# Laboratorio Sistemas Inteligentes

Grupo B02

Rosa María Fernández-Baíllo Callejas

# Índice

# Tarea 1

## En qué consistió

La tarea 1 pedía un procesamiento inicial de un nivel de sokoban introducido por consola. Junto al nivel lo acompaña un parámetro "-l" o "-level" indicador de que el siguiente elemento es el nivel.

En esta tarea se definía por tanto el dominio del problema y se pedía un tratamiento inicial de este string definiendo su número de filas, columnas, identificación y localización por coordenadas de elementos del nivel (paredes, cajas, objetivos, jugador) y finalmente un identificador del nivel formado por la codificación en MD5 de la tupla de coordenadas (jugador, cajas).

## Decisiones tomadas

De las decisiones tomadas a destacar para esta primera tarea no hay mucho de lo que hablar.

Se puede mencionar el tratamiento del nivel en una lista en la que cada elemento es una fila en lugar de una simple matriz de dos dimensiones, aunque la lista de filas es bidimensional en si ya que se puede recorrer cada string dentro de esa lista. Se pasa por una serie de ifs para ver qué tipo de carácter del nivel es para tenerlos guardados en listas juntos y con un método aparte se imprimen con el formato correcto y la codificación pedida para el ID.

La comprobación inicial de parámetros se ha dejado hecha pensando en ampliar más opciones de entrada aparte del "-l" o "-level" pensando a futuro porque no costaba nada.

Las comprobaciones extras que se añadieron están puestas en el main, pero hay preparado un método para usarse modularizado según sean los cambios para la Tarea 2.

El método walls_closed no está completo, tarde y sin tiempo para pensarlo bien y rectificar (ya estaba hecho el pull request a la rama main) me he enterado por un compañero que los niveles no solo pueden ser cuadrados o rectangulares, cosa que no se mencionó en nuestro laboratorio...

Para esta nueva comprobación se me ocurre de primeras un método recursivo que según la fila del nivel en la que esté compruebe en una dirección u otra si el siguiente carácter es #, esto recursivo hasta los límites del nivel con un valor de control boolean isDentro() para cubrir posibles niveles en forma de E o con múltiples capas horizontales que se unen solo por un pasillo. Aun así, esto no está escrito aún ya que me parece complicado y debe haber una forma mejor.

# Tarea 2

## En qué consistió

La tarea número dos pide implementar dos nuevas funcionalidades T2T y T2S.

- T2T consiste en comprobar si el nivel introducido por consola es objetivo, es decir, si está completado el nivel y todas las cajas están en los objetivos.
- T2S es más complejo y pide generar los sucesores al nivel introducido, los sucesores serán los posibles movimientos que el jugador puede hacer:
  - Moverse en vertical u horizontal (u = up, d = down, l = left, r = right)
  - Si hay una caja en su adyacente para ese movimiento empujarla si se pudiera.

Además, se pide sacar los sucesores posibles usando una vez más la codificación del nivel nuevo del sucesor al cambiar las posiciones. Y la petición extra de las comprobaciones para ver si el nivel es válido siguen en el aire viendo que es demasiado complicado ver si las paredes están cerradas sin añadir mucha complejidad

## Cómo se ha resuelto y decisiones tomadas

Tras un cambio de perspectiva a como se explica en el enunciado que el jugador puede hacer los movimientos (u, U, r, R, d, D, l, L) he optado por simplificarlo haciendo solo las direcciones y si hay o no caja procesarlo como corresponda dejando la acción en minúscula o mayúscula.

Antes de esto estuve pegándome bastante con problemas ya que no se comprobaban si había paredes o cajas en la siguiente posición y empezar de nuevo fue la mejor decisión ya que se simplifica bastante.

Viéndolo paso a paso empieza con la definición de las acciones en orden y un diccionario con el cambio de coordenadas para cada movimiento para que sea sencillo cambiar el orden en la lista acciones o añadir nuevos movimientos

```
acciones = ['u', 'r', 'd', 'l']
sucesores = []

movimientos = {
    'u': (-1, 0), 'r': (0, 1),
    'd': (1, 0), 'l': (0, -1)
}
```

Seguido del bucle para iterar por las distintas opciones de movimientos y con un if que abarca hasta el final de método para ver si la nueva posición del jugador aplicado un movimiento está en pared.

```
for accion in acciones:
    dr, dc = movimientos[accion]
    jugador_nuevo = (level_parser.player[0] + dr, level_parser.player[1] + dc)
    nuevo_estado = level_parser

    # Intentar mover el jugador
    if jugador_nuevo not in level_parser.walls:
```

Una vez tenemos que la posición a la que vamos es válida comprobamos si es solo movimiento del jugador mirando si esa coordenada está en la lista de que guarda las cajas, si no es un movimiento de jugador y se añade como sucesor con el nuevo código de id.

```
# Intentar mover el jugador
if jugador_nuevo not in level_parser.walls:
    # Si la casilla está vacía, solo se mueve el jugador
    if jugador_nuevo not in level_parser.boxes:
        sucesores.append((accion, nuevo_estado.encode_id2(jugador_nuevo, nuevo_estado.boxes), 1))
```

Si no se cumple el not in boxes entonces la nueva posición del jugador es una caja y no vale como movimiento ni como nueva posición de jugador. Vamos a ver si se puede mover la caja y si se puede hacemos una acción de empuje.

Guardamos esta nueva posición como una caja, porque hemos visto que es una caja y miramos si la siguiente posición con el mismo desplazamiento de acción no es ni otra caja ni una pared, diciendo así que podemos mover la caja y seguir con el empuje.

Si se puede actualizamos la posición de la caja quitándola de la lista y añadiéndola en la nueva posición, actualizamos el jugador y lo devolvemos como a la lista con la acción en mayúscula.

```
# Si hay una caja intentamos moverla
else:
    caja_nueva = (jugador_nuevo[0] + dr, jugador_nuevo[1] + dc)
    # Verificar que la casilla siguiente a la caja esté libre
    if caja_nueva not in level_parser.walls and caja_nueva not in level_parser.boxes:

        nuevas_cajas = level_parser.boxes.copy()
        nuevas_cajas.remove(jugador_nuevo)
        nuevas_cajas.append(caja_nueva)
        nuevas_cajas.sort() #ordenar cajas por si se mueven de fila

        sucesores.append((accion.upper(), nuevo_estado.encode_id2(jugador_nuevo, nuevas_cajas), 1))
```

En cuanto a la parte de T2T es solo comprobar que las coordenadas de la lista cajas están en la lista de objetivos para saber que está solucionado el nivel.

```
def is_objetive(level_parser):
    for box in level_parser.boxes:
        if box not in level_parser.targets:
            return False
    return True
```

En esta tarea también se cambian los métodos de la tarea 1 a una clase llamada levelParser como clase para guardar y gestionar niveles como objetos y estos métodos se moverán futuramente a la clase level parser dejando sokoban más limpio ya que el método usado para T2S deberá duplicarse y adaptarse para la tarea 3 como veremos.

# Tarea 3

## En que consistió

La tarea 3 pide desarrollar el algoritmo de búsqueda dado y todas las clases y métodos que lo rodean como Nodo, Frontera y gestión de estados visitados.

La frontera se pide que sea una estructura ordenada para almacenar los nodos del árbol de búsqueda pidiendo que se ordenen por valor (y añado de después por id como siguiente criterio) y que se inserten ya ordenados para evitar reordenar tras cada inserción.

Los nodos se pide una definición concreta que llama a que sea una clase y se reutilice el generar sucesores de la tarea 2 con cambios para funcionar con nodos en lugar de conjuntos en una lista como se tenía. Y se deja la heurística en 0.00 para todas las estrategias de esta tarea. Nota que esto no se va a calcular ni cuando se añada la heurística en la tarea 4 porque si no fallan las pruebas.

El árbol se presenta en una figura con las estrategias de expansión a tener en cuenta siendo anchura (BFS), profundidad (DFS) y coste uniforme (UC).

## Cómo se ha resuelto y decisiones tomadas

Lo primero ha sido ver qué clases se van a implementar y cuales no, por ejemplo, la frontera tiene su propia clase para gestionar la lista que es permitiendo más fácil la inserción ordenada y devolución de nodos con la librería heapq.

```
class Frontera:
    def __init__(self):
        self.frontera = []
        heapq.heapify(self.frontera)

def add(self, nodo : Nodo)-> None:
    heapq.heappush(self.frontera, (nodo.valor, nodo.id, nodo))

def pop(self) -> Nodo:
    _, _, nodo = heapq.heappop(self.frontera)
    return nodo
```

Y por otro lado se ha visto innecesario hacer una clase Problema para gestionar estado inicial y estado objetivo como esto está hecho de antes y se puede gestionar pasando la estrategia directamente al algoritmo de búsqueda y reducir la complejidad ciclomática.

Para el algoritmo de búsqueda ha habido cambios múltiples veces por tema de datos dejados fuera de las indicaciones o las pruebas, las más notables siendo DFS empieza en 1 el valor por la división para calcular el valor después, o que no se calcula heurística salvo para voraz y A*, o que el redondeo tiene que ser comprobado si estamos con DFS.

El resto ha sido solucionar problemas con la generación de sucesores para nodos adaptando el método de la tarea 2 para que no arrastre desvíos en las comprobaciones de una iteración a otra con cosas como deepcopy para los posibles nuevos estados.

El algoritmo en si se ha dejado sencillo, introduciendo el nodo inicial (ya creado en el main con el nivel que se haya metido, la estrategia y la profundidad máxima.

Tenemos en primer lugar un contador general para los id de los nodos, la inicialización de la frontera, el conjunto para guardar los estados visitados y un boolean a false para ver si hemos llegado a la solución. Y se añade el estado inicial a la frontera para empezar el bucle de la búsqueda.

```
def algoritmo_busqueda(nodoInicial, estrategia, max_prof):

    estrategia = estrategia.upper()
    id_nodo = 0

    if estrategia == 'DFS':
        nodoInicial.valor = 1.0

    frontera = Frontera()

    visitados = set()
    solucion = False

    frontera.add(nodoInicial)
```

Mientras la frontera tenga nodos y no tengamos la solución del nivel se irán sacando nodos de la frontera, si no son objetivo sacamos sus sucesores ponemos como nodos con el valor que les toque en función de la estrategia y se meten a la frontera

```
while not frontera.is_empty() and not solucion:
    nodo = frontera.pop()

    if sokoban.is_objetive(nodo.estado):
        solucion = True

    elif nodo.estado.id not in visitados and nodo.profundidad < int(max_prof):
        visitados.add(nodo.estado.id)

        for sucesor in nodo.generar_sucesores():
            id_nodo += 1
            #nodo id, padre, estado, profundidad, costo, heuristica, accion
            nodo_sucesor = Nodo(id_nodo, nodo, sucesor.estado, nodo.profundidad + 1, nodo.costo + 1, 0, sucesor.accion)

            if estrategia == 'A*' or estrategia == 'GREEDY': #solo t4
                nodo_sucesor.heuristica = nodo_sucesor.calcular_heuristica()

            nodo_sucesor.valor = valor_estrategia(nodo_sucesor, estrategia)

            frontera.add(nodo_sucesor)
```

```
def valor_estrategia(nodo, estrategia):
    valor = 0.0
    if estrategia == 'A*':
        valor = nodo.costo + nodo.heuristica
    elif estrategia == 'GREEDY':
        valor = nodo.heuristica
    elif estrategia == 'BFS':
        valor = nodo.profundidad
    elif estrategia == 'DFS':
        valor = 1/(1 + nodo.profundidad)
    elif estrategia == 'UC':
        valor = nodo.costo

    return valor
```

Cuando tengamos la solución o en su defecto no queden nodos para mirar de la frontera salimos del bucle y construimos el camino a la solución del nivel y lo devolvemos al main.

```
camino = []
if solucion:
    camino = nodo.funcion_camino()

return camino #camino vacío no hay solución
```

Una vez tenemos el camino solución lo imprimimos con el formato pedido de decimales y el decimal extra si estamos con DFS para las pruebas.

```
def __str__(self) -> str:
    costo_redondeado = round(self.costo, 2)
    heuristica_redondeada = round(self.heuristica, 2)
    valor_redondeado = round(self.valor, 2)

    #<Node ID>, <State ID>, <Parent ID>, <Accion>, <Profundidad>, <Costo>, <Heuristica>, <Valor>
    return (f"{self.id}, {self.estado.id}, {self.parent.id if self.parent else 'None'}, {self.accion}, "
            f"{self.profundidad}, {costo_redondeado:.2f}, {heuristica_redondeada:.2f}, {self.redondear_tercer_decimal(self.valor):.2f}")
```

```
def redondear_tercer_decimal(self, valor):
    aux_value = valor * 1000
    tercer_decimal = int(aux_value) % 10
    if tercer_decimal >= 5:
        return math.ceil(valor * 100) / 100
    else:
        return math.floor(valor * 100) / 100
```

A destacar de esta tarea, en el bucle del *algoritmo_busqueda*, simplemente que los visitados se comprueban guardando el id codificado del nivel en MD5 del jugador y las cajas de la tarea 1. Nodo.estado es un objeto level parser o nivel para que se entienda (le cambiaré el nombre) y el id se lo he metido como un parámetro más para que se identifique mejor por ese código que leer todos los caracteres del nivel para compararlo.

# Tarea 4

## En qué consistió

Esta tarea pide añadir dos estrategias más para la búsqueda, A* y voraz, además de añadir el cálculo de heurística Manhattan para estas dos según la formula dada en el libro de laboratorio.

## Cómo se ha resuelto y decisiones tomadas

Para implementar estas estrategias basta con añadir su calculo del valor al método correspondiente que ya teníamos y crear un método que calcule la heurística para cambiarla si estamos en estas estratégias:

```python
def valor_estrategia(nodo, estrategia):
    valor = 0.0
    if estrategia == 'A*':
        valor = nodo.costo + nodo.heuristica
    elif estrategia == 'GREEDY':
        valor = nodo.heuristica
    elif estrategia == 'BFS':
        valor = nodo.profundidad
    elif estrategia == 'DFS':
        valor = 1/(1 + nodo.profundidad)
    elif estrategia == 'UC':
        valor = nodo.costo

    return valor
```

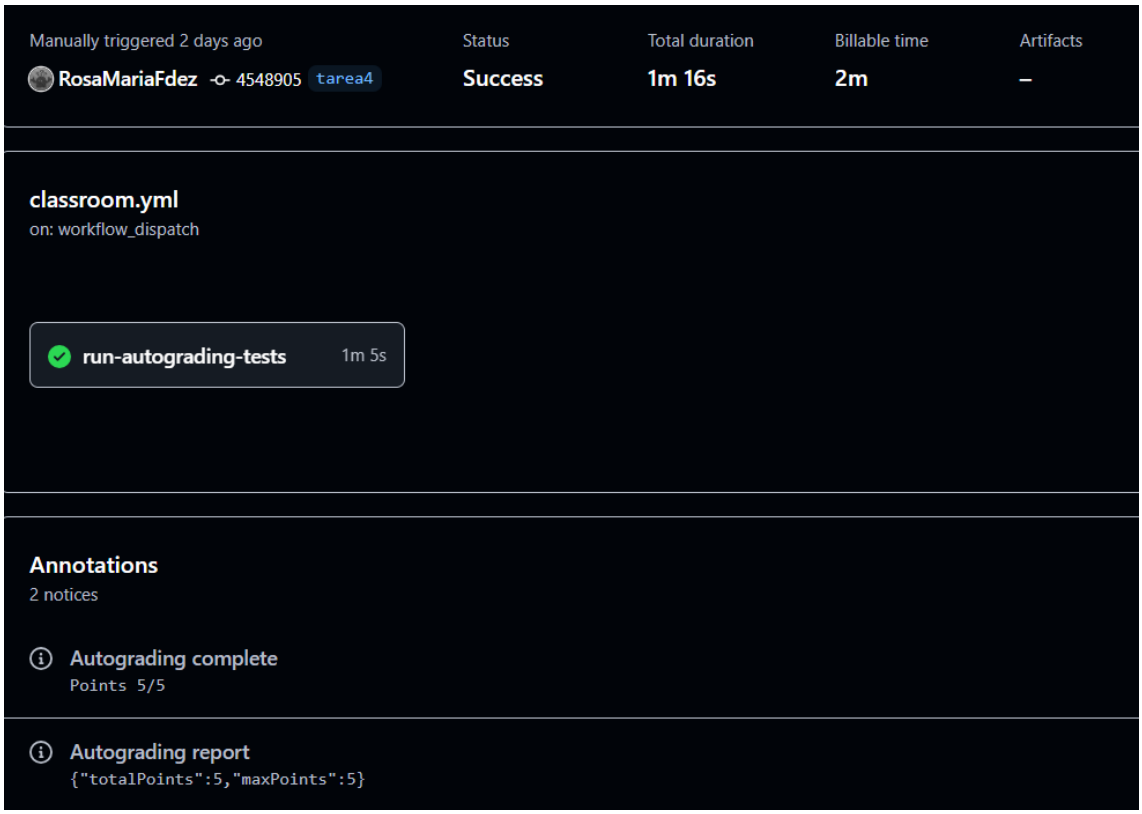
```python
def calcular_heuristica(self):
    heuristica = 0.0
    for box in self.estado.boxes:
    # distancia mínima de esta caja a objetivo
        heuristica += min(
            abs(box[0] - target[0]) + abs(box[1] - target[1])
            for target in self.estado.targets
    )
    return heuristica
```

Y en el algoritmo de búsqueda llamarlo si estamos en las nuevas estrategias

```python
if estrategia == 'A*' or estrategia == 'GREEDY': #solo t4
    nodo_sucesor.heuristica = nodo_sucesor.calcular_heuristica()
```

# Código final

El código final se puede encontrar en [este github](), pasando todas las pruebas de los test e igual que se expone en este documento a falta de quizá unas modificaciones de organización de algún método para mayor claridad.

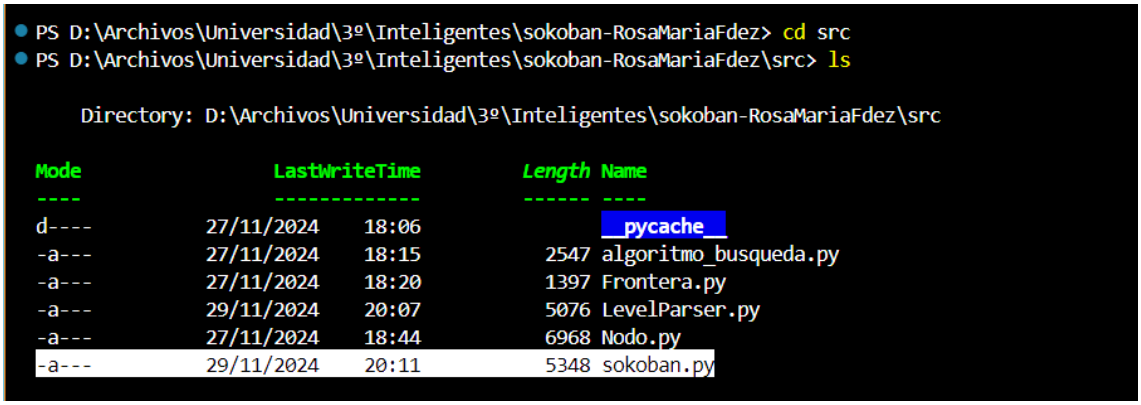


# Manual de usuario

Para usar el código realizado para este laboratorio basta con descargarlo del repositorio o clonarlo y ejecutarlo por la línea de comandos, bien sea dentro de un IDE como Visual Studio Code o la terminal de sistema.

## 1) Set up

Es necesario moverse a la carpeta **src** donde están los ficheros .py del código para tener disponible **sokoban.py**, nuestro archivo principal para ejecutar cualquier función implementada




## 2) Construcción de comando

Para ejecutar sokoban.py se debe seguir las siguientes estructuras dependiendo de la tarea a ejecutar:

- **Tarea 1**

| python | **sokoban.py** | T1 | -l (o -level) | 'nivel' |
|---|---|---|---|---|
| interprete | archivo | Tarea para ejecutar | Parámetro para indicar que después va el nivel | Nivel metido entre comillas simples o dobles |

Esto nos devolverá como se pedía en la tarea 1 la información del nivel introducido

Ejemplo:

python **sokoban.py** T1 -l '#####\n#.$@#\n# * #\n#####'

- **Tarea 2**

| python | **sokoban.py** | T2S o T2T | -l (o -level) | 'nivel' |
|---|---|---|---|---|
| interprete | archivo | T2S para generar los sucesores del nivel introducido o T2T para saber si el nivel introducido está resuelto o no | Parámetro para indicar que después va el nivel | Nivel metido entre comillas simples o dobles |

Ejecutando T2S veremos por pantalla una lista con los sucesores del nivel introducido (posibles movimientos) con su acción, id codificado y coste.

Ejecutando T2T veremos por pantalla TRUE o FALSE en respuesta a si el nivel metido está solucionado o no.

Ejemplo:

python **sokoban.py** T2T -l '#####\n#*@ #\n# * #\n#####'

python **sokoban.py** T2S -l '#####\n#.$@#\n#$. #\n#####'

- **Tarea 3**

No lo puedo poner en tabla por falta de espacio así que siguiendo el modelo de la tarea 1 y 2 cambiaremos solo para indicar T3 y añadiremos después del nivel:

| ... | -s | (aquí la estrategia elegida) | -d | Número para la profundidad |
|---|---|---|---|---|
| ... | Parámetro, indica que después va la estrategia | Puede ser:<br>DFS<br>BFS<br>UC | Parámetro para saber que después va la profundidad máxima | Debe ser un entero |

Ejemplo:
python3 **sokoban.py** T3 -l '##########\#### @# #\#### # #\#### $# #\n# $. .## #\n# ### $#\n# ### .#\n##########' -s BFS -d 100

*Es importante conocer la notación usada para la representación de los niveles, incluida al final del manual*

- **Tarea 4**

Ya que la tarea 4 solo añade algunas cosas para la 3 se utiliza el mismo comando solo que con las estrategias nuevas añadidas en esta parte, que son A* o GREEDY, así que **se deja T3**

| | | | | |
|---|---|---|---|---|
| ... | -s | (aquí la estrategia elegida) | -d | Número para la profundidad |
| ... | Parámetro, indica que después va la estrategia | Puede ser:<br>A*<br>GREEDY | Parámetro para saber que después va la profundidad máxima | Debe ser un entero |

Ejemplo:

python **sokoban.py** T3 -l '##########\n#### @# #\n#### # #\n#### $# #\n# $. .## #\n# ### $#\n# ### .#\n##########' -s A* -d 100

## Representación de un nivel

El nivel de Sokoban se representa como una cadena de caracteres donde cada carácter tiene un significado específico relacionado con los elementos del tablero

| Símbolo | Elemento | Descripción |
|---|---|---|
| **#** | Pared | Representa los límites o barreras del tablero |
| **@** | Jugador | El personaje controlado para mover las cajas |
| **$** | Caja | Objeto que el jugador debe empujar hacia los objetivos |
| **.** | Objetivo | Lugar donde deben colocarse las cajas |
| ***** | Caja sobre objetivo | Ya hay una caja colocada sobre un objetivo aquí |
| **+** | Jugador sobre objetivo | El jugador está situado en una celda objetivo |
| **Espacio ‘ ‘** | Espacio vacío | Celda sin elementos por donde el jugador puede moverse o empujar cajas |
| **/n** | Fin de fila del nivel | Indica que es el final de esta fila del nivel y que los caracteres posteriores corresponden a otra fila |

Por ejemplo, esta cadena de caracteres:

"#####\n#  #\n#@$.#\n#  #\n#####"

Para representar este nivel:

```
#####
#   #
#@$.#
#   #
#####
```

# Opinión personal

Lo que opinemos o no los alumnos sobre el laboratorio es irrelevante para los profesores que dirigen la asignatura como ya hemos visto del curso anterior.

Si bien es cierto que este año el problema a solucionar era menos complicado de configurar sin tener que redimensionar mapas, ha sido bien compensado añadiendo más documentación al código y pasando de ser un trabajo en equipo a un trabajo individual. ¿Para demostrar qué?

Me gustaría que se escuchase a los estudiantes cuando presentan problemas de manera formal, amable y comunicativa como se hizo para evitar conflictos y situaciones desesperantes que resulta bizarro ver en una universidad en estos años. De otra forma se alimenta un fuego y rabia entre la gente que solo nos envenena y separa del buen ambiente que se ha intentado buscar siempre entre la dirección de la escuela y la delegación de alumnos que no puede sino ver estos casos de manera impotente.

Nada en contra la asignatura ni los profesores, todo lo contrario, agradecida por la ayuda de Jesús Oviedo y de David Carneros el curso pasado, pero me siento terriblemente impotente y resentida por las situaciones que se han creado y se siguen haciendo más y más conforme pasa el tiempo. Esta no es la manera.

Como fan de los juegos de puzles me ha gustado la temática del laboratorio de este año y habría sido aún mejor si el trabajo lo hubiera podido hacer con alguien más a mi lado ya que, trabajos de esta envergadura hacen que cuando te atascas y no ves el fallo estás solo, y solo te quedas, costándote demasiado tiempo cada error.

En ingeniería informática son muy raros los trabajos que requieren que se trabaje individualmente y esto es algo en lo que se nos ha insistido siempre para aprender a trabajar en equipos.

La temática de trabajo me ha gustado, la forma de hacerlo no, y la evolución que está siguiendo la asignatura mucho menos, pero da igual porque es solo una opinión, y por suerte para mi ya tenía decidido no hacer la rama de computación.