devolver la misma solución en los dos casos.

## Problemas encontrados durante el desarrollo del trabajo

Inicialmente las acciones no recibían ningún parámetro de entrada, de tal forma que las filas, columnas y las actualizaciones de éstas se llevaban a cabo en los estados. Dichos estados estaban compuestos por lo tanto del tablero, fila, columna y número de huecos del tablero.

```
class Pentominos(probee.ProblemaEspacioEstados):#Como
    def __init__(self, filas, cols):
        matriz = np.empty((filas,cols), dtype=np.str_)
        for i in range(matriz.shape[0]):
            for j in range(matriz.shape[1]):
                matriz[i][j] = '0'

        acciones = [poner_Figura_I()]
        estado_inicial = [matriz, 0, 0, filas*cols]
        super().__init__(acciones, estado_inicial)
```



Pero esto no nos funcionaba y pasamos a meterle las filas y las columnas (recorridas mediante dos for) a las acciones. Una vez implementado esto tuvimos un problema con la clase búsqueda\_espacio\_estados.py ya que dentro del clase ListaNodos en el método \_\_contains\_\_ se comparan los estados, y la primera posición de nuestros estados es una matriz, con lo que devolvía una matriz de True/False cuando el método solo puede devolver o True o False, por lo que tuvimos que comparar por separado los distintos componentes del estado y ponerle un .any al comparar la matriz para que devolviese True/False.

```
class ListaNodos(collections.deque):
    def añadir(self, nodo):
        self.append(nodo)

    def vaciar(self):
        self.clear()

    def __contains__(self, nodo):
        # return any(x.estado == nodo.estado
        #             for x in self)
        return any(numpy.array_equal(x.estado[0],nodo.estado[0]) and x.estado[1] == nodo.estado[1] and x.estado[2] == nodo.estado[2]
                    for x in self)
```

Por último tuvimos un problema a la hora crear la lista de acciones, y es que no sabíamos demasiado bien como se concatenaban listas en Python. Esto lo solucionamos concatenando las acciones con el símbolo “+”.

## Fortalezas y debilidades

- **Fortalezas**

- Menú inicial sencillo e intuitivo

Hemos implementado un menú inicial muy sencillo e intuitivo que permita al usuario indicar el punto de partida del pentominós. Podrá decidir el número de filas y columnas de la matriz a rellenar, así como el número máximo de fichas I (tanto verticales como horizontales) que se pueden usar para encontrar la solución.

- Posibilidad de limitar el número de fichas I que se pueden usar

El usuario podrá decidir cuántas fichas I (tanto verticales como horizontales) que se podrá usar para encontrar la solución. De esta manera puede limitar el uso de esta ficha para que el algoritmo no la use de forma excesiva para encontrar la solución.

- No solapamiento de fichas

La implementación del código se ha realizado de forma que no haya solapamiento de fichas para rellenar todos los espacios del tablero. Así, podrá verse claramente la representación de las diferentes fichas que se han utilizado para llegar a la solución.

- Código optimizado y organizado

El código implementado para la realización del trabajo ha sido optimizado y claramente organizado para que sea mucho más cómodo y fácil poder interpretarlo y encontrar posibles errores y fallos de programación. Además, que haya sido optimizado, hace que el código no sea excesivo ni sobrecargado.

- Factoría externa para las acciones de las fichas

Hemos implementado una factoría externa para las acciones de las fichas que es usada en la clase principal. De esta manera queda el código de una manera mucho más limpia y optimizada.

- Solución detallada paso a paso de las acciones aplicadas

La solución muestra paso a paso las acciones aplicadas hasta llegar a la solución final. De esta manera podemos ver de forma sencilla las fichas que se han usado en cada paso.

- **Debilidades**

- No se ha encontrado heurística apropiada  
No hemos conseguido encontrar una heurística apropiada y óptima para el ejercicio.
- Tiempo elevado para encontrar solución con algunos algoritmos  
El tiempo para encontrar una solución es muy elevado en algunos algoritmos como puede ser *búsqueda en anchura* o *algoritmo A\**.

## Posibles mejoras

Las posibles mejoras a realizar en el trabajo son las dos debilidades.

Por un lado encontrar una heurística apropiada que haga que el tiempo para encontrar la solución con el *algoritmo A\** sea menor y además el resultado sea óptimo.

Por otro lado, habría que conseguir reducir significativamente el tiempo para encontrar una solución con el algoritmo de *búsqueda en anchura*, ya que actualmente ese tiempo es excesivamente elevado.

Como última posible mejora, se pueden implementar todas las figuras rotadas y su simetría, para que pueda haber mayor capacidad de elección a la hora de seleccionar una figura.

## Conclusiones

Tras haber realizado el trabajo, hemos podido observar y comprobar de forma experimental las diferentes soluciones que se encuentran para un mismo ejercicio dependiendo del algoritmo empleado.

Hemos probado que la parte de programación e implementación ha sido correcta, ya que para todos los algoritmos probados, se ha encontrado alguna solución.

En el caso de haber encontrado una heurística apropiada al problema, también podríamos haber reducido el tiempo de espera para encontrar una solución y haber probado a resolver el ejercicio con todas las figuras posibles, así como con sus rotaciones y simetrías.

Debido a que el tiempo de espera para encontrar la solución es excesivamente elevado, no hemos implementado todas las rotaciones y simetrías de todas las figuras, ya que con las 13 posibles figuras a elegir que hemos implementado, algunos algoritmos se quedaban buscando una solución durante largo tiempo.

## Referencias bibliográficas

[www.youtube.com](http://www.youtube.com)

[www.stackoverflow.com](http://www.stackoverflow.com)